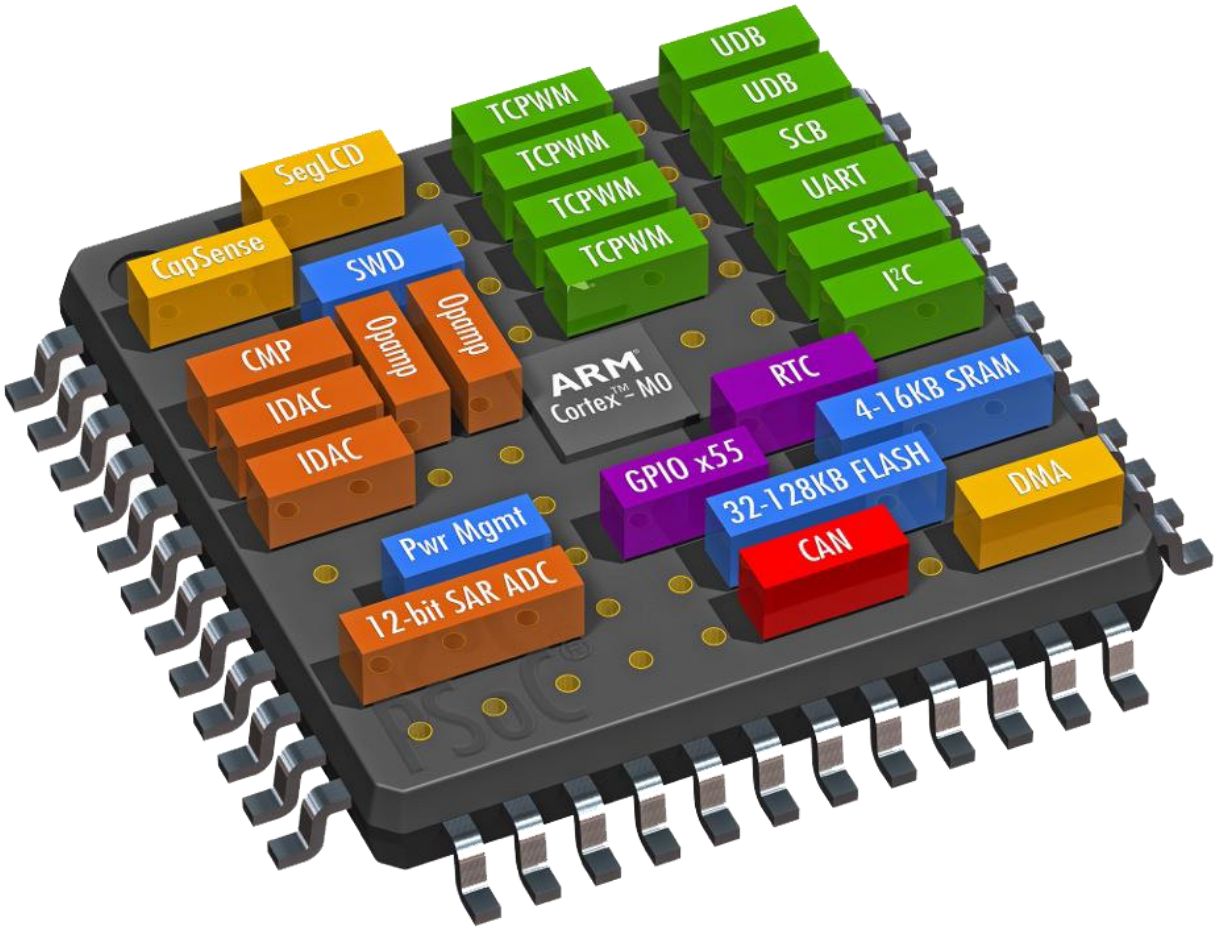
Using the Cooperative Scheduler Pattern with the State Machine Pattern in Resource-Constrained Embedded Systems



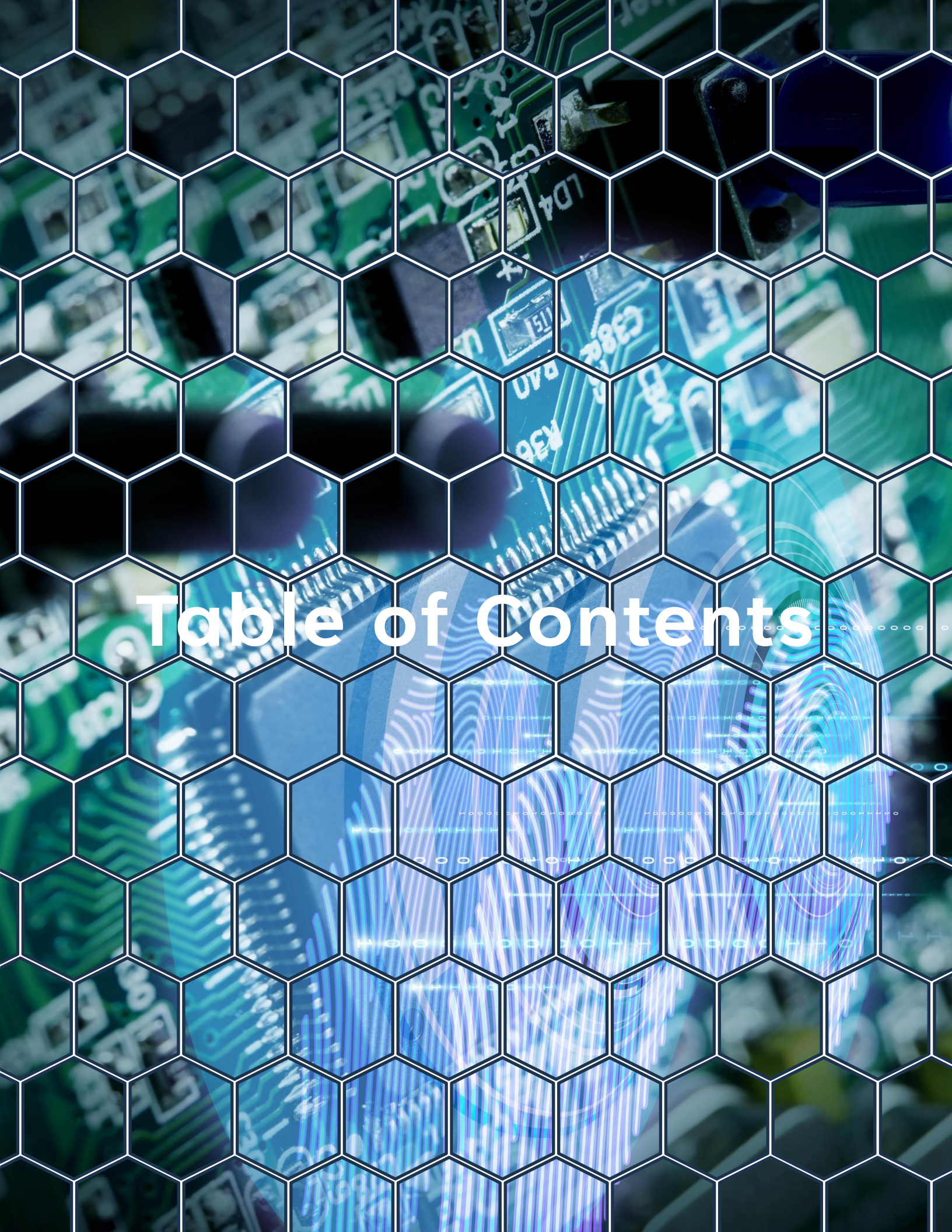
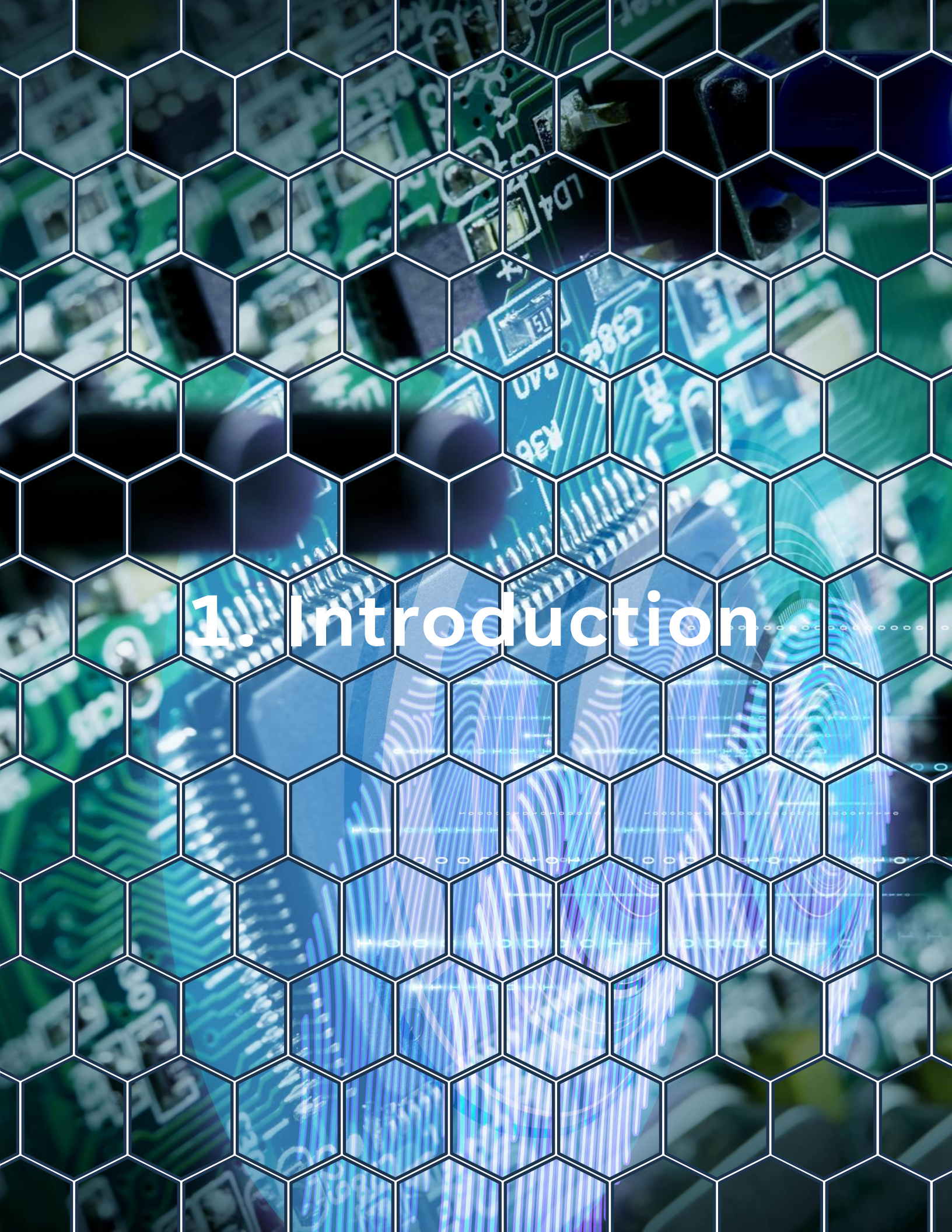


Table of Contents

Table of Contents



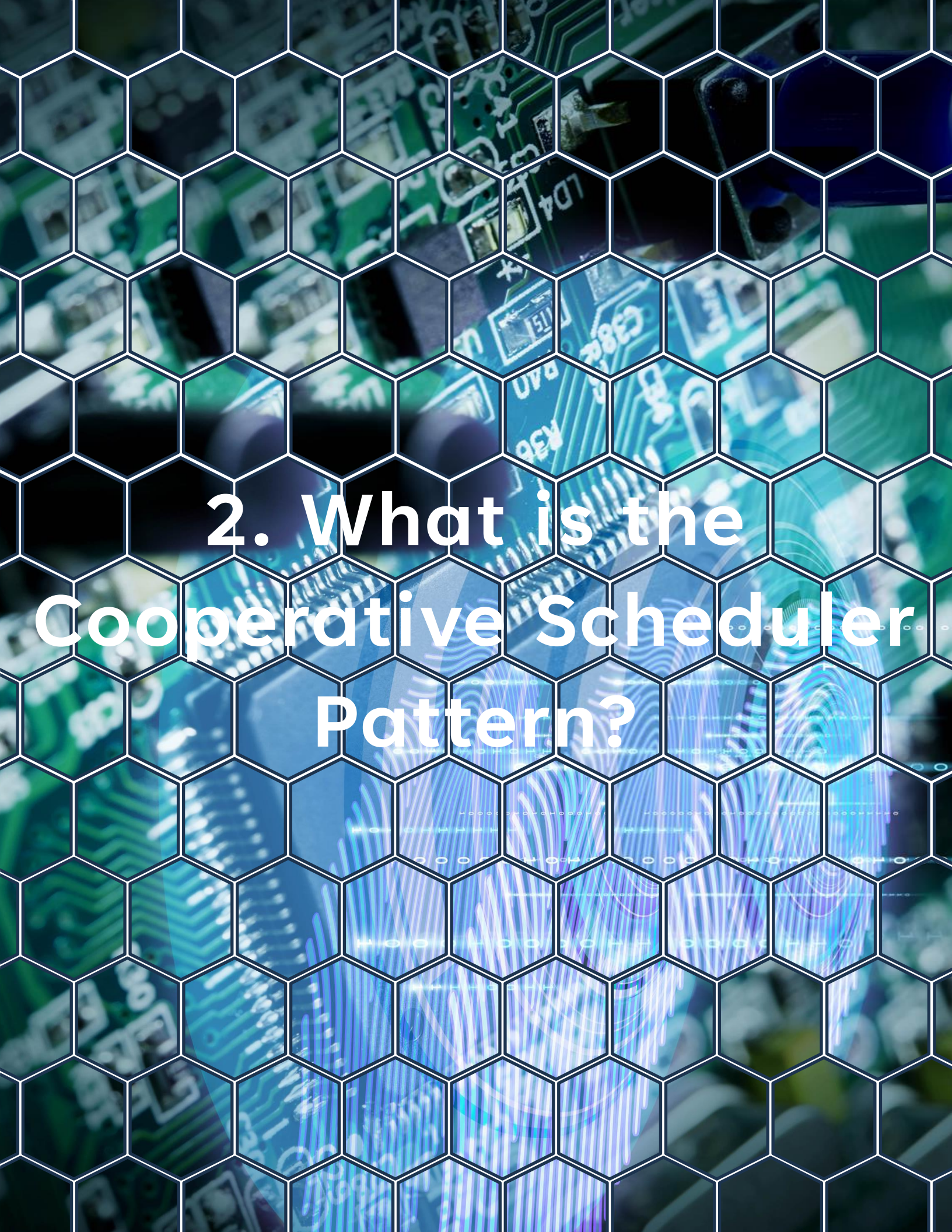
1. Introduction

1. Introduction

In embedded systems development, particularly on small MCUs like the **ATtiny1616**, developers often face significant resource constraints. These devices typically offer **no hardware support for multitasking**, possess **limited RAM (often ~2KB or less)**, and feature a **single-core architecture**. While real-time operating systems (RTOS) provide powerful multitasking capabilities, their overhead and complexity make them unsuitable for these low-end applications.

1. Introduction

To handle multiple time-sensitive tasks—such as reading sensors via ADC, communicating over USART or I2C, and controlling GPIOs—embedded developers must design software that emulates concurrency. The most effective way to accomplish this is through a combination of **Cooperative Scheduler** and **State Machine** design patterns. This article explores the use of these patterns together and provides a real-world implementation on the ATtiny1616 platform.



2. What is the Cooperative Scheduler Pattern?

2. What is the Cooperative Scheduler Pattern?

The **Cooperative Scheduler Pattern** is a task management technique that enables pseudo-concurrent execution of multiple tasks on a single-core, non-threaded system. In this model, each task is executed in small, non-blocking chunks, and control is voluntarily returned to a central loop (the scheduler) once a task completes or pauses itself.

2. What is the Cooperative Scheduler Pattern?

Characteristics:

Non-preemptive: Tasks must explicitly yield control.

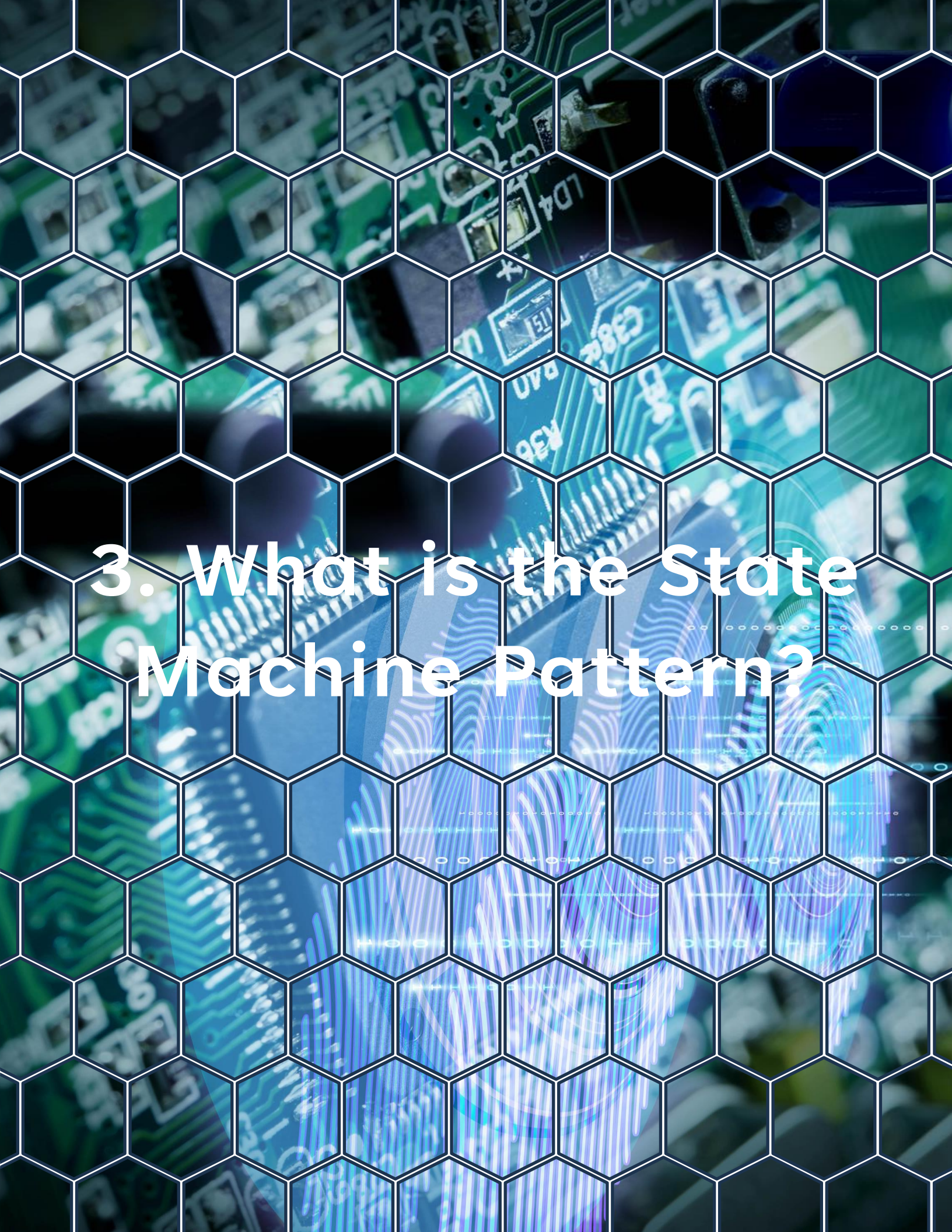
Round-robin structure: All tasks are called repeatedly in sequence.

No context switching: Unlike in RTOS, no stack switching or task isolation is needed.

ISR-safe: Interrupt Service Routines (ISRs) are used only to trigger events, not to execute long logic.

2. What is the Cooperative Scheduler Pattern?

This pattern is ideal for MCUs with minimal memory and no OS support, as it imposes very little overhead and keeps task execution highly deterministic.



3. What is the State Machine Pattern?

3. What is the State Machine Pattern?

The **State Machine Pattern** models a task as a series of well-defined states and transitions. Each task maintains a **state variable**, and depending on internal logic or external events (e.g., flags, timers, or ISR triggers), it transitions to the next state.

3. What is the State Machine Pattern?

Benefits:

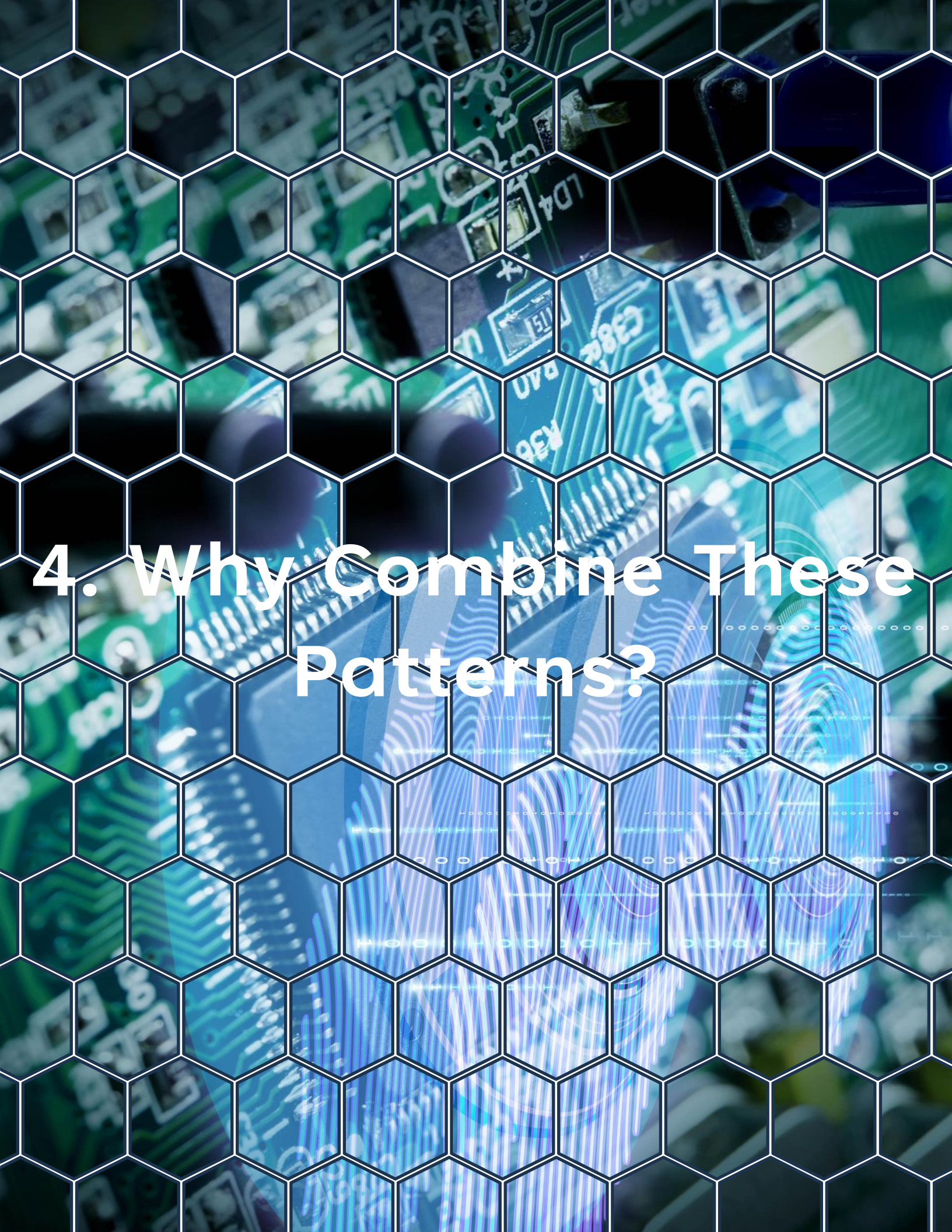
Modular design: Easier to isolate functionality per task.

Improved readability: Code becomes easier to follow and maintain.

Event-driven behavior: Tasks react to timers, flags, or ISR signals efficiently.

Testability: Each state can be validated independently.

State machines are especially effective in embedded systems for **handling peripherals**, **managing protocol layers**, or **sequencing hardware operations**.



4. Why Combine These Patterns?

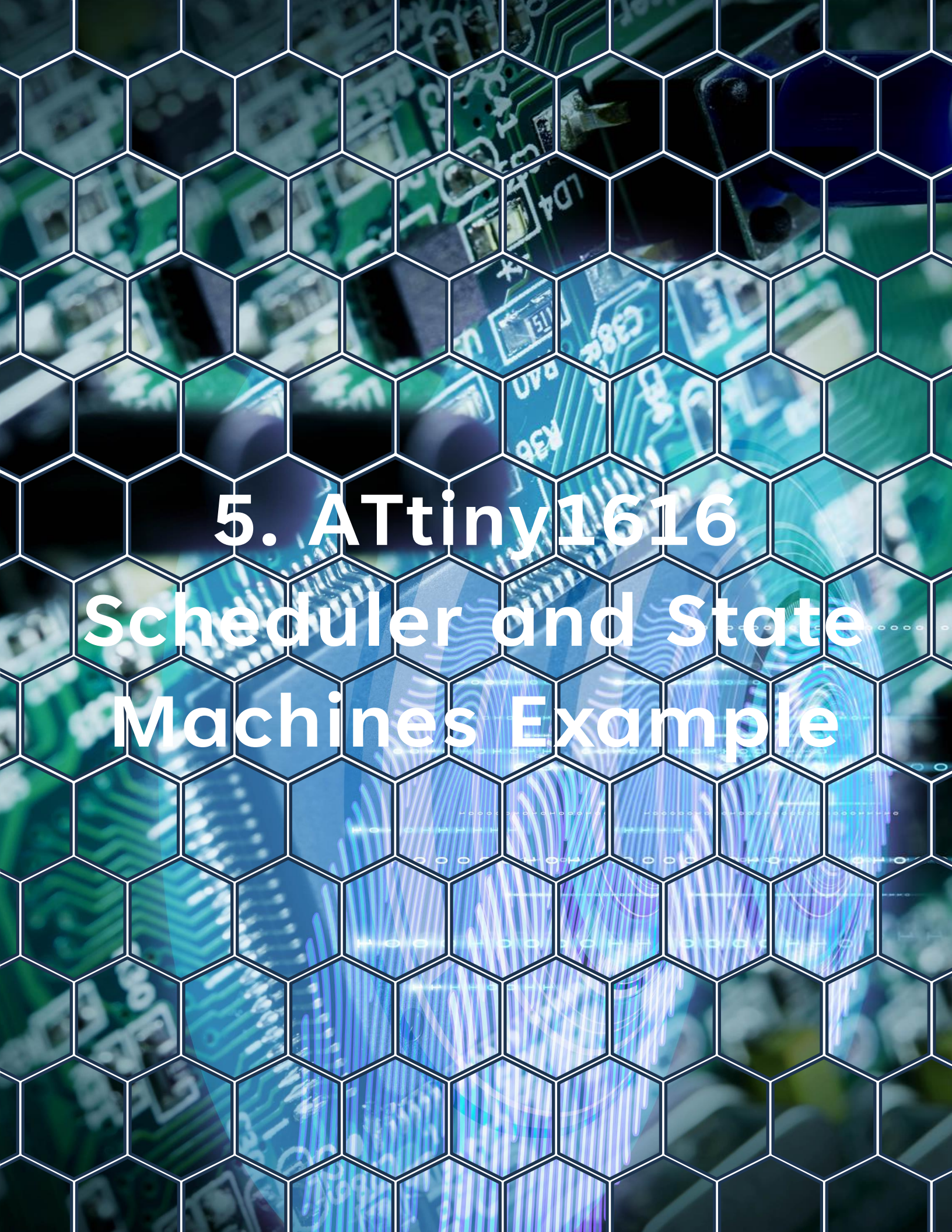
4. Why Combine These Patterns?

When used together, the **Cooperative Scheduler** manages *when* tasks execute, while State Machines manage *how* they behave. This division of responsibility allows the system to:

- Avoid **blocking operations** that would delay other tasks.
- **Simulate** concurrency without relying on an RTOS.
- Make logic more **predictable** and **traceable** during debugging.
- Keep **ISR execution fast**, with flags used for signaling instead of heavy logic.

4. Why Combine These Patterns?

By using this architecture, you can execute multiple high-level behaviors (e.g., ADC reading, reading, USART echoing, I2C polling) in a single-core MCU **without race conditions** or **preemption risks**.



5. ATtiny1616 Scheduler and State Machines Example

5. ATtiny1616 Scheduler and State Machines Example

Let's implement three pseudo-concurrent tasks using a cooperative scheduler and per-task state machines on an ATtiny1616:

- **ADC Task**: Triggered by a timer every 500ms.
- **I2C Task**: Simulated read cycle every 2s.
- **USART Task**: Sends a periodic message.

⚠ All code is written for AVR-GCC using bare-metal programming, targeting ATtiny1616.

5. ATtiny1616 Scheduler and State Machines Example

Task State Definitions

```
typedef enum { ADC_IDLE, ADC_START, ADC_WAIT, ADC_DONE } adc_state_t;
typedef enum { I2C_IDLE, I2C_START, I2C_WAIT, I2C_DONE } i2c_state_t;
typedef enum { USART_IDLE, USART_SEND, USART_DONE } usart_state_t;
```

Global Flags and State Variables

```
volatile bool adc_triggered = false;
adc_state_t adc_state = ADC_IDLE;
i2c_state_t i2c_state = I2C_IDLE;
usart_state_t usart_state = USART_IDLE;
```

Timer Overflow ISR (for ADC trigger)

```
ISR(TCA0_OVF_vect) {
    adc_triggered = true;
    TCA0.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;
}
```

5. ATtiny1616 Scheduler and State Machines Example

ADC Task (Non-blocking)

```c
void adc_task(void) {
    static uint16_t result = 0;

    switch (adc_state) {
        case ADC_IDLE:
            if (adc_triggered) {
                adc_triggered = false;
                adc_state = ADC_START;
            }
            break;

        case ADC_START:
            ADC0.COMMAND = ADC_STCONV_bm;
            adc_state = ADC_WAIT;
            break;

        case ADC_WAIT:
            if (ADC0.INTFLAGS & ADC_RESRDY_bm) {
                result = ADC0.RES;
                adc_state = ADC_DONE;
            }
            break;

        case ADC_DONE:
            // Process result (e.g., format/send)
            adc_state = ADC_IDLE;
            break;
    }
}
```

5. ATtiny1616 Scheduler and State Machines Example

I2C Task (Simulated State Machine)

```c
void i2c_task(void) {
    static uint16_t delay_counter = 0;

    switch (i2c_state) {
        case I2C_IDLE:
            delay_counter = 0;
            i2c_state = I2C_START;
            break;

        case I2C_START:
            // Begin fake transaction
            i2c_state = I2C_WAIT;
            break;

        case I2C_WAIT:
            if (++delay_counter > 2000) { // Simulate delay
                i2c_state = I2C_DONE;
            }
            break;

        case I2C_DONE:
            // Simulate result handling
            i2c_state = I2C_IDLE;
            break;
    }
}
```

5. ATtiny1616 Scheduler and State Machines Example

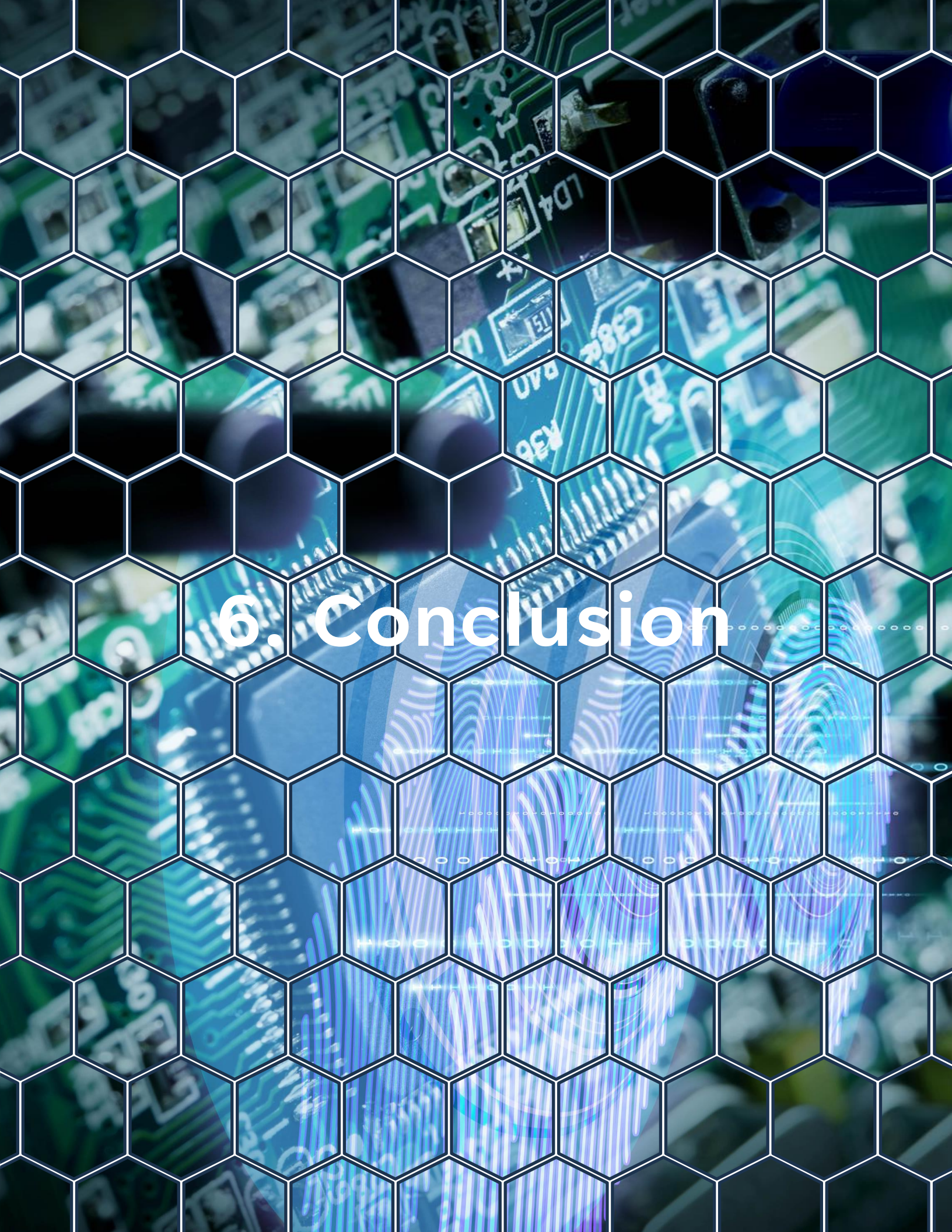
USART Task

```c
void usart_task(void) {
    static const char *msg = "Hello from USART\n";
    static uint8_t idx = 0;

    switch (usart_state) {
        case USART_IDLE:
            usart_state = USART_SEND;
            idx = 0;
            break;

        case USART_SEND:
            if (USART0.STATUS & USART_DREIF_bm) {
                USART0.TXDATAL = msg[idx++];
                if (msg[idx] == '\0') {
                    usart_state = USART_DONE;
                }
            }
            break;

        case USART_DONE:
            usart_state = USART_IDLE;
            break;
    }
}
```

5. ATtiny1616 Scheduler and State Machines Example

Main Loop: Cooperative Scheduler

```c
int main(void) {
    adc_init();
    usart_init();
    init_timer();
    sei();  // Enable interrupts

    while (1) {
        adc_task();
        i2c_task();
        usart_task();
        // Optional: power-saving or watchdog reset
    }
}
```



6. Conclusion

6. Conclusion

In resource-constrained embedded systems, where multitasking is not natively supported and memory is minimal, combining the **Cooperative Scheduler** and **State Machine** patterns offers a clean, modular, and safe way to implement pseudo-concurrent tasks. This architecture not only eliminates the need for an RTOS but also improves code maintainability and timing predictability—critical traits in real-time embedded designs.

6. Conclusion

By structuring each task as a non-blocking state machine and executing them cooperatively within the main loop, developers gain full control over timing, ISR behavior, and system responsiveness—all while keeping the system footprint extremely low. Whether you're reading sensors, polling buses, or managing communication, this method remains a tried-and-true strategy in the embedded engineer's toolbox.