ARM-Embedded-Path

Multiple LEDs

Pavel Pys

October 29, 2025

Analysis: The `LED_Init()` Function

Goal: Configure 4 different LEDs (PC13, PA0, PA12, PB9) as digital outputs (Push-Pull, 2MHz) for CMSIS code.

Our Function (The "Patient"):

- `void LED_Init(void)`

The 3 Initialization Steps:

- **Step 1:** Enable clock (RCC)
- **Step 2:** Understand GPIO registers (CRL vs. CRH)
- **Step 3:** Configure pins robustly (Read-Modify-Write)

Step 1: Enable Clock (RCC)

Problem: Every peripheral (GPIO, UART, ...) is "off" after reset to save power. It has no clock.

Solution: We must explicitly enable the clock for GPIO ports A, B and C in the **RCC** (Reset and Clock Control) module.

Where? (STM32F1):

- The "fast" GPIO ports are on the **APB2** bus.
- The register is: `RCC->APB2ENR` (APB2 Enable Register)

The Code:

```
void LED_Init(void) {
    // 1. Clock (All 3 ports in one operation)
    RCC->APB2ENR |= (RCC_APB2ENR_IOPAEN | // Port A
    RCC_APB2ENR_IOPBEN | // Port B
    RCC_APB2ENR_IOPCEN);  // Port C
    ...
}
```

Step 2: GPIO Registers (CRL & CRH)

The "STM32F1 Problem": There's no simple "Direction" register.

- Per port (e.g., `GPIOA`) there are two 32-bit configuration registers.
- `CRL` (Config Register **Low**): Controls pins **0 to 7**
- `CRH` (Config Register **High**): Controls pins **8 to 15**

Each pin uses 4 bits:

- `MODE[1:0]`: (Mode) Input, Output 10MHz, 2MHz, 50MHz
- `CNF[1:0]`: (Config) Push-Pull, Open-Drain, Pull-up, ...

Our Targets (Output, 2MHz, Push-Pull):

- `MODE` = `0b10` (Output 2MHz) -> `GPIO_CRx_MODEx_1`
- `CNF` = `0b00` (Push-Pull) -> (Standard, 0)

Step 2: Assigning Our 4 Pins

Due to the CRL/CRH division, our 4 pins end up in 4 different register positions:

- **PA0** (Pin 0):
  - Port A -> `GPIOA`
  - Pin 0-7 -> `CRL`
  - **Target: `GPIOA->CRL`** (Bits 0-3)
- **PB9** (Pin 9):
  - Port B -> `GPIOB`
  - Pin 8-15 -> `CRH`
  - **Target: `GPIOB->CRH`** (Bits 4-7)
- **PA12** (Pin 12):
  - Port A -> `GPIOA`
  - Pin 8-15 -> `CRH`
  - **Target: `GPIOA->CRH`** (Bits 16-19)
- **PC13** (Pin 13):
  - Port C -> `GPIOC`
  - Pin 8-15 -> `CRH`
  - **Target: `GPIOC->CRH`** (Bits 20-23)

Step 3: Robust Configuration (RMW)

Problem: We need to change 4 bits (e.g., for PA12), *without* destroying the other 28 bits (e.g., for PA8, PA9, ...) in the same `CRH` register.

Solution: Read-Modify-Write (RMW)

- **1. Read:** Read the current 32-bit value of the register.
- **2. Modify:** Clear (`&= M̃ASK`) only our 4 bits and set (`|= VALUE`) them anew.
- **3. Write:** Write the modified value back.

Step 3: Robust Configuration (RMW) - Code Example

The "One-Liner" Pattern (Example PA12):

```
// Register = (Register AND (NOT MASK)) OR VALUE;
// 1. MASK: The bits we want to clear (CNF12 and MODE12)
// (GPIO_CRH_CNF12 | GPIO_CRH_MODE12)
// 2. VALUE: The new value (Output 2MHz)
// (GPIO_CRH_MODE12_1)
GPIOA->CRH = (GPIOA->CRH & ~(GPIO_CRH_CNF12 |
GPIO_CRH_MODE12))
| (GPIO_CRH_MODE12_1);
```

Complete `LED_Init` Code (Part 1/2)

```
void LED_Init(void) {
    // 1. Enable clock for all 3 ports
    RCC->APB2ENR |= (RCC_APB2ENR_IOPAEN |
    RCC_APB2ENR_IOPBEN |
    RCC_APB2ENR_IOPCEN);

    // 2. PC13 (in CRH) as Output 2MHz (MODE=10, CNF=00)
    // RMW operation for Pin 13
    GPIOC->CRH = (GPIOC->CRH & ~(GPIO_CRH_MODE13 |
GPIO_CRH_CNF13))
    | (GPIO_CRH_MODE13_1);

    // 3. PA0 (in CRL) as Output 2MHz (MODE=10, CNF=00)
    // RMW operation for Pin 0
    GPIOA->CRL = (GPIOA->CRL & ~(GPIO_CRL_MODE0 |
GPIO_CRL_CNF0))
    | (GPIO_CRL_MODE0_1);
```

Complete `LED_Init` Code (Part 2/2)

```
    // 4. PA12 (in CRH) as Output 2MHz (MODE=10, CNF=00)
    // RMW operation for Pin 12
    GPIOA->CRH = (GPIOA->CRH & ~(GPIO_CRH_MODE12 |
GPIO_CRH_CNF12))
    | (GPIO_CRH_MODE12_1);

    // 5. PB9 (in CRH) as Output 2MHz (MODE=10, CNF=00)
    // RMW operation for Pin 9
    GPIOB->CRH = (GPIOB->CRH & ~(GPIO_CRH_MODE9 |
GPIO_CRH_CNF9))
    | (GPIO_CRH_MODE9_1);
}
```

Analysis: The LED-API (`LED_On` / `LED_Off`)

Goal: Understand how our abstraction functions control the hardware (GPIO pins).

The 2 Core Concepts:

- **The `BSRR` Register:** Why is it better than `ODR`?
- **Active-Low vs. Active-High:** Why is LED 0 (PC13) "inverted"?

Our 4 LEDs:

- `LED 0` → `PC13` (On-Board LED)
- `LED 1` → `PA0` (External)
- `LED 2` → `PA12` (External)
- `LED 3` → `PB9` (External)

The Tool: GPIO Bit Set/Reset Register (`BSRR`)

Problem: What happens if an interrupt (e.g., SysTick) modifies the `ODR` register *exactly when* `main` is also changing it? (Race Condition!)

Solution: The `BSRR` Register

- `BSRR` is **atomic**. Each assignment is uninterruptible.
- It's a "Write-Only" 32-bit register.

How `BSRR` Works

Functionality:

- **Bits 0-15 (Set):** Write 1 to `BS0` → `ODR0` becomes 1 (HIGH).
- **Bits 16-31 (Reset):** Write 1 to `BR0` → `ODR0` becomes 0 (LOW).

CMSIS Macros:

```
// Set pin 0 to HIGH (write to bit 0)
GPIOA->BSRR = GPIO_BSRR_BS0;

// Set pin 0 to LOW (write to bit 16)
GPIOA->BSRR = GPIO_BSRR_BR0;
```

Code Analysis: `void LED_On(number)`

Our function "translates" logical numbers into hardware commands.

```
void LED_On(uint8_t LEDNumber){
    if (LEDNumber == 0) {
        GPIOC->BSRR = GPIO_BSRR_BR13; // Set PC13 LOW
    }
    if (LEDNumber == 1) {
        GPIOA->BSRR = GPIO_BSRR_BS0;  // Set PA0 HIGH
    }
    if (LEDNumber == 2) {
        GPIOA->BSRR = GPIO_BSRR_BS12; // Set PA12 HIGH
    }
    if (LEDNumber == 3) {
        GPIOB->BSRR = GPIO_BSRR_BS9;  // Set PB9 HIGH
    }
}
```

The Secret: Active-High vs. Active-Low

Question: Why does `LED_On(0)` set the pin to **LOW** (`BR13`)?

Answer: The hardware wiring!

- **Active-High (LED 1, 2, 3):**
  - Circuit: `PIN -> Resistor -> LED -> GND`
  - For current to flow (LED on), pin must be **HIGH** (3.3V).
  - `LED_On` → `BSRR = BSx` (Set)
- **Active-Low (LED 0 / PC13):**
  - Circuit: `3.3V -> Resistor -> LED -> PIN`
  - For current to flow (LED on), pin must be **LOW** (GND).
  - `LED_On` → `BSRR = BRx` (Reset)

Code Analysis: `void LED_Off(number)`

The `LED_Off` function logically does the exact opposite.

```
void LED_Off(uint8_t LEDNumber){
    if (LEDNumber == 0) {
        GPIOC->BSRR = GPIO_BSRR_BS13; // Set PC13 HIGH (LED off)
    }
    if (LEDNumber == 1) {
        GPIOA->BSRR = GPIO_BSRR_BR0;  // Set PA0 LOW (LED off)
    }
    if (LEDNumber == 2) {
        GPIOA->BSRR = GPIO_BSRR_BR12; // Set PA12 LOW (LED off)
    }
    if (LEDNumber == 3) {
        GPIOB->BSRR = GPIO_BSRR_BR9;  // Set PB9 LOW (LED off)
    }
}
```

Summary & Best Practices

- **Atomicity:** Always use `BSRR` instead of `ODR` to avoid race conditions between `main` and ISRs.
- **Abstraction:** A "driver" (your API) is good because it hides hardware details (ports, pins, active-low/high) from the main program.
- **Readability:** `LED_On(1)` is much clearer than `GPIOA->BSRR = GPIO_BSRR_BS0;`
- **Code Style (Optional):**
  - A `switch-case` statement would be (slightly) more efficient and more readable than four separate `if` statements.

The Great Debug Hunt: A Case Study

Why our 4-LED blinker kept crashing

- **Goal:** A simple blinker with 4 LEDs and `Delay_ms(1000)`.
- **Problem:** Program crashed, but only sometimes.
- **Insight:** It's (almost) always the hardware.

Patient: STM32F103 "Blue Pill" (CMSIS, Bare-Metal)
Symptoms:

- Debugger always opens `startup_...s` file.
- Blinking is "weird", "shortened" or "asynchronous".

The Mystery: Clues & Contradictions

We faced a series of clues that contradicted each other.

Clue 1: The Time Factor (Main Symptom)

- `LED_Mode(MODE_BLINKYALL, 100);` → **Works!**
- `LED_Mode(MODE_BLINKYALL, 500);` → **CRASH!** (Reset)

Clue 2: The "Heisenberg" Effect (Debugger "lies")

- **Without debugger:** Program crashes (reset loop).
- **With debugger:** Program (e.g., temp sensor) suddenly ran error-free!

Clue 3: Power Source Swap

- Power via **ST-Link (3.3V pin)** → **CRASH!**
- Power via **PC USB port (5V pin)** → **Works!**

Suspect 1: Stack Overflow

Theory: The 1ms `SysTick_Handler` (1000x per second) bombards the small default stack.

- Each interrupt "saves" 8 registers (32 bytes) to stack.
- When `main()` $\rightarrow$ `LED_Mode()` $\rightarrow$ `Delay_ms()` runs, stack is already deep.
- SysTick interrupt causes overflow $\rightarrow$ `HardFault`.

Status: DEBUNKED!

- Problem occurred with **temp sensor project** too, which used a "dumb" `for` loop and **no SysTick**.
- So not a software error from interrupts.

Suspect 2: The Watchdog (IWDG)

Theory: The symptom (`Delay(100)` works, `Delay(500)` crashes) is watchdog 101.

- A **hardware-activated** IWDG (via option bytes) always runs.
- Its timeout is > 100ms but < 500ms (e.g., 250ms).
- `Delay_ms(500)` blocks code $\rightarrow$ watchdog not "fed" $\rightarrow$ it "bites" $\rightarrow$ hardware reset!

Status: DEBUNKED! (Despite perfect symptoms)

- "Wiping" (resetting option bytes) should have disabled IWDG. But error remained (when powered via ST-Link).
- Adding `IWDG->KR = 0xAAAA;` should have fed it. Error remained too.

Resolution: Brown-Out-Reset (BOR)

The real culprit: Power supply! The combination of "Clue 2" and "Clue 3" was key:

1. Every `Delay_ms()` loop (whether `for` or `SysTick`) forces CPU to **100% load**.
2. 100% CPU load $\rightarrow$ maximum power consumption.
3. **ST-Link** (especially clones) provides very **little stable current** via 3.3V pin (e.g., < 50mA).
4. This continuous load (even if small) slowly dropped voltage (`VDD`) over time.
5. `Delay(100)` was short enough for `VDD` to stay stable.
6. `Delay(500)` was long enough for `VDD` to fall below threshold (e.g., 2.7V).
7. **Brown-Out-Reset (BOR)** protection detected undervoltage $\rightarrow$ **Hardware reset!**

Final Proof (The 2 Power Paths)

Why did the debugger "lie"?

- In **debug mode** (live data), ST-Link stabilizes voltage (or CPU enters low-power debug state). **BOR is prevented**.

Why did switching to USB cable solve everything?

- ST-Link (3.3V) *bypasses* the voltage regulator.
- PC USB port (5V) goes *through* the 3.3V regulator on Blue Pill board.
- This regulator is strong enough to power CPU at 100% load.

Remember: ST-Link 3.3V pin is for flashing only, not for operation under load!

What We Learned (The 3 Rules)

Rule 1: It's (almost) always the power supply.

- An "unexplainable" reset loop is 90% watchdog or brown-out reset (BOR).
- Before doubting code for 4 hours, check power supply (GND, VCC, ST-Link vs. USB) for 1 minute.

Rule 2: The debugger isn't an innocent observer.

- A debugger **changes** the system (timing, power, watchdogs).
- If a bug "disappears" when debugger runs, it's almost certainly watchdog or power issue.

Rule 3: Blocking delays are "evil".

- Our `for` loop `Delay_ms()` is a "heater". It maximizes power consumption by keeping CPU at 100%.
- (A `SysTick` delay with `__WFI()` [Wait-For-Interrupt] would put CPU to sleep and use almost no power.)

The Mode Concept (State Machines)

Goal: Clean up our main program (`main`) and cleanly control complex sequences (like blinking, alarms).

The Problem (Without Modes):

- The `while(1)` loop quickly becomes huge and unreadable.
- `if (button1_pressed) { ... }`
- `if (timer > 500) { ... }`
- `if (uart_data == 'A') { ... }`
- Manual `if/else` logic becomes error-prone ("spaghetti code").

The Solution: A "State Machine"

- We define clear "states" (modes) the system can be in.
- (e.g., `MODE_BLINKYALL`, `MODE_ALARM_FAST`)

Step 1: Define Modes with `enum`

We use `typedef enum` (enumeration) to give our modes meaningful names instead of "magic numbers" (0, 1, 2).

Why `enum`?

- **Readable:** `MODE_BLINKYALL` is clearer than `1`.
- **Safe:** Compiler warns us if we forget a mode.
- **Maintainable:** Easy to add new modes (e.g., `MODE_SOS`).

The Code (Global, e.g., in `main.c`):

```c
/* Definition of our states (modes).
* The type "LED_Mode_t" can now be used like "int".
*/
typedef enum {
    MODE_BLINKYALL,  // Internal value = 0
    MODE_ALARM_FAST  // Internal value = 1
} LED_Mode_t;
```

Step 2: The Mode Function (`LED_Mode`)

Central function deciding "What happens in which mode?"

- Takes desired mode (`LED_Mode_t`) as parameter
- Encapsulates all logic for that mode
- `switch` statement acts as perfect "switch operator"

API Function:

```
void LED_Mode(LED_Mode_t mode, uint32_t delay) {
    switch (mode) {
        case MODE_BLINKYALL:
        // Logic for BlinkyAll
        break;
        case MODE_ALARM_FAST:
        // Logic for AlarmFast
        break;
    }
}
```

Step 3: The Clean `main`

Tiny, readable `while(1)` loop - complexity hidden in `LED_Mode`

Final `main`:

```c
int main(void){
    // 1. Initializations
    SysTick_Config(SystemCoreClock / 1000);
    LED_Init();
    LED_Off(0); LED_Off(1);
    LED_Off(2); LED_Off(3);

    // 2. Main loop
    while (1){
        LED_Mode(MODE_BLINKYALL, 1000);
    }
}
```

Summary: Why This Approach?

Advantages of the Mode Concept (State Machine)

- **Readability:** `main` describes "what", not "how".
- **Maintainability:**
  - New mode? $\rightarrow$ Simply add a new `case`.
  - Bug in "Alarm"? $\rightarrow$ Only check `case MODE_ALARM_FAST`.
- **Error Safety:** `break;` enforces clean separation. (Our "fall-through" bug was the best lesson for that!)
- **Extensibility:** This is the foundation for any complex firmware (e.g., USB devices, menu navigation, ...).