ARM-Embedded-Path

LED: Polling vs State-Machine

Pavel Pys

October 29, 2025

Core Concept: Polling vs. State Machine

Why our code becomes "deaf" and how to fix it

- **Goal:** Do two things simultaneously.
  - Run a light pattern (`MODE_CHRISTMAS`)
  - *WHILE* reacting to a button press.
- **Problem:** Our `Delay_ms()` function is pure poison. It "blocks" the entire CPU.

Agenda

- **Concept 1: Simple Polling (Project 5)**
  - Analogy: The child in the back seat
- **The Trap: Polling + Blocking Delays**
  - Analogy: The cook staring at water
  - (Your problem from Project 5b)
- **Concept 2: Non-Blocking State Machine (Project 6/7)**
  - Analogy: The professional cook with 5 timers
- **The 3 Pillars** of a State Machine (static, timer, switch)
- **Comparison** & Conclusion

Concept 1: Simple Polling ("Asking")

Polling = The CPU actively queries the status.

Analogy: The child in the back seat.

- The CPU (driver) turns around 1000x per second asking: "Button pressed?" ... "Button pressed?" ... "Button pressed?" ...
- 99.9% of the time the answer is "No".
- The driver is 100% busy asking.

Evaluation (for simple things):

- Works great!
- The loop is extremely fast, reaction is immediate.
- CPU is 100% busy, but the system feels reactive.

Concept 1: The "Good" Polling Code (Project 5)

The "good" polling loop (Project 5):

```
while (1) {
    // 1. Query (Poll)
    if ((GPIOB->IDR & GPIO_IDR_IDR0) == 0) {
        // 2. Fast action
        LED_On(0);
    } else {
        // 3. Fast action
        LED_Off(0);
    }
    // Loop repeats immediately (in microseconds)
}
```

Analysis:

- Actions (`LED_On`/`LED_Off`) are **extremely fast**.
- They take only nanoseconds.
- Loop never blocks $\rightarrow$ System is reactive.

The Trap: Polling + Blocking Delays

What happens when loop actions are "slow"? (Your code suggestion)

Analogy: The cook staring at water.

- The cook (CPU) checks if button is pressed. (No)
- Starts `MODE_CHRISTMAS`.
- First instruction: `Delay_ms(300)`.
- **Cook now stares at clock for 300ms.**
- Is **deaf, blind and blocked**.
- You (the button) get pressed. **Cook doesn't notice.**
- Only when *entire* mode (5 seconds) is finished, does he check the button again.

The Trap: The "Broken" Polling Code

The "broken" polling loop:

```
while (1) {
    // This query happens only every 5 seconds!
    if ((GPIOB->IDR & ... ) == 0) {
        // Blocks 1 second!
        LED_Mode(MODE_ALARM_FAST, 300);
    } else {
        // Blocks 5 seconds!
        LED_Mode(MODE_CHRISTMAS, 300);
    }
}
```

Problem: Worst-case reaction time to button press is **5 seconds**. System is "dead".

Concept 2: Non-Blocking State Machine

Goal: The `while(1)` loop must *never* block. Every iteration must finish in microseconds.

Analogy: The professional cook (multitasking).

- The cook (CPU) starts the light pattern.
- Turns on **only the 1st LED**.
- Sets a **timer** for 300ms.
- **Immediately leaves** (`return`).
- The `while(1)` loop (the "boss") continues.
- The boss checks 1000x per second:
    - 1. "Was button pressed?" (Reacts immediately!)
    - 2. "Did the light pattern timer ring?"
- After 300ms timer rings. Boss calls the pattern function, which **switches only the 2nd LED**, resets the timer and **returns immediately**.

The 3 Pillars of a "Non-Blocking" SM

How does the function "remember" where it was?

1. "Progress" (State)

- A variable storing the next step.
- **`static int state = 0;`**
- `static` is key! Variable "survives" function call.

2. "Memory" (Timestamp)

- When did we set the timer?
- **`static uint32_t last_tick = 0;`**

3. "Timer" (Timer Check)

- The "Non-Blocking Delay".
- **`if ((g_msTicks - last_tick) < delay) { return; }`**
- Most important line! "If time not up, goodbye!"

"Non-Blocking" Function (Part 1: Framework)

```
void LED_Mode_Christmas_Update(void) {
    // Pillars 1 & 2: "Memory"
    static uint32_t last_tick = 0;
    static int state = 0;
    const uint32_t delay = 300;

    // Pillar 3: "Timer" (Timer Check)
    if ((g_msTicks - last_tick) < delay) {
        return; // IMMEDIATE EXIT! (Non-Blocking)
    }

    // TIME'S UP! (Timer rang)
    last_tick = g_msTicks; // Reset timer

    // Execute logic & remember progress (next slide)
    // ...
}
```

"Non-Blocking" Function (Part 2: Logic)

```
// (Continuation of \texttt{LED_Mode_Christmas_Update})
switch (state) {
    case 0: // Step 0: LED 1 ON
    LED_PIN_1_PORT->BSRR = LED_PIN_1_BIT_S;
    break;
    case 1: // Step 1: LED 2 ON
    LED_PIN_2_PORT->BSRR = LED_PIN_2_BIT_S;
    break;
    // ... (etc. for all 8 steps)
    case 7: // Step 7: Pause (all OFF)
    break;
}

// Remember progress (State)
state++;
if (state > 7) {
    state = 0; // Start over
}
```

The "Boss" (Project 7: Super-State-Machine)

The `main`-loop becomes the "boss" (dispatcher), deciding which "worker" (mode) is active.

- We need a "global" variable storing the current state of the "boss".
- `App_Mode_t` is our `enum` from before (`MODE_CHRISTMAS`, `MODE_ALARM`).

The "Brain" Variable:

```
// Global variable storing "boss status"
static App_Mode_t g_current_mode = MODE_CHRISTMAS;
```

Boss Tasks:

1. **Always:** Check button for `g_current_mode` changes ("HR department").
2. **Always:** Call the *currently active* worker ("delegate work").

The "Boss" (`main` Code)

```
while (1) {
    // Boss Task 1: Check button (IMMEDIATE reaction)
    Button_Check_For_Mode_Change();

    // Boss Task 2: Call active worker
    switch (g_current_mode) {
        case MODE_CHRISTMAS:
        LED_Mode_Christmas_Update();
        break;
        case MODE_ALARM:
        LED_Mode_Alarm_Update();
        break;
    }
    // Loop repeats immediately (in microseconds)
}
```

Comparison: Blocking vs. Non-Blocking (Part 1)

Blocking (with `Delay_ms`)

- **Analogy:** Cook stares at egg timer for 10 minutes.
- **CPU Load:** 100% (heats up).
- **Reactivity: Catastrophic**. Reaction (e.g., to button) takes seconds.
- **Multitasking: Impossible**. Only one thing can happen.
- **Complexity:** Very simple to write (linear).

Comparison: Blocking vs. Non-Blocking (Part 2)

Non-Blocking (State Machine)

- **Analogy:** Professional cook with 5 timers reacting to guests between tasks.
- **CPU Load:** $< 1\%$ (can use `__WFI` and sleep).
- **Reactivity: Immediate**. Reaction in microseconds.
- **Multitasking: Built-in**. The "boss" can manage 10 "workers" (lights, buttons, UART, ...).
- **Complexity: Very high** initially (must learn to "think in timers").

Conclusion

- **Blocking Delays (`Delay_ms`)** are the **archenemy** of every reactive firmware.
- **State Machines** (Non-Blocking) are the **only** way to control multiple things "simultaneously" and reactively.