ARM-Embedded-Path

LED - UART

Pavel Pys

October 29, 2025

Overview

- Introduction & Motivation
- Architecture & Basics
- Excursus: Register Magic
- Practice & Exercise
- Debugging & Outlook
- Architecture (Memory)

CMSIS Training: UART & Timing

Task: The "Morse-Bot"

From UART Input to a Timed LED Signal

Recap: What is CMSIS?

- **Problem:** Every ARM chip has different addresses for peripherals (GPIO, Timer...).
- **CMSIS-Core (Cortex-M):** Uniform access to core functions (e.g., Interrupts, `SysTick`).
- **CMSIS-Device (STM32F103):** Provides header files (`stm32f103x6.h`).

What was learned (Blinky):

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

Motivation: The 1kB "Blinky"

The Observation (Comparison):

Arduino (e.g., ESP8266)

- "Blinky" (with `delay()`)
- **~250+ KByte Flash**
- Simple: `digitalWrite(D1, HIGH);`

CMSIS (STM32F103)

- "Blinky" (with SysTick)
- **~1 KByte Flash**
- More complex: `GPIOC->BSRR = ...;`

Where are the remaining 249 KByte?!

Motivation: Control instead of Convenience

Answer: There is no "overhead".

1. **No Framework Overhead**
   - **Arduino:** `digitalWrite(PIN_NAME);`
   - *What happens:* Calls 10+ C++ functions, translates pin names to port/bit, checks for timer conflicts...
   - **CMSIS:** `GPIOC->BSRR = GPIO_BSRR_BR13;`
   - *What happens:* Compiles down to 3-4 machine instructions.
2. **No Operating System (RTOS) / Stacks**
   - **ESP8266:** *Always* loads a FreeRTOS and the Wi-Fi stack just to execute `delay()`.
   - **The CMSIS code:** The `Delay_ms()` is 3 lines of C code.

→**Conclusion:** Convenience is traded for **Control, Efficiency**, and **Minimal Memory Footprint**.

Architecture 1: Timing with the SysTick Timer

- **What is SysTick?** A simple 24-bit "countdown" timer located *directly within the Cortex-M Core*.
- **What is it for?** Ideal as a "time base" (ticker) for a simple `delay` function.
- **CMSIS-Core Function:** `SysTick_Config(uint32_t ticks)`

The Idea:

1. The SysTick is configured to trigger an interrupt every **1ms**.
2. `SysTick_Config(SystemCoreClock / 1000);`
3. `SystemCoreClock` is a global variable (from `system_stm32f10xx.c`) holding the CPU clock (e.g., 8MHz HSI).
4. $8,000,000/1000 = 8000$ ticks.
5. The SysTick counts from 8000 down to 0, triggers an interrupt (`SysTick_Handler`), and restarts.

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

Step 1: Clock (RCC)

- A clock signal is needed for:
  - `USART1` (the peripheral itself)
  - `GPIOA` (for the pins)
  - `AFIO` (Alternate Function IO)
- Register: `RCC->APB2ENR`

Step 2: GPIO (Pins)

- The "Blue Pill" board uses `PA9` (TX) and `PA10` (RX) for `USART1`.
- Configuration (in the `GPIOA->CRH` register):
  - `PA9 (TX)`: **Alternate Function, Push-Pull**, 10MHz
  - `PA10 (RX)`: **Input, Floating**

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

Step 3: Baud Rate (BRR)

- A divisor must be calculated $\rightarrow$ See Excursus!
- Register: `USART1->BRR`

Step 4: Activation (CR1)

- Enable the UART (`UE`), Transmitter (`TE`), and Receiver (`RE`).
- Register: `USART1->CR1`

Excursus: Bit Masking (The Register Magic)

The Problem:

- The goal is to configure Pin **PA9** (e.g., to `1001`).
- The 32-bit register `GPIOA->CRH` must be modified.
- **Objective:** Change *only* the 4 bits for Pin 9, without destroying the other 7 pins!

The Register Layout (`CRH`):

Pin	...	Pin 10				**Pin 9**				Pin 8			
Bits	...	11	10	9	8	**7**	**6**	**5**	**4**	3	2	1	0

$\rightarrow$**Bits 7, 6, 5, and 4 must be modified.**

Excursus: Bit Masking (Part 1: The Concept)

The "Read-Modify-Write" Operation

- It is not possible to write only 4 bits. The entire 32-bit register must be read, modified, and written back.
- This operation always has two steps:
  1. **Clearing (Masking):** Set the target bits to 0 (with `&= ~...`).
  2. **Setting (Writing):** Set the new bits to 1 (with `|= ...`).

Step 1: Setting the old bits for Pin 9 to 0.

```
// Short form (which is used):
GPIOA->CRH &= ~(0xF << 4);
```

Excursus: Bit Masking (Part 1: Detailed Analysis)

Analysis of: `GPIOA->CRH &=  (0xF « 4);`

- `0xF` ⇒ `0b1111`
- *(The 4-bit "pin mask")*

- `(0xF « 4)` ⇒ `0b...0000 1111 0000`
- *(Mask is shifted to the position of Pin 9: Bits 4-7)*

- `˜(0xF « 4)` ⇒ `0b...1111 0000 1111`
- *(The "clear mask". Only bits 4-7 are 0)*

- `...  &= ...` ⇒ (Bitwise AND assignment)
- *(Everything ANDed with 1 remains. Everything ANDed with 0 becomes 0.)*

Excursus: Bit Masking (Part 2: Setting)

Step 2: Setting the new bits for Pin 9.

- The register now has zeroes at the Pin 9 position.

```
GPIOA->CRH |= (0x9 << 4);
```

What is happening here?

- `0x9` ⇒ `0b1001` (The new value for 10MHz AF-Out)
- « 4 ⇒ "Shift the value to bit position 4"
- `(0x9 « 4)` ⇒ `0b...10010000` (The "set mask")
- `|=` ⇒ "Bitwise OR assignment"
- *(The zeroes are overwritten with 1001)*

Excursus: Baud Rate (Part 1: The Formula)

The Problem:

- The goal is to set 9600 Baud (Bits/s).
- A **divisor** must be calculated that divides the system clock (f_{CLK}) down to 9600.

The Formula (from RM0008):

- $\text{Baud} = f_{CLK}/(16 \times \text{USARTDIV})$

Step 1: Solve for `USARTDIV`

- $f_{CLK} = 8,000,000$ (The 8MHz HSI)
- $\text{USARTDIV} = 8,000,000/(16 \times 9600)$
- $\text{USARTDIV} = 8,000,000/153,600$
- USARTDIV = **52.0833...**

Excursus: Baud Rate (Part 2: The `BRR` Register)

Problem: How is **52.083** stored in a 16-bit register?

Solution: The `BRR` (Baud Rate Register) is split:

Bits	**[15 : 4]** (12 Bits)	**[3 : 0]** (4 Bits)
Content	`DIV_Mantissa`	`DIV_Fraction`
Meaning	Integer Part	Fractional Part (16ths)

Step 2: Splitting the number `52.083`

- **Mantissa:** The integer part is **52** $\Rightarrow$ `0x34`
- **Fraction:** The fractional part is **0.083...**
- *Conversion to 4-bit value:* $0.083 \times 16 = 1.33...$
- *Rounding up:* **1** $\Rightarrow$ `0x01`

Excursus: Baud Rate (Part 3: The Code)

Step 3: Assembling the parts in code

- Mantissa = `0x34` (52)
- Fraction = `0x01` (1)

```
USART1->BRR = (0x34 << 4) | 0x01;
```

What is happening here?

- `(0x34 « 4)` ⇒ `0x340`
- The mantissa (52) is shifted left by 4 bits.
- *(Binary: ...0011 0100 0000)*

- `| 0x01` ⇒ `0x341`
- The fraction (1) is ORed into the empty 4 bits `[3:0]`.
- *(Binary: ...0011 0100 0001)*

Excursus: Baud Rate (Part 4: The Result)

Analysis of: `USART1->BRR = 0x341;`

- The register now holds the value `0x341`.
- The hardware reads this as:
  - Mantissa $=$ `0x34` $= 52$
  - Fraction $=$ `0x1` $= 1$
- The resulting divisor is: $52 + (1/16) =$ **52.0625**

Comparison:

- Desired divisor: 52.0833...
- Achieved divisor: 52.0625

$\rightarrow$ *The deviation is $<$ 0.1%, which is perfect for UART.*

Practice: The "Echo-Bot" (Part 1: The Logic)

Test: All Parts (SysTick, LED, UART) Together

Objective: The MCU should send back (echo) every received UART character and toggle the LED upon reception.

The core `main()` loop:

```c
while (1){
    // 1. Wait (blocking) until a character arrives
    char c = UART1_GetChar();
    // 2. Send the same character back immediately (Echo)
    UART1_SendChar(c);
    // 3. Toggle the LED as visual feedback
    LED_Toggle();
}
```

Practice: The "Echo-Bot" (Part 2: Setup & Test)

1. **Hardware Setup (Wiring)**
   - Connect your USB-UART Converter (FTDI, CH340, etc.)
   - Converter **TX** $\rightarrow$ Blue Pill **PA10 (RX)**
   - Converter **RX** $\rightarrow$ Blue Pill **PA9 (TX)**
   - Converter **GND** $\rightarrow$ Blue Pill **GND**
2. **Host Setup (PC Terminal)**
   - Use a Serial Terminal (e.g., PuTTY, CoolTerm, VS Code).
   - Settings: **9600** Baud, 8-N-1, No Flow Control.
3. **Acceptance Criteria (Verification)**
   - If you type **'A'** in the terminal...
   - ...you must see **'A'** echoed back.
   - ...the PC13 LED must toggle **once**.

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

Objective: Extend the "Echo-Bot" to a "Morse-Bot".

Requirement: Receive a character via UART, output it as Morse code on the LED.

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 two new functions:
2. `void morse_dit(void)`
3. `void morse_dah(void)`
4. *(These will call `LED_On()`, `LED_Off()` and `Delay_ms()`)*
5. Create a function `void morse_char(char c)`.
6. Use a `switch (c)` statement.
7. Implement (e.g.): 'S' (...), 'O' (--), 'A' (.-)
8. Modify your `main()` loop:
9. `char c = UART1_GetChar();`
10. `UART1_SendChar(c);` // Echo (good for debugging)
11. `morse_char(c);`
12. `Delay_ms(DIT_MS * 3);` // Pause after the letter

Debugging (Trap 1): UART-TX (Sending) Failed

Problem: The "life sign" test (`UART1_SendChar('!')`) sends nothing. The TX line (PA9) remains silent.

The Cause: Register Typo

- `CRH` = Control Register **High** (Pins 8-15)
- `CRL` = Control Register **Low** (Pins 0-7)

The Faulty Code:

```
// Pin 9 (TX) Configuration
// Clears bits for Pin 9 in CRH (Correct)
GPIOA->CRH &= ~(0xF << 4);
// Writes bits for Pin 1 in CRL (FATAL MISTAKE!)
GPIOA->CRL |= (0x9 << 4);
```

Realization: Pin 9 (TX) was 0000 (Input Analog) and could not transmit. Pin 1 was configured instead (uselessly).

Debugging (Trap 2): UART-RX (The Problem)

Problem: The code (e.g., "Echo-Bot") appears "frozen". The terminal remains empty.

The (Supposed) Error:

```c
while (1){
    // THE PROGRAM IS WAITING HERE!
    char c = UART1_GetChar();
    // These lines are NEVER reached
    // until a character is received.
    UART1_SendChar('!');
    LED_Toggle();
}
```

Debugging (Trap 2): UART-RX (The Realization)

Realization: This is not a bug, this is the design!

- The `UART1_GetChar()` function is **blocking**.
- It waits in its *own* `while` loop for the `RXNE` flag.
- The program "hangs" as expected, waiting for input.

Best Debug Strategy (The "Life Sign" Test):

- **Always** test Transmission (TX) first and in isolation (e.g., in a 1-second loop), *before* waiting for Reception (RX).

Debugging (Trap 3): C-Syntax & Compiler Errors

Problem 1 (Fatal): Linker Error: `undefined reference to 'tolwer'`

- **Cause:** Simple typo. The function is `tolower()`.
- **Realization:** The linker (`ld`) fails if a function was declared (or implicitly assumed) but never *defined* (implemented).

Problem 2 (Warning): Compiler Warning: `expected 'char' but argument is 'char *'`

- **Cause:** `UART1_SendChar("R");` instead of `UART1_SendChar('R');`
- **Central C-Realization:**
  - `'R'` (single quotes): Is a `char`. A single number (ASCII value).
  - `"R"` (double quotes): Is a `char*`. A pointer to a string (`{'R', '\0'}`) in memory.

Recap: What Was Learned

Learned:

1. `SysTick` is the CMSIS standard for timing (`delay`).
2. `volatile` is essential for shared variables (ISR vs. main).
3. **Bit Masking:** The `&= ~..` and `|= ...` dance (Read-Modify-Write).
4. **Baud Rate:** How to calculate f_{CLK} and the divisor (Mantissa/Fraction) into the `BRR` register.
5. **C-Syntax:** The difference between `'c'` (char) and `"c"` (char*).
6. **Debugging:** `CRH` vs. `CRL`, and testing `TX` before `RX`.

Outlook: The Problem with "Blocking"

The BIG Problem with the current code:

- `UART1_GetChar()` is **blocking**.
- `Delay_ms()` is **blocking**.
- *The system "freezes" while it waits or flashes Morse code. It misses new UART data during this time!*

Outlook (Part 3): The Solution

- **Interrupts:** UART reception in the background (ISR).
- **Ring Buffers:** Buffering data.
- **State Machines:** Flashing Morse code without blocking the `main` loop.

Part 3: The "Blocking" Problem

Analysis of our Morse-Bot (Part 2):

- The CPU is "deaf" while it is sending Morse code.
- It is completely stuck inside the `Delay_ms()` function.
- During this time, it cannot call `UART1_GetChar()` to check for new data.

The Consequence: UART Overrun

1. User types 'S' $\rightarrow$ `morse_char('s')` starts (blocks for 500ms).
2. User types 'O' (while 'S' is still flashing).
3. The 'O' byte hits a "deaf" MCU. The UART hardware register (`DR`) is overwritten before the CPU can read it.
4. **Data loss.**

Part 3: The Solution: Decoupling

The Goal: Decouple Reception from Processing.

- **Reception (Fast):** Must happen in the background, immediately when data arrives.
- $\rightarrow$ *This is the job of an **Interrupt**.*
- **Processing (Slow):** The `main()` loop can do this when it has time (e.g., flashing the Morse code).

The Analogy: The "Mailbox"

- The **Interrupt (ISR)** is the *Mailman* who quickly drops a letter in the box (he doesn't wait).
- The **Ring Buffer** is the *Mailbox* itself.
- The **Main Loop** is *You*, checking the mailbox whenever you have a free moment.

Architecture: What is a Ring Buffer?

Also called: Circular Buffer or FIFO (First-In, First-Out)

- A simple array in RAM (e.g., 32 bytes).
- Two "pointers" (which are just index variables):
  - `head`: Where the ISR *writes the next byte to*.
  - `tail`: Where the `main` loop *reads the next byte from*.
- When `head` or `tail` reach the end of the array, they "wrap around" back to 0.

States:

- Buffer Empty: `head == tail`
- Buffer Full: `head` is "just behind" `tail`

Architecture: Ring Buffer (The Critical Risk)

WARNING: Race Condition

- `head` is modified by the **ISR context**.
- `tail` is modified by the **main context**.

The Problem:

- The compiler does not know these are modified by two different contexts and will "optimize" access.
- It might cache the value in a CPU register, assuming it won't change "randomly". The `main` loop might never see the change the ISR made.

Architecture: Ring Buffer (The Solution)

The Solution (Mandatory):

- The index variables **must** be declared as `volatile`.
- `volatile uint32_t head;`
- `volatile uint32_t tail;`
- This forces the compiler to re-read the variable from RAM *every single time* it is accessed, even if it looks "inefficient".
- It prevents the "stale register" optimization bug.

Step 1: Enable the UART Interrupt (The "Bell")

Goal: Stop actively waiting for `RXNE` (Polling).

Solution: Tell the UART to trigger an interrupt as soon as the `RXNE` flag (Receive Register Not Empty) is set.

How? In the `UART1_Init()` function.

- A bit must be set in `USART1->CR1` (Control Register 1).
- The bit is called: **`RXNEIE`** (RXNE Interrupt Enable)

The Code (Addition to `UART1_Init()`):

```
// At the end of UART1_Init(), after enabling UE, TE, RE:
// Enable the "Receive Register Not Empty" Interrupt
USART1->CR1 |= USART_CR1_RXNEIE;
```

Step 2: Inform the "Boss" (The NVIC)

Problem: Just because the UART "rings the bell" (`RXNEIE`) doesn't mean the CPU (the "boss") is listening.

Solution: The NVIC (Nested Vectored Interrupt Controller)

- The NVIC is the interrupt manager inside the Cortex-M Core.
- It manages *all* interrupt sources (Timers, UARTs, GPIOs...).
- We must tell the NVIC: "Hey, the bell from `USART1` is important. Turn it on."

The Code (At the start of `main()`)

```
// We must tell the NVIC to globally enable
// the USART1 interrupt.
// (This is a CMSIS-Core function)
NVIC_EnableIRQ(USART1_IRQn);
```

Step 3: The "Mailman" (The ISR)

Goal: The Interrupt Service Routine (ISR) must be *lightning fast*.

What happens (The Chain):

1. A byte arrives.
2. `RXNE` flag is set by hardware.
3. `RXNEIE` bit is active $\rightarrow$ UART sends signal to NVIC.
4. `USART1_IRQn` is enabled in NVIC $\rightarrow$ CPU stops `main()`, jumps to ISR.

The Implementation (in `stm32f1xx_it.c`)

Step 3: The "Mailman" (The ROBUST ISR)

Problem: The ISR can be triggered by errors (like Overrun). If an error flag is not cleared, the UART will lock up.
Solution: Check for (and clear) errors FIRST.

Step 3: The "Mailman" (The ROBUST ISR)

```
void USART1_IRQHandler(void){
    // --- 1. Error Handling (Overrun, Noise, etc.) ---
    if (USART1->SR & (USART_SR_ORE | USART_SR_NE |
    USART_SR_FE))
        {
            // Reading SR (above) and then DR (below)
            (void)(USART1->DR & 0xFF);
        }
    // --- 2. "Happy Path" (Data Reception) ---
    if ( (USART1->SR & USART_SR_RXNE) &&    (USART1->CR1
    & USART_CR1_RXNEIE) )
        {
            // Reading DR clears the RXNE flag
            uint8_t byte = (uint8_t)(USART1->DR & 0xFF);

            RingBuffer_Write(&g_rxBuffer, byte);
        }
}
```

Excursus: Flash vs. RAM (The "Endurance" Question)

Your Question: "Doesn't the ring buffer kill my Flash endurance?"

Answer: No! It doesn't affect Flash at all.

Flash (Program Memory)

- **Usage:** Code (`.bin`), constants
- **Property:** Non-Volatile
- **Limitation:** Limited write cycles
- *Only used during flashing*

SRAM (Working Memory)

- **Usage:** Variables, buffer arrays
- **Property:** Volatile
- **Advantage:** Unlimited write cycles
- *Used continuously*

Conclusion: Ring buffer resides entirely in SRAM. RAM writes are "free" and cause no wear.