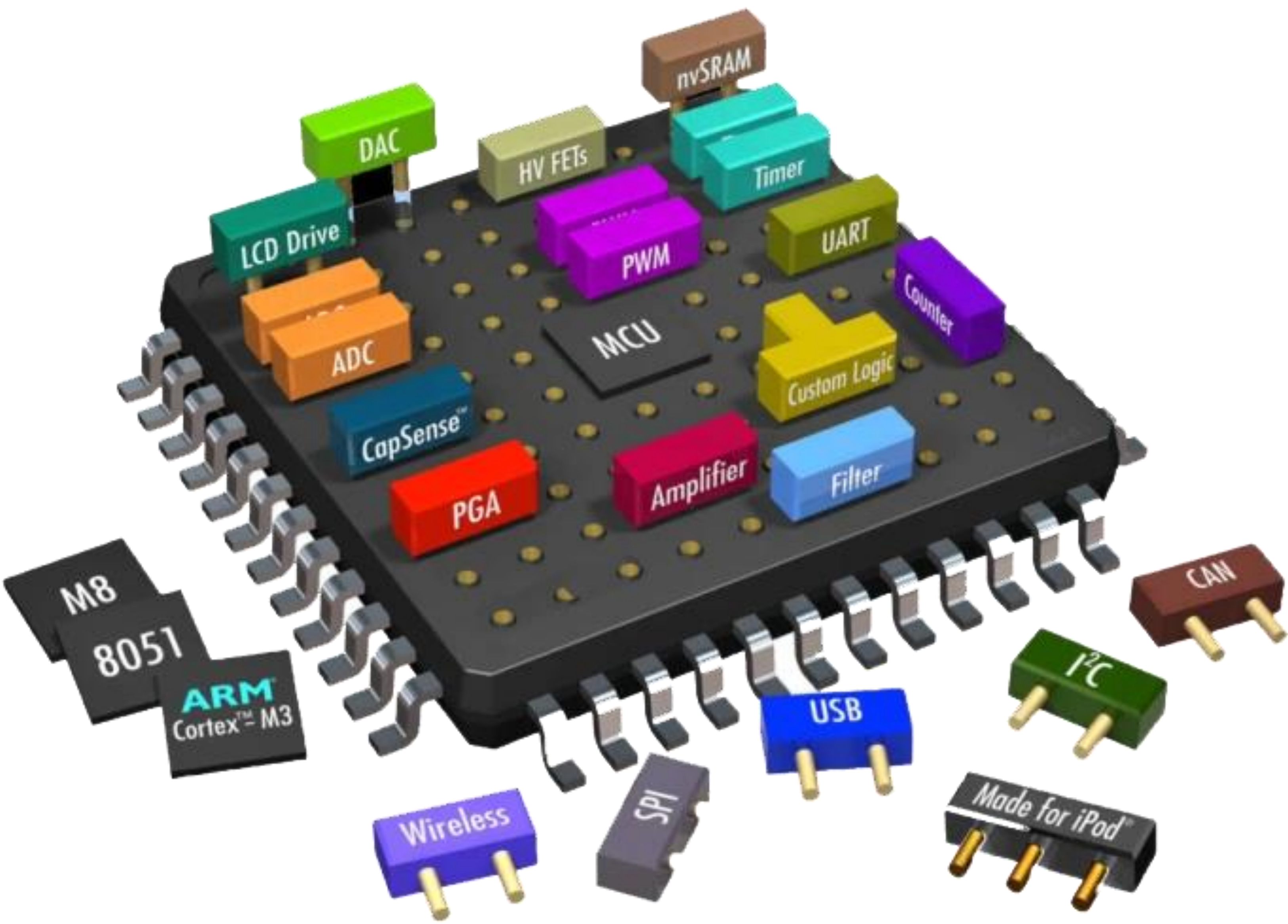
Design Patterns in Resource-Constrained Embedded Systems



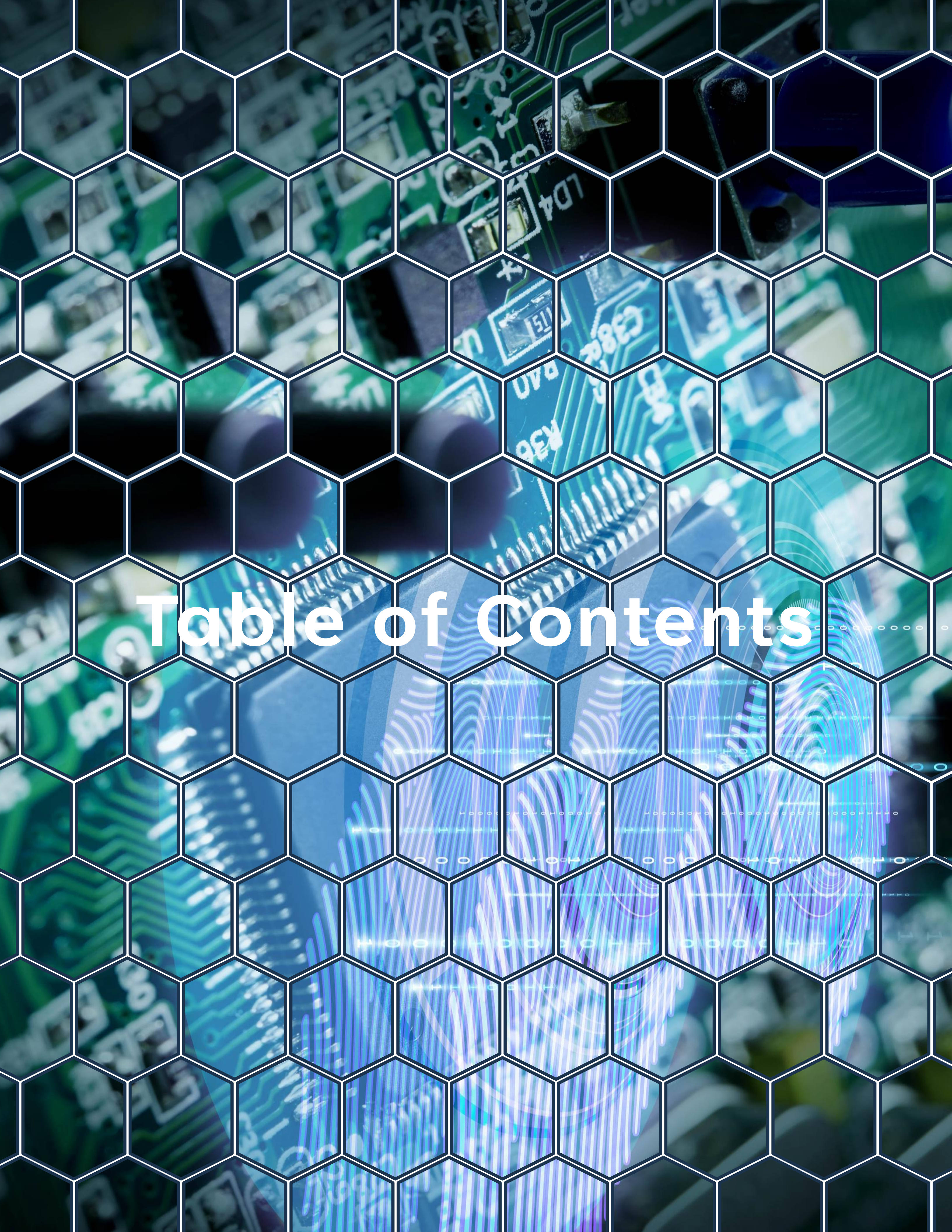
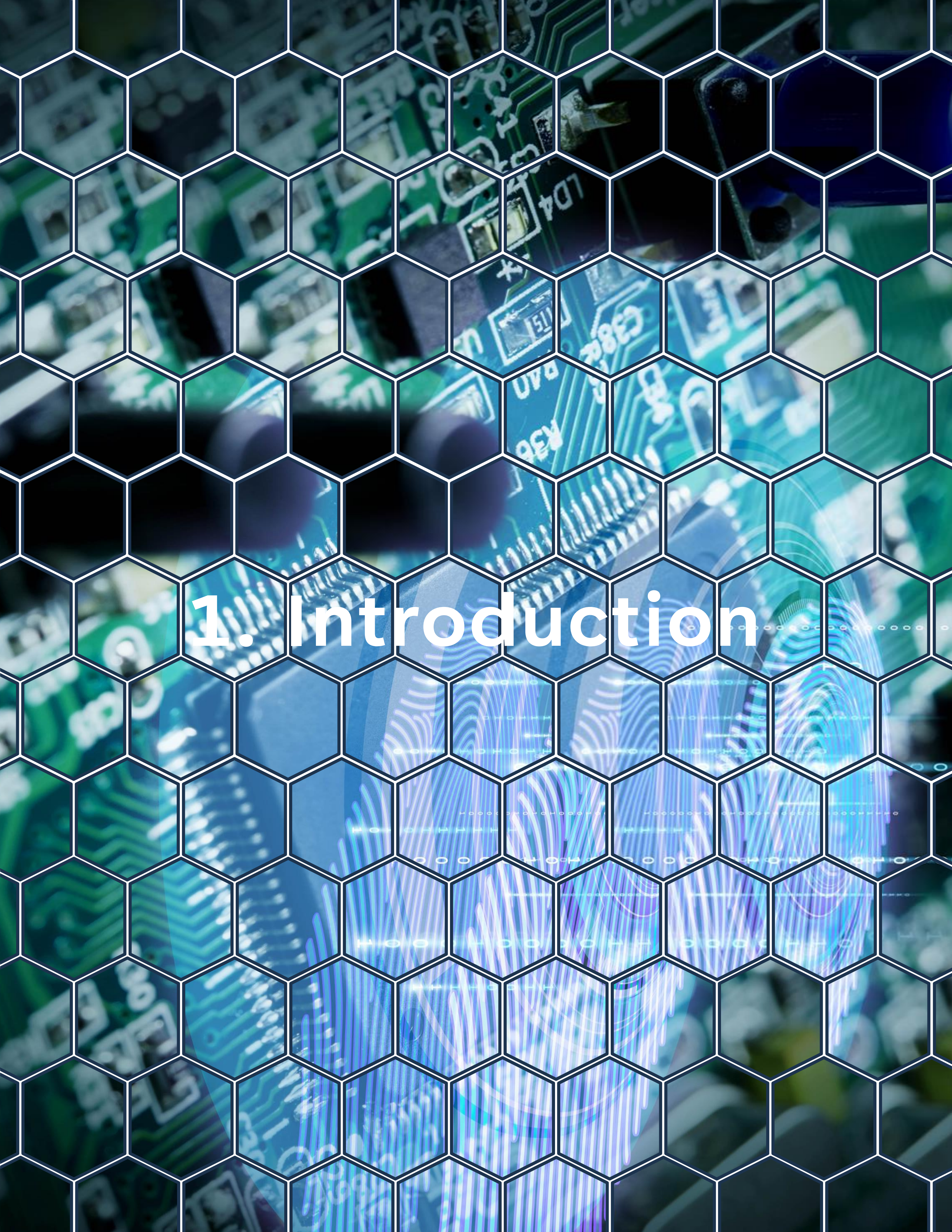


Table of Contents

Table of Contents



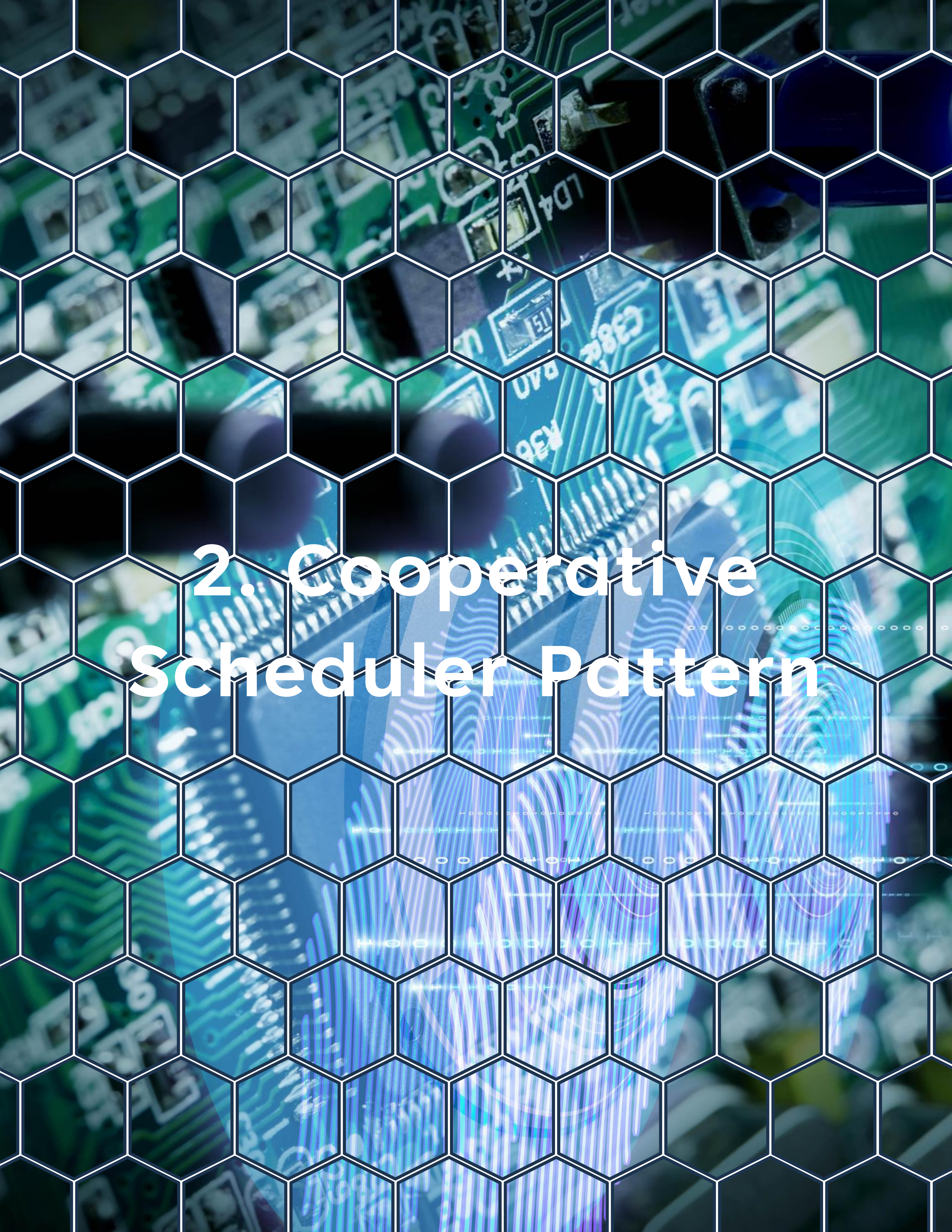
1. Introduction

1. Introduction

Designing embedded firmware for resource-constrained systems is a delicate balance between performance, power efficiency, and code clarity. Unlike application-level development on full-fledged operating systems, these systems operate with limited SRAM (often less than 2KB), single-core processors, and no built-in support for multitasking. As such, firmware engineers must architect software with deterministic behavior, minimal stack usage, and absolute control over timing.

1. Introduction

Design patterns, when properly applied, become powerful tools to manage complexity and reuse tested architectural principles. However, not all patterns translate well to bare-metal embedded systems. This article explores patterns that have been adapted to low-level environments such as AVR microcontrollers (e.g., ATtiny1616), where interrupts, non-blocking execution, and low-power states are the rule, not the exception.



2. Cooperative Scheduler Pattern

2. Cooperative Scheduler Pattern

Purpose

Implements multitasking in a non-preemptive way by explicitly yielding control between tasks.

Use Case in Embedded

Enables simple task scheduling on single-core MCUs without RTOS support. Tasks must complete quickly or yield periodically to keep the system responsive.

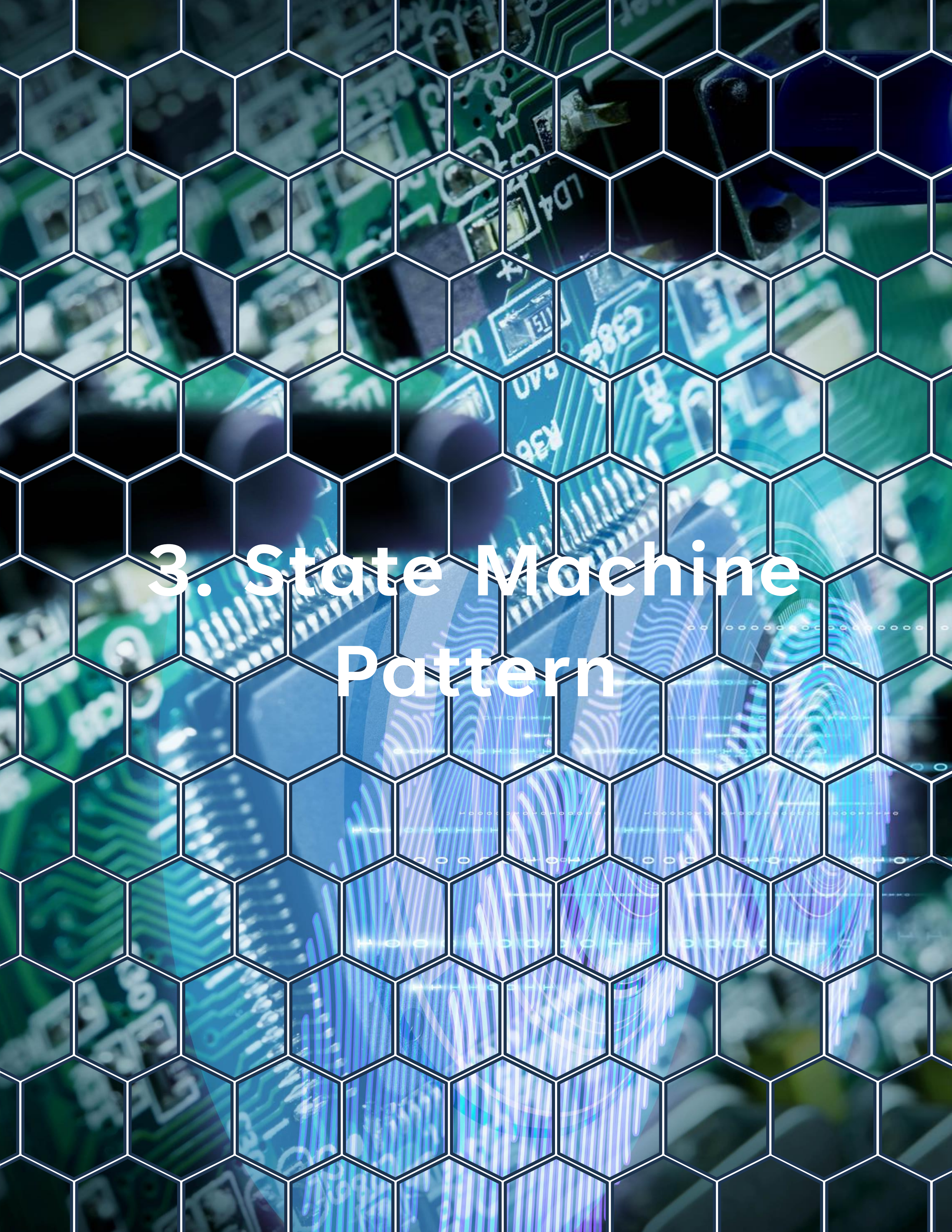
Best Practice Notes

- Avoid blocking delays (`_delay_ms()`).
- Design tasks to run for a bounded time.
- Use timer flags or counters to defer execution.

2. Cooperative Scheduler Pattern

Example

```c
typedef void (*TaskFunc)(void);

typedef struct {
    TaskFunc func;
    uint16_t interval_ms;
    uint16_t elapsed_ms;
} Task;

void task_led_toggle(void);
void task_uart_poll(void);

Task scheduler[] = {
    {task_led_toggle, 500, 0},
    {task_uart_poll, 10, 0}
};

#define NUM_TASKS (sizeof(scheduler)/sizeof(Task))

void run_scheduler(uint16_t tick_ms) {
    for (uint8_t i = 0; i < NUM_TASKS; ++i) {
        scheduler[i].elapsed_ms += tick_ms;
        if (scheduler[i].elapsed_ms >= scheduler[i].interval_ms) {
            scheduler[i].elapsed_ms = 0;
            scheduler[i].func(); // Cooperative call
        }
    }
}

// Timer ISR increments system tick and triggers run_scheduler()
```



3. State Machine Pattern

3. State Machine Pattern

Purpose

Encapsulates system behavior into defined states and transitions, promoting clear logic separation.

Use Case in Embedded

Used for debouncing buttons, protocol parsing, and user interface handling.

Best Practice Notes

- Use enums for states.
- Transition logic must remain non-blocking.
- Each state must do minimal work per call.

3. State Machine Pattern

Example: Button Debounce FSM

```c
typedef enum { IDLE, DEBOUNCING, PRESSED } ButtonState;
static ButtonState btn_state = IDLE;
static uint16_t debounce_counter = 0;

void fsm_button_tick(bool input_signal) {
    switch (btn_state) {
        case IDLE:
            if (input_signal) {
                btn_state = DEBOUNCING;
                debounce_counter = 10;
            }
            break;

        case DEBOUNCING:
            if (--debounce_counter == 0) {
                if (input_signal) {
                    btn_state = PRESSED;
                    // Do something: button press confirmed
                } else {
                    btn_state = IDLE;
                }
            }
            break;

        case PRESSED:
            if (!input_signal)
```

3. State Machine Pattern

```
void fsm_button_tick(bool input_signal) {
    switch (btn_state) {
        case IDLE:
            if (input_signal) {
                btn_state = DEBOUNCING;
                debounce_counter = 10;
            }
            break;

        case DEBOUNCING:
            if (--debounce_counter == 0) {
                if (input_signal) {
                    btn_state = PRESSED;
                    // Do something: button press confirmed
                } else {
                    btn_state = IDLE;
                }
            }
            break;

        case PRESSED:
            if (!input_signal)
                btn_state = IDLE;
            break;
    }
}
```

4. Event Loop Pattern

4. Event Loop Pattern

Purpose

Processes events in a loop, deferring action logic to event handlers.

Use Case in Embedded

Ideal for communication stacks, central control loops, or serial command processing.

Best Practice Notes

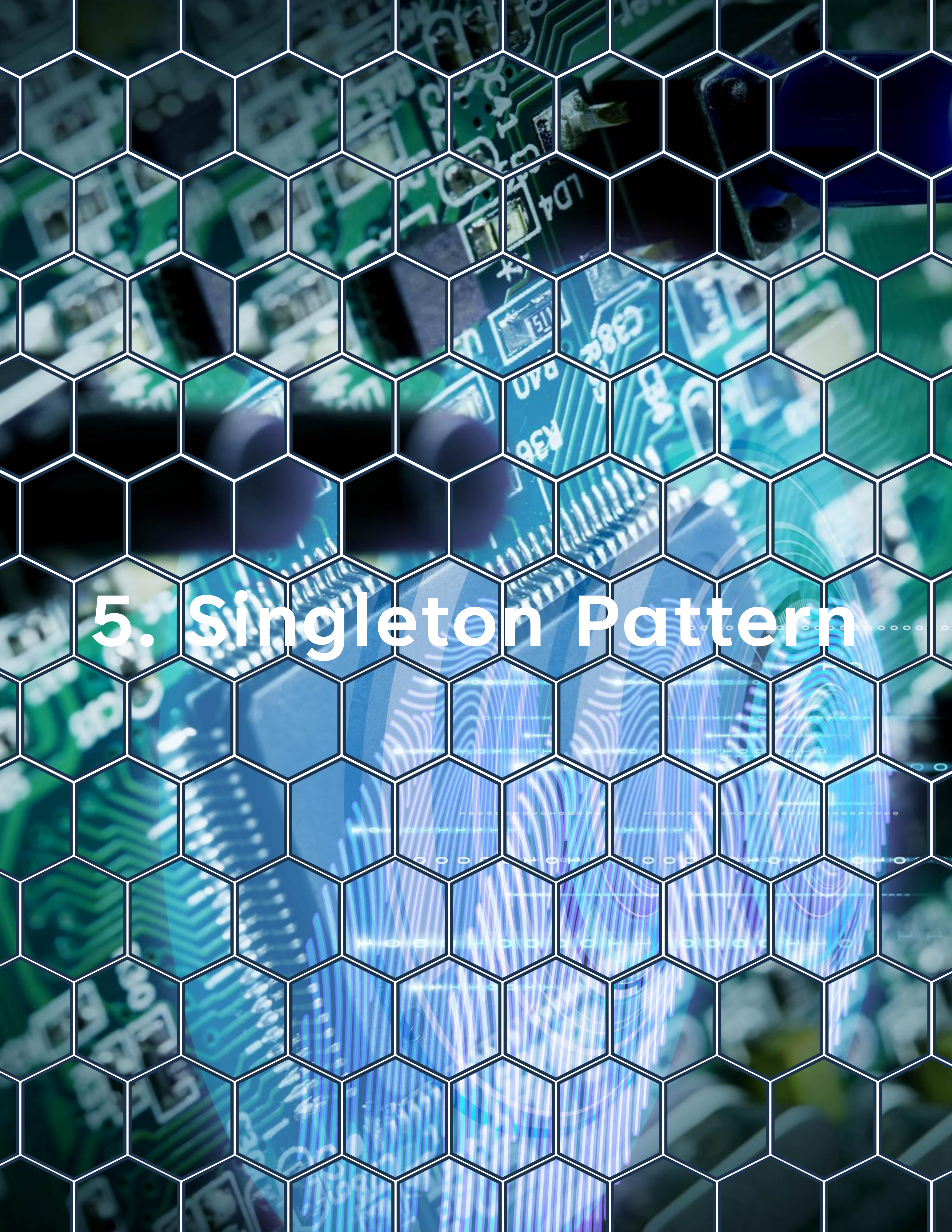
- Use event flags or queues.
- Keep handlers fast and deterministic.

4. Event Loop Pattern

Example: Central Event Loop

```c
typedef enum { EVT_NONE, EVT_UART_RX, EVT_TEMP_READY } EventType;
volatile EventType current_event = EVT_NONE;

void event_loop(void) {
    while (1) {
        switch (current_event) {
            case EVT_UART_RX:
                process_uart_data();
                break;

            case EVT_TEMP_READY:
                read_temperature_sensor();
                break;

            default:
                break;
        }
        current_event = EVT_NONE;
    }
}

// ISR or polling logic sets current_event
```

5. Singleton Pattern



5. Singleton Pattern

Purpose

Ensures a peripheral or manager is accessed through a single instance.

Use Case in Embedded

Used to abstract hardware peripherals like UART, ADC, or SPI, where duplicate access is unsafe.

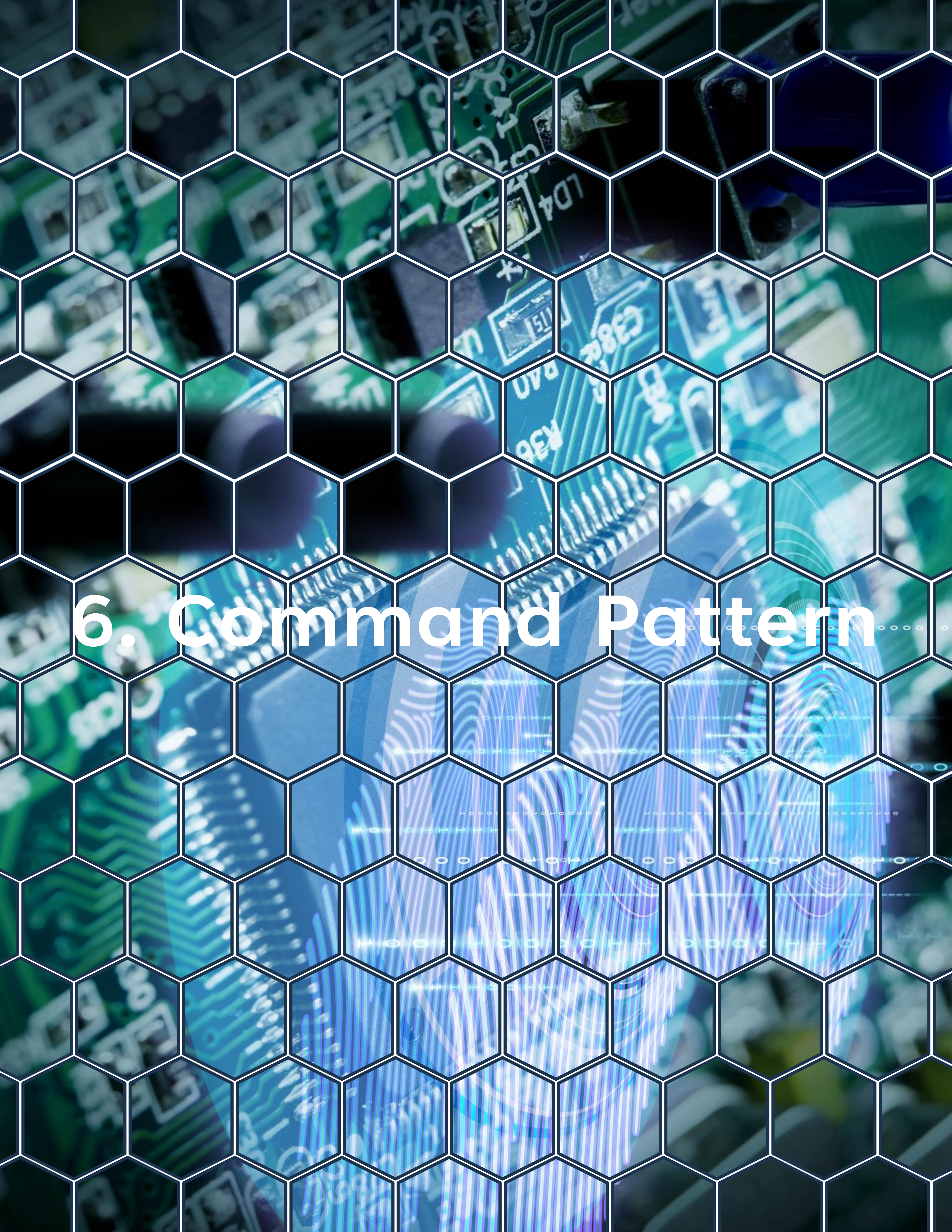
Best Practice Notes

- Implement as a static function-scoped instance.
- Avoid dynamic memory.

5. Singleton Pattern

Example: UART Singleton

```c
typedef struct {
    uint8_t tx_buffer[64];
    uint8_t rx_buffer[64];
} UartDriver;

UartDriver* get_uart_instance(void) {
    static UartDriver uart;
    return &uart;
}
```



6. Command Pattern

6. Command Pattern

Purpose

Encapsulates actions as objects to decouple command issuance from execution.

Use Case in Embedded

Used in CLI interpreters, menu systems, or motor control logic.

Best Practice Notes

- Define command structs or enums.
- Use function pointers to execute commands.

6. Command Pattern

Example: CLI Command Handler

```c
typedef void (*CommandFunc)(void);

typedef struct {
    const char* name;
    CommandFunc execute;
} Command;

void cmd_led_on(void)  { PORTB.OUTSET = PIN0_bm; }
void cmd_led_off(void) { PORTB.OUTCLR = PIN0_bm; }

Command commands[] = {
    {"LEDON",  cmd_led_on},
    {"LEDOFF", cmd_led_off}
};

void handle_command(const char* input) {
    for (int i = 0; i < sizeof(commands)/sizeof(Command); ++i) {
        if (strcmp(input, commands[i].name) == 0) {
            commands[i].execute();
            return;
        }
    }
}
```

7. Observer Pattern

7. Observer Pattern

Purpose

Allows multiple modules to react to state changes in another module.

Use Case in Embedded

Used in sensor threshold alerting or decoupled UI updates.

Best Practice Notes

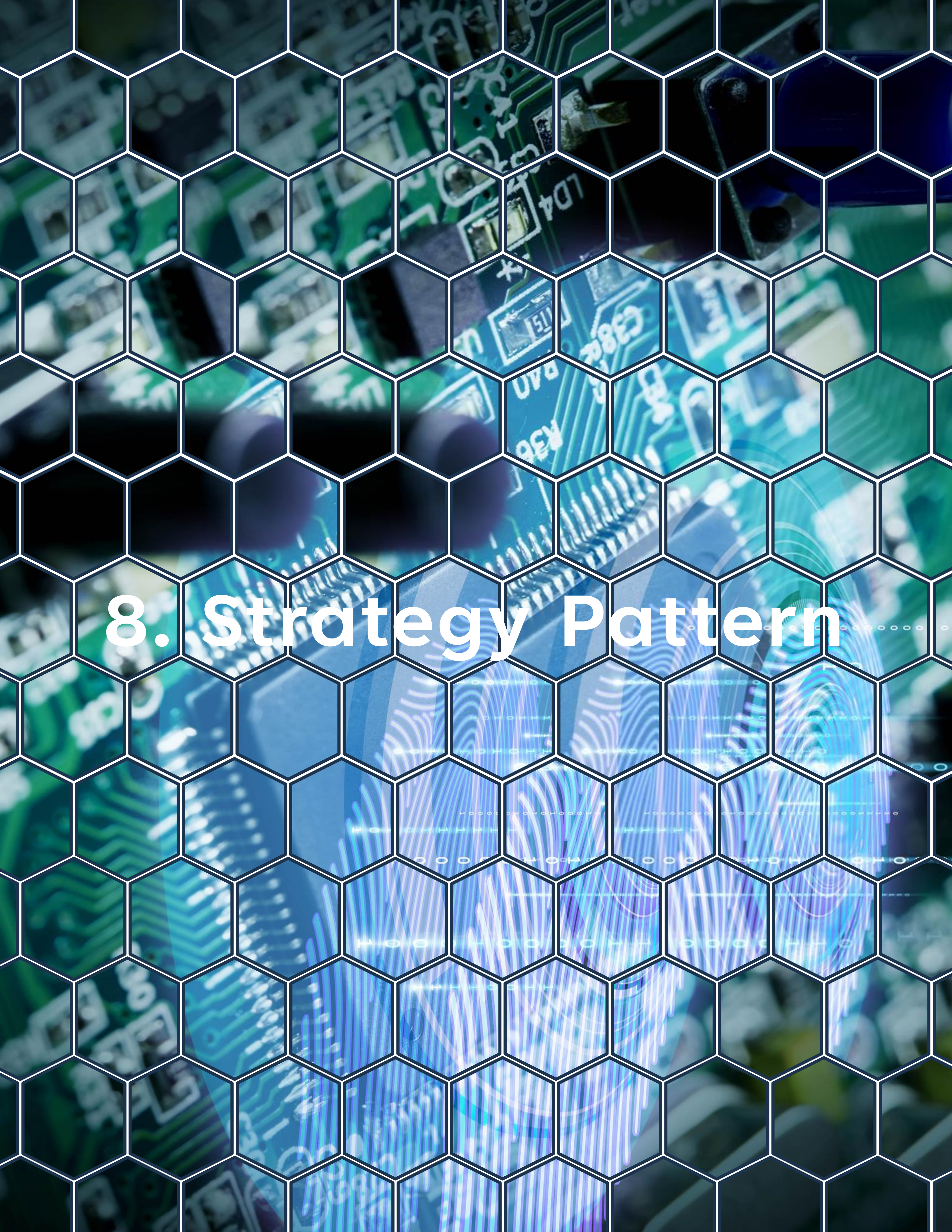
- Use function pointers or callback registries.
- Limit callback duration.

7. Observer Pattern

Example: Temperature Observer

```c
typedef void (*TempCallback)(int16_t temp);
#define MAX_OBSERVERS 4
static TempCallback observers[MAX_OBSERVERS];

void register_temp_observer(TempCallback cb) {
    for (int i = 0; i < MAX_OBSERVERS; ++i) {
        if (!observers[i]) {
            observers[i] = cb;
            break;
        }
    }
}

void notify_temp_change(int16_t temp) {
    for (int i = 0; i < MAX_OBSERVERS; ++i) {
        if (observers[i]) observers[i](temp);
    }
}
```

8. Strategy Pattern



8. Strategy Pattern

Purpose

Allows runtime selection of behavior among multiple algorithms.

Use Case in Embedded

Used for selecting communication methods, power modes, or filtering techniques.

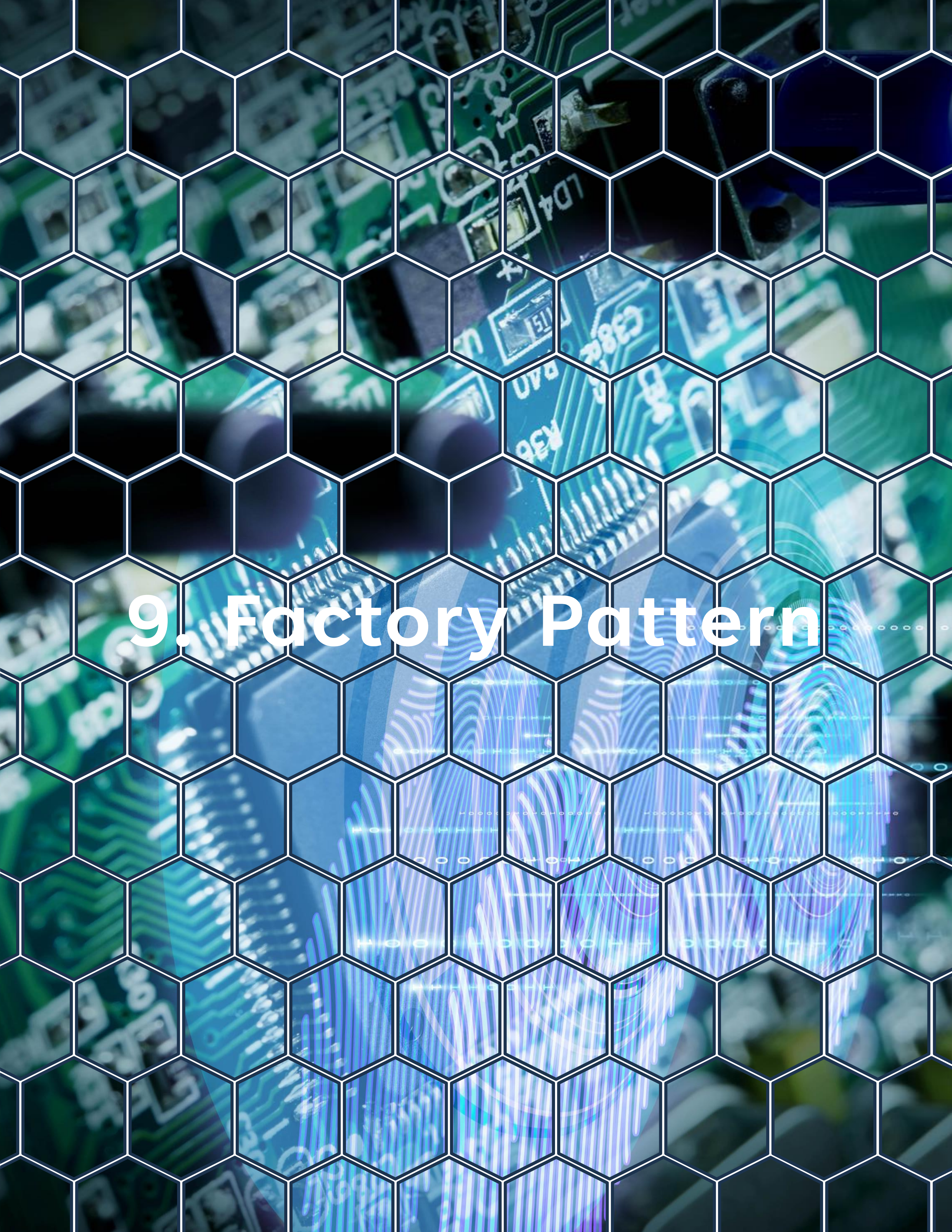
Best Practice Notes

- Define interfaces via function pointers.
- Keep strategy structs static.

8. Strategy Pattern

Example: Power Mode Strategy

```
typedef void (*SleepStrategy)(void);

void sleep_idle(void)    {
    SLPCTRL.CTRLA = SLPCTRL_SMODE_IDLE_gc |
                    SLPCTRL_SEN_bm;
    __sleep();
}
void sleep_standby(void) {
    SLPCTRL.CTRLA = SLPCTRL_SMODE_STDBY_gc |
                    SLPCTRL_SEN_bm;
    __sleep();
}

typedef struct {
    const char* name;
    SleepStrategy sleep_func;
} PowerStrategy;

PowerStrategy modes[] = {
    {"IDLE",    sleep_idle},
    {"STANDBY", sleep_standby}
};

void enter_sleep_mode(PowerStrategy* strategy) {
    strategy->sleep_func();
}
```



9. Factory Pattern

9. Factory Pattern

Purpose

Provides a way to create objects without exposing the instantiation logic.

Use Case in Embedded

Used for abstracting different driver backends.

Best Practice Notes

- Use config constants or init structs to drive instantiation.
- Avoid malloc—use statically allocated buffers.

9. Factory Pattern

Example: Sensor Factory

```c
typedef enum { SENSOR_DHT, SENSOR_SHT } SensorType;

SensorBase* create_sensor(SensorType type) {
    static DHTSensor dht;
    static SHTSensor sht;

    switch (type) {
        case SENSOR_DHT: return &dht.base;
        case SENSOR_SHT: return &sht.base;
        default: return NULL;
    }
}
```

10. Active Object Pattern

10. Active Object Pattern

Purpose

Separates method execution into its own context (thread or task).

Use Case in Embedded

Used with queues in RTOS environments or ISR-to-task communication.

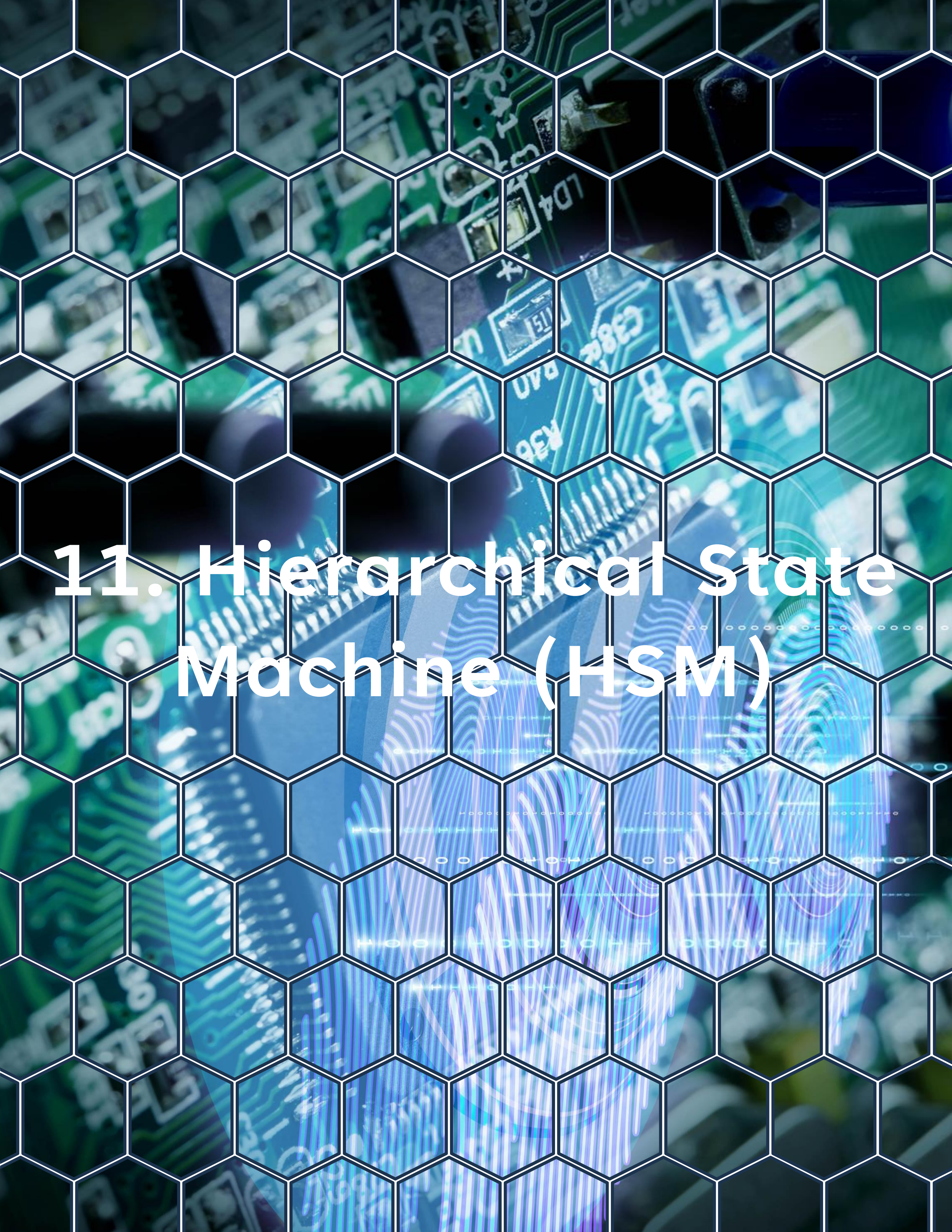
Best Practice Notes

- Use FreeRTOS queues or ring buffers.
- Decouple ISR logic from heavy processing.

10. Active Object Pattern

Example: Logger Task

```c
#define LOG_QUEUE_SIZE 8
QueueHandle_t log_queue;

void logger_task(void* params) {
    char log_entry[64];
    while (1) {
        if (xQueueReceive(log_queue, &log_entry, portMAX_DELAY)) {
            write_to_flash(log_entry);
        }
    }
}

void log_async(const char* msg) {
    xQueueSend(log_queue, msg, 0);
}
```



11. Hierarchical State Machine (HSM)

11. Hierarchical State Machine (HSM)

Purpose

Extends FSMs by supporting state nesting and reuse.

Use Case in Embedded

Used in communication stacks, display logic, or robotic behaviors.

Best Practice Notes

- Model common substates (e.g., Error, Idle).
- Manage transitions cleanly with events.

11. Hierarchical State Machine (HSM)

Example: BT State Tree

```
typedef enum {
    STATE_DISCONNECTED, STATE_IDLE, STATE_STREAMING, STATE_ERROR
} BTState;

typedef enum {
    EVT_CONNECT, EVT_STREAM_START, EVT_STREAM_STOP,
    EVT_DISCONNECT, EVT_ERROR
} BTEvent;

static BTState state = STATE_DISCONNECTED;

void handle_event(BTEvent event) {
    switch (state) {
        case STATE_DISCONNECTED:
            if (event == EVT_CONNECT) state = STATE_IDLE;
            break;

        case STATE_IDLE:
            if (event == EVT_STREAM_START) state = STATE_STREAMING;
            else if (event == EVT_DISCONNECT) state = STATE_DISCONNECTED;
            break;

        case STATE_STREAMING:
            if (event == EVT_STREAM_STOP) state = STATE_IDLE;
            else if (event == EVT_DISCONNECT) state = STATE_DISCONNECTED;
            break;

        case STATE_ERROR:
            // do some error handling
            break;
    }
```

11. Hierarchical State Machine (HSM)

```c
} BTState;

typedef enum {
    EVT_CONNECT, EVT_STREAM_START, EVT_STREAM_STOP,
    EVT_DISCONNECT, EVT_ERROR
} BTEvent;

static BTState state = STATE_DISCONNECTED;

void handle_event(BTEvent event) {
    switch (state) {
        case STATE_DISCONNECTED:
            if (event == EVT_CONNECT) state = STATE_IDLE;
            break;

        case STATE_IDLE:
            if (event == EVT_STREAM_START) state = STATE_STREAMING;
            else if (event == EVT_DISCONNECT) state = STATE_DISCONNECTED;
            break;

        case STATE_STREAMING:
            if (event == EVT_STREAM_STOP) state = STATE_IDLE;
            else if (event == EVT_DISCONNECT) state = STATE_DISCONNECTED;
            break;

        case STATE_ERROR:
            // do some error handling
            break;
    }

    if (event == EVT_ERROR)
        state = STATE_ERROR;
}
```

12. Object Pool Pattern

12. Object Pool Pattern

Purpose

Avoids fragmentation and allocation overhead by reusing pre-allocated objects.

Use Case in Embedded

Used for packet buffers, command structs, or job queues.

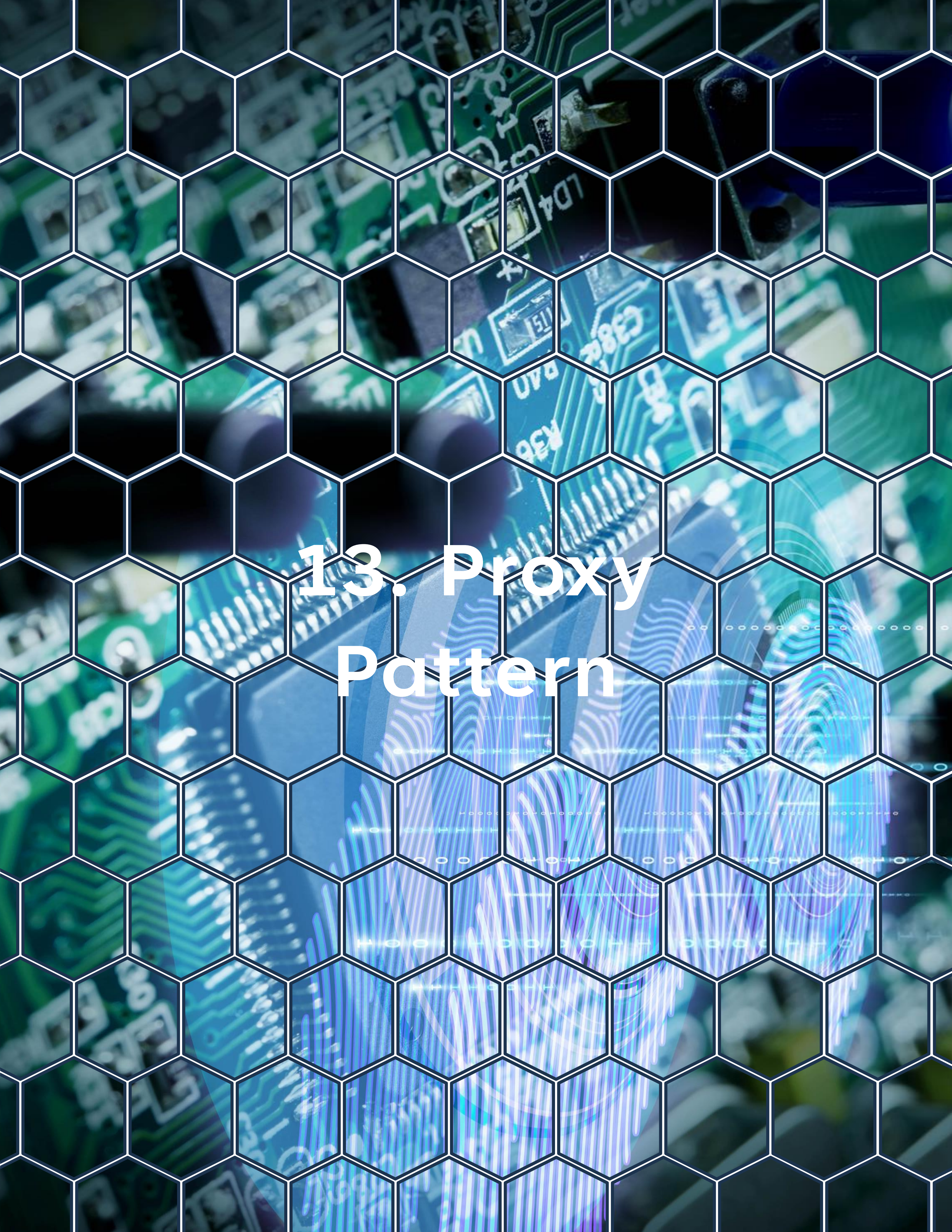
Best Practice Notes

- Use circular buffers or free lists.
- Never use malloc/free in ISR or runtime.

12. Object Pool Pattern

Example: UART Packet Pool

```c
#define POOL_SIZE 8
static UartPacket pool[POOL_SIZE];
static bool used[POOL_SIZE] = {false};

UartPacket* allocate_packet(void) {
    for (int i = 0; i < POOL_SIZE; ++i) {
        if (!used[i]) {
            used[i] = true;
            return &pool[i];
        }
    }
    return NULL;
}

void release_packet(UartPacket* pkt) {
    int index = pkt - pool;
    if (index >= 0 && index < POOL_SIZE)
        used[index] = false;
}
```



13. Proxy Pattern

13. Proxy Pattern

Purpose

Acts as a controlled interface to access complex or sensitive resources.

Use Case in Embedded

Used for EEPROM, Flash, or I2C abstraction with added safety or caching.

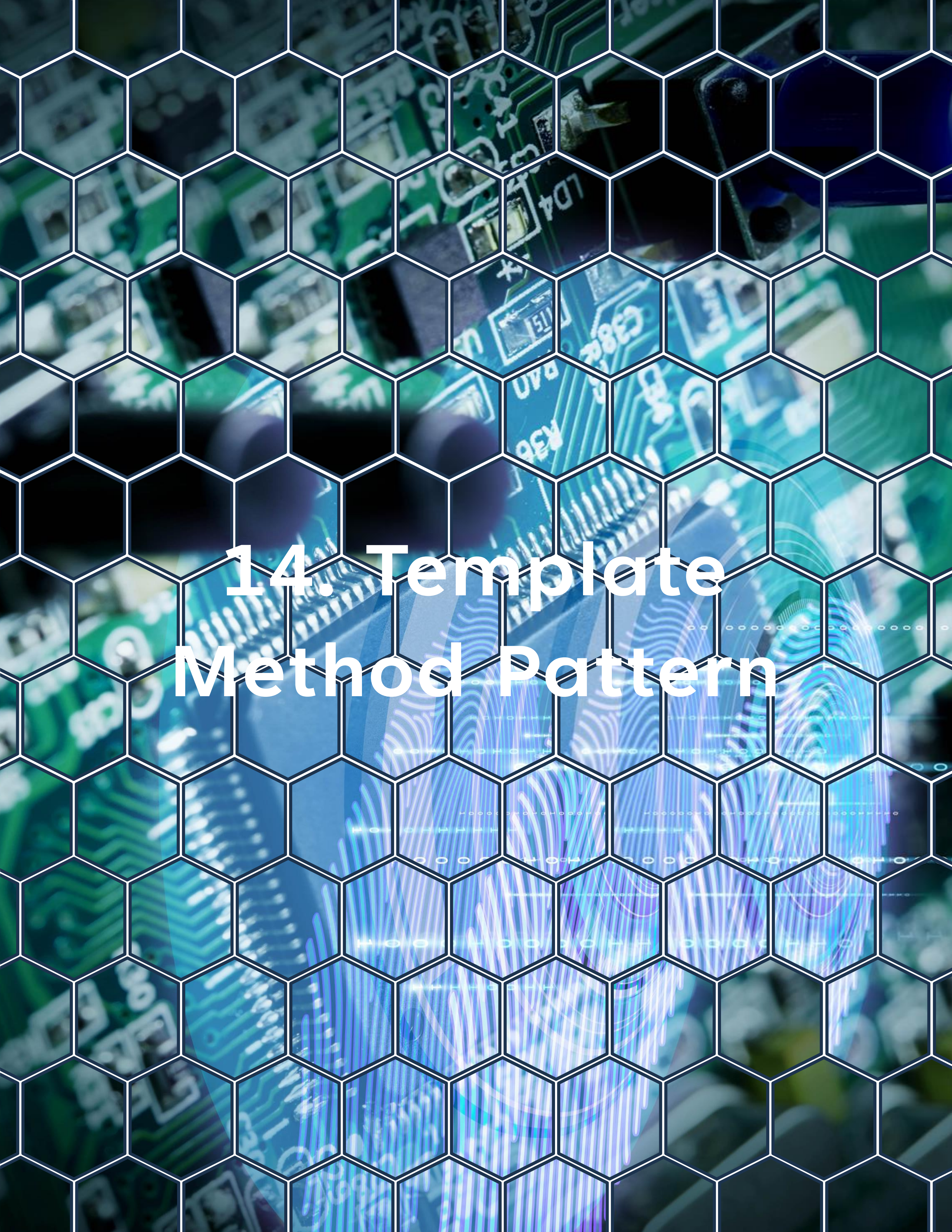
Best Practice Notes

- Use static structs and clear boundaries.
- Implement protection for concurrent or unsafe access.

13. Proxy Pattern

Example: EEPROM Proxy

```
uint8_t eeprom_proxy_read(uint16_t addr) {
    // Adds bounds check and wear leveling
    if (addr >= EEPROM_SIZE) return 0xFF;
    return EEPROM[map_address(addr)];
}
```



14. Template Method Pattern

14. Template Method Pattern

Purpose

Defines the skeleton of an operation, deferring specific steps to subclasses.

Use Case in Embedded

Used to standardize interface behavior across hardware-specific drivers.

Best Practice Notes

- Use function pointers in base structs.
- Keep the "template" flow centralized.

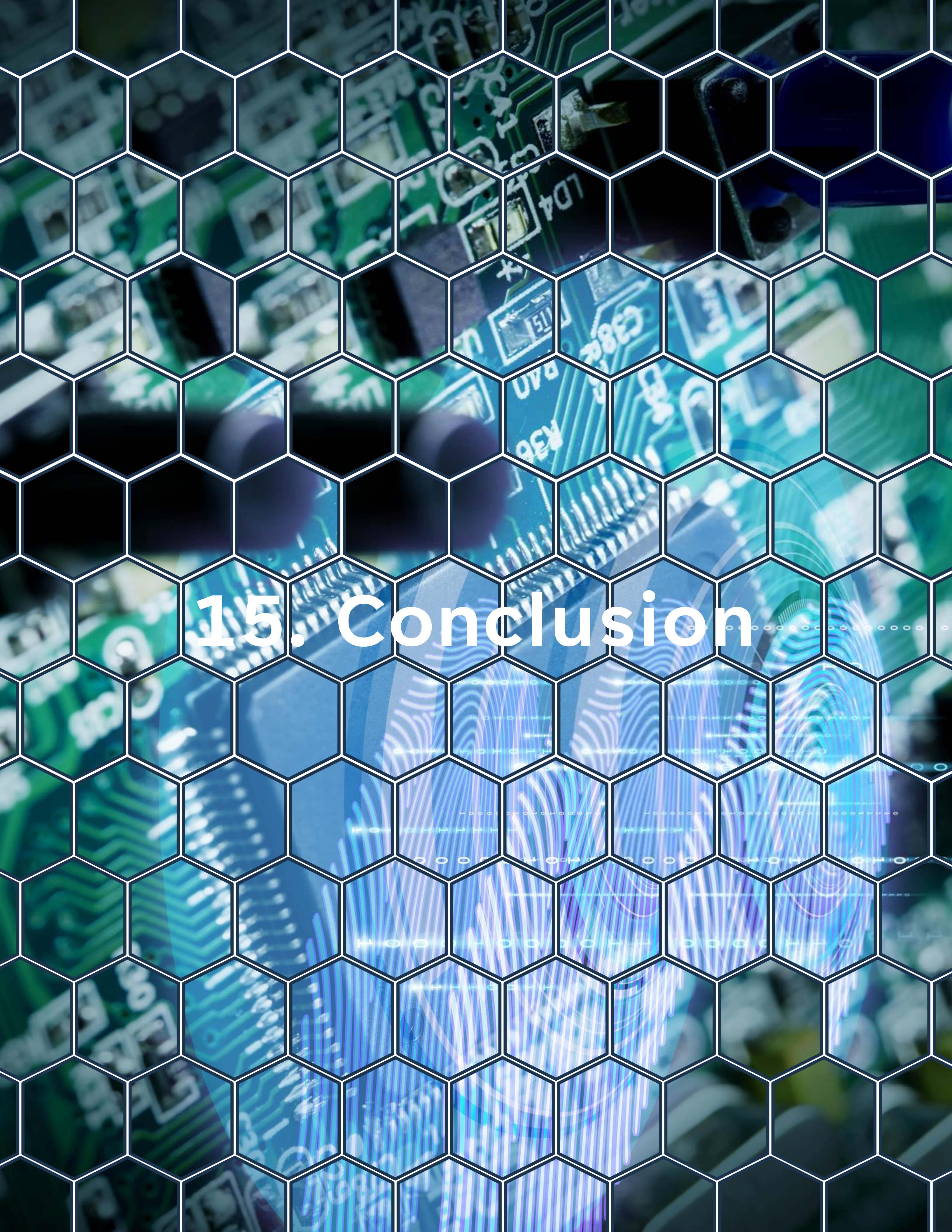
14. Template Method Pattern

Example: Sensor Read Template

```c
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>

// === Sensor Interface ===
typedef struct {
    void    (*start)(void);
    bool    (*ready)(void);
    int16_t (*read)(void);
} SensorInterface;

// === Template Method ===
void read_sensor(SensorInterface* sensor) {
    sensor->start();
    // Blocking wait — replace with timeout logic if needed
    while (!sensor->ready());
    int16_t value = sensor->read();
    // Replace with actual processing
    printf("Sensor Value: %d\n", value);
}

// === DHT22 Implementation ===
void dht22_start(void)      { printf("DHT22: Start\n"); }
bool dht22_ready(void)      { return true; }  // Simulate immediate readiness
int16_t dht22_read(void)    { return 245; }   // Dummy value

SensorInterface DHT22 = {
    .start = dht22_start,
    .ready = dht22_ready,
    .read  = dht22_read
};

// === SHT3x Implementation ===
```

14. Template Method Pattern

```c
    // Blocking wait – replace with timeout logic if needed
    while (!sensor->ready());
    int16_t value = sensor->read();
    // Replace with actual processing
    printf("Sensor Value: %d\n", value);
}

// === DHT22 Implementation ===
void dht22_start(void)      { printf("DHT22: Start\n"); }
bool dht22_ready(void)      { return true; }  // Simulate immediate readiness
int16_t dht22_read(void)    { return 245; }   // Dummy value

SensorInterface DHT22 = {
    .start = dht22_start,
    .ready = dht22_ready,
    .read  = dht22_read
};

// === SHT3x Implementation ===
void sht3x_start(void)      { printf("SHT3x: Start\n"); }
bool sht3x_ready(void)      { return true; }
int16_t sht3x_read(void)    { return 278; }

SensorInterface SHT3X = {
    .start = sht3x_start,
    .ready = sht3x_ready,
    .read  = sht3x_read
};

// === Main Usage ===
int main(void) {
    read_sensor(&DHT22);    // Uses DHT22 sequence
    read_sensor(&SHT3X);    // Uses SHT3x sequence
    return 0;
}
```



15. Conclusion

15. Conclusion

Design patterns offer powerful abstractions and reusable structures for embedded software development, especially under tight memory, power, and performance constraints. When adapted correctly, they provide maintainable and scalable architectures even for bare-metal microcontrollers like the ATtiny1616.

However, using them effectively requires a deep understanding of the system's limitations. Avoid dynamic memory, keep execution non-blocking, and prioritize deterministic behavior. These patterns are not just academic concepts—they're essential tools for delivering reliable embedded firmware in the real world.