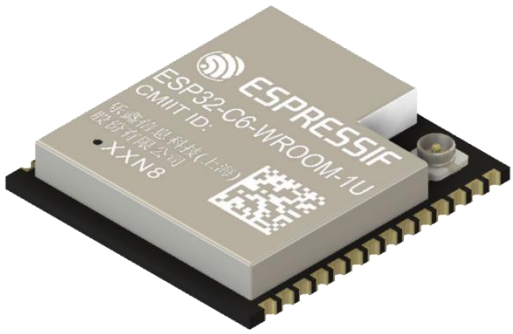
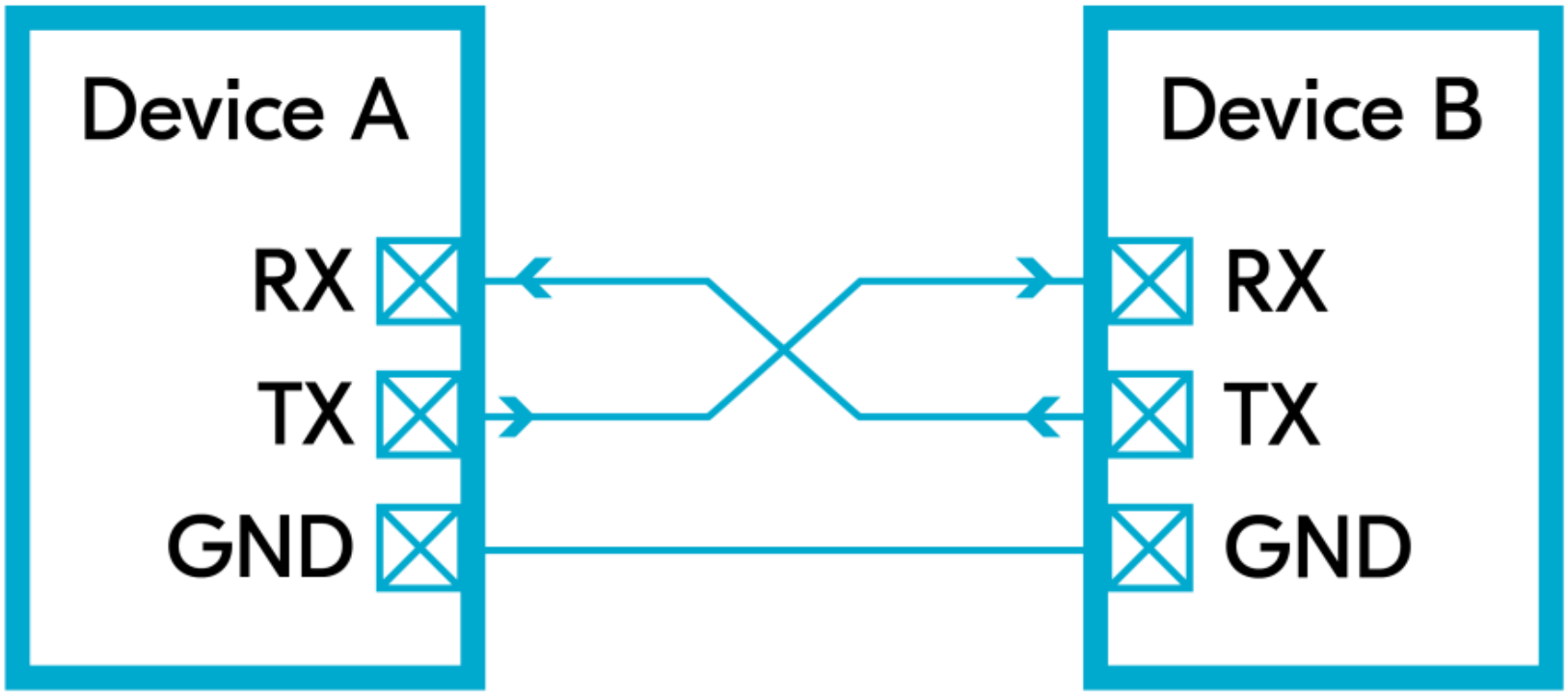
UART on ESP32S3 with ESP-IDF + FreeRTOS: A Practical Guide





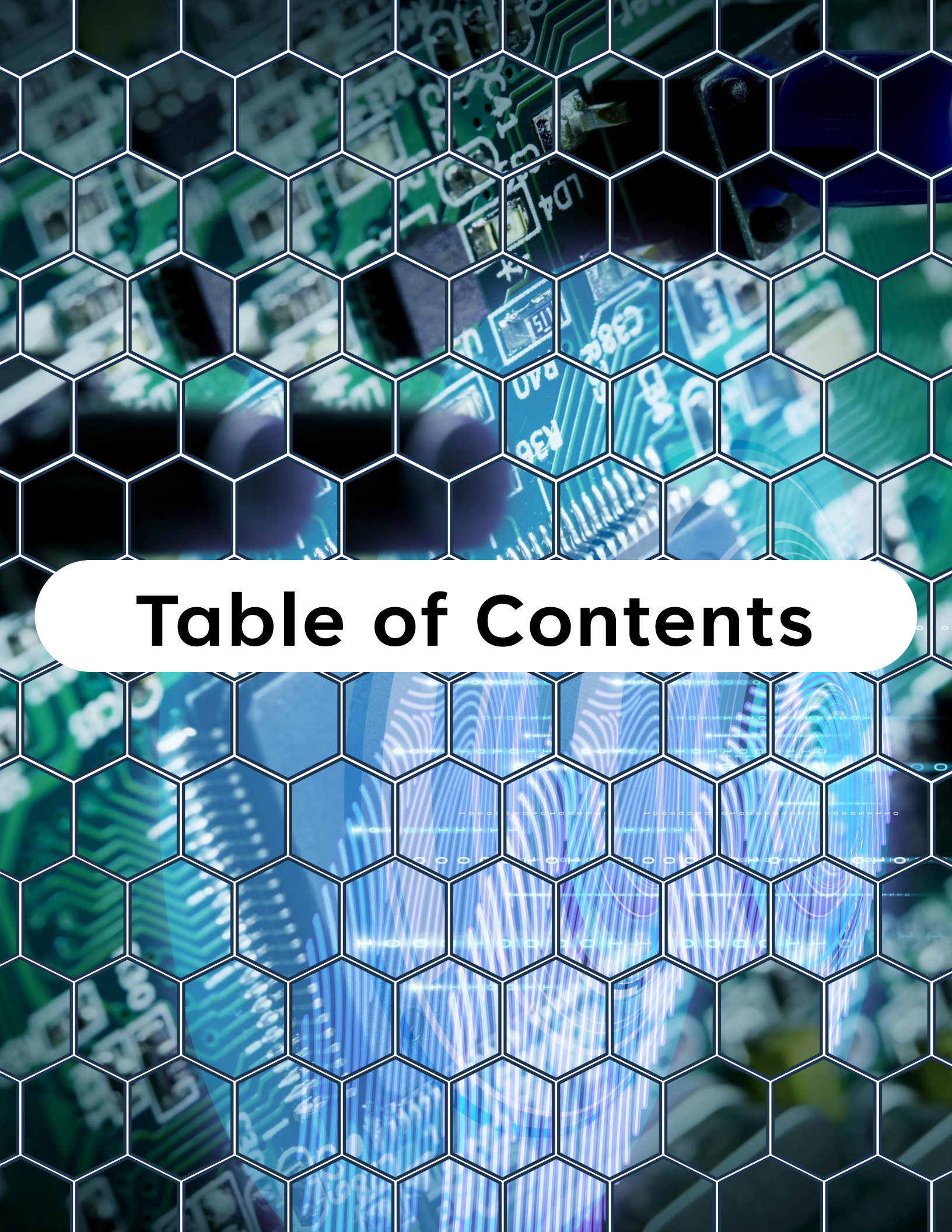
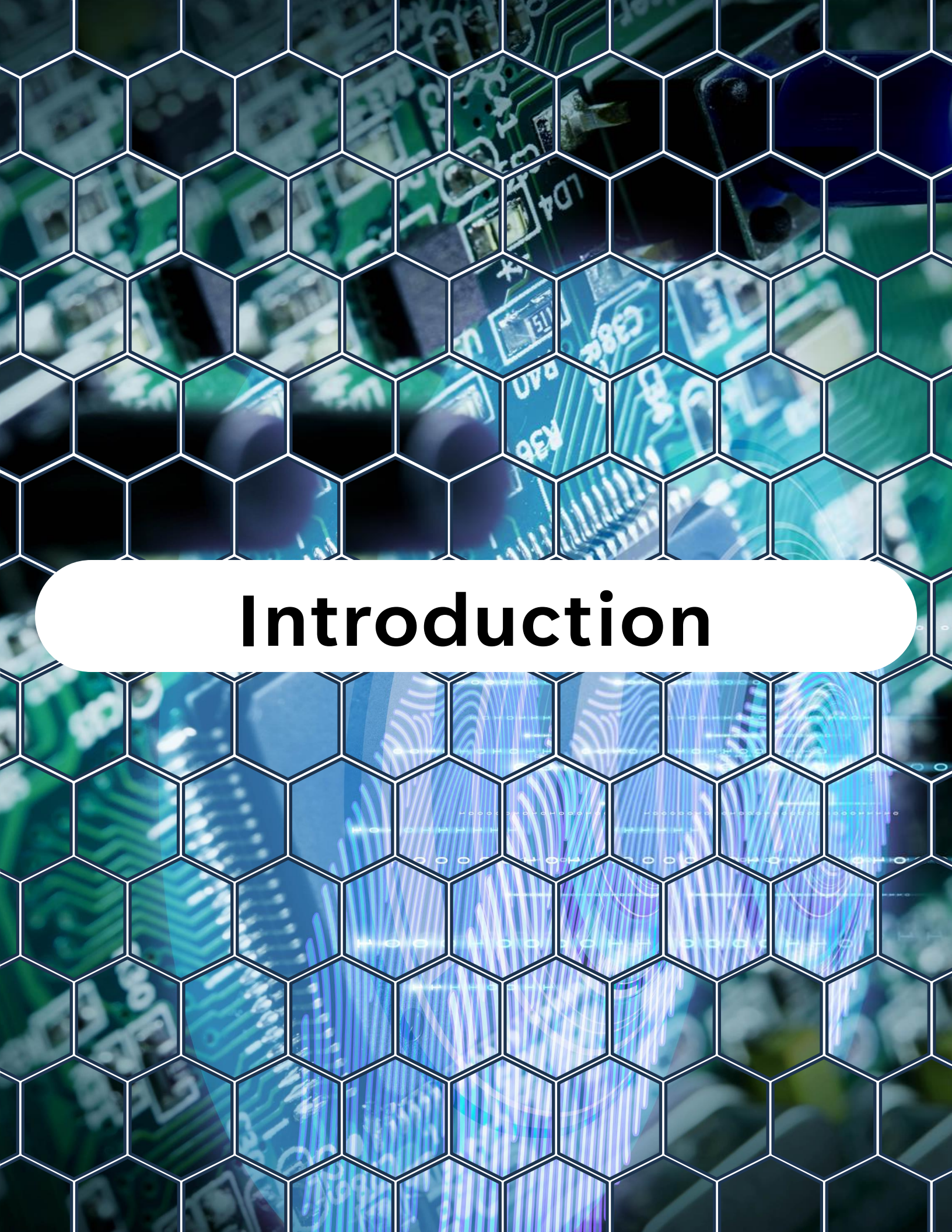


Table of Contents

Table of Contents

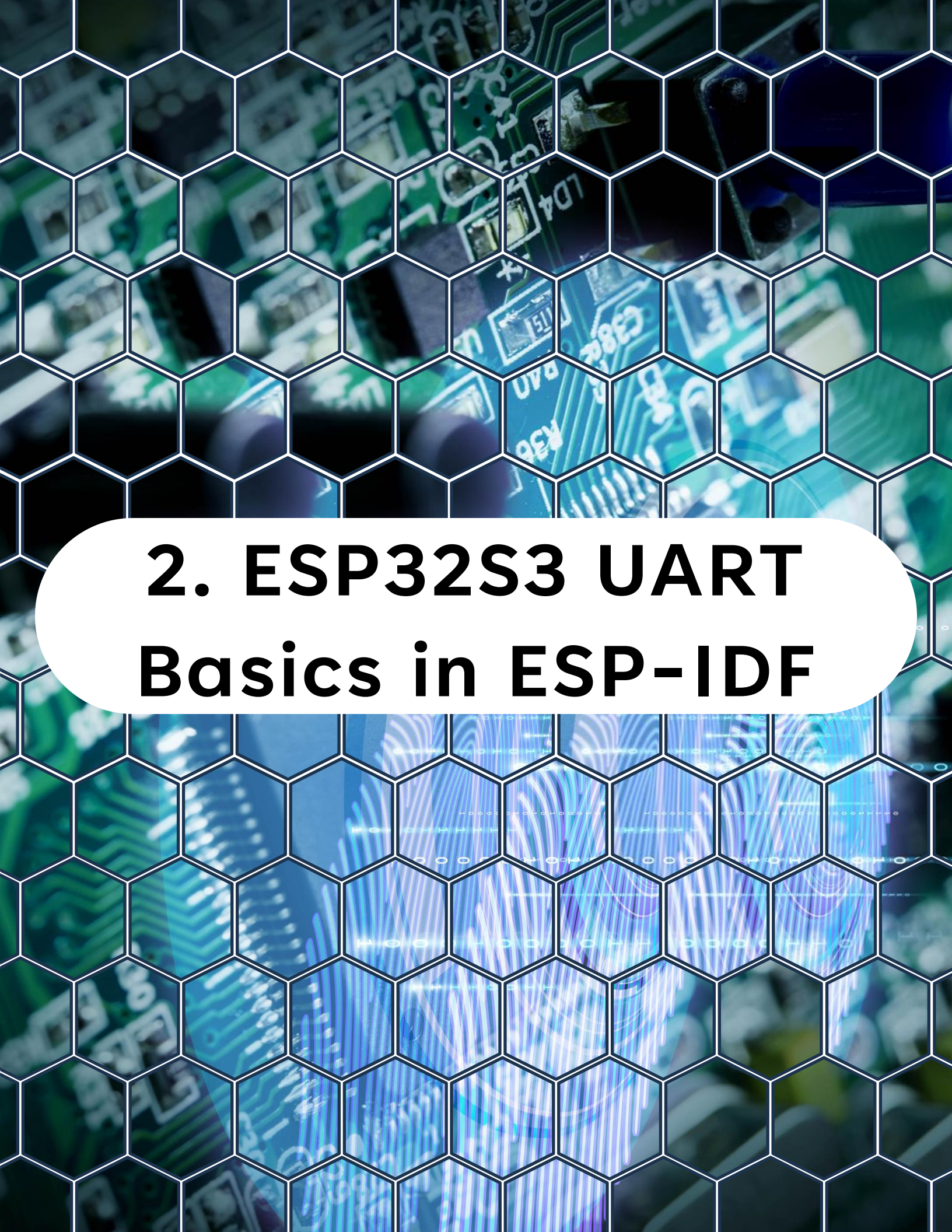


Introduction

Introduction

UART is still one of the most practical ways to communicate with "the outside world" in IoT: cellular modems, GNSS modules, RS-485 transceivers, external MCUs, and even a PC terminal via USB-UART. On the ESP32S3, using UART correctly is less about calling the right driver functions and more about architecture: you must avoid blocking your system, handle bursts, define message boundaries (framing), and recover from errors.

This guide focuses on ESP-IDF's UART driver plus FreeRTOS patterns that scale: event-driven RX, decoupled TX, safe buffering, and clear ownership of data flow. You will see multiple code snippets and a complete end-to-end reference design.



2. ESP32S3 UART Basics in ESP-IDF

2. ESP32S3 UART Basics in ESP-IDF

2.1 UART peripherals and typical usage

ESP32S3 commonly exposes UART0/1/2 (naming depends on IDF version and target). UART0 is frequently used for console/programming on many boards, so for "outside world" devices you often select UART1 or UART2 to avoid conflicts with logs and flashing.

Practical recommendation:

- Use UART0 for logs/console (default) unless you deliberately re-route console.
- Use UART1/2 for external modules.

2. ESP32S3 UART Basics in ESP-IDF

2.2 Pin routing via GPIO matrix

ESP32S3 can route UART signals to many GPIOs.

You configure pins with `uart_set_pin()`.

Notes:

- Always share ground with the external device.
- Confirm voltage levels (ESP32S3 GPIO is 3.3V logic).
- If your module is 1.8V, you need a level shifter.

2. ESP32S3 UART Basics in ESP-IDF

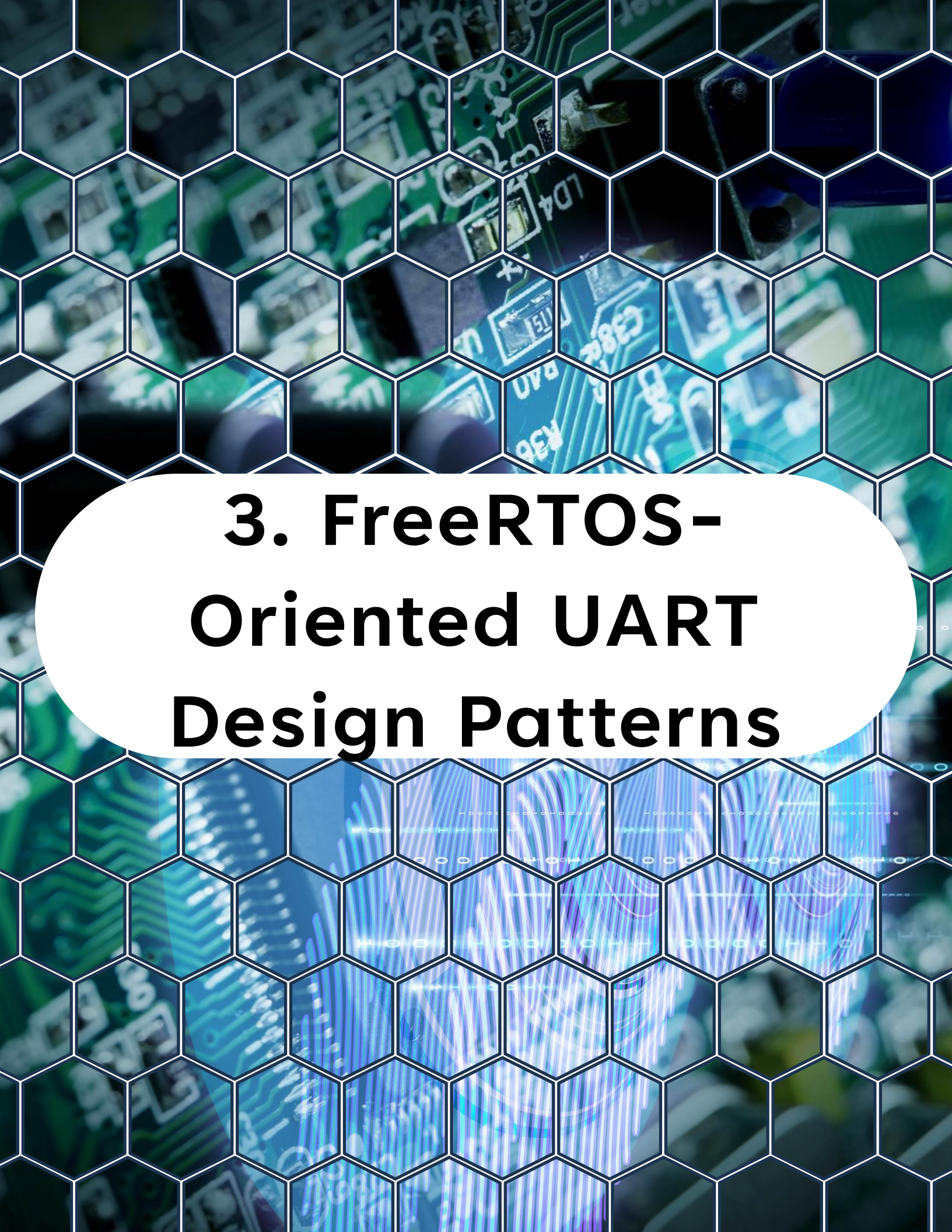
2.3 Serial format and flow control

UART format choices:

- Baud rate: 9600 to >2M depending on device
- Data bits: usually 8
- Parity: none/even/odd (device dependent)
- Stop bits: 1 is typical

Flow control:

- None: simple, but can lose data under bursts
- RTS/CTS: strongly recommended for high throughput, bursty devices, or when CPU is busy



3. FreeRTOS-Oriented UART Design Patterns

3. FreeRTOS-Oriented UART Design Patterns

3.1 The problem with "just read in a loop“

If you do blocking reads/writes inside a high-priority task, you can:

- starve other tasks
- trigger watchdog resets
- lose data due to small buffers and slow parsing
- make the system hard to debug

3. FreeRTOS-Oriented UART Design Patterns

3.2 A scalable pattern: split responsibilities

A robust UART design usually has:

1. **UART RX task:** reads bytes from UART driver and pushes them to a parser buffer or stream buffer
2. **Parser task (or logic task):** turns bytes into messages/commands/events
3. **UART TX task:** pulls outgoing frames from a queue and writes them to UART

Benefits:

- RX stays fast and deterministic
- parsing can be slower without overflowing UART FIFO
- TX is serialized and can be rate-limited
- application logic stays clean

3. FreeRTOS-Oriented UART Design Patterns

3.3 Data decoupling primitives

Use these FreeRTOS primitives:

- **Queue:** for message objects or pointers to allocated buffers
- **StreamBuffer** or **RingBuffer:** for byte streams and burst absorption
- **EventGroups:** for signaling "connected", "ready", "error", etc.

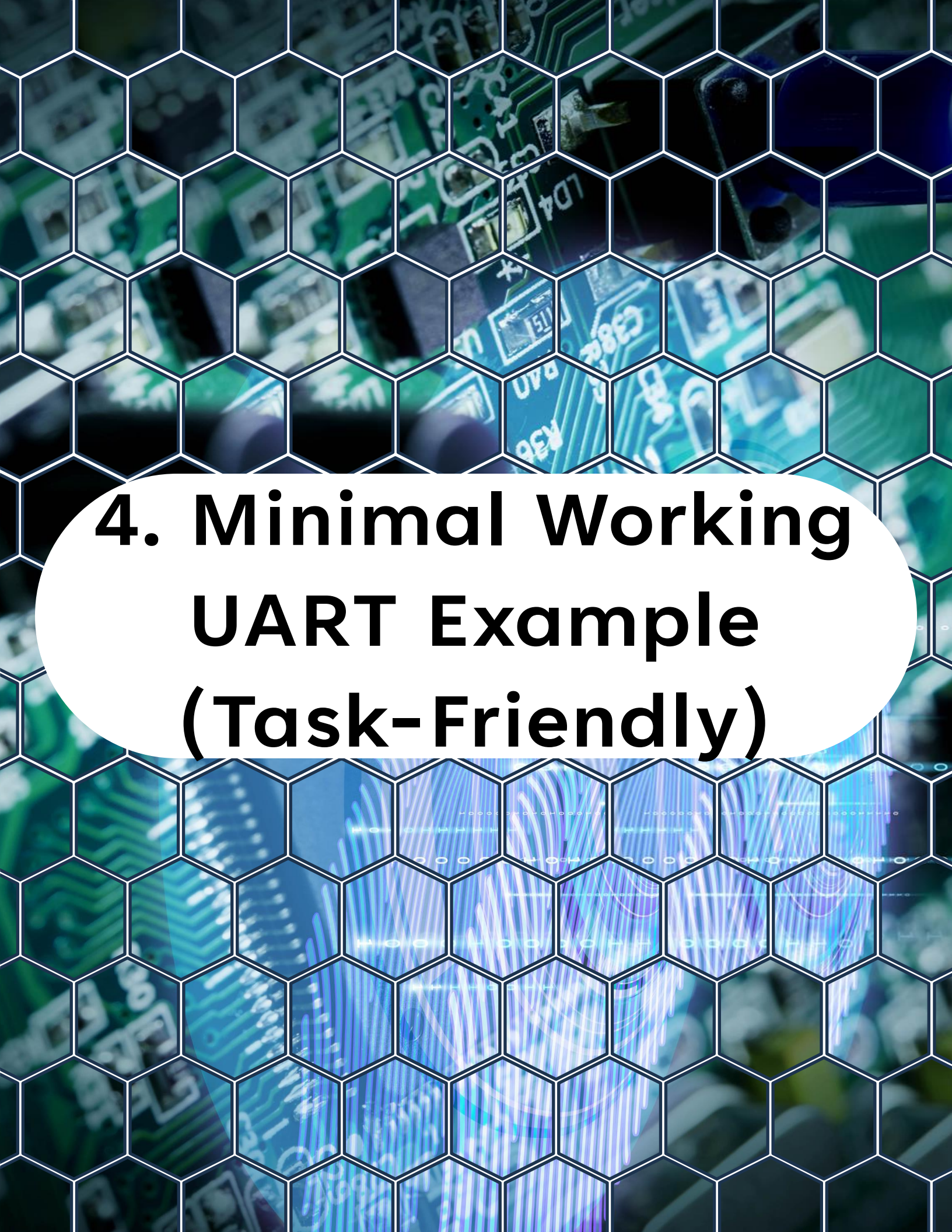
Rule of thumb:

- If you need to transport bytes, consider `StreamBuffer`.
- If you need to transport discrete messages, use `Queue`.

3. FreeRTOS-Oriented UART Design Patterns

3.4 Priorities and timeouts

- RX task: medium-high priority (not the highest in the system)
- Parser task: medium priority
- TX task: medium priority
- Always use timeouts (ticks), not infinite blocks everywhere



4. Minimal Working UART Example (Task-Friendly)

4. Minimal Working UART Example (Task-Friendly)

This is the baseline: configure UART, install driver, create a task that reads with timeouts, and provide a simple write function. It is not yet "production-grade", but it is safe for beginners.

4.1 Minimal init

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

// UART configuration parameters
#define UART_PORT          UART_NUM_1
#define UART_TX_PIN        GPIO_NUM_17
#define UART_RX_PIN        GPIO_NUM_18
#define UART_RTS_PIN       UART_PIN_NO_CHANGE
#define UART_CTS_PIN       UART_PIN_NO_CHANGE

#define UART_BAUD_RATE     115200
#define UART_RX_BUF_SIZE   2048
#define UART_TX_BUF_SIZE   2048

static const char *TAG = "uart_min";

// Initialize UART with minimal configuration
static void uart_min_init(void) {
```

4. Minimal Working UART Example (Task-Friendly)

```c
// Initialize UART with minimal configuration
static void uart_min_init(void) {
    uart_config_t cfg = {
        .baud_rate = UART_BAUD_RATE,
        .data_bits = UART_DATA_8_BITS,
        .parity    = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_DEFAULT,
    };

    // Install UART driver
    ESP_ERROR_CHECK(uart_driver_install(UART_PORT, UART_RX_BUF_SIZE,
                                        UART_TX_BUF_SIZE, 0, NULL, 0));

    // Set UART parameters
    ESP_ERROR_CHECK(uart_param_config(UART_PORT, &cfg));

    // Set UART pins
    ESP_ERROR_CHECK(uart_set_pin(UART_PORT, UART_TX_PIN, UART_RX_PIN,
                                 UART_RTS_PIN, UART_CTS_PIN));

    // Optional: reduce log noise if UART shares pins with console.
    ESP_LOGI(TAG, "UART initialized on port %d", UART_PORT);
}
```

4. Minimal Working UART Example (Task-Friendly)

4.2 Minimal RX task (timeout-based)

```c
static void uart_rx_task(void *arg) {

    uint8_t buf[256];

    while (1) {
        // Read up to sizeof(buf) bytes, wait up to 20 ms.
        int n = uart_read_bytes(UART_PORT, buf, sizeof(buf),
                                pdMS_TO_TICKS(20));
        if (n > 0) {
            // For demonstration: print as a hex dump or
            // forward to parser.
            ESP_LOGI(TAG, "RX %d bytes", n);
            // In real systems, do NOT do heavy logging in
            // RX loops at high rates.
        }

        // Always yield to avoid tight-looping if nothing arrives.
        vTaskDelay(pdMS_TO_TICKS(10));
    }
}
```

4. Minimal Working UART Example (Task-Friendly)

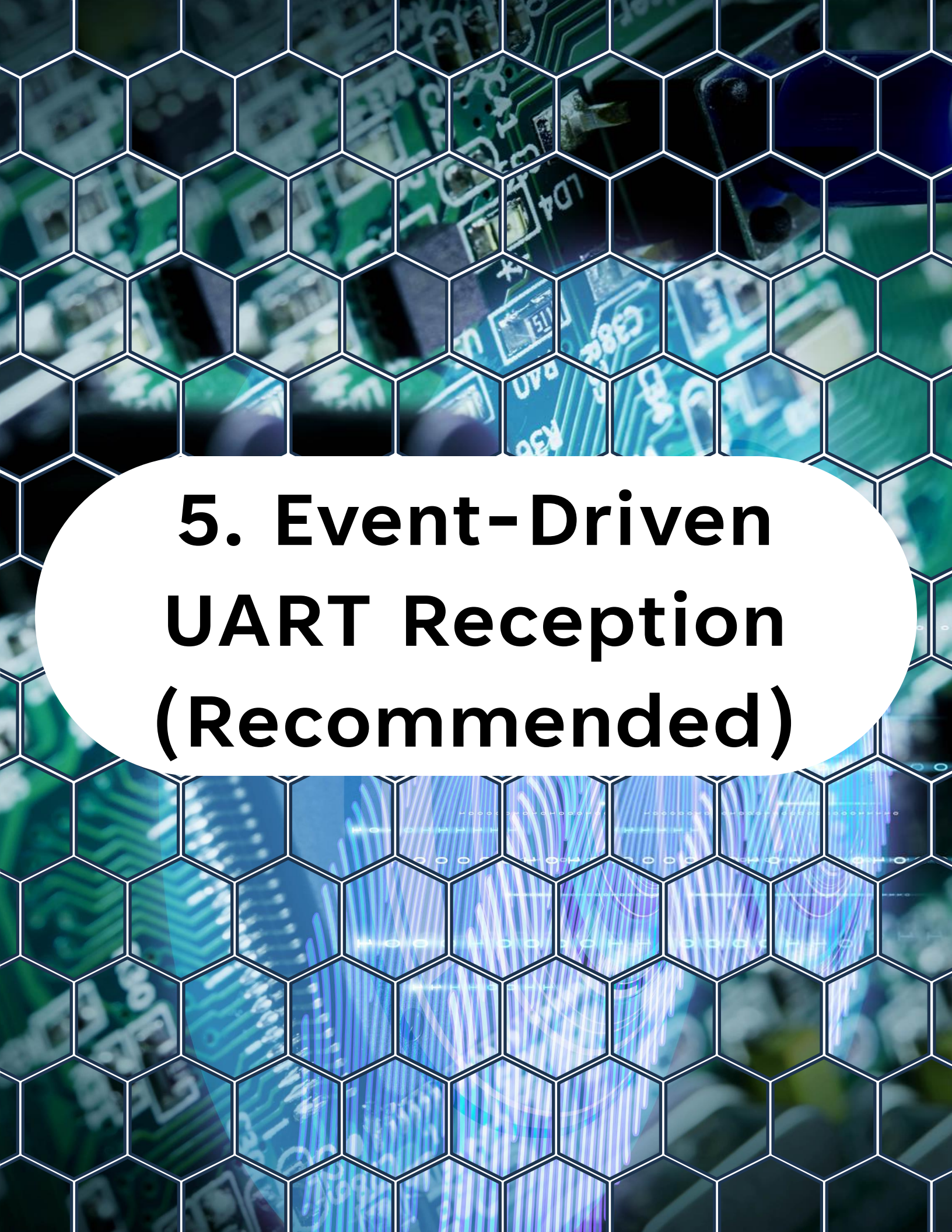
4.3 Minimal TX function

```c
static int uart_send_bytes(const uint8_t *data, size_t len) {
    // uart_write_bytes may block until there is space in the
    // TX buffer.
    // Keep payloads small here; production design uses a
    // TX task + queue.
    return uart_write_bytes(UART_PORT, (const char *)data, len);
}
```

4.4 app_main

```c
void app_main(void) {
    uart_min_init();

    xTaskCreate(uart_rx_task, "uart_rx_task", 4096, NULL, 10, NULL);

    const char *hello = "Hello from ESP32S3 UART\r\n";
    uart_send_bytes((const uint8_t *)hello, strlen(hello));
}
```

This works, but for real IoT modules you should move to the event-driven pattern in the next section.



5. Event-Driven UART Reception (Recommended)

5. Event-Driven UART Reception (Recommended)

ESP-IDF UART driver can post events into a queue. This is usually the best beginner-to-intermediate approach because it is robust and avoids polling.

5.1 Install driver with an event queue

```
#define UART_EVT_QUEUE_LEN 20

// UART event queue handle
static QueueHandle_t uart_evt_queue;

// Initialize UART with event queue
static void uart_evt_init(void) {
    // UART configuration structure
    uart_config_t cfg = {
        .baud_rate = UART_BAUD_RATE,
        .data_bits = UART_DATA_8_BITS,
        .parity    = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_DEFAULT,
    };

    // Install UART driver with event queue
    ESP_ERROR_CHECK(uart_driver_install(UART_PORT,
                                        UART_RX_BUF_SIZE,
                                        UART_TX_BUF_SIZE,
```

5. Event-Driven UART Reception (Recommended)

```c
#define UART_EVT_QUEUE_LEN 20

// UART event queue handle
static QueueHandle_t uart_evt_queue;

// Initialize UART with event queue
static void uart_evt_init(void) {
    // UART configuration structure
    uart_config_t cfg = {
        .baud_rate = UART_BAUD_RATE,
        .data_bits = UART_DATA_8_BITS,
        .parity    = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_DEFAULT,
    };

    // Install UART driver with event queue
    ESP_ERROR_CHECK(uart_driver_install(UART_PORT,
                                        UART_RX_BUF_SIZE,
                                        UART_TX_BUF_SIZE,
                                        UART_EVT_QUEUE_LEN,
                                        &uart_evt_queue,
                                        0));

    // Set UART parameters
    ESP_ERROR_CHECK(uart_param_config(UART_PORT, &cfg));

    // Set UART pins
    ESP_ERROR_CHECK(uart_set_pin(UART_PORT, UART_TX_PIN, UART_RX_PIN,
                                 UART_RTS_PIN, UART_CTS_PIN));
}
```

5. Event-Driven UART Reception (Recommended)

5.2 RX task driven by UART events

Key idea:

- Wait on the event queue.
- When UART_DATA arrives, read exactly what is available.
- Handle overflow events by flushing input and resetting the event queue.

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

static void uart_event_rx_task(void *arg)
{
    uart_event_t evt;
    uint8_t *rx = (uint8_t *)malloc(1024);
    if (!rx) {
        ESP_LOGE(TAG, "malloc failed");
        vTaskDelete(NULL);
        return;
    }

    while (1) {
        // Wait for UART event.
        if (xQueueReceive(uart_evt_queue, &evt, portMAX_DELAY) == pdTRUE) {
            switch (evt.type) {
                case UART_DATA: {
                    int to_read = evt.size;
                    while (to_read > 0) {
                        int chunk = (to_read > 1024) ? 1024 : to_read;
```

5. Event-Driven UART Reception (Recommended)

```
                    while (to_read > 0) {
                        int chunk = (to_read > 1024) ? 1024 : to_read;
                        // Read bytes from UART
                        int n = uart_read_bytes(UART_PORT, rx, chunk, pdMS_TO_TICKS(20));
                        if (n > 0) {
                            // Forward bytes to parser or stream buffer (recommended).
                            // Keep this part fast.
                            // Example: process_rx_bytes(rx, n);
                        }
                        to_read -= n;
                        if (n == 0) break;
                    }
                    break;
                }

                case UART_FIFO_OVF:
                    ESP_LOGW(TAG, "UART FIFO overflow");
                    uart_flush_input(UART_PORT);
                    xQueueReset(uart_evt_queue);
                    break;

                case UART_BUFFER_FULL:
                    ESP_LOGW(TAG, "UART buffer full");
                    uart_flush_input(UART_PORT);
                    xQueueReset(uart_evt_queue);
                    break;

                case UART_PARITY_ERR:
                    ESP_LOGW(TAG, "UART parity error");
                    break;

                case UART_FRAME_ERR:
                    ESP_LOGW(TAG, "UART frame error");
                    break;

                default:
                    // Other events exist (break, pattern detection, etc.)
                    break;
                }
            }
        }

        // Cleanup
        free(rx);
        vTaskDelete(NULL);
}
```

5. Event-Driven UART Reception (Recommended)

5.3 Why overflow handling matters

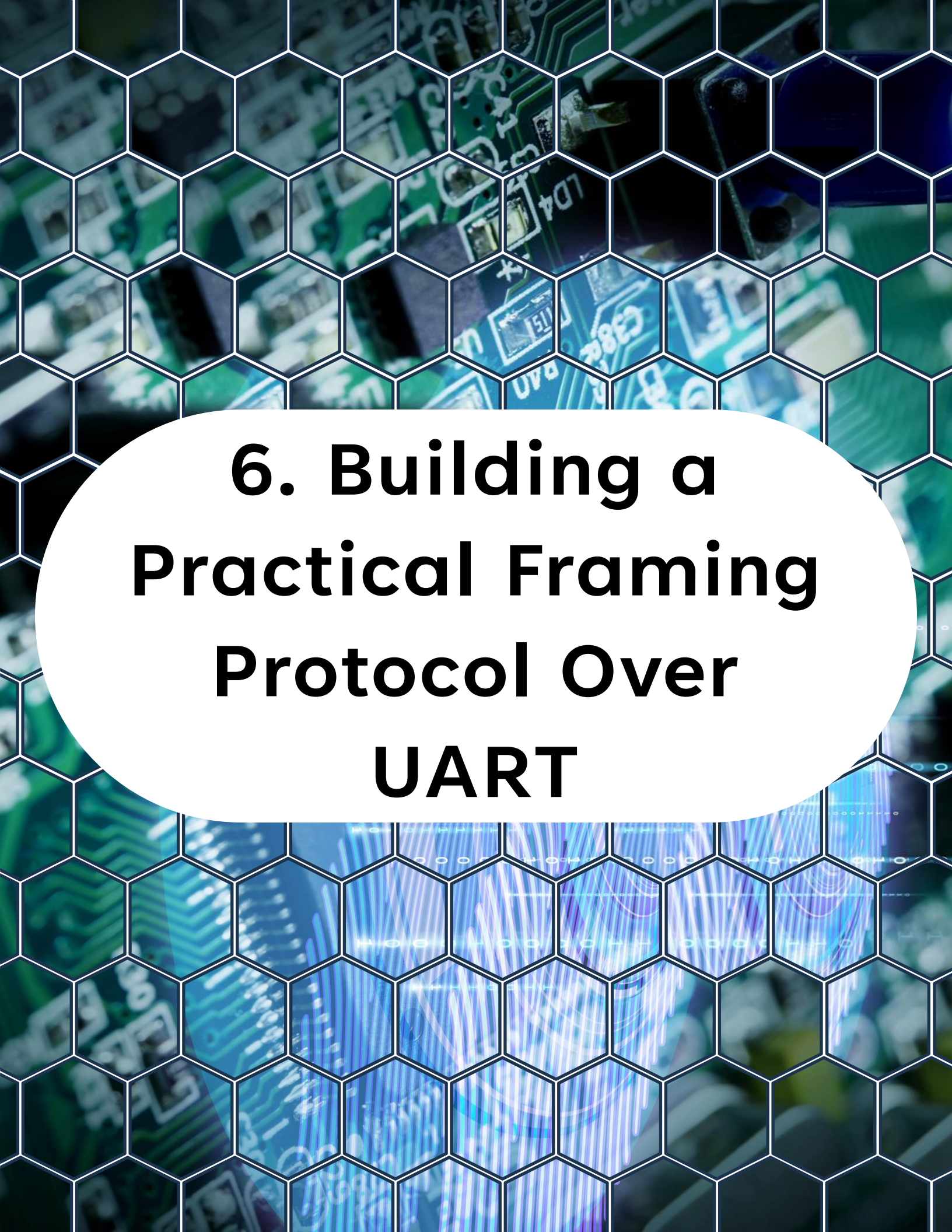
When your external module bursts data faster than your firmware parses it:

- the UART HW FIFO can overflow
- the driver RX ring buffer can fill up

  If you do not handle it, you will read corrupted streams and your protocol layer will desync.

At minimum:

- log the condition
- flush and reset queue (as shown)
- increment counters for diagnostics



6. Building a Practical Framing Protocol Over UART

6. Building a Practical Framing Protocol Over UART

UART is a byte stream. Your application needs message boundaries.

Two common framing options:

1. Delimiter-based (ASCII): lines ending in "\n“
2. Length-prefixed (binary): [len][payload][crc]

6.1 Delimiter-based line protocol (great for beginners)

Example:

- ESP receives: "LED ON\n“
- ESP replies: "OK\n“

6. Building a Practical Framing Protocol Over UART

A simple line accumulator:

```c
typedef struct {
    char line[256];
    size_t len;
} line_acc_t;

static void line_acc_reset(line_acc_t *a)
{
    a->len = 0;
    a->line[0] = '\0';
}

static int line_acc_push(line_acc_t *a, const uint8_t *data, size_t n)
{
    // Returns 1 when a full line is ready, 0 otherwise.
    for (size_t i = 0; i < n; i++) {
        char c = (char)data[i];

        if (c == '\r') continue; // ignore CR
        if (c == '\n') {
            a->line[a->len] = '\0';
            return 1;
        }

        if (a->len < sizeof(a->line) - 1) {
            a->line[a->len++] = c;
        } else {
            // Line too long -> reset (or handle overflow policy)
            line_acc_reset(a);
        }
    }
    return 0;
}
```

6. Building a Practical Framing Protocol Over UART

Then, in your parser task or RX processing:

```c
static void handle_command_line(const char *line)
{
    if (strcmp(line, "PING") == 0) {
        const char *resp = "PONG\n";
        uart_write_bytes(UART_PORT, resp, strlen(resp));
    } else if (strcmp(line, "VERSION") == 0) {
        const char *resp = "ESP32S3_APP v1\n";
        uart_write_bytes(UART_PORT, resp, strlen(resp));
    } else {
        const char *resp = "ERR\n";
        uart_write_bytes(UART_PORT, resp, strlen(resp));
    }
}
```

6. Building a Practical Framing Protocol Over UART

6.2 Length-prefixed frames (better for binary payloads)

A simple format:

- 2 bytes: payload length (uint16_t, little-endian)
- N bytes: payload
- 2 bytes: CRC16 (optional but recommended)

You can implement a small state machine parser:

```
typedef enum {
    RX_WAIT_LEN1,
    RX_WAIT_LEN2,
    RX_WAIT_PAYLOAD,
    RX_WAIT_CRC1,
    RX_WAIT_CRC2
} rx_state_t;

typedef struct {
    rx_state_t st;
```

6. Building a Practical Framing Protocol Over UART

```
typedef struct {
    rx_state_t st;
    uint16_t len;
    uint16_t got;
    uint8_t  payload[512];
    uint16_t crc;
    uint16_t crc_rx;
} frame_rx_t;

static void frame_rx_init(frame_rx_t *f)
{
    f->st = RX_WAIT_LEN1;
    f->len = 0;
    f->got = 0;
    f->crc = 0;
    f->crc_rx = 0;
}
```

Then feed bytes through it. For beginners, delimiter-based is usually enough unless you need binary data.

7. TX Architecture for Reliable Sending

7. TX Architecture for Reliable Sending

7.1 Why a TX task is worth it

If multiple tasks call `uart_write_bytes()` directly:

outputs can interleave (corrupting protocol)

tasks can block unpredictably

you cannot easily rate-limit or prioritize

A TX task solves this: one owner writes to UART.

7.2 TX queue pattern

Define a message object:

```
typedef struct {
    size_t len;
    uint8_t data[256];
} uart_tx_msg_t;

static QueueHandle_t uart_tx_queue;
```

7. TX Architecture for Reliable Sending

Initialize:

```
uart_tx_queue = xQueueCreate(10, sizeof(uart_tx_msg_t));
```

TX task:

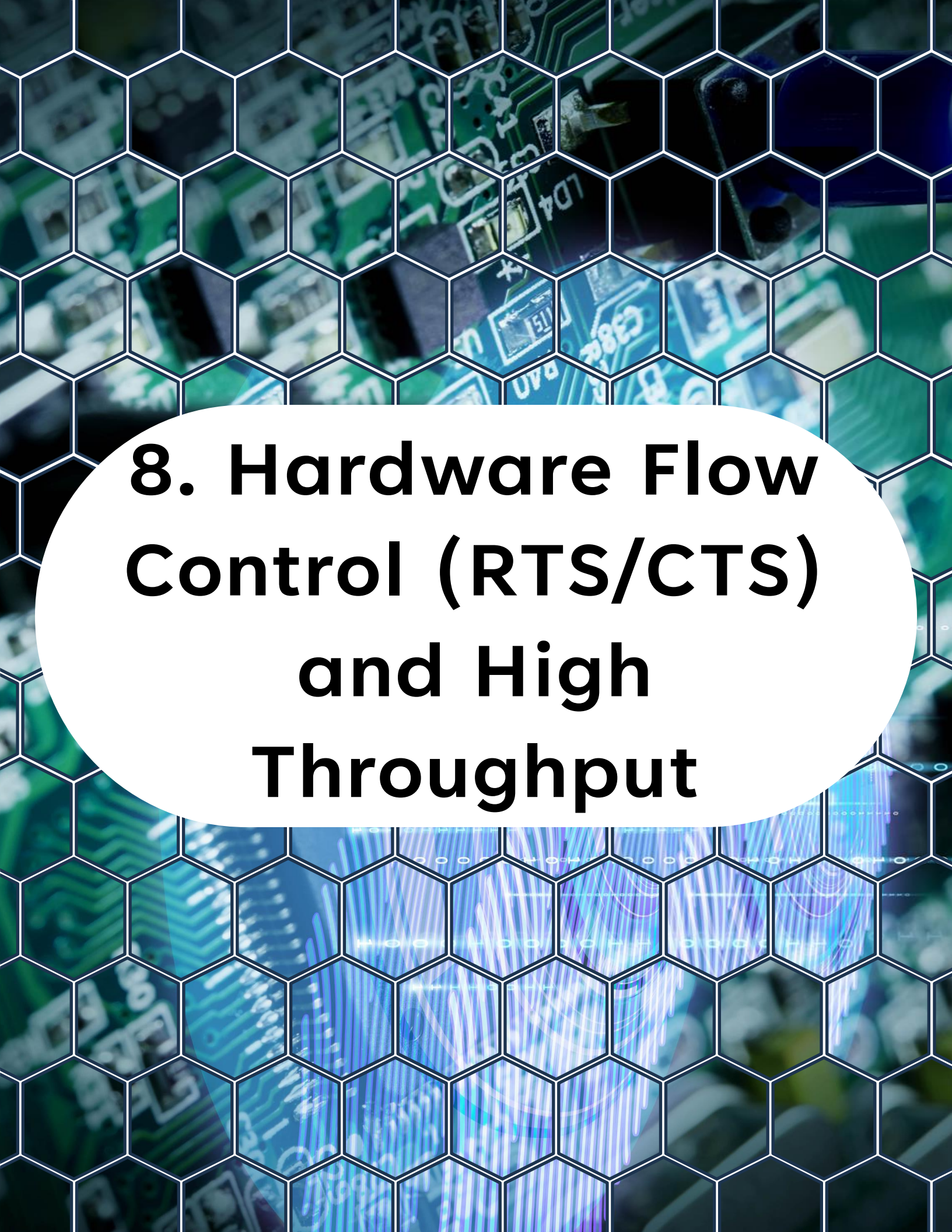
```
static void uart_tx_task(void *arg)
{
    uart_tx_msg_t msg;

    while (1) {
        // Wait for a message to send
        if (xQueueReceive(uart_tx_queue, &msg, portMAX_DELAY) == pdTRUE) {
            // Write and optionally wait for completion
            uart_write_bytes(UART_PORT, (const char *)msg.data, msg.len);

            // Optional: ensure data left UART hardware before next messag
            uart_wait_tx_done(UART_PORT, pdMS_TO_TICKS(100));
        }
    }
}
```

7. TX Architecture for Reliable Sending

Enqueue from application code:

```c
static bool uart_send_line_async(const char *s)
{
    uart_tx_msg_t msg;
    size_t n = strlen(s);
    if (n >= sizeof(msg.data)) return false;

    memcpy(msg.data, s, n);
    msg.len = n;

    return (xQueueSend(uart_tx_queue, &msg, pdMS_TO_TICKS(10)) == pdTRUE);
}
```

If you need large payloads, do not copy into fixed arrays; instead queue pointers to allocated buffers (and define clear ownership/free policy).



8. Hardware Flow Control (RTS/CTS) and High Throughput

8. Hardware Flow Control (RTS/CTS) and High Throughput

8.1 When you should enable RTS/CTS

Enable hardware flow control when:

- baud rate is high (e.g., 921600+)
- external module bursts logs/data (GNSS NMEA at high update rates, modems)
- your firmware occasionally has CPU spikes (Wi-Fi, TLS, heavy parsing)
- you cannot tolerate drops

8.2 ESP-IDF configuration for RTS/CTS

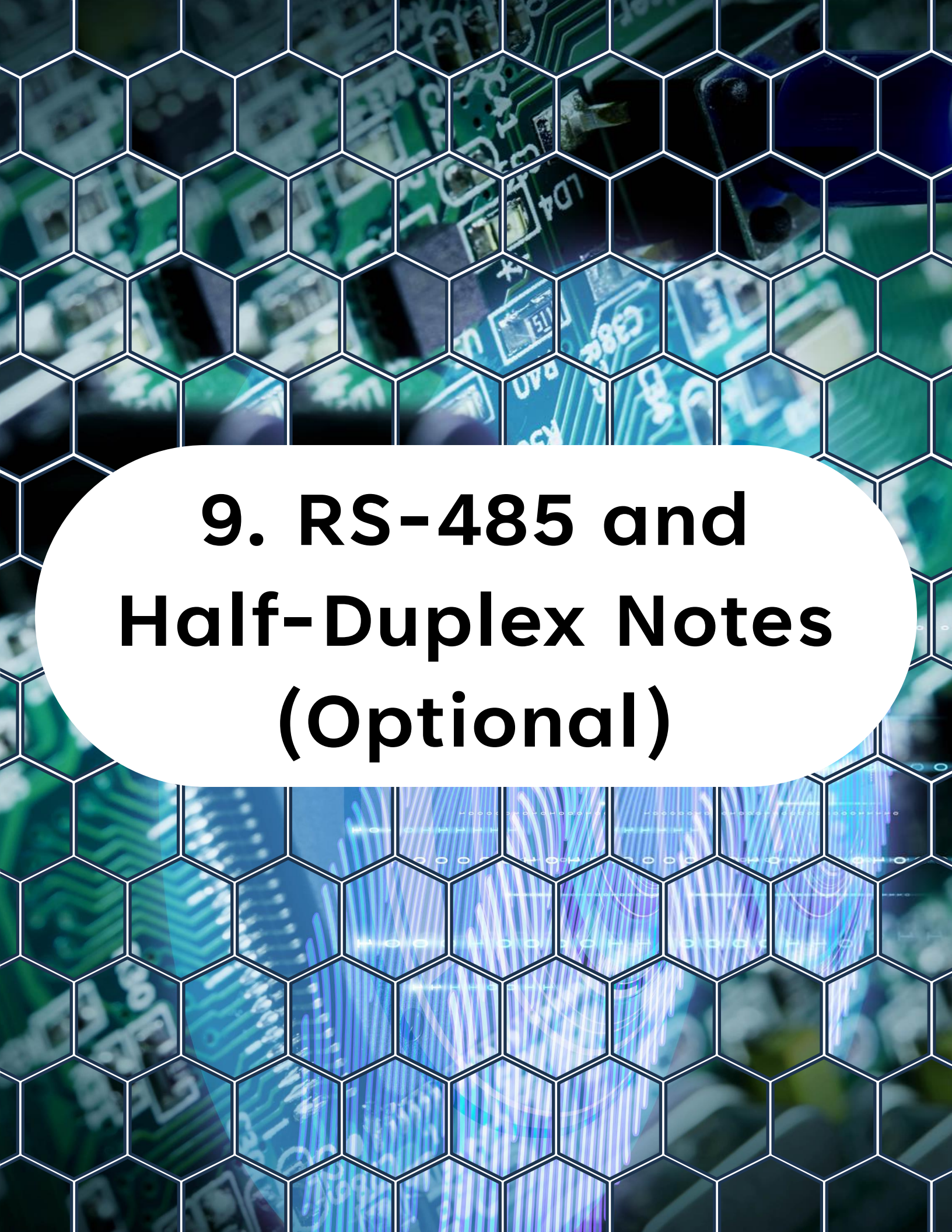
```c
#define UART_RTS_PIN GPIO_NUM_19
#define UART_CTS_PIN GPIO_NUM_20

static void uart_enable_hw_flow(void)
{
    // UART configuration structure with hardware flow control enabled
    uart_config_t cfg = {
        .baud_rate = UART_BAUD_RATE,
        .data_bits = UART_DATA_8_BITS,
        .parity    = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
```

8. Hardware Flow Control (RTS/CTS) and High Throughput

```c
#define UART_RTS_PIN GPIO_NUM_19
#define UART_CTS_PIN GPIO_NUM_20

static void uart_enable_hw_flow(void)
{
    // UART configuration structure with hardware flow control enabled
    uart_config_t cfg = {
        .baud_rate = UART_BAUD_RATE,
        .data_bits = UART_DATA_8_BITS,
        .parity    = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_CTS_RTS,
        .rx_flow_ctrl_thresh = 122, // tune based on buffering
        .source_clk = UART_SCLK_DEFAULT,
    };

    // Install UART driver with event queue
    ESP_ERROR_CHECK(uart_param_config(UART_PORT, &cfg));

    // Set UART pins with RTS and CTS
    ESP_ERROR_CHECK(uart_set_pin(UART_PORT, UART_TX_PIN, UART_RX_PIN,
                                 UART_RTS_PIN, UART_CTS_PIN));
}
```

8.3 Test it

Force your parser to sleep for 200-500 ms and verify you do not lose frames. With flow control disabled, you typically will.



9. RS-485 and Half-Duplex Notes (Optional)

9. RS-485 and Half-Duplex Notes (Optional)

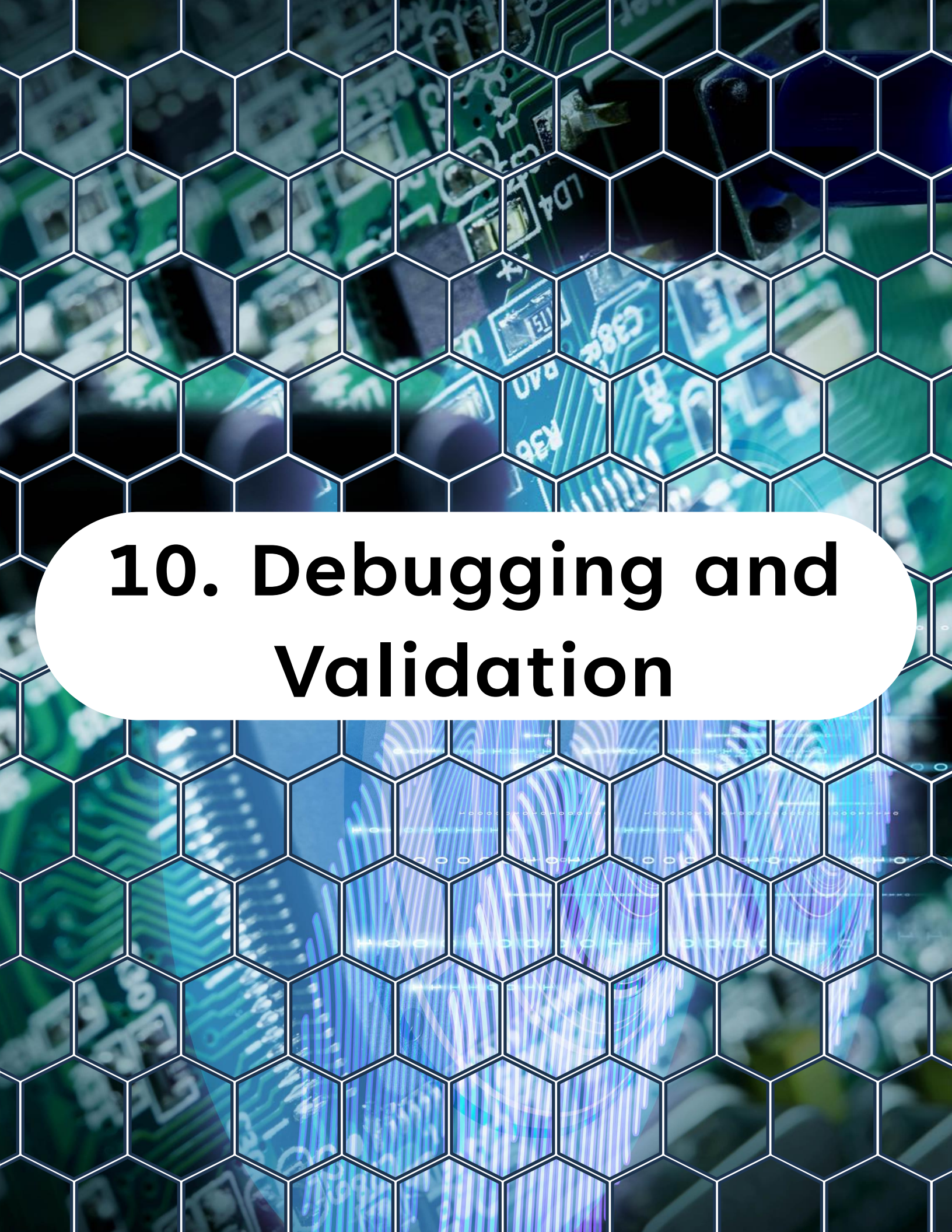
If you are connecting to industrial devices over RS-485:

UART becomes half-duplex (shared line)

you must toggle DE/RE on the transceiver

timing matters

ESP-IDF has RS-485 support modes for UART (varies by IDF version). Even if you do manual DE pin toggling, keep the TX serialized with a TX task.



10. Debugging and Validation

10. Debugging and Validation

10.1 Basic electrical checks

- TX from ESP goes to RX on module (crossed)
- RX from ESP goes to TX on module
- shared GND is mandatory
- confirm module logic voltage
- confirm baud/parity/stop match exactly

10.2 Use a logic analyzer or scope

Look for:

idle level high (typical UART)

correct bit timing for your baud rate

unexpected glitches/noise

10. Debugging and Validation

10.3 Separate logging UART from device UART

If you use the same UART for logs and device traffic, it becomes confusing quickly.

Recommendation:

- keep ESP-IDF logging on UART0
- use UART1/2 for device link

10.4 Loopback test

For quick sanity:

connect TX and RX together on the same UART port

send a known string and verify you receive it back

11. Performance, Power, and Reliability Tips

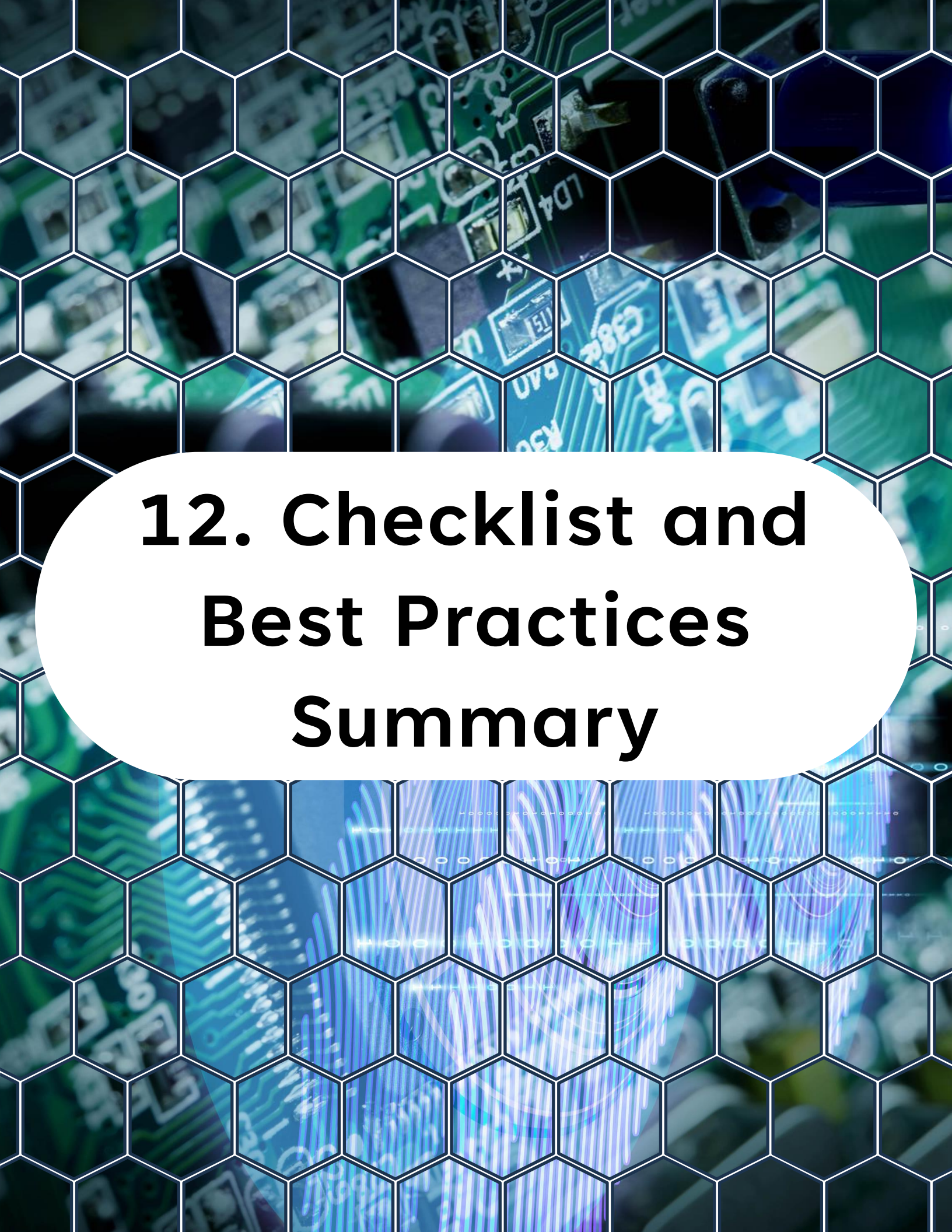
11. Performance, Power, and Reliability Tips

1. Buffer sizing:
    - Size RX buffer based on worst-case burst while your parser is busy.
    - If your module can send 2 KB bursts, do not set RX buffer to 256 bytes.
2. Avoid heavy work in RX task:
    - no JSON parsing in RX task
    - no long logs in RX task
    - forward to parser task
3. Watchdog safety:
    - do not run tight loops without delays or blocking calls
    - keep tasks responsive with timeouts

11. Performance, Power, and Reliability Tips

4. Use counters for field diagnostics:

    - fifo_overflow_count
    - buffer_full_count
    - frame_err_count
    - parity_err_count



12. Checklist and Best Practices Summary

12. Checklist and Best Practices Summary

Configuration checklist:

1. Correct pins, crossed TX/RX, shared GND
2. Matching baud/parity/stop bits
3. Consider RTS/CTS for bursty or high baud devices
4. Use a dedicated UART port for external devices if possible

FreeRTOS checklist:

1. RX task is event-driven and fast
2. Parsing is in a separate task
3. TX is owned by one task, fed by a queue
4. Timeouts are used (avoid infinite blocks everywhere)

12. Checklist and Best Practices Summary

5. Buffer sizes are chosen by worst-case burst, not guesswork

Protocol checklist:

1. Define framing (newline or length-prefixed)
2. Handle partial frames and resync policy
3. Add CRC if corruption matters
4. Handle overflow recovery deterministically

"Thanks for watching!

If you enjoyed this video,

make sure to hit the like button and subscribe to stay updated with my latest content.

Don't forget to check out my other videos for more tips and tutorials on Embedded C, Python, hardware designs, etc. Keep exploring, keep learning, and I'll see you in the next video!"

DISCOVER, LEARN, SUCCEED!

https://www.linkedin.com/in/yamil-garcia
https://www.youtube.com/@LearningByTutorials
https://github.com/god233012yamil