ARM-Embedded-Path

LED - UART

Pavel Pys

October 28, 2025

Overview

- Einleitung & Motivation
- Architektur & Grundlagen
- Exkurs: Register-Magie
- Praxis & Übung
- Debugging & Ausblick
- Aufgabe 3: Non-Blocking Design (Interrupts)
- Aufgabe 3: Non-Blocking Design (Interrupts)
- Architektur (Speicher)

CMSIS Schulung: UART & Timing

Aufgabe: Der "Morse-Bot"

Von der UART-Eingabe zum getimeten LED-Signal

Recap: Was war CMSIS?

- **Problem:** Jeder ARM-Chip hat andere Adressen für Peripherien (GPIO, Timer...).
- **CMSIS-Core (Cortex-M):** Einheitlicher Zugriff auf Kern-Funktionen (z.B. Interrupts, `SysTick`).
- **CMSIS-Device (STM32F103):** Stellt Header-Dateien (`stm32f103x6.h`) bereit.

Was wir gelernt haben (Blinky):

1. Takt für Peripherie (Port C) aktivieren: `RCC->APB2ENR`
2. GPIO-Pin (PC13) konfigurieren: `GPIOC->CRH`
3. Pin schalten: `GPIOC->ODR` (oder `BSRR`)

Motivation: Der 1kB "Blinky"

Die Beobachtung (Vergleich):

Arduino (z.B. ESP8266)

- "Blinky" (mit `delay()`)
- **$\tilde{2}$50+ KByte Flash**
- Einfach: `digitalWrite(D1, HIGH);`

CMSIS (STM32F103)

- "Blinky" (mit SysTick)
- **$\tilde{1}$ KByte Flash**
- Komplexer: `GPIOC->BSRR = ...;`

Wo sind die restlichen 249 KByte?!

Motivation: Kontrolle statt Komfort

Antwort: Wir haben keinen "Overhead".

1. **Kein Framework-Overhead**
   - **Arduino:** `digitalWrite(PIN_NAME);`
   - *Was passiert:* Ruft 10+ C++ Funktionen auf, übersetzt Pin-Namen in Port/Bit, prüft Timer-Konflikte...
   - **CMSIS:** `GPIOC->BSRR = GPIO_BSRR_BR13;`
   - *Was passiert:* Wird zu 3-4 Maschinenbefehlen übersetzt.

2. **Kein Betriebssystem (RTOS) / Stacks**
   - **ESP8266:** Lädt *immer* ein FreeRTOS und den WLAN-Stack, nur um `delay()` auszuführen.
   - **Unser Code:** Unser `Delay_ms()` sind 3 Zeilen C-Code.

→**Fazit:** Wir tauschen Komfort gegen **Kontrolle, Effizienz** und **minimalen Speicherbedarf**.

Architektur 1: Timing mit dem SysTick-Timer

- **Was ist SysTick?** Ein einfacher 24-Bit "Countdown"-Timer, der *direkt im Cortex-M Kern* sitzt.
- **Wofür?** Ideal als "Taktgeber" für eine simple `delay`-Funktion.
- **CMSIS-Core Funktion:** `SysTick_Config(uint32_t ticks)`

Die Idee:

1. Wir konfigurieren den SysTick so, dass er alle **1ms** einen Interrupt auslöst.
2. `SysTick_Config(SystemCoreClock / 1000);`
3. `SystemCoreClock` ist eine globale Variable (aus `system_stm32f10xx.c`) mit unserem CPU-Takt (8MHz HSI).
4. 8.000.000/1000 = 8000 Ticks.
5. Der SysTick zählt von 8000 auf 0, löst einen Interrupt aus (`SysTick_Handler`) und startet neu.

Architektur 2: Was ist UART (Bare-Metal)?

Schritt 1: Takt (RCC)

- Wir brauchen Takt für:
  - `USART1` (die Peripherie selbst)
  - `GPIOA` (für die Pins)
  - `AFIO` (Alternate Function IO)
- Register: `RCC->APB2ENR`

Schritt 2: GPIO (Pins)

- "Blue Pill" nutzt `PA9` (TX) und `PA10` (RX) für `USART1`.
- Konfiguration (im `GPIOA->CRH` Register):
  - `PA9 (TX)`: **Alternate Function, Push-Pull**, 10MHz
  - `PA10 (RX)`: **Input, Floating**

Architektur 2: Was ist UART (Bare-Metal)? (Forts.)

Schritt 3: Baudrate (BRR)

- Wir müssen einen Divisor (Teiler) berechnen → Siehe Exkurs!
- Register: `USART1->BRR`

Schritt 4: Aktivierung (CR1)

- UART aktivieren (`UE`), Senden (`TE`), Empfang (`RE`).
- Register: `USART1->CR1`

Exkurs: Bit-Masking (Die Register-Magie)

Das Problem:

- Wir wollen Pin **PA9** konfigurieren (z.B. auf `1001`).
- Wir müssen das 32-Bit Register `GPIOA->CRH` bearbeiten.
- **Ziel:** Ändere *nur* die 4 Bits für Pin 9, ohne die anderen 7 Pins zu zerstören!

Das Register-Layout (`CRH`):

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

→**Wir müssen die Bits 7, 6, 5 und 4 bearbeiten.**

Exkurs: Bit-Masking (Teil 1: Das Konzept)

Der "Read-Modify-Write" Vorgang

- Man kann nicht nur 4 Bits schreiben. Man muss das ganze 32-Bit-Register lesen, ändern und zurückschreiben.
- Dieser Vorgang hat immer zwei Schritte:
  1. **Löschen (Masking):** Die Ziel-Bits auf 0 setzen (mit `&= ~...`).
  2. **Setzen (Writing):** Die neuen Bits auf 1 setzen (mit `|= ...`).

Schritt 1: Die alten Bits für Pin 9 auf 0 setzen.

```
// Kurzform (die wir verwenden):
GPIOA->CRH &= ~(0xF << 4);
```

Exkurs: Bit-Masking (Teil 1: Detail-Analyse)

Analyse von: `GPIOA->CRH &=  (0xF << 4);`

- `0xF` $\Rightarrow$ `0b1111`
- *(Unsere 4-Bit "Pin-Maske")*
- `(0xF << 4)` $\Rightarrow$ `0b...0000 1111 0000`
- *(Maske wird an die Position von Pin 9 geschoben: Bits 4-7)*
- $\tilde{}$`(0xF << 4)` $\Rightarrow$ `0b...1111 0000 1111`
- *(Die "Lösch-Maske". Nur die Bits 4-7 sind 0)*
- `...  &= ...` $\Rightarrow$ (Bitweise UND-Zuweisung)
- *(Alles, was mit 1 per UND verknüpft wird, bleibt. Alles, was mit 0 per UND verknüpft wird, wird 0.)*

Exkurs: Bit-Masking (Teil 2: Setzen)

Schritt 2: Die neuen Bits für Pin 9 setzen.

- Das Register hat jetzt Nullen an der Position von Pin 9.

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

Was passiert hier?

- `0x9` ⇒ `0b1001` (Unser neuer Wert für 10MHz AF-Out)
- « `4` ⇒ "Schiebe den Wert an Bit-Position 4"
- `(0x9 « 4)` ⇒ `0b...10010000` (Die "Setz-Maske")
- `|=` ⇒ "Bitweise ODER-Zuweisung"
- *(Die Nullen werden mit 1001 überschrieben)*

Exkurs: Baudrate (Teil 1: Die Formel)

Das Problem:

- Wir wollen 9600 Baud (Bits/s) einstellen.
- Wir müssen einen **Divisor** (Teiler) berechnen, der unseren Systemtakt (f_{CLK}) auf 9600 herunterbricht.

Die Formel (aus RM0008):

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

Schritt 1: Nach `USARTDIV` auflösen

- $f_{CLK} = 8.000.000$ (Unser 8MHz HSI)
- $\text{USARTDIV} = 8.000.000/(16 \times 9600)$
- $\text{USARTDIV} = 8.000.000/153.600$
- $\text{USARTDIV} = \mathbf{52.0833...}$

Exkurs: Baudrate (Teil 2: Das `BRR` Register)

Problem: Wie speichern wir **52.083** in einem 16-Bit Register?

Lösung: Das `BRR` (Baud Rate Register) ist aufgeteilt:

Bits	**[15 : 4]** (12 Bits)	**[3 : 0]** (4 Bits)
Inhalt	`DIV_Mantissa`	`DIV_Fraction`
Bedeutung	Ganzzahl-Teil	Bruch-Teil (16tel)

Schritt 2: Unsere Zahl `52.083` aufteilen

- **Mantisse:** Der Ganzzahl-Teil ist **52** $\Rightarrow$ `0x34`
- **Fraktion:** Der Bruch-Teil ist **0.083...**
- *Umrechnung in 4-Bit-Wert:* $0.083 \times 16 = 1.33...$
- *Wir runden auf:* **1** $\Rightarrow$ `0x01`

Exkurs: Baudrate (Teil 3: Der Code)

Schritt 3: Die Teile im Code zusammensetzen

- Mantisse = `0x34` (52)
- Fraktion = `0x01` (1)

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

Was passiert hier?

- `(0x34 « 4)` ⇒ `0x340`
- Die Mantisse (52) wird 4 Bits nach links geschoben.
- *(Binär: ...0011 0100 0000)*
- `| 0x01` ⇒ `0x341`
- Die Fraktion (1) wird mit ODER in die leeren 4 Bits `[3:0]` eingefügt.
- *(Binär: ...0011 0100 0001)*

Exkurs: Baudrate (Teil 4: Das Ergebnis)

Analyse von: `USART1->BRR = 0x341;`

- Das Register hat jetzt den Wert `0x341`.
- Die Hardware liest das als:
  - Mantisse = `0x34` = 52
  - Fraktion = `0x1` = 1
- Der resultierende Divisor ist: $52 + (1/16) = \mathbf{52.0625}$

Vergleich:

- Gewünschter Divisor: 52.0833...
- Erreichter Divisor: 52.0625

$\rightarrow$*Die Abweichung ist $< 0.1\%$, was für UART perfekt ist.*

Praxis: Der "Echo-Bot"

Test 2: Alle Teile (SysTick, LED, UART) zusammen

Ziel: Der MCU soll jedes empfangene UART-Zeichen zurücksenden (Echo) und dabei die LED togglen.

Auszug aus der `main()`-Schleife:

```
        while (1)
        {
            // 1. Warte blockierend, bis ein Zeichen
    reinkommt
            char c = UART1_GetChar();
            // 2. Sende dasselbe Zeichen sofort zurück (
    Echo)
            UART1_SendChar(c);
            // 3. Toggle die LED als visuelles Feedback
            LED_Toggle();
        }
```

Übung: "Morse-Bot" (Teil 1: Regeln)

Ziel: Erweitere den "Echo-Bot" zum "Morse-Bot".

Anforderung: Empfange ein Zeichen per UART, gib es als Morse-Code auf der LED aus.

Basis-Timing:

- `DIT_MS = 100` (z.B. 100 Millisekunden)

Morse-Regeln:

- **Dit (Punkt):** LED AN (1 * `DIT_MS`), LED AUS (1 * `DIT_MS`)
- **Dah (Strich):** LED AN (3 * `DIT_MS`), LED AUS (1 * `DIT_MS`)
- **Pause (Buchstaben):** (3 * `DIT_MS`)
- **Pause (Wort/Space):** (7 * `DIT_MS`)

Übung: "Morse-Bot" (Teil 2: Aufgaben)

Aufgaben:

1. Erstelle zwei neue Funktionen:
2. `void morse_dit(void)`
3. `void morse_dah(void)`
4. *(Diese rufen `LED_On()`, `LED_Off()` und `Delay_ms()` auf)*
5. Erstelle eine Funktion `void morse_char(char c)`.
6. Nutze ein `switch (c)` Statement.
7. Implementiere (z.B.): 'S' (`...`), 'O' (`---`), 'A' (`.-`)
8. Ändere deine `main()`-Schleife:
9. `char c = UART1_GetChar();`
10. `UART1_SendChar(c);` // Echo (gut für Debugging)
11. `morse_char(c);`
12. `Delay_ms(DIT_MS * 3);` // Pause nach dem Buchstaben

Debugging (Falle 1): UART-TX (Senden) ging nicht

Problem: Der "Lebenszeichen"-Test (`UART1_SendChar('!')`) sendet nichts. Die TX-Leitung (PA9) bleibt still.

Die Ursache: Register-Tippfehler

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

Der fehlerhafte Code:

```
    // Pin 9 (TX) Konfiguration
    GPIOA->CRH &= ~(0xF << 4); // Löscht Bits für Pin
 9 im CRH (Korrekt)
    GPIOA->CRL |= (0x9 << 4); // Schreibt Bits für
Pin 1 im CRL (FATALER FEHLER!)
```

Erkenntnis: Pin 9 (TX) war 0000 (Input Analog) und konnte nicht senden. Pin 1 wurde stattdessen (sinnlos) konfiguriert.

Debugging (Falle 2): UART-RX (Problem)

Problem: Der Code (z.B. "Echo-Bot") scheint "eingefroren". Das Terminal bleibt leer.

Der (vermeintliche) Fehler:

```
while (1)
{
    // HIER WARTET DAS PROGRAMM!
    char c = UART1_GetChar();

    // Diese Zeilen werden NIE erreicht,
    // bis ein Zeichen empfangen wird.
    UART1_SendChar('!');
    LED_Toggle();
}
```

Debugging (Falle 2): UART-RX (Erkenntnis)

Erkenntnis: Das ist kein Fehler, das ist das Design!

- Unsere Funktion `UART1_GetChar()` ist **blockierend**.
- Sie wartet in ihrer *eigenen* `while`-Schleife auf das `RXNE`-Flag.
- Das Programm "hängt" wie erwartet und wartet auf Input.

Beste Debug-Strategie (Der "Lebenszeichen"-Test):

- Teste Senden (TX) *immer* zuerst und isoliert, z.B. mit einer 1-Sekunden-Schleife, *bevor* du auf Empfang (RX) wartest.

Debugging (Falle 3): C-Syntax & Compiler-Fehler

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

- **Ursache:** Simpler Tippfehler. Es heiSSt `tolower()`.
- **Erkenntnis:** Der Linker (`ld`) bricht ab, wenn eine Funktion zwar deklariert (oder implizit angenommen), aber nirgends *definiert* wurde.

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

- **Ursache:** `UART1_SendChar("R");` statt `UART1_SendChar('R');`
- **Zentrale C-Erkenntnis:**
  - `'R'` (einfach): Ist ein `char`. Eine einzelne Zahl (ASCII-Wert).
  - `"R"` (doppelt): Ist ein `char*`. Ein Zeiger auf einen String (`{'R', '\0'}`) im Speicher.

Recap: Was wir gelernt haben

Gelernt:

1. `SysTick` ist die CMSIS-Standard für Timing (`delay`).
2. `volatile` ist essenziell bei geteilten Variablen (ISR vs. main).
3. **Bit-Masking:** Der `&= ~...` und `|= ...` Tanz (Read-Modify-Write).
4. **Baudrate:** Wie man f_{CLK} und Divisor (Mantisse/Fraktion) ins `BRR` Register rechnet.
5. **C-Syntax:** Der Unterschied zwischen `'c'` (char) und `"c"` (char*).
6. **Debugging:** `CRH` vs. `CRL`, und `TX`-Test vor `RX`-Test.

Ausblick: Das Problem mit "Blocking"

Das GROSSE Problem unseres jetzigen Codes:

- `UART1_GetChar()` ist **blockierend**.
- `Delay_ms()` ist **blockierend**.
- *Das System "steht still", während es wartet oder morst. Es verpasst in dieser Zeit neue UART-Daten!*

Ausblick (Teil 3): Die Lösung

- **Interrupts:** UART-Empfang im Hintergrund (ISR).
- **Ringbuffer:** Daten zwischenspeichern.
- **State Machines:** Morsen, ohne die `main`-Schleife zu blockieren.

Teil 3: Das "Blocking"-Problem

Analyse unseres Morse-Bots (Teil 2):

- Die CPU ist "taub", während sie morst.
- Sie hängt in `Delay_ms()` fest.
- In dieser Zeit kann sie `UART1_GetChar()` nicht aufrufen.

Das Problem: UART Overrun

1. User tippt 'S' → `morse_char('s')` startet (blockiert ca. 500ms).
2. User tippt 'O' (während 'S' morst).
3. Das 'O' trifft auf einen "tauben" MCU. Das UART-Hardware-Register (DR) wird überschrieben, bevor die CPU es lesen kann.
4. **Datenverlust.**

Die Lösung: Entkopplung von Empfang & Verarbeitung.

- **Empfang (schnell):** Muss im Hintergrund passieren (Interrupt).

Architektur: Interrupt & Ringbuffer

Unsere neue Architektur (Der "Briefkasten"):

1. **Der UART RX-Interrupt (Die "Klingel")**
   - Wir konfigurieren den UART so, dass er einen Interrupt (`USART1_IRQHandler`) auslöst, sobald ein Byte ankommt (`RXNE`-Flag).
   - Die CPU stoppt kurz ihre Hauptarbeit (Morsen).
2. **Der Ringbuffer (Der "Briefkasten")**
   - Die Interrupt-Routine (ISR) ist extrem schnell:
   - Sie liest das Byte aus `USART1->DR`.
   - Sie "wirft" das Byte in einen Puffer (den Ringbuffer).
   - Sie ist sofort wieder weg.
3. **Die `main()`-Schleife (Der "Leser")**
   - Die `main()`-Schleife (Morse-Logik) schaut in ihrer Freizeit in den Ringbuffer, ob "Post" da ist und holt sich das nächste Byte ab.

Architektur: Was ist ein Ringbuffer?

Auch: Circular Buffer oder FIFO (First-In, First-Out)

- Ein einfaches Array im RAM (z.B. 32 Bytes groSS).
- Zwei "Zeiger" (eigentlich nur Index-Variablen):
  - `head`: Wo die ISR das *nächste Byte hineinschreibt*.
  - `tail`: Wo die `main`-Loop das *nächste Byte herausliest*.
- Wenn `head` oder `tail` das Ende des Arrays erreichen, "wrappen" sie zurück auf 0.
- **Zustände:**
  - Puffer leer: `head == tail`
  - Puffer voll: `head` ist "kurz vor" `tail`

WICHTIG (Race Condition):

- `head` wird von einer ISR geändert.
- `tail` wird von `main` geändert.
- → Die `head`/`tail` Index-Variablen müssen **`volatile`** sein!

Schritt 1: UART-Interrupt aktivieren (Die "Klingel")

Ziel: Wir wollen nicht mehr aktiv auf `RXNE` warten (Polling).

Lösung: Wir sagen dem UART, er soll einen Interrupt auslösen, sobald das `RXNE`-Flag (Receive Register Not Empty) gesetzt wird.

Wie? In unserer `UART1_Init()`-Funktion.

- Wir müssen ein Bit im `USART1->CR1` (Control Register 1) setzen.
- Das Bit heiSSt: **`RXNEIE`** (RXNE Interrupt Enable)

Der Code (Ergänzung in `UART1_Init()`):

```
    // Am Ende von UART1_Init(), nach dem Aktivieren
von UE, TE, RE:
    // Aktiviere den "Receive Register Not Empty"
Interrupt
    USART1->CR1 |= USART_CR1_RXNEIE;
```

Schritt 2: Den "Chef" (NVIC) informieren

Problem: Nur weil der UART "klingelt", heiSSt das nicht, dass die CPU (der "Chef") zuhört.

Lösung: Der NVIC (Nested Vectored Interrupt Controller)

- Der NVIC ist der Interrupt-Manager direkt im Cortex-M Kern.
- Er verwaltet *alle* Interrupt-Quellen (Timer, UARTs, GPIOs...).
- Wir müssen dem NVIC sagen: "Hey, die Klingel von `USART1` ist uns wichtig. Stell sie an."

Der Code (Am Anfang von `main()`)

```
    // Wir müssen dem NVIC sagen, dass er den USART1-
Interrupt
    // global aktivieren soll.
    // (CMSIS-Core Funktion)
    NVIC_EnableIRQ(USART1_IRQn);
```

Schritt 3: Der "Postbote" (Die ISR)

Ziel: Die Interrupt Service Routine (ISR) muss *blitzschnell* sein.
Was passiert:

1. Ein Byte kommt an.
2. `RXNE`-Flag wird von Hardware gesetzt.
3. `RXNEIE`-Bit ist an $\rightarrow$ UART sendet Signal an NVIC.
4. `USART1_IRQn` ist im NVIC an $\rightarrow$ CPU stoppt `main()`, springt zur ISR.

Die Implementierung (in `stm32f1xx_it.c`)

Schritt 3: Der "Postbote" (Die ISR)

```
// Prototyp der Funktion ist in startup_stm32...s
void USART1_IRQHandler(void)
{
    // 1. Prüfen, ob WIRKLICH ein RXNE-Interrupt der
Grund war
    if ( (USART1->SR & USART_SR_RXNE) )
    {
        // 2. Byte aus dem Datenregister lesen
        //    (WICHTIG: Das löscht das RXNE-Flag
automatisch!)
        uint8_t byte = (uint8_t)(USART1->DR & 0xFF);
        // 3. Byte in unseren "Briefkasten" werfen
        RingBuffer_Write(&g_rxBuffer, byte);
    }
}
```

Exkurs: Flash vs. RAM (Die "Endurance"-Frage)

Deine Frage: "Killt der Ringbuffer nicht meine Flash Endurance?"

Antwort: Nein! Er killt sie *null*, weil er *nicht* im Flash liegt.

1. **Flash (Programmspeicher)**
   - **Analogie:** Festplatte
   - **Nutzung:** Dein Code (`.bin`), `const`
   - **Eigenschaft:** Non-Volatile
   - **Problem: Limitierte** Zyklen (10k-100k), langsames Schreiben.
   - *Wird nur beim Flashen genutzt.*

2. **SRAM (Arbeitsspeicher)**
   - **Analogie:** RAM / Arbeitsspeicher
   - **Nutzung:** Alle C-Variablen (`g_msTicks`, unser Puffer-Array)
   - **Eigenschaft:** Volatile
   - **Vorteil: Unlimitierte** Zyklen, extrem schnelles Schreiben.
   - *Wird permanent genutzt.*

Fazit: Unser Ringbuffer liegt komplett im SRAM.