# ARM-Embedded-Path

## LED - UART

Pavel Pys

October 27, 2025

# Overview

- Introduction
- SysTick & UART
- Practice

## CMSIS Training: UART & Timing

### Task: The "Morse-Bot"

From UART input to timed LED signal

## Recap: What was CMSIS?

- **Problem:** Each ARM chip has different peripheral addresses (GPIO, Timer...).
- **CMSIS-Core (Cortex-M):** Unified access to core functions (e.g., interrupts, `SysTick`).
- **CMSIS-Device (STM32F103):** Provides header files (`stm32f103x6.h`).

**What we learned (Blinky):**

1. Enable peripheral clock (Port C): `RCC->APB2ENR`
2. Configure GPIO pin (PC13): `GPIOC->CRH`
3. Toggle pin: `GPIOC->ODR` (or `BSRR`)

**Today:** Same principle for UART + Timing.

# From Arduino to CMSIS: `Serial` & `delay`

**Arduino world (Abstraction):**

- `Serial.begin(9600);` *Magic?*
- `char c = Serial.read();` *Blocking? Interrupt?*
- `delay(100);` *How? Active waiting? Timer?*

**CMSIS/Bare-Metal world (Control):**

- We must configure EVERYTHING ourselves:
  1. **Clock** for UART (e.g., 8MHz from HSI).
  2. **GPIO pins** (PA9/PA10) for "Alternate Function" (AF).
  3. **Baud rate** (9600) manually calculated and written to `BRR` (Baud Rate Register).
  4. **UART** itself enabled (Transmit/Receive).
- For `delay()` we use a standard Cortex-M timer: The **SysTick**.

# Architecture 1: Timing with SysTick Timer

- **What is SysTick?** A simple 24-bit "countdown" timer located *directly in the Cortex-M core* (not with peripherals).
- **What for?** Ideal as a "time base" for a system (e.g., for an RTOS or, as here, for a simple `delay` function).
- **CMSIS-Core function:** `SysTick_Config(uint32_t ticks)`

**The idea:**

1. We configure SysTick to trigger an interrupt exactly **1000 times per second** (every 1ms).
2. `SysTick_Config(SystemCoreClock / 1000);`
3. `SystemCoreClock` is a global variable (from `system_stm32f10xx.c`) containing our CPU clock (8MHz HSI).
4. $8,000,000/1000 = 8000$ ticks.
5. SysTick counts down from 8000 to 0, triggers an **interrupt**, and restarts.

# Architecture 2: What is UART (Bare-Metal)?

- **UART:** Universal Asynchronous Receiver/Transmitter.
- **STM32F103:** Has `USART1`, `USART2`, `USART3`. We use `USART1`.

**Step 1: Clock (RCC)**

- `USART1` is on the `APB2` bus.
- `GPIOA` (for pins) is also on the `APB2` bus.
- We need clock for: `USART1`, `GPIOA` and `AFIO` (Alternate Function IO).
- Register: `RCC->APB2ENR`

**Step 2: GPIO (Pins)**

- "Blue Pill" uses `PA9` (TX) and `PA10` (RX) for `USART1`.
- These pins must be configured:
  - `PA9 (TX)`: **Alternate Function, Push-Pull**, 50MHz
  - `PA10 (RX)`: **Input, Floating** (or Pull-up)
- Register: `GPIOA->CRH` (Control Register High, for pins 9 & 10)

# Architecture 2: UART (Cont.)

### Step 3: Baud Rate (BRR)

- Formula (from RM0008 Ref. Manual):
- Baud $= f_{CLK}/(16 \times \text{USARTDIV})$
- $f_{CLK}$ (clock for APB2/USART1) = 8MHz (HSI)
- USARTDIV $= 8,000,000/(16 \times 9600) \approx 52.083$
- Mantissa = 52. Fraction $= 0.083 \times 16 \approx 1$.
- Register: `USART1->BRR = (52 << 4) | 1;`

### Step 4: Activation (CR1)

- Enable UART (`UE`), enable transmit (`TE`), enable receive (`RE`).
- Register: `USART1->CR1`

# Practice: SysTick (Part 1: Globals & ISR)

We need a global variable that increments in the interrupt.

```
        // IMPORTANT: volatile! Tells the compiler:
        // "This variable can change at any time."
        volatile uint32_t g_msTicks = 0;
```

**The Interrupt Service Routine (ISR) in `main.c`:**

```
        // This is the "callback" from the SysTick
    interrupt
        // The name is predefined in the startup code (
    vector table).
        void SysTick_Handler(void) {
            g_msTicks++; // Every millisecond +1
        }
```

# Practice: SysTick (Part 2: `Delay_ms()`)

**Our `delay` function:**

```
void Delay_ms(uint32_t ms) {
    uint32_t start = g_msTicks;

    // Wait until target difference is reached
    while ((g_msTicks - start) < ms) {
        // Active waiting (Polling)
        // Better (energy saving): __WFI(); (Wait
For Interrupt)
    }
}
```

# Practice: SysTick (Part 2: `Delay_ms()`)

**Initialization in `main()`:**

```
    // SystemCoreClock is (hopefully) 8000000
    SysTick_Config(SystemCoreClock / 1000); // 1ms
Tick
```

# Practice: UART Configuration Part 1 (Clock & GPIO)

**Excerpt for `UART1_Init()`:**

```
    // 1. Enable clocks (RCC)
    // RM0008, Section 7.3.7 (APB2 peripheral clock enable register)
    RCC->APB2ENR |= RCC_APB2ENR_USART1EN | // UART1 Clock
    RCC_APB2ENR_IOPAEN   | // GPIOA Clock
    RCC_APB2ENR_AFIOEN;  // Alternate Function Clock
```

# Practice: UART Configuration Part 1 (Clock & GPIO)

```
    // 2. GPIOs (PA9 TX, PA10 RX)
    // RM0008, Section 9.2.2 (GPIO port configuration
 register high)

    // PA9 (TX): AF Push-Pull, 50MHz (Mode: 11, CNF:
10 -> 1011)
    GPIOA->CRH &= ~(0xF << 4); // Clear bits for pin
9
    GPIOA->CRH |=  (0xB << 4); // Set 1011

    // PA10 (RX): Input Floating (Mode: 00, CNF: 01
-> 0100)
    GPIOA->CRH &= ~(0xF << 8); // Clear bits for pin
10
    GPIOA->CRH |=  (0x4 << 8); // Set 0100
```

# Practice: UART Configuration Part 2 (Baud Rate & Activation)

**Excerpt for `UART1_Init()` (Cont.):**

```
// 3. Baud rate (BRR) - RM0008, Section 27.3.4
// 8MHz HSI / (16 * 9600) = 52.083
// Mantissa = 52 (0x34), Fraction = 1 (0x1)
USART1->BRR = (0x34 << 4) | 0x01;
```

```
// 4. Enable UART (CR1) - RM0008, Section 27.3.7
USART1->CR1 |= USART_CR1_UE | // Enable UART
USART_CR1_TE | // Enable Transmitter
USART_CR1_RE;  // Enable Receiver
```

# Practice: UART Polling (Part 1: Flags)

**Polling:** We actively check the status bits (flags).

**Flags in `USART1->SR` (Status Register):**

- `USART_SR_TXE` (Transmit Data Register Empty): *"Transmit buffer is empty, you may send the next byte."*
- `USART_SR_RXNE` (Read Data Register Not Empty): *"Attention! A new byte is in the receive buffer!"*

# Practice: UART Polling (Part 2: Functions)

**Functions (e.g., in `main.c` or `uart.c`):**

```c
    // Send a single character (blocking)
    void UART1_SendChar(char c) {
        // Wait until transmit register is empty (TXE
 flag)
        while (!(USART1->SR & USART_SR_TXE));
        USART1->DR = c; // Write data to Data
Register
    }
```

```c
    // Receive a single character (blocking)
    char UART1_GetChar(void) {
        // Wait until data is received (RXNE flag)
        while (!(USART1->SR & USART_SR_RXNE));
        return (char)(USART1->DR & 0xFF); // Read
data and return it
    }
```

# Exercise: "Morse-Bot" (Part 1: Rules)

**Goal:** Combine all parts.

**Base Timing:**

- `DIT_MS = 100` (e.g., 100 milliseconds)

**Morse Rules:**

- **Dit (Dot):** LED ON (1 * `DIT_MS`), LED OFF (1 * `DIT_MS`)
- **Dah (Dash):** LED ON (3 * `DIT_MS`), LED OFF (1 * `DIT_MS`)
- **Pause (Letters):** (3 * `DIT_MS`)
- **Pause (Word/Space):** (7 * `DIT_MS`)

# Exercise: "Morse-Bot" (Part 2: Tasks)

**Tasks:**

1. Create GPIO init for PC13 (as in Exercise 1).
2. Create SysTick init and `Delay_ms` function.
3. Create `UART1_Init`, `SendChar`, `GetChar` functions.
4. Write functions: `morse_dit()` and `morse_dah()` (use `Delay_ms`).
5. Write a function `morse_char(char c)` using a `switch` statement (e.g., for 'A', 'B', 'C', 'S', 'O').
6. **Main Loop:** Read a character with `UART1_GetChar()`, send it back via `UART1_SendChar()` (echo!) and call `morse_char()`.

# Recap: What we learned

1. `SysTick` is the CMSIS standard for timing bases (e.g., `delay`).
2. `volatile` is essential for shared variables between ISR and Main Loop.
3. Peripheral configuration (UART) always follows the pattern:
   - **Clock (RCC)** $\rightarrow$ **Pins (GPIO)** $\rightarrow$ **Peripheral Config**
4. Register programming requires constant reference to the **Reference Manual (RM0008)**.

# Outlook: Problems & Next Steps

**Problem with current solution:**

- `UART1_GetChar()` is **blocking**. The CPU only waits and cannot do anything else (e.g., Morse).
- `Delay_ms()` is **blocking**. The CPU only waits and cannot receive UART characters during that time.

**Outlook (Part 3):**

- Efficient reception with **UART Interrupts** (The `RXNE` flag triggers an ISR).
- We fill a buffer (ring buffer) in the background.
- We build a "Non-Blocking" Morse machine (State Machine).