State Machine Pattern in Embedded Systems (Function Pointer Approach)

1. Core Concepts of the State Machine Pattern

In junior-level embedded programming, state machines are typically implemented using giant `switch-case` statements. As the system grows, this approach leads to "spaghetti code" that is hard to maintain, difficult to test, and suffers from non-deterministic execution time (the CPU takes longer to reach `case 50` than `case 1`).

The **Function Pointer State Machine Pattern** (often popularized by frameworks like Quantum Leaps / Miro Samek) solves this by treating every state as a distinct function.

For a robust architecture, the following principles apply:

- **State as a Function:** Each state is a dedicated `static` function that takes the state machine object and an event as parameters.
- **O(1) Execution Time:** The core engine simply dereferences the current state's function pointer. There is no `switch-case` lookup, guaranteeing deterministic execution time.
- **State Transitions:** A transition is performed simply by reassigning the function pointer to a new state function.
- **Context Encapsulation:** All variables related to the state machine are kept inside a `struct` (the context), avoiding global variables.

---

2. STM32 Example Implementation

In this example, we will design an Industrial Heater Controller. It has three states: `IDLE`, `HEATING`, and `COOLING`. Instead of a massive switch statement in `main()`, each state has its own cleanly separated function.

2.1. The Context and Types: `Heater.h`

This defines the events, the state function signature, and the object itself.

```c
#ifndef HEATER_H
#define HEATER_H

#include <stdint.h>
#include <stdbool.h>

// 1. Define the Events that can trigger state transitions
typedef enum {
    EV_BTN_PRESS,
    EV_TEMP_REACHED,
    EV_TIMEOUT
} SystemEvent_t;

// 2. Forward declaration of the Heater object
typedef struct Heater_t Heater_t;

// 3. The State Function Pointer Signature
// Every state function must match this signature
typedef void (*StateFunc_t)(Heater_t *const me, SystemEvent_t event);

// 4. The State Machine Context Object
struct Heater_t {
    StateFunc_t currentState; // Pointer to the current state function
    uint16_t temperature;     // Encapsulated variable (Context)
    uint32_t heating_time;
};

// 5. Public API
void Heater_Init(Heater_t *const me);
void Heater_DispatchEvent(Heater_t *const me, SystemEvent_t event);

#endif // HEATER_H
```

2.2. The Concrete States: `Heater.c`

Here, we implement the individual states. Notice how clean and isolated the logic for each state is. If you need to add a "FAULT" state tomorrow, you just add a new function without touching the existing ones (Open/Closed Principle).

```c
#include "Heater.h"
#include "stm32f4xx_hal.h"
```

```
// Forward declarations of state functions
static void State_Idle(Heater_t *const me, SystemEvent_t event);
static void State_Heating(Heater_t *const me, SystemEvent_t event);
static void State_Cooling(Heater_t *const me, SystemEvent_t event);

// --- STATE IMPLEMENTATIONS ---

static void State_Idle(Heater_t *const me, SystemEvent_t event) {
    switch (event) {
        case EV_BTN_PRESS:
            // Action: Turn on the heater hardware
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_SET);
            me->heating_time = 0;

            // Transition to HEATING
            me->currentState = State_Heating;
            break;

        default:
            // Ignore other events in this state
            break;
    }
}

static void State_Heating(Heater_t *const me, SystemEvent_t event) {
    switch (event) {
        case EV_TEMP_REACHED:
        case EV_TIMEOUT: // Fallthrough: Both events trigger the same tra
            // Action: Turn off the heater hardware, turn on cooling fan
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_RESET);
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_SET); // Fan ON

            // Transition to COOLING
            me->currentState = State_Cooling;
            break;

        default:
            break;
    }
}

static void State_Cooling(Heater_t *const me, SystemEvent_t event) {
    switch (event) {
        case EV_TIMEOUT:
```

```c
            // Action: Turn off cooling fan
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_RESET); // Fan OFF

            // Transition back to IDLE
            me->currentState = State_Idle;
            break;

        default:
            break;
    }
}

// --- PUBLIC API ---

void Heater_Init(Heater_t *const me) {
    me->temperature = 25;
    me->heating_time = 0;

    // Set the initial default state
    me->currentState = State_Idle;
}

// The core engine: O(1) Dispatcher
void Heater_DispatchEvent(Heater_t *const me, SystemEvent_t event) {
    // Simply call the function pointer.
    // No massive switch-case lookup required!
    if (me->currentState != NULL) {
        me->currentState(me, event);
    }
}
```

2.3. System Integration: `main.c`

The application layer simply feeds events into the dispatcher. The dispatcher handles the rest.

```c
#include "stm32f4xx_hal.h"
#include "Heater.h"

Heater_t myHeater;

int main(void) {
    HAL_Init();
    // SystemClock_Config() and MX_GPIO_Init() assumed here.
    // Initialize the state machine
    Heater_Init(&myHeater);

    while (1) {
        // Example: Polling a button (in a real system, this might be eve
        if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET) {
            HAL_Delay(50); // Debounce

            // Feed the event into the state machine
            Heater_DispatchEvent(&myHeater, EV_BTN_PRESS);
        }

        // Example: Simulated ADC Temperature Check
        myHeater.temperature = Get_ADC_Temperature();
        if (myHeater.temperature > 100) {
            Heater_DispatchEvent(&myHeater, EV_TEMP_REACHED);
        }

        HAL_Delay(10);
    }
}
```

---

3. Architectural Warnings and Best Practices

- **Entry and Exit Actions:** The basic implementation above executes actions *during* the transition. In highly complex, safety-critical systems (like ISO 26262 ECU firmware), it is often better to implement "Entry" and "Exit" actions. This ensures that a state initializes itself cleanly regardless of where it was transitioned from.
- **State Explosion:** If your system has more than 15-20 states, a flat state machine becomes difficult to manage. At that point, you should upgrade to a **Hierarchical State Machine (HSM)** where states can have parent/child relationships (e.g., UML statecharts).
- **Interrupt Safety:** Do not call `Heater_DispatchEvent` directly from an ISR (Interrupt Service Routine) if it is also called from `main()`. This will cause Race Conditions. Instead, have the ISR push the `SystemEvent_t` into an RTOS Queue or a Ring Buffer, and let the main loop dequeue and dispatch it.