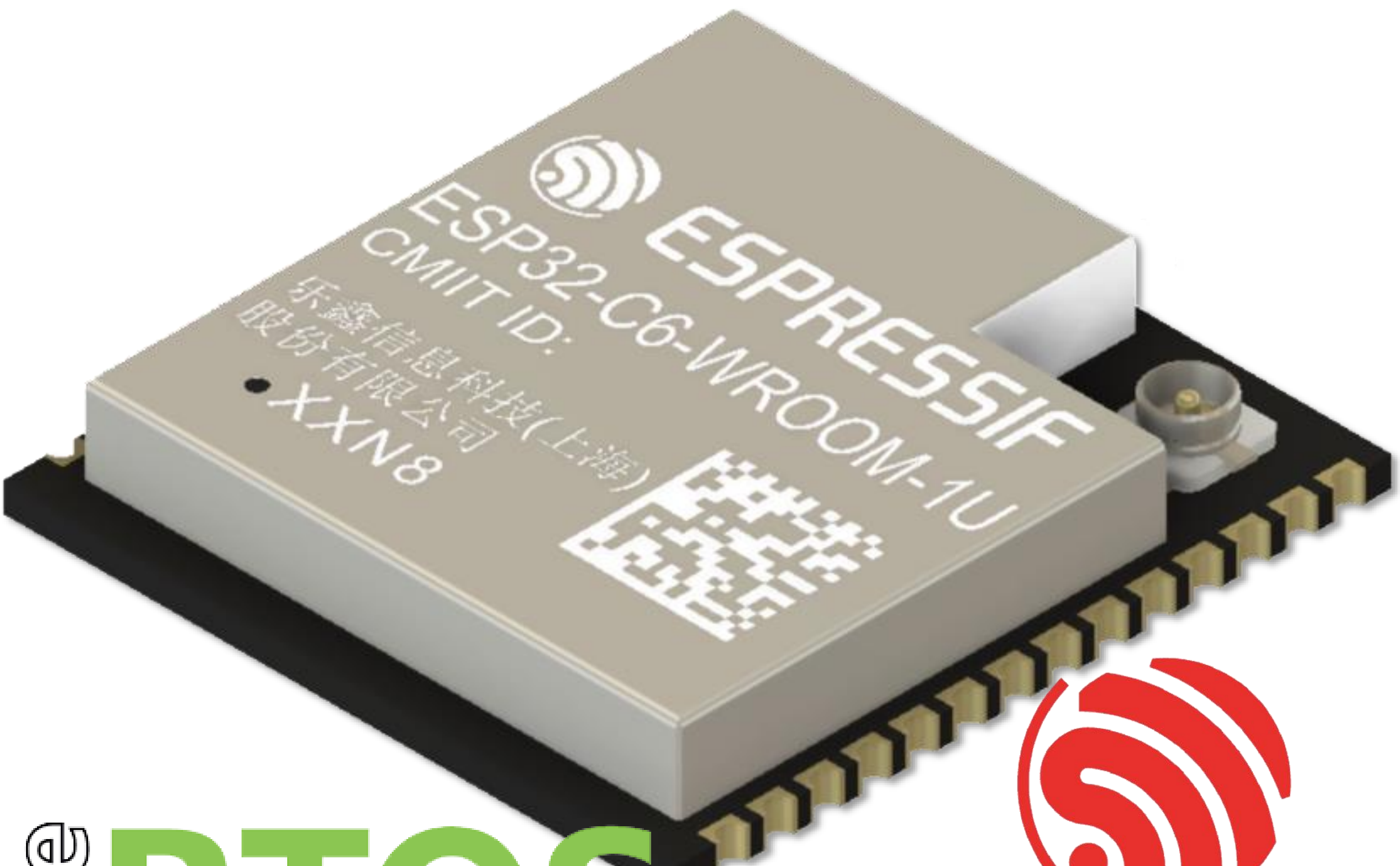
Mastering FreeRTOS on ESP32:

A Comprehensive Guide for Embedded IoT Engineers



freeRTOS

ESP-IDF

Table of Contents

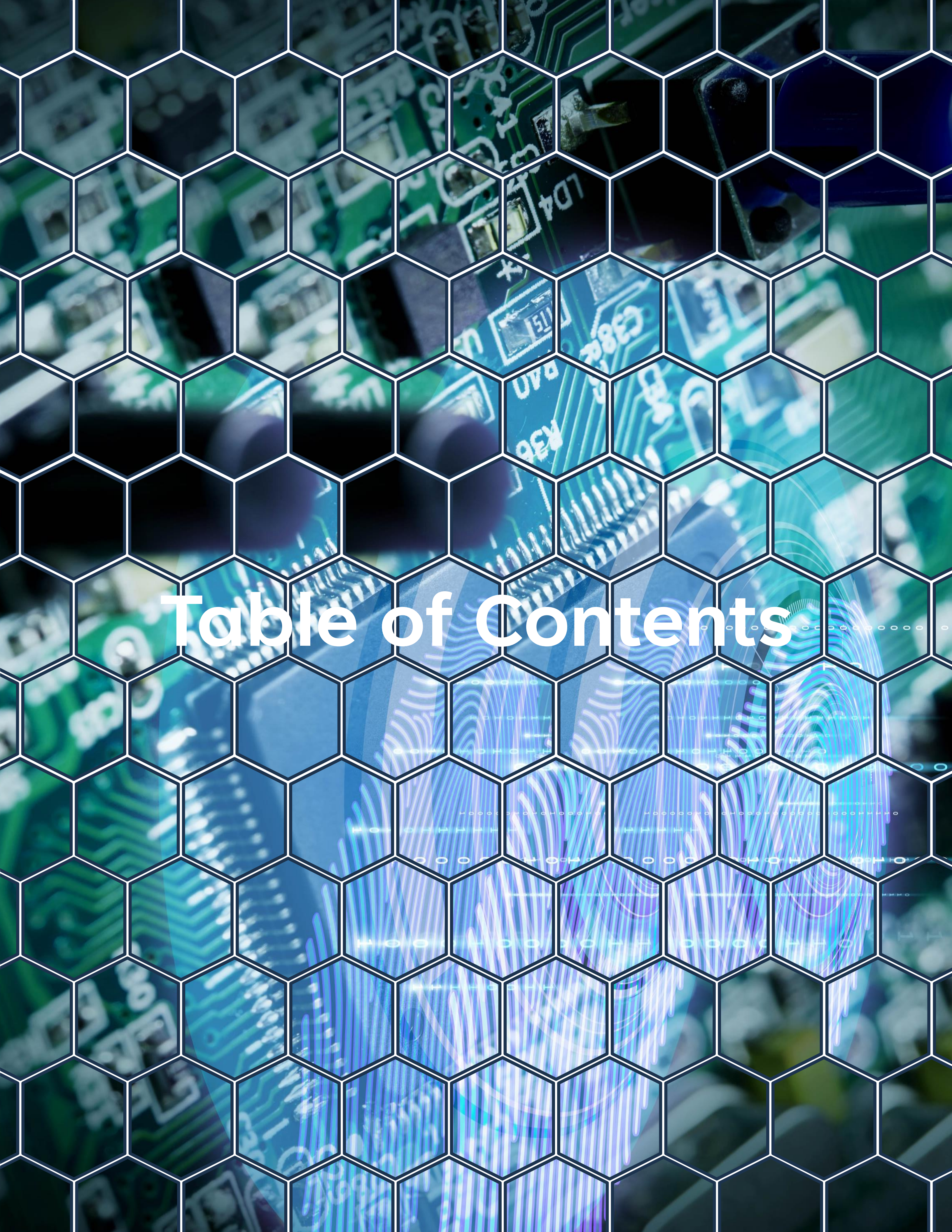
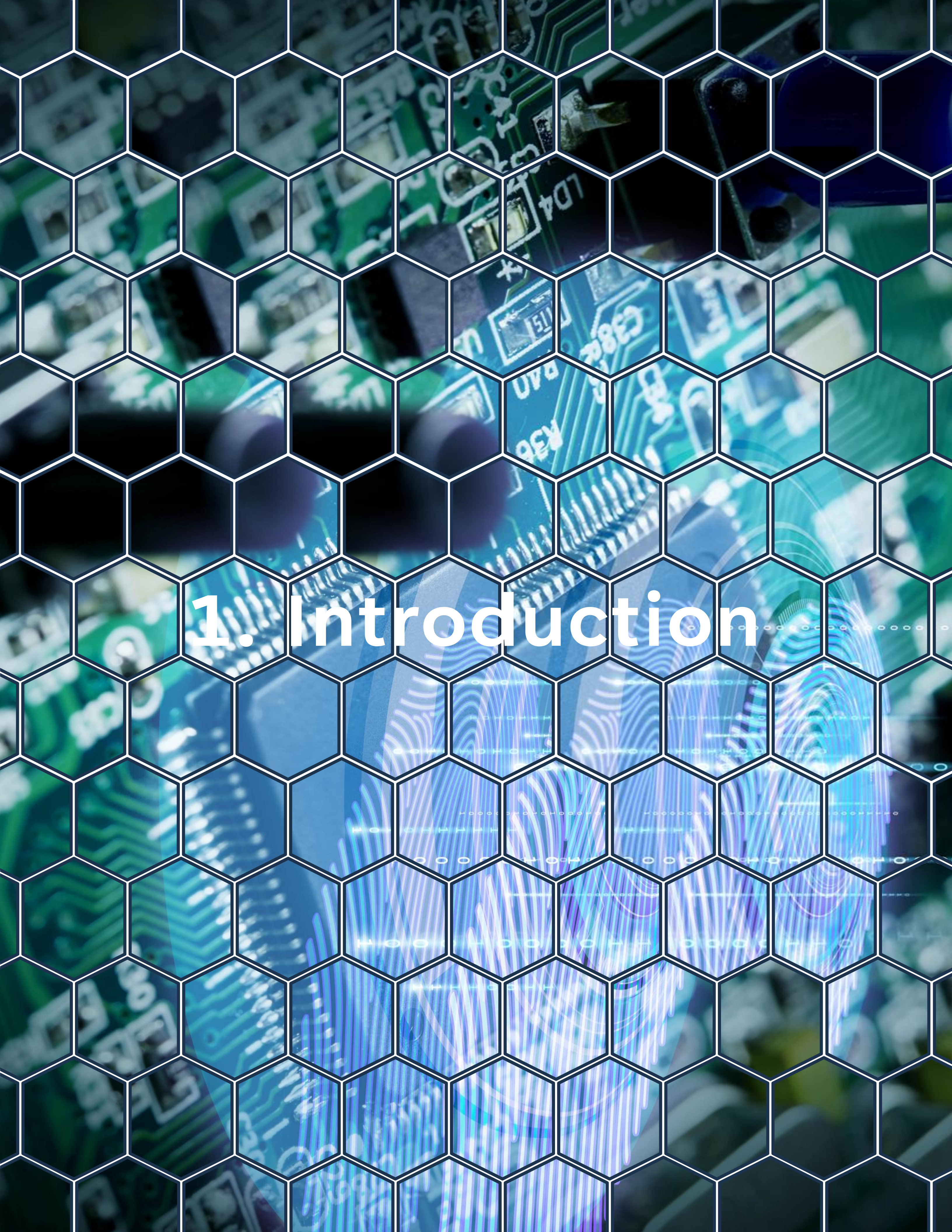


Table of Contents

1. Introduction



1. Introduction

The ESP32 microcontroller has emerged as a cornerstone of modern IoT development due to its powerful dual-core CPU, rich peripheral set, and robust support for real-time operating systems like FreeRTOS.

This guide aims to bridge the gap between FreeRTOS theory and practical ESP32 development, enabling embedded engineers to build scalable, responsive, and maintainable real-time applications.

1. Introduction

Whether you're building smart home devices, wearable tech, or industrial monitoring systems, mastering FreeRTOS on ESP32 is essential for creating deterministic behavior and efficient multitasking in your firmware.

2. Getting Started with FreeRTOS on ESP32

2. Getting Started with FreeRTOS ESP32

Concept and Use CaseFree

RTOS is a real-time operating system kernel for embedded devices that makes it easy to manage multiple tasks efficiently. With the ESP32's support through the ESP-IDF framework, FreeRTOS is used for developing real-time, concurrent, and multitasking applications in IoT.

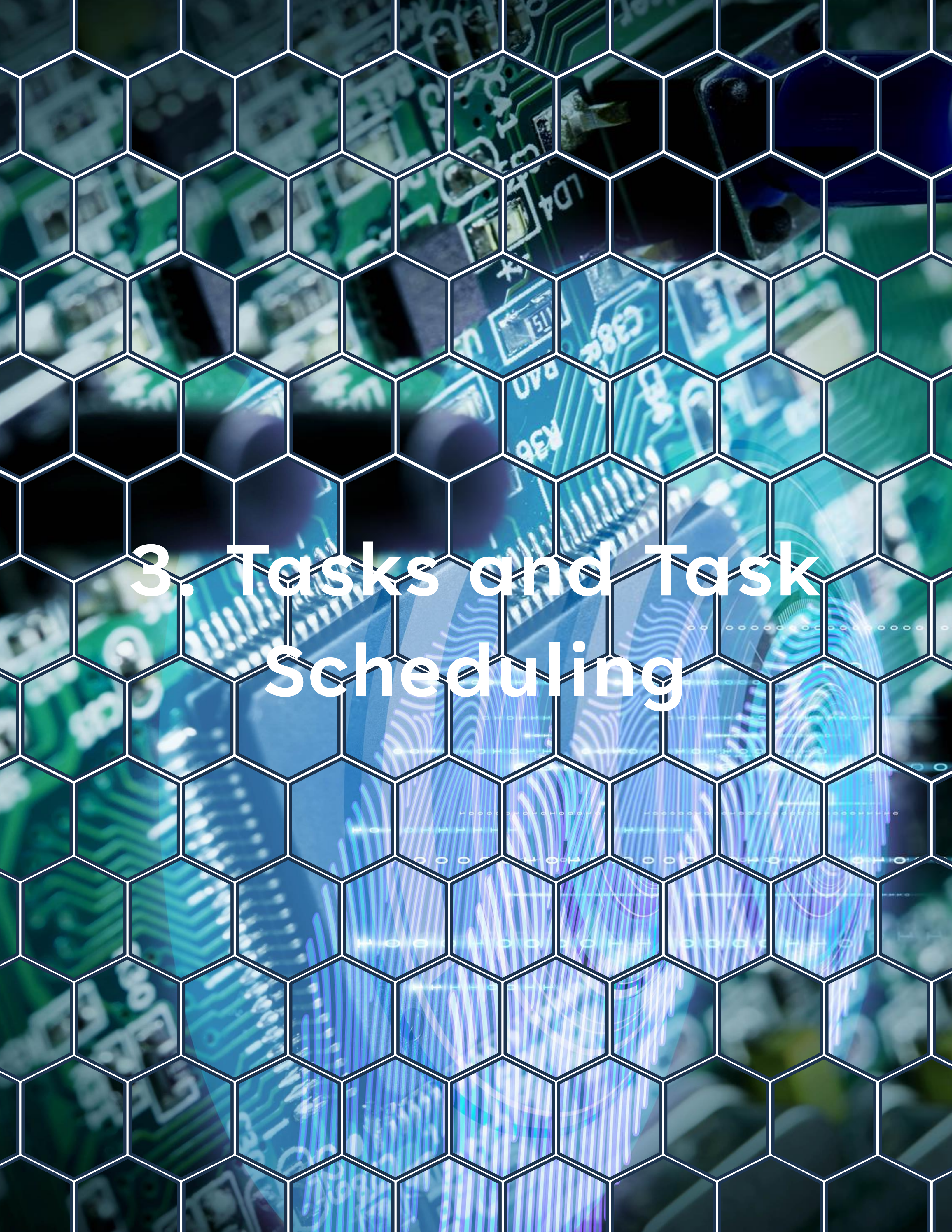
2. Getting Started with FreeRTOS ESP32

Best Practice

Set up your ESP-IDF environment correctly and configure FreeRTOS settings such as tick rate, timer task stack size, and core affinity to match your application requirements.

```
idf.py create-project freertos_demo
cd freertos_demo
idf.py menuconfig
```

Enable FreeRTOS options under Component config -> FreeRTOS. Once configured, you can begin creating tasks, queues, and other kernel objects directly in your application.



3. Tasks and Task Scheduling

3. Tasks and Task Scheduling

Concept and Use Case

A task in FreeRTOS is akin to a thread in traditional operating systems. Tasks enable concurrent operations such as reading sensor data while handling communication. Task scheduling in FreeRTOS is priority-based and preemptive by default.

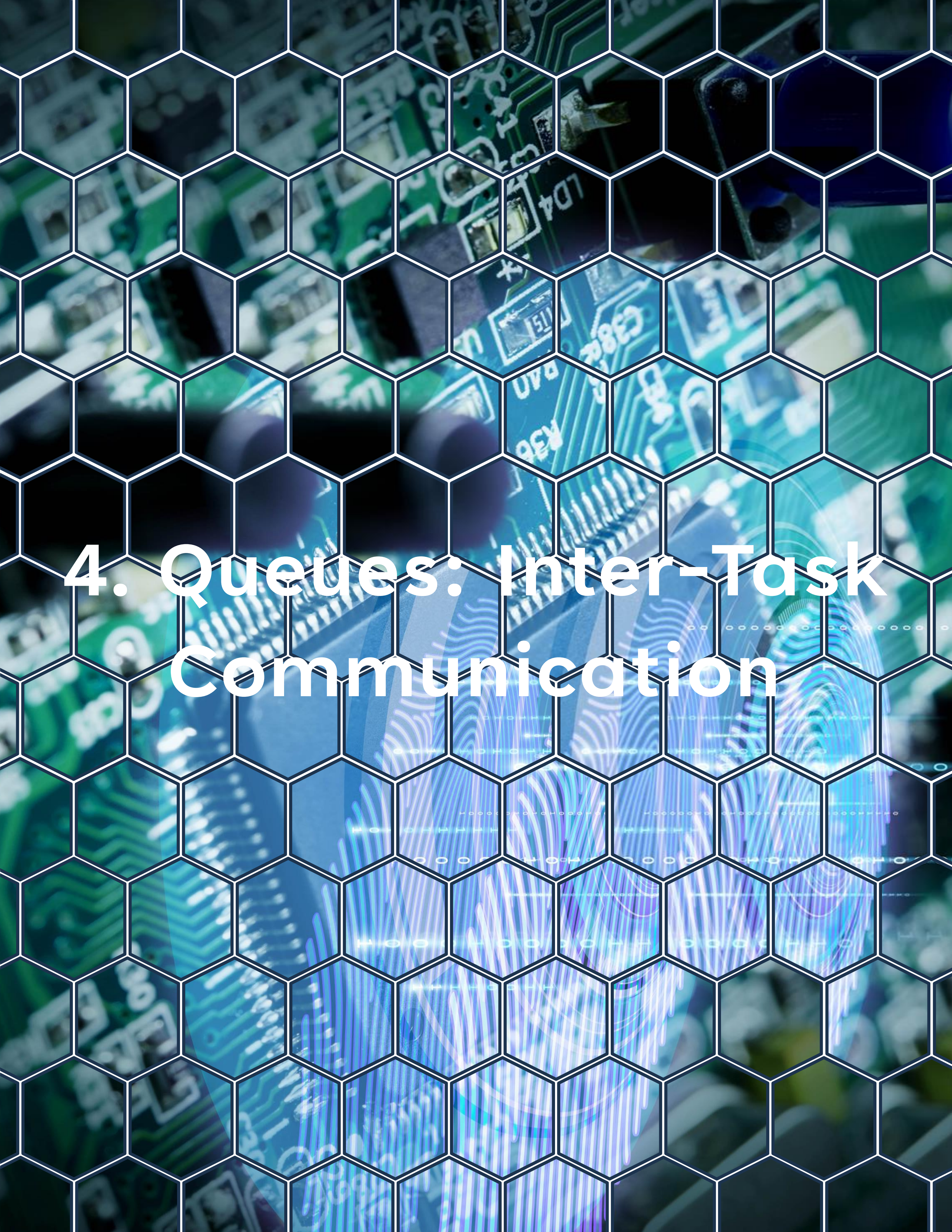
Best Practice

Keep tasks short and non-blocking, assign appropriate priorities, and ensure proper use of delays or synchronization mechanisms to yield CPU time.

3. Tasks and Task Scheduling

Creating a Task Code example

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void vSensorTask(void *pvParameters) {
    while (1) {
        printf("Reading sensor data...\n");
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void app_main(void) {
    xTaskCreate(vSensorTask, "SensorTask", 2048, NULL, 5, NULL);
}
```



4. Queues: Inter-Task Communication

4. Queues: Inter-Task Communication

Concept and Use Case

Queues allow safe data sharing between tasks. Each queue is a thread-safe FIFO buffer, enabling one task to send data and another to receive it.

Best Practice

Use queues to decouple tasks, and avoid large queue item sizes to minimize memory usage. Always check return values for success/failure.

4. Queues: Inter-Task Communication

Using a Queue Code example

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"

typedef struct {
    int temperature;
    int humidity;
} SensorData;

QueueHandle_t sensorQueue;

void vProducerTask(void *pvParameters) {
    SensorData data = {25, 60};
    while (1) {
        xQueueSend(sensorQueue, &data, portMAX_DELAY);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void vConsumerTask(void *pvParameters) {
    SensorData received;
    while (1) {
        if (xQueueReceive(sensorQueue, &received, portMAX_DELAY)) {
            printf("Temp: %d, Humidity: %d\n", received.temperature,
                received.humidity);
        }
    }
```

4. Queues: Inter-Task Communication

```c
typedef struct {
    int temperature;
    int humidity;
} SensorData;

QueueHandle_t sensorQueue;

void vProducerTask(void *pvParameters) {
    SensorData data = {25, 60};
    while (1) {
        xQueueSend(sensorQueue, &data, portMAX_DELAY);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void vConsumerTask(void *pvParameters) {
    SensorData received;
    while (1) {
        if (xQueueReceive(sensorQueue, &received, portMAX_DELAY)) {
            printf("Temp: %d, Humidity: %d\n", received.temperature,
                received.humidity);
        }
    }
}

void app_main(void) {
    sensorQueue = xQueueCreate(10, sizeof(SensorData));
    xTaskCreate(vProducerTask, "Producer", 2048, NULL, 5, NULL);
    xTaskCreate(vConsumerTask, "Consumer", 2048, NULL, 5, NULL);
}
```

5. Semaphores and Mutexes

5. Semaphores and Mutexes

Concept and Use Case

Semaphores are signaling mechanisms for synchronizing tasks or ISRs with tasks. Mutexes are binary semaphores with priority inheritance, used to protect shared resources.

Best Practice

Use binary semaphores for signaling and mutexes for protecting shared resources. Avoid holding a mutex for long durations.

5. Semaphores and Mutexes

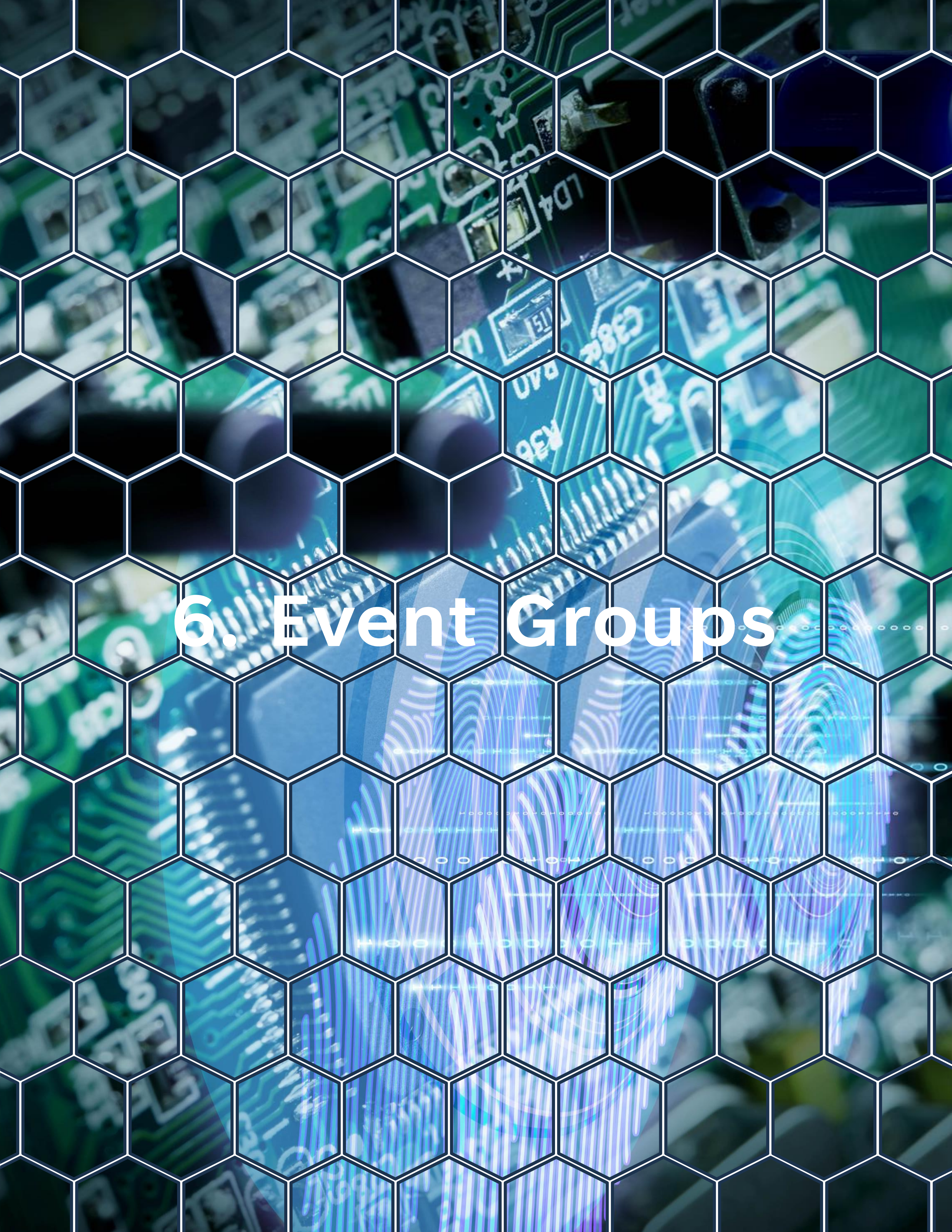
Semaphore Code example

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

SemaphoreHandle_t xSemaphore;

void IRAM_ATTR vISRHandler(void* arg) {
    xSemaphoreGiveFromISR(xSemaphore, NULL);
}

void vTaskHandler(void* arg) {
    while (1) {
        if (xSemaphoreTake(xSemaphore, portMAX_DELAY)) {
            printf("Interrupt received!\n");
        }
    }
}

void app_main(void) {
    xSemaphore = xSemaphoreCreateBinary();
    gpio_install_isr_service(0);
    gpio_set_direction(GPIO_NUM_0, GPIO_MODE_INPUT);
    gpio_set_intr_type(GPIO_NUM_0, GPIO_INTR_POSEDGE);
```

5. Semaphores and Mutexes

```c
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "driver/gpio.h"

SemaphoreHandle_t xSemaphore;

void IRAM_ATTR vISRHandler(void* arg) {
    xSemaphoreGiveFromISR(xSemaphore, NULL);
}

void vTaskHandler(void* arg) {
    while (1) {
        if (xSemaphoreTake(xSemaphore, portMAX_DELAY)) {
            printf("Interrupt received!\n");
        }
    }
}

void app_main(void) {
    xSemaphore = xSemaphoreCreateBinary();
    gpio_install_isr_service(0);
    gpio_set_direction(GPIO_NUM_0, GPIO_MODE_INPUT);
    gpio_set_intr_type(GPIO_NUM_0, GPIO_INTR_POSEDGE);
    gpio_isr_handler_add(GPIO_NUM_0, vISRHandler, NULL);
    xTaskCreate(vTaskHandler, "Task", 2048, NULL, 5, NULL);
}
```

6. Event Groups



6. Event Groups

Concept and Use Case

Event groups manage multiple flags (bits) for synchronizing task execution based on the occurrence of multiple events. They are ideal for coordinating the readiness of several components or subsystems before proceeding.

Best Practice

Use `xEventGroupWaitBits` to block tasks until all required events occur. Use pdTRUE to clear bits upon return when the event has been handled. Combine bits with macros for clarity.

6. Event Groups

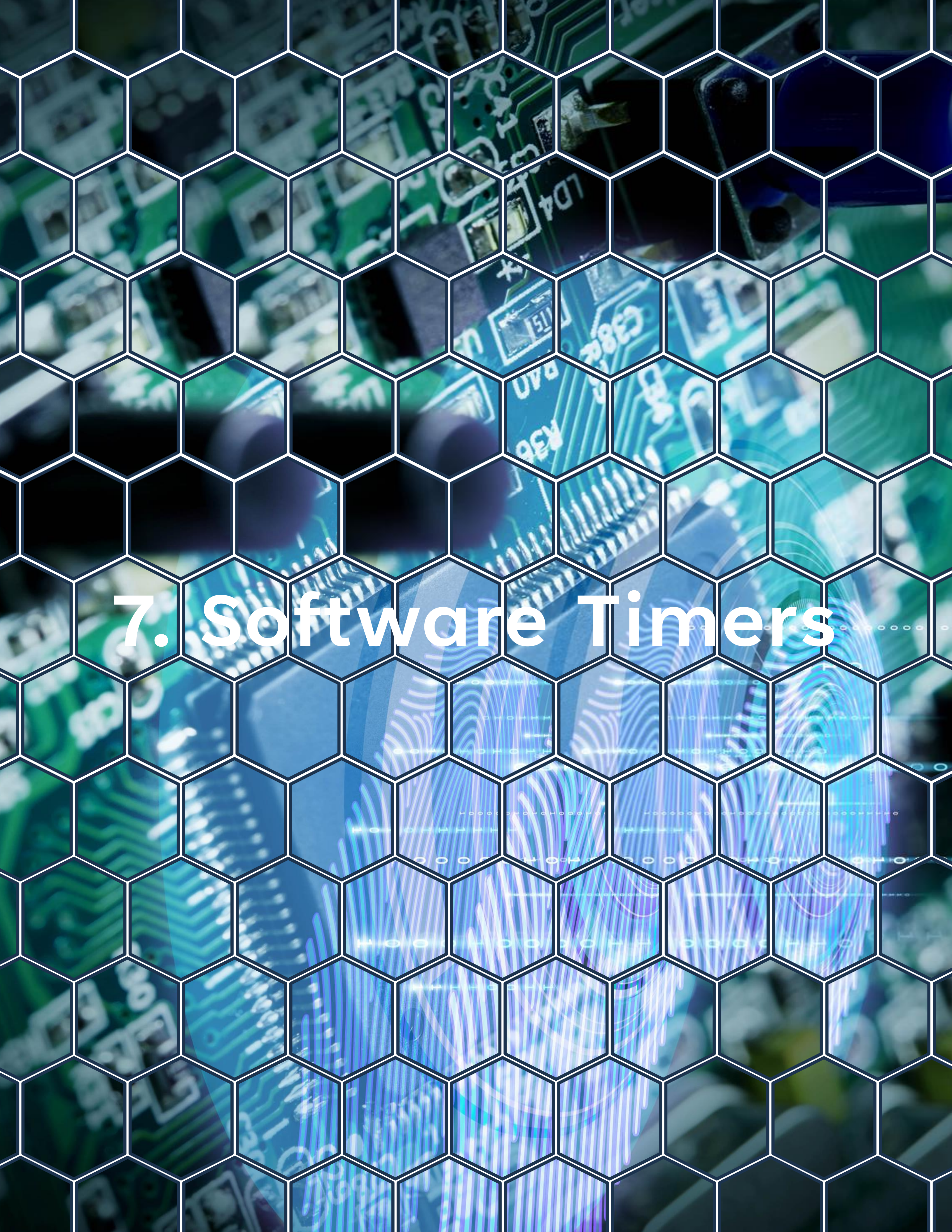
Event Groups Code example

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"

EventGroupHandle_t xEventGroup;
#define BIT_WIFI_CONNECTED BIT0
#define BIT_SENSOR_READY  BIT1

void vTaskA(void* arg) {
    vTaskDelay(pdMS_TO_TICKS(1000));
    printf("WiFi Connected\n");
    xEventGroupSetBits(xEventGroup, BIT_WIFI_CONNECTED);
}

void vTaskB(void* arg) {
    vTaskDelay(pdMS_TO_TICKS(2000));
    printf("Sensor Ready\n");
    xEventGroupSetBits(xEventGroup, BIT_SENSOR_READY);
}

void vTaskC(void* arg) {
    EventBits_t uxBits;
    uxBits = xEventGroupWaitBits(xEventGroup,
                                 BIT_WIFI_CONNECTED | BIT_SENSOR_READY,
                                 pdTRUE, pdTRUE, portMAX_DELAY);
    printf("System is fully ready!\n");
}

void app_main(void) {
```

6. Event Groups

```c

EventGroupHandle_t xEventGroup;
#define BIT_WIFI_CONNECTED BIT0
#define BIT_SENSOR_READY  BIT1

void vTaskA(void* arg) {
    vTaskDelay(pdMS_TO_TICKS(1000));
    printf("WiFi Connected\n");
    xEventGroupSetBits(xEventGroup, BIT_WIFI_CONNECTED);
}

void vTaskB(void* arg) {
    vTaskDelay(pdMS_TO_TICKS(2000));
    printf("Sensor Ready\n");
    xEventGroupSetBits(xEventGroup, BIT_SENSOR_READY);
}

void vTaskC(void* arg) {
    EventBits_t uxBits;
    uxBits = xEventGroupWaitBits(xEventGroup,
                                 BIT_WIFI_CONNECTED | BIT_SENSOR_READY,
                                 pdTRUE, pdTRUE, portMAX_DELAY);
    printf("System is fully ready!\n");
}

void app_main(void) {
    xEventGroup = xEventGroupCreate();
    xTaskCreate(vTaskA, "WiFiTask", 2048, NULL, 5, NULL);
    xTaskCreate(vTaskB, "SensorTask", 2048, NULL, 5, NULL);
    xTaskCreate(vTaskC, "StartupTask", 2048, NULL, 5, NULL);
}
```

7. Software Timers



7. Software Timers

Concept and Use Case

Software timers allow execution of callback functions at specified intervals, without occupying task resources. They are well-suited for periodic or one-shot operations like status checks or timeout handling.

Best Practice

Avoid placing heavy operations in timer callbacks. Use timers when execution can be deferred or doesn't need a dedicated task.

7. Software Timers

Software Timer Code example

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/timers.h"

void vTimerCallback(TimerHandle_t xTimer) {
    printf("Timer triggered!\n");
}

void app_main(void) {
    TimerHandle_t xTimer = xTimerCreate("MyTimer",
                                         pdMS_TO_TICKS(1000),
                                         pdTRUE,
                                         NULL,
                                         vTimerCallback);

    if (xTimer != NULL) {
        xTimerStart(xTimer, 0);
    }
}
```

8. Task Notifications

8. Task Notifications

Concept and Use Case

Task notifications are optimized for lightweight signaling between tasks. They can act as binary semaphores, counting semaphores, or simple event flags.

Best Practice

Use when only a single receiver task is involved. Avoid mixing notification types for clarity and maintainability.

8. Task Notifications

Task Notifications Code example

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

TaskHandle_t xTaskHandle = NULL;

void vNotifierTask(void *arg) {
    vTaskDelay(pdMS_TO_TICKS(2000));
    xTaskNotifyGive(xTaskHandle);
}

void vWaiterTask(void *arg) {
    while (1) {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        printf("Notification received!\n");
    }
}

void app_main(void) {
    xTaskCreate(vWaiterTask, "Waiter", 2048, NULL, 5, &xTaskHandle);
    xTaskCreate(vNotifierTask, "Notifier", 2048, NULL, 5, NULL);
}
```

9. Event Loop Pattern

9. Event Loop Pattern

Concept

The event loop pattern in ESP32, when built using the esp_event library, allows decoupled and modular event handling across multiple system components. Instead of manually creating queues and writing custom dispatchers, developers can use esp_event_loop_create and esp_event_post to register handlers for different event types and IDs, creating a centralized yet extensible architecture.

9. Task Notifications

Typical use cases include

- System-level event management (e.g., Wi-Fi, Bluetooth, sensor states)
- Application state transitions
- Dispatching asynchronous tasks or messages across modules

9. Task Notifications

Best Practice

- Define your **event base** and event **IDs** clearly.
- Use `esp_event_handler_register` to bind specific handlers to your base and ID.
- Always check return values from `esp_event_post` to avoid silent failures.
- Use lightweight handlers or defer heavy processing to a task if needed.

9. Task Notifications

Event Loop Pattern Code example

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_event.h"
#include "esp_log.h"

static const char *TAG = "event_loop";

// Define a custom event base
ESP_EVENT_DEFINE_BASE(APP_EVENTS);

// Event IDs
typedef enum {
    EVENT_BUTTON_PRESS,
    EVENT_SENSOR_UPDATE,
    EVENT_TIMEOUT
} app_event_id_t;

// Handler function
static void app_event_handler(void* handler_arg, esp_event_base_t base,
    int32_t id, void* event_data) {
    switch (id) {
        case EVENT_BUTTON_PRESS:
            ESP_LOGI(TAG, "Handled: Button Pressed");
            break;
        case EVENT_SENSOR_UPDATE:
            ESP_LOGI(TAG, "Handled: Sensor Updated");
            break;
        case EVENT_TIMEOUT:
            ESP_LOGI(TAG, "Handled: Timeout Occurred");
            break;
        default:
            ESP_LOGW(TAG, "Unhandled Event ID: %d", id);
            break;
```

9. Task Notifications

```
// Handler function
static void app_event_handler(void* handler_arg, esp_event_base_t base,
    int32_t id, void* event_data) {
    switch (id) {
        case EVENT_BUTTON_PRESS:
            ESP_LOGI(TAG, "Handled: Button Pressed");
            break;
        case EVENT_SENSOR_UPDATE:
            ESP_LOGI(TAG, "Handled: Sensor Updated");
            break;
        case EVENT_TIMEOUT:
            ESP_LOGI(TAG, "Handled: Timeout Occurred");
            break;
        default:
            ESP_LOGW(TAG, "Unhandled Event ID: %d", id);
            break;
    }
}

// Task that posts button press event
void button_task(void *arg) {
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(3000));
        esp_event_post(APP_EVENTS, EVENT_BUTTON_PRESS, NULL, 0, portMAX_DELAY);
    }
}

// Task that posts sensor update event
void sensor_task(void *arg) {
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(5000));
        esp_event_post(APP_EVENTS, EVENT_SENSOR_UPDATE, NULL, 0, portMAX_DELAY);
    }
}

// Task that periodically posts timeout event
void timeout_task(void *arg) {
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(10000));
```

9. Task Notifications

```c
void sensor_task(void *arg) {
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(5000));
        esp_event_post(APP_EVENTS, EVENT_SENSOR_UPDATE, NULL, 0, portMAX_DELAY);
    }
}

// Task that periodically posts timeout event
void timeout_task(void *arg) {
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(10000));
        esp_event_post(APP_EVENTS, EVENT_TIMEOUT, NULL, 0, portMAX_DELAY);
    }
}

void app_main(void) {
    esp_event_loop_args_t loop_args = {
        .queue_size = 10,
        .task_name = "app_event_loop",
        .task_priority = uxTaskPriorityGet(NULL),
        .task_stack_size = 4096,
        .task_core_id = tskNO_AFFINITY
    };

    esp_event_loop_handle_t app_event_loop;

    ESP_ERROR_CHECK(esp_event_loop_create(&loop_args, &app_event_loop));
    ESP_ERROR_CHECK(esp_event_handler_register_with(app_event_loop,
        APP_EVENTS, ESP_EVENT_ANY_ID, app_event_handler, NULL));

    xTaskCreate(button_task, "button_task", 2048, NULL, 5, NULL);
    xTaskCreate(sensor_task, "sensor_task", 2048, NULL, 5, NULL);
    xTaskCreate(timeout_task, "timeout_task", 2048, NULL, 5, NULL);
}
```

10. Memory Management in FreeRTOS

10. Memory Management in FreeRTOS

Concept and Use Case

FreeRTOS dynamically allocates memory for tasks, queues, semaphores, and other kernel objects. On the ESP32, memory allocation is handled through heap_caps_malloc() or pvPortMalloc() depending on the configuration. It's crucial to monitor heap usage in resource-constrained environments.

10. Memory Management in FreeRTOS

Best Practice

Always check for NULL after memory allocation.

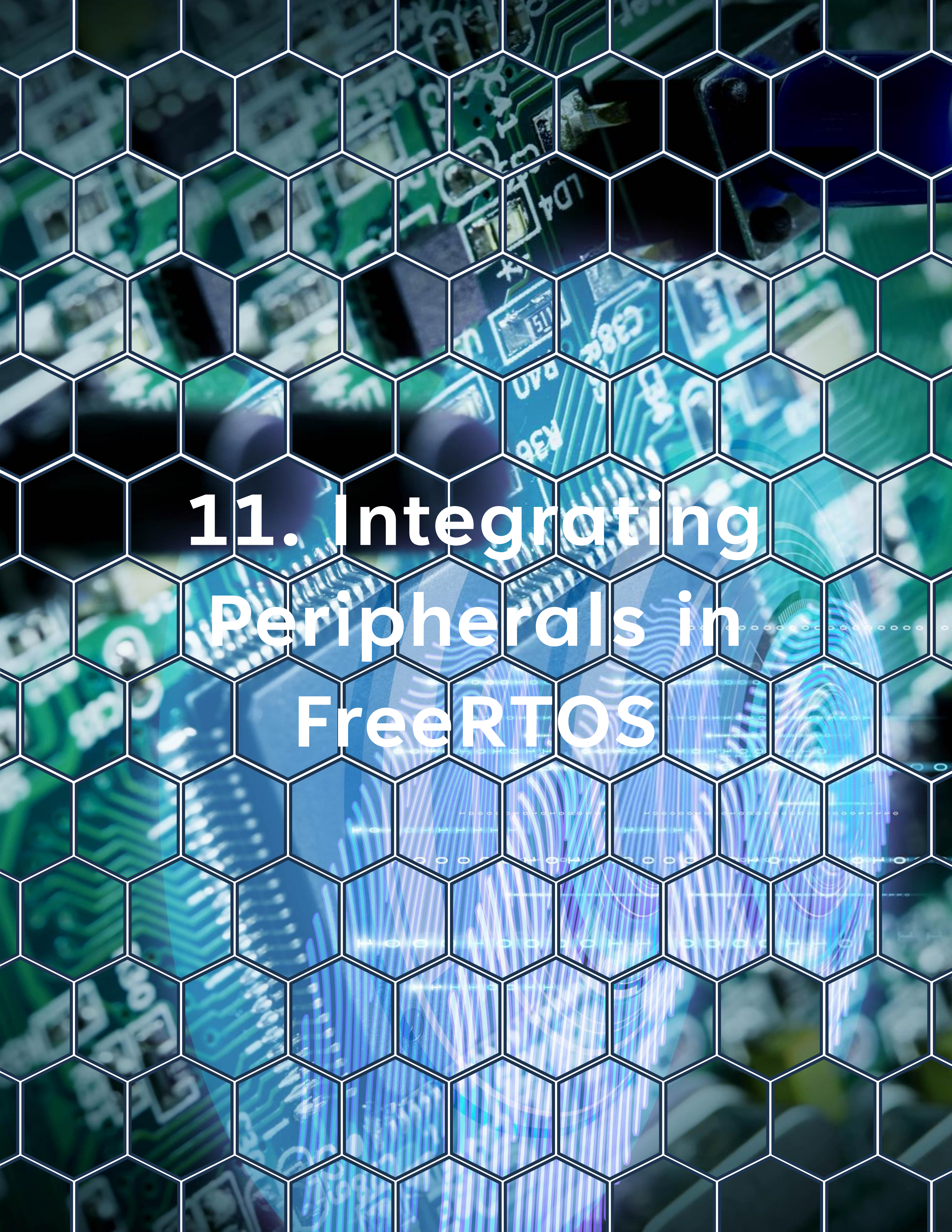
Use static allocation if determinism is critical.

Use ESP-IDF's heap functions to monitor usage and detect leaks.

10. Memory Management in FreeRTOS

Code example

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "esp_heap_caps.h"

void vMemoryTask(void *pvParameters) {
    void *ptr = heap_caps_malloc(1024, MALLOC_CAP_DEFAULT);
    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
    } else {
        printf("Memory allocated at %p\n", ptr);
        heap_caps_free(ptr);
    }
    vTaskDelete(NULL);
}

void app_main(void) {
    xTaskCreate(vMemoryTask, "MemoryTask", 2048, NULL, 5, NULL);
}
```



11. Integrating Peripherals in FreeRTOS

11. Integrating Peripherals in FreeRTOS

Concept and Use Case

Integrating peripherals like UART or I2C into FreeRTOS tasks enables concurrent and asynchronous communication with sensors, modules, or other MCUs. Tasks can independently handle their peripherals without blocking each other.

Best Practice

Initialize peripherals before starting their tasks. Use proper synchronization mechanisms when sharing peripherals. Avoid long blocking delays inside ISR callbacks.

11. Integrating Peripherals in FreeRTOS

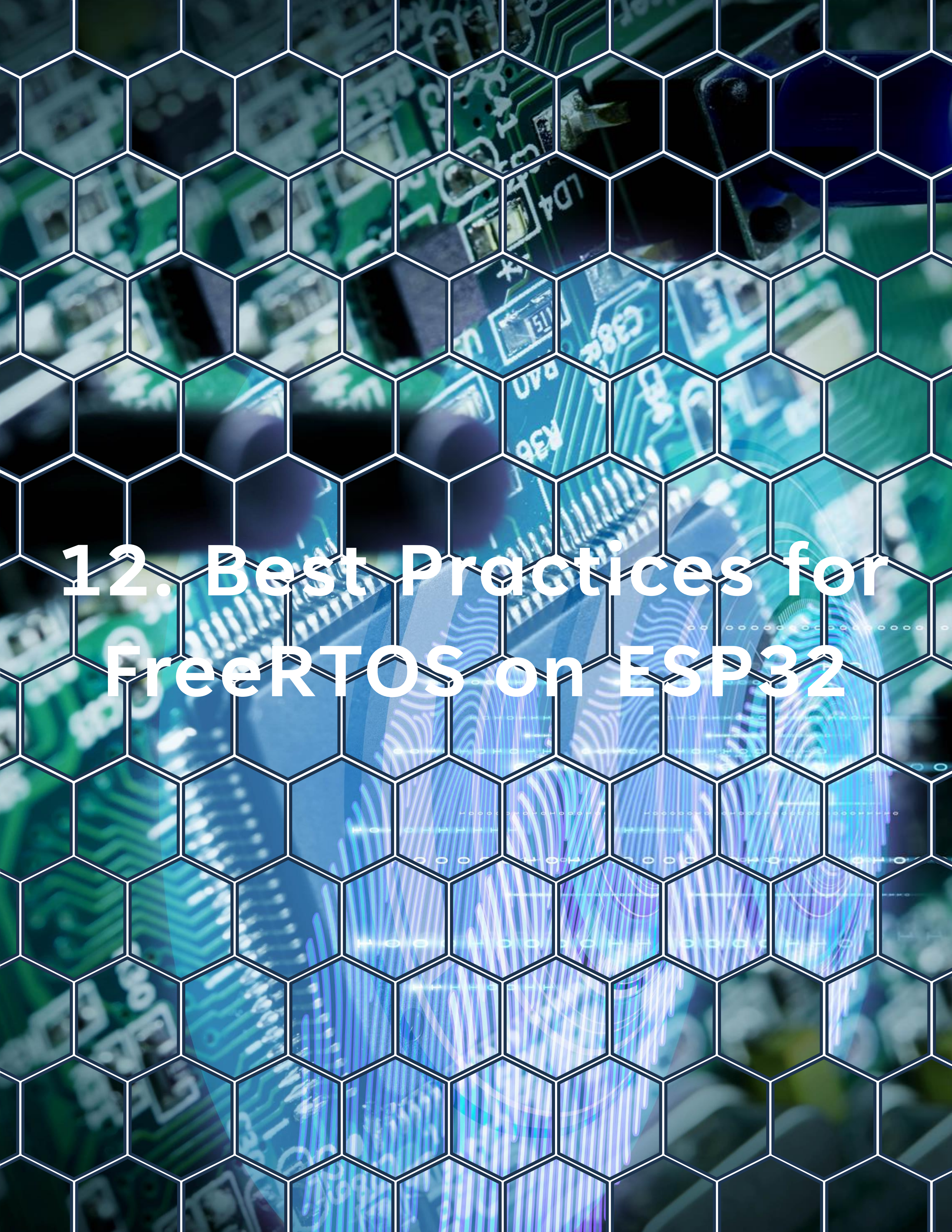
Code example

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

#define BUF_SIZE (1024)

void uart_event_task(void *pvParameters) {
    uint8_t data[BUF_SIZE];
    while (1) {
        int len = uart_read_bytes(UART_NUM_1, data, BUF_SIZE,
            20 / portTICK_PERIOD_MS);
        if (len > 0) {
            data[len] = '\0';
            printf("Received [%d bytes]: %s\n", len, data);
        }
    }
}

void app_main(void) {
    uart_config_t uart_config = {
        .baud_rate = 115200,
        .data_bits = UART_DATA_8_BITS,
        .parity    = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE
    };
    uart_driver_install(UART_NUM_1, BUF_SIZE * 2, 0, 0, NULL, 0);
    uart_param_config(UART_NUM_1, &uart_config);
```

11. Integrating Peripherals in FreeRTOS

```c
#include "freertos/task.h"
#include "driver/uart.h"

#define BUF_SIZE (1024)

void uart_event_task(void *pvParameters) {
    uint8_t data[BUF_SIZE];
    while (1) {
        int len = uart_read_bytes(UART_NUM_1, data, BUF_SIZE,
            20 / portTICK_PERIOD_MS);
        if (len > 0) {
            data[len] = '\0';
            printf("Received [%d bytes]: %s\n", len, data);
        }
    }
}

void app_main(void) {
    uart_config_t uart_config = {
        .baud_rate = 115200,
        .data_bits = UART_DATA_8_BITS,
        .parity    = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE
    };
    uart_driver_install(UART_NUM_1, BUF_SIZE * 2, 0, 0, NULL, 0);
    uart_param_config(UART_NUM_1, &uart_config);
    uart_set_pin(UART_NUM_1, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE,
        UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);

    xTaskCreate(uart_event_task, "uart_event_task", 4096, NULL, 10, NULL);
}
```

12. Best Practices for FreeRTOS on ESP32



12. Best Practices for FreeRTOS on ESP32

Concept and Use Case

Following FreeRTOS best practices on ESP32 ensures system reliability, efficient memory usage, and responsiveness in IoT systems. Proper use of synchronization, task management, and peripheral handling is crucial.

12. Best Practices for FreeRTOS on ESP32

Best Practice

- Use uxTaskGetStackHighWaterMark() to monitor stack usage.
- Prefer static allocation for predictable memory usage.
- Encapsulate peripheral logic inside its own task.
- Minimize ISR execution time; defer work to tasks.
- Use watchdog timers to catch stuck tasks.

12. Best Practices for FreeRTOS on ESP32

Code example

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void monitored_task(void *pvParameters) {
    while (1) {
        printf("Stack watermark: %lu\n",
            uxTaskGetStackHighWaterMark(NULL));
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

void app_main(void) {
    xTaskCreate(monitored_task, "Monitor", 2048, NULL, 5, NULL);
}
```

13. Summary and Additional Resources

13. Summary and Additional Resources

FreeRTOS empowers embedded engineers to build reliable, real-time applications on the ESP32 platform. By leveraging its multitasking capabilities and synchronization primitives, developers can achieve responsive and deterministic designs suited for a wide range of IoT use cases.

13. Summary and Additional Resources

Recommended Resources

ESP-IDF FreeRTOS Documentation:

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html

FreeRTOS.org API Reference:

https://www.freertos.org/a00106.html