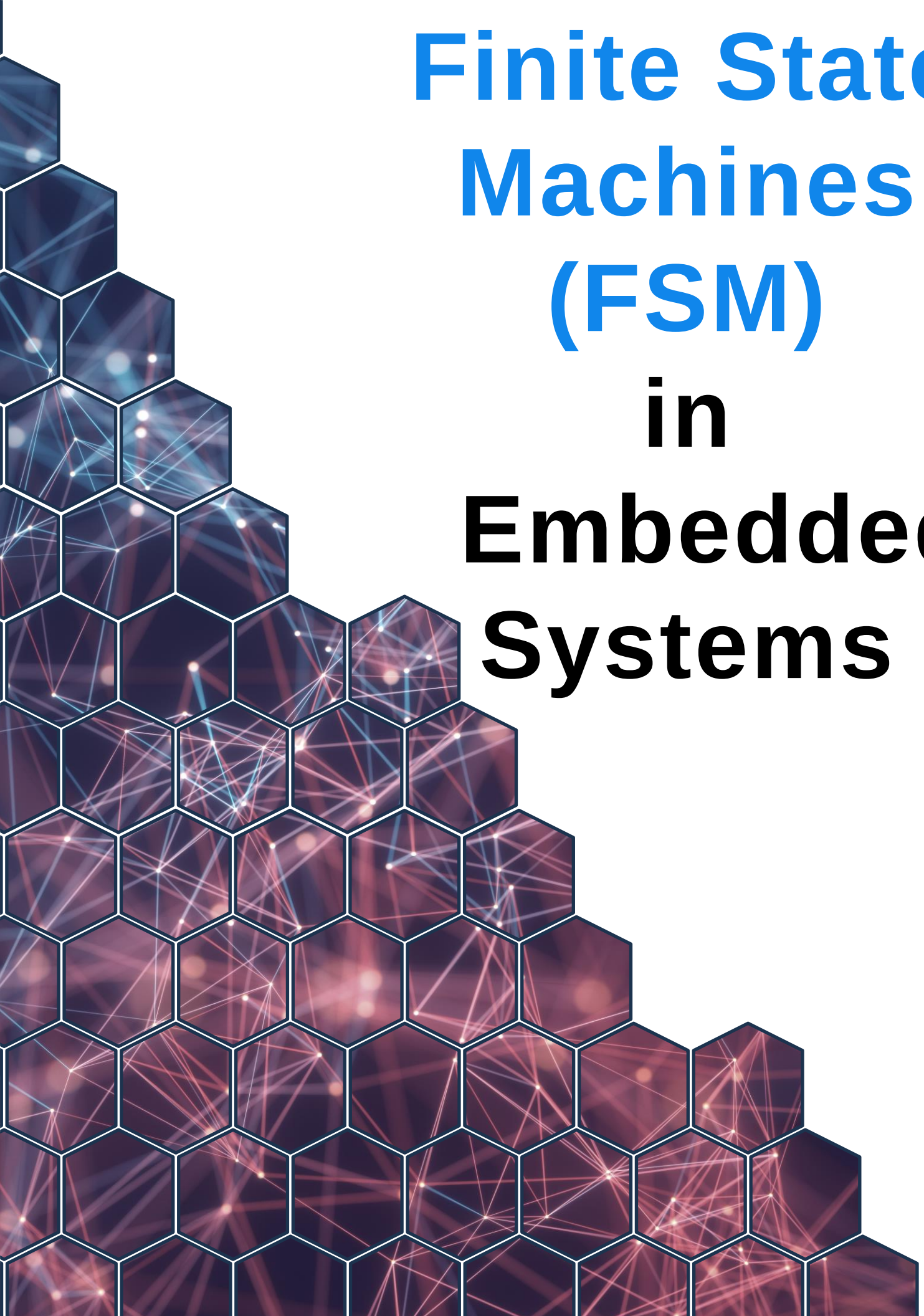
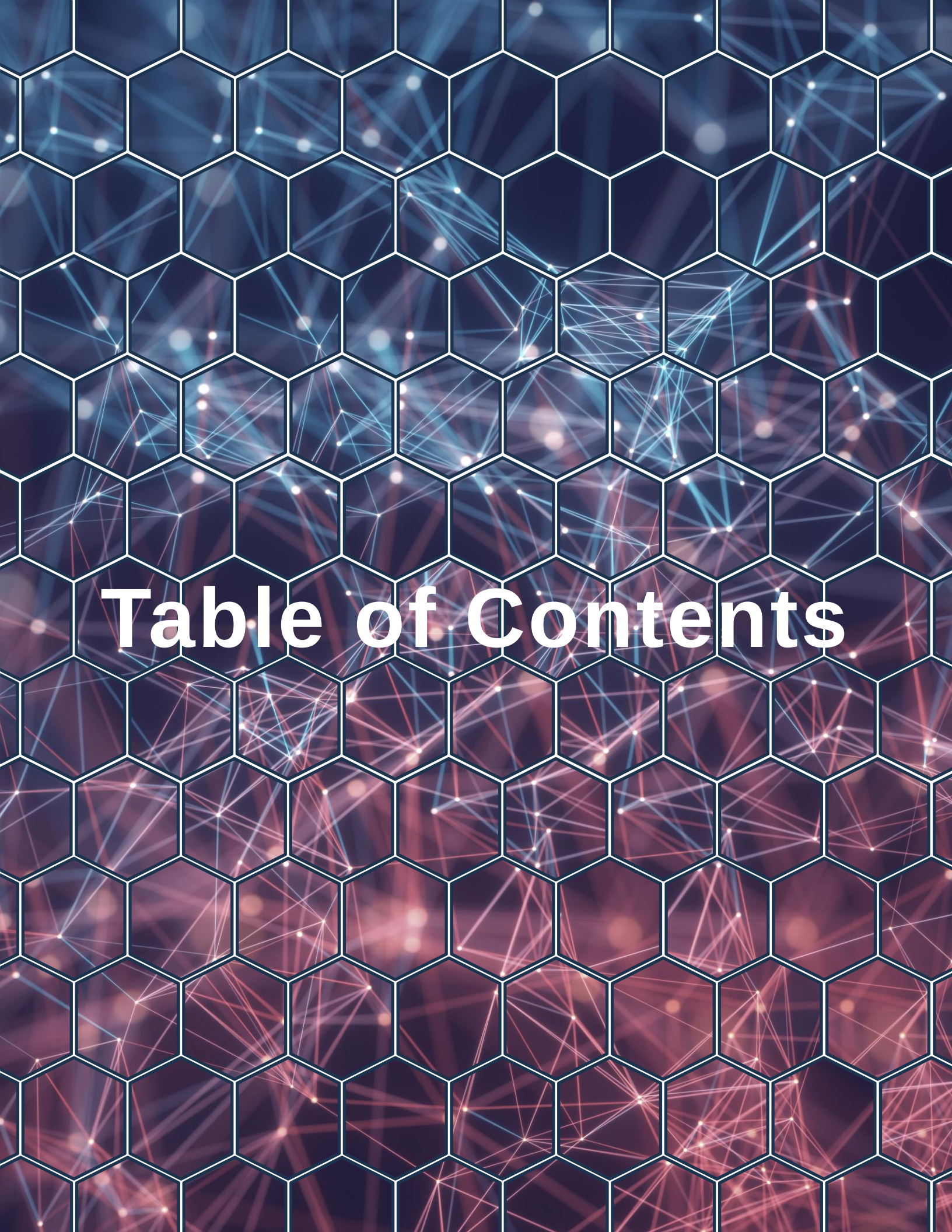
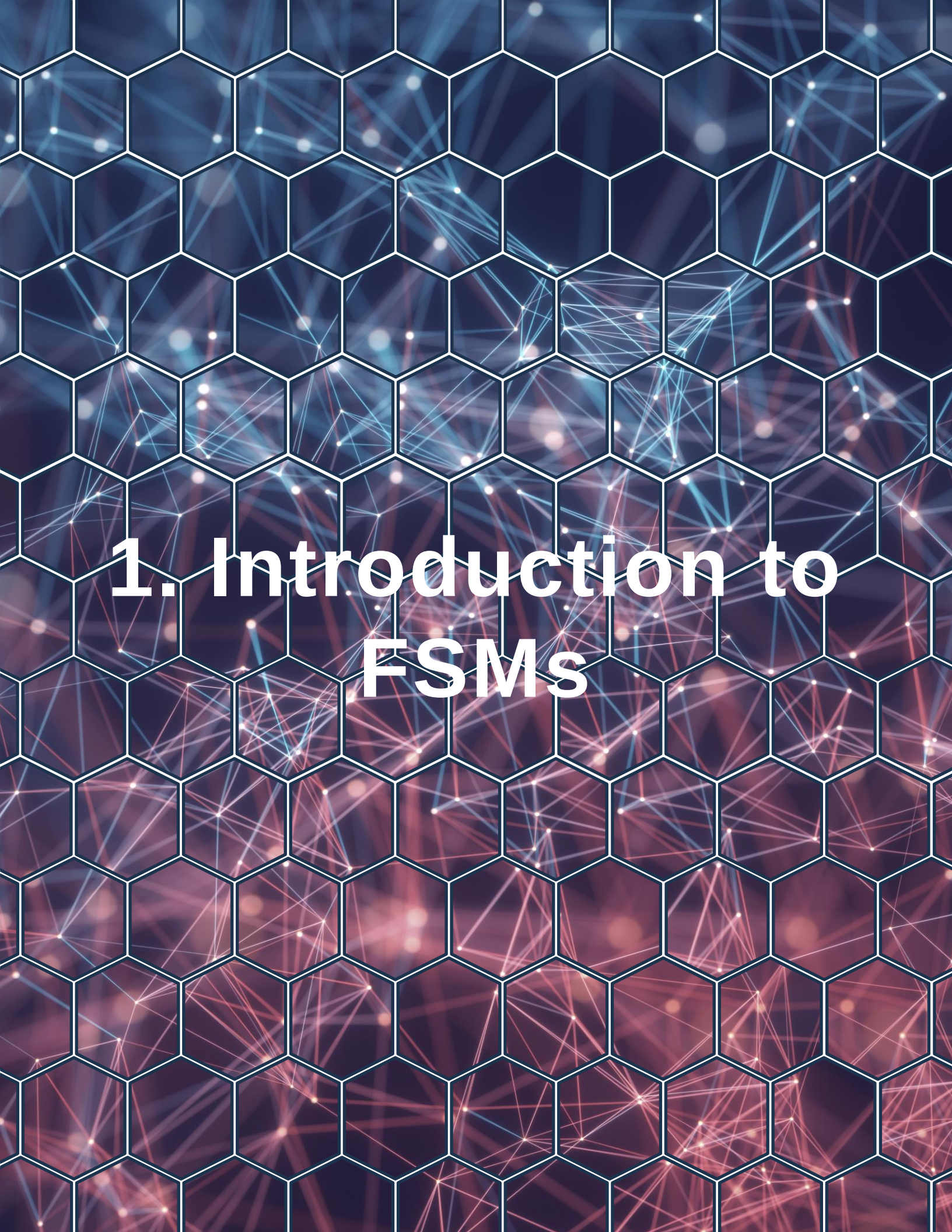


# Finite State Machines (FSM) in Embedded Systems



# Table of Contents

# Table of Contents



# 1. Introduction to FSMs

# 1. Introduction to FSMs

Finite State Machines (FSMs) are mathematical models used to represent systems that transition between a finite number of states based on external events or conditions. In embedded systems, FSMs are widely used for tasks that involve decision-making, control logic, and sequencing actions, making them indispensable in fields like automotive control systems, robotics, and consumer electronics.

**Types of FSMs**

FSMs are primarily categorized into two types:

- **Moore Machines**: Outputs depend only on the current state.
- **Mealy Machines**: Outputs depend on both the current state and external inputs.

SMs simplify complex processes by modeling

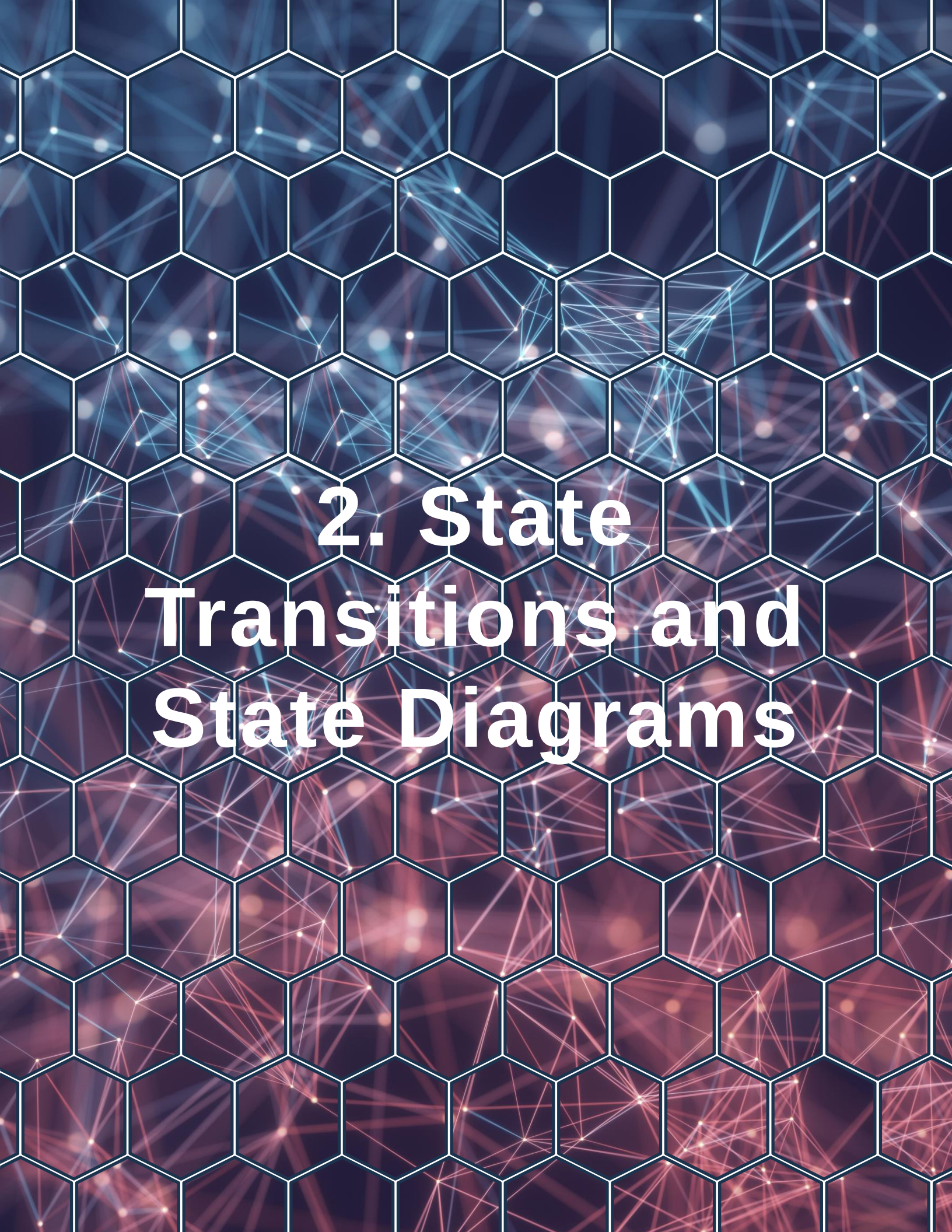
# 1. Introduction to FSMs

## Types of FSMs

FSMs are primarily categorized into two types:

- **Moore Machines**: Outputs depend only on the current state.
- **Mealy Machines**: Outputs depend on both the current state and external inputs.

FSMs simplify complex processes by modeling behavior into states and transitions, providing clarity and ease of debugging.



# 2. State Transitions and State Diagrams

# 2. State Transitions and State Diagrams

FSMs operate based on three main components:

- **States**: Represent distinct conditions (e.g., "Idle", "Running").
- **Transitions**: Define valid changes between states based on events or inputs.
- **Actions**: Activities triggered during state entry, exit, or transitions.

**State Diagram Example**

Consider a traffic light controller with three states: **Red**, **Green**, and **Yellow**. The following diagram outlines its behavior:

Red → Green → Yellow → Red


3. Event-Driven
FSM
Implementation

# 3. Event-Driven FSM Implementation

## Switch-Case Based FSM

FSMs in embedded systems are often implemented using switch-case statements for simplicity.

## Example: Basic FSM in Embedded C

```c
typedef enum {
    STATE_RED,
    STATE_GREEN,
    STATE_YELLOW
} State;

State currentState = STATE_RED;

void fsm_run() {
    switch (currentState) {
        case STATE_RED:
            // Actions for Red
            turn_on_red_light();
            if (timer_expired()) currentState = STATE_GREEN;
            break;

        case STATE_GREEN:
            // Actions for Green
            turn_on_green_light();
            if (timer_expired()) currentState = STATE_YELLOW;
```

# 3. Event-Driven FSM Implementation

## Example: Basic FSM in Embedded C

```c
typedef enum {
    STATE_RED,
    STATE_GREEN,
    STATE_YELLOW
} State;

State currentState = STATE_RED;

void fsm_run() {
    switch (currentState) {
        case STATE_RED:
            // Actions for Red
            turn_on_red_light();
            if (timer_expired()) currentState = STATE_GREEN;
            break;

        case STATE_GREEN:
            // Actions for Green
            turn_on_green_light();
            if (timer_expired()) currentState = STATE_YELLOW;
            break;

        case STATE_YELLOW:
            // Actions for Yellow
            turn_on_yellow_light();
            if (timer_expired()) currentState = STATE_RED;
```

# 3. Event-Driven FSM Implementation

```
8
9  void fsm_run() {
10     switch (currentState) {
11         case STATE_RED:
12             // Actions for Red
13             turn_on_red_light();
14             if (timer_expired()) currentState = STATE_GREEN;
15             break;
16
17         case STATE_GREEN:
18             // Actions for Green
19             turn_on_green_light();
20             if (timer_expired()) currentState = STATE_YELLOW;
21             break;
22
23         case STATE_YELLOW:
24             // Actions for Yellow
25             turn_on_yellow_light();
26             if (timer_expired()) currentState = STATE_RED;
27             break;
28     }
29 }
```

This implementation loops through states based on external triggers (timers).

# 4. Advanced FSM Design Techniques

# 4. Advanced FSM Design Techniques

## 1. Hierarchical FSMs

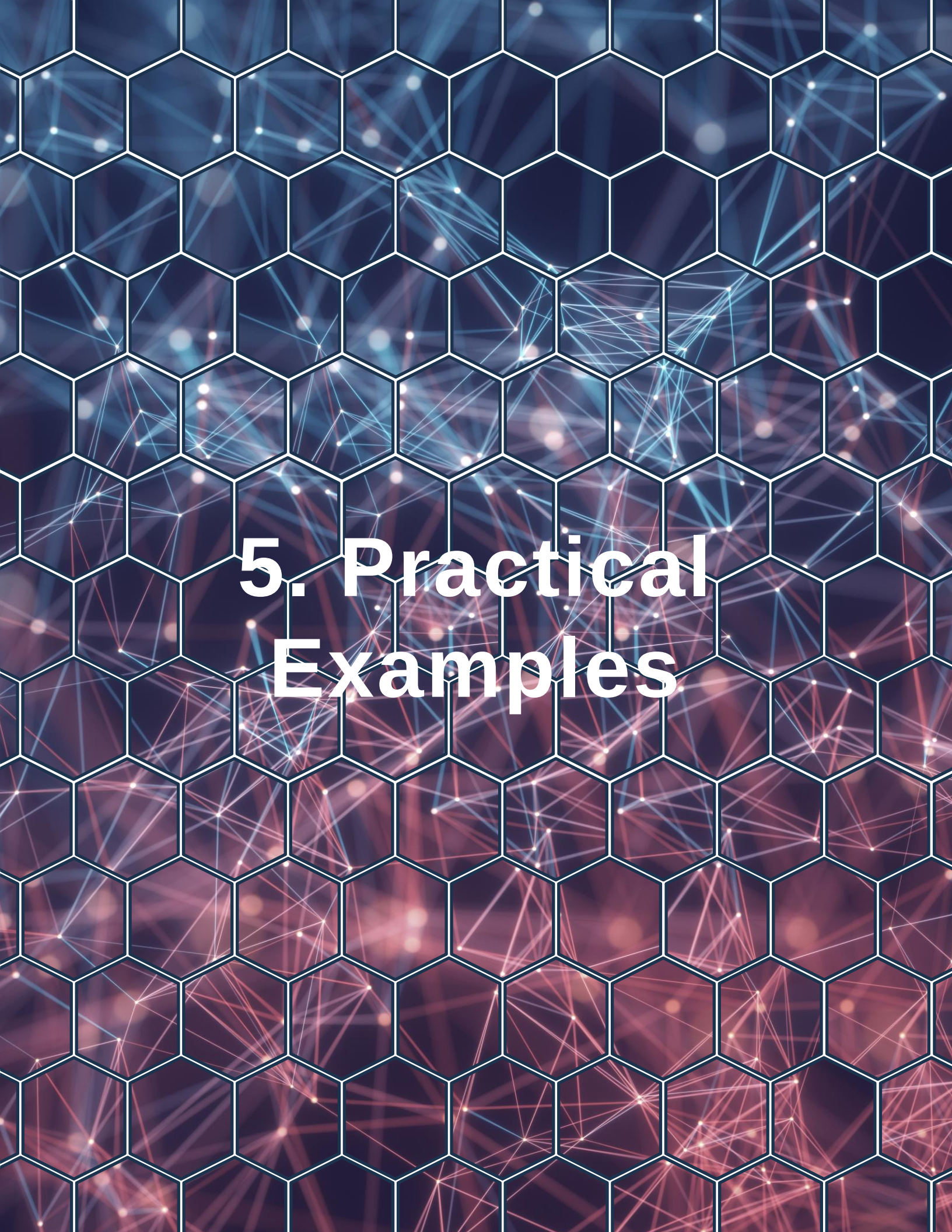
For complex systems, FSMs can be nested to reduce redundancy. Example: An oven controller may have a high-level state for "Cooking" and nested states for "Preheat," "Bake," and "Cool Down.“

## 2. State Encoding Techniques

Efficient encoding methods (binary, gray code) can optimize memory utilization and speed in FSM implementations.

## 3. Parallel State Machines

Used in applications where multiple states operate simultaneously, such as dual-motor controls.


5. Practical
Examples

# 5. Practical Examples

## Keypad Lock System

Detects a valid sequence of inputs before unlocking a door.

```
typedef enum { LOCKED, UNLOCKED, ERROR } LockState;
LockState lockState = LOCKED;

char code[4] = {'1', '2', '3', '4'}; // Predefined code
char input[4];
int index = 0;

void keypad_fsm(char keypress) {
    if (lockState == LOCKED) {
        if (keypress == code[index]) {
            index++;
            if (index == 4) lockState = UNLOCKED; // All digits match
        } else {
            lockState = ERROR; // Wrong input
        }
    }

    if (lockState == ERROR || lockState == UNLOCKED) {
        reset_input();
        index = 0;
    }
}
```

# 5. Practical Examples

## Motor Control System

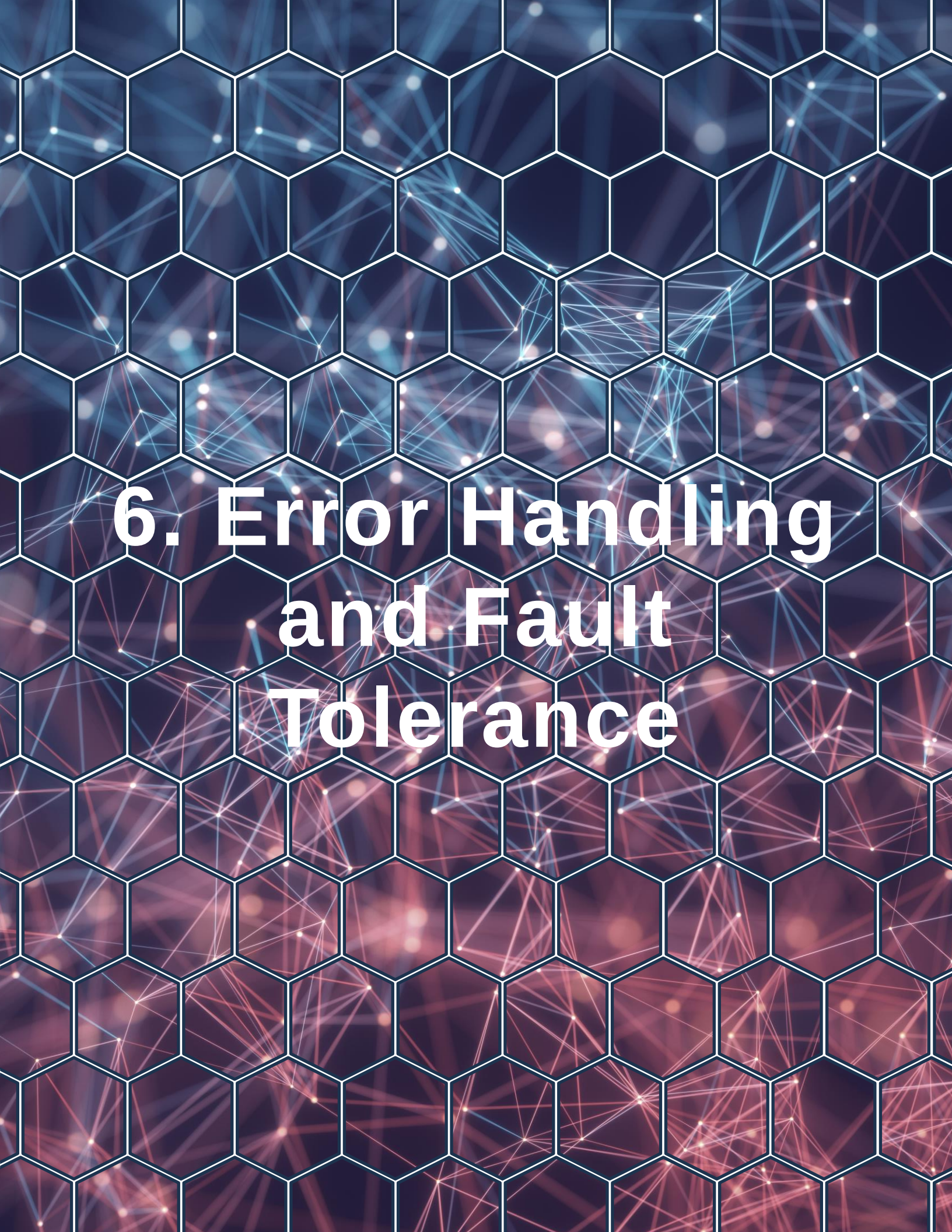
FSM for controlling motor states (Start, Stop, and Idle).

```
typedef enum { IDLE, START, STOP } MotorState;
MotorState motorState = IDLE;

void motor_fsm() {
    switch (motorState) {
        case IDLE:
            if (start_button_pressed()) motorState = START;
            break;

        case START:
            start_motor();
            if (stop_button_pressed()) motorState = STOP;
            break;

        case STOP:
            stop_motor();
            motorState = IDLE;
            break;
    }
}
```


6. Error Handling and Fault Tolerance

# 6. Error Handling and Fault Tolerance

FSMs can handle **unexpected** inputs or failures by defining:

- **Default states**: Returning to a safe state if an error occurs.
- **Timeouts**: Transitioning automatically after a certain period of inactivity.
- **Watchdog timers**: Monitoring and resetting the system during faults.

# 9. Conclusion

# 9. Conclusion

Finite State Machines (FSMs) simplify embedded system design by providing structured approaches to model state transitions and behaviors. They improve reliability, scalability, and maintainability, especially for applications requiring sequential logic. FSMs are widely used in areas like robotics, motor control, and security systems. By using best practices such as modular design, logging, and error handling, FSMs become even more effective in complex systems, ensuring stability and efficiency throughout operation.