ARM-Embedded-Path

CMSIS-Basics

Pavel Pys

October 25, 2025

Overview

- Introduction
- ARM-Architecture
  - General Information
  - ARM-Architecture
  - CMSIS vs HAL
- Practice
  - Blink-Test:Mixed

Introduction
What is the target of this journey?

In these slides, I want to document my learning progress in handling ARM microcontrollers, in my case from the company ST-Microelectronics. Ultimately, this slide set should become a reference work. - Hanover 21.10.2025

Introduction
What is the ARM architecture?

- A microprocessor architecture developed by the British computer company Acorn in 1983. Initially, ARM stood for Acorn RISC Machine, and was later changed to Advanced RISC Machines.
- The company does not manufacture the chips itself, but instead grants different licenses to semiconductor development companies, which then manufacture based on this architecture.

Introduction
What is the ARM architecture?

Today, many renowned chip manufacturers build their chips on the ARM architecture.

Notable manufacturers:

- Apple
- Qualcomm Inc.
- Samsung Electronics
- Huawei Technologies Co. Ltd.
- STMicroelectronics
- ...

Introduction
Market share of ARM chips

The market share of ARM-based chips is very large, but depends on the system. In mobile phones, it was already about 98% in 2005 (at least one ARM processor).

In data and server centers, ARM is currently growing rapidly, though its goal of reaching 50% market share by the end of 2025 is considered ambitious by some analysts.

Introduction
What are the advantages of ARM?

The ARM architecture offers several advantages:

- ARM uses the RISC principle.
- ARM cores are small and can be easily combined.
- Low costs and licensing flexibility.
- Large ecosystem.
- High performance per watt (efficiency).
- Good security features.
- Wide range of applications.

ARM
What is a Microcontroller Architecture?

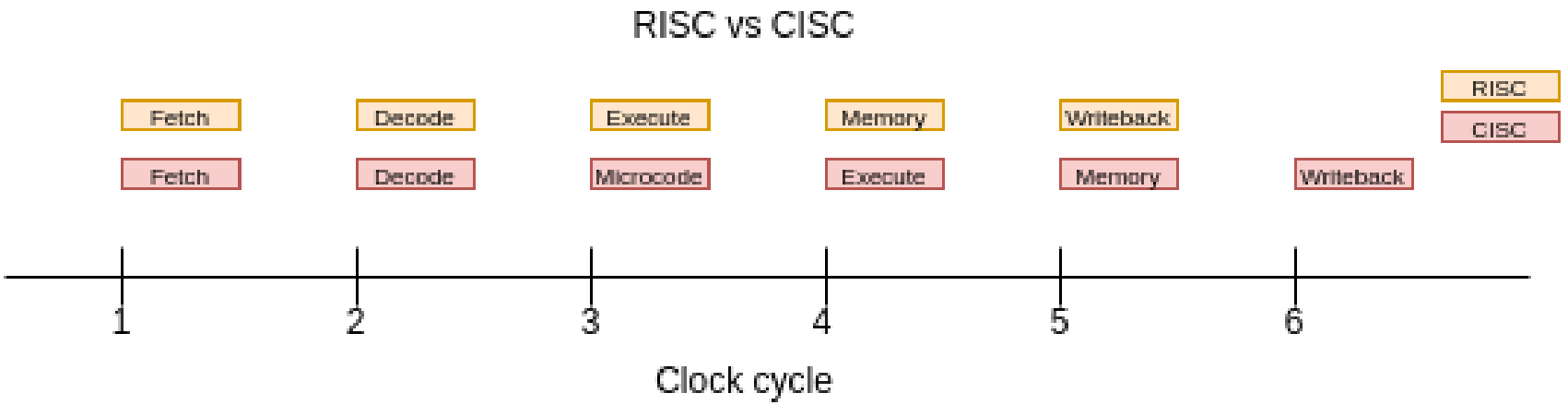
A microcontroller architecture describes the internal structure and functionality of a microcontroller meaning how the individual components on the chip are interconnected and how they work together.

The architecture consists of: CPU, memory, bus system, peripherals, clock source, and power supply as well as reset logic.

ARM
What is a Microcontroller Architecture?

In summary: A microcontroller architecture is the blueprint of how CPU, memory, peripherals, buses, and clock sources work together on a single chip to execute tasks efficiently.

ARM
RISC vs CISC



The graphic shows the pipelines of RISC and CISC. RISC processes instructions in parallel (one new instruction per clock cycle), while CISC processes longer and more complex instructions sequentially.

ARM
Architecture Structure

Register-based design (e.g., 16-32 registers) with a pipeline architecture for parallel instruction execution.

Harvard or Von Neumann structure depending on the type.

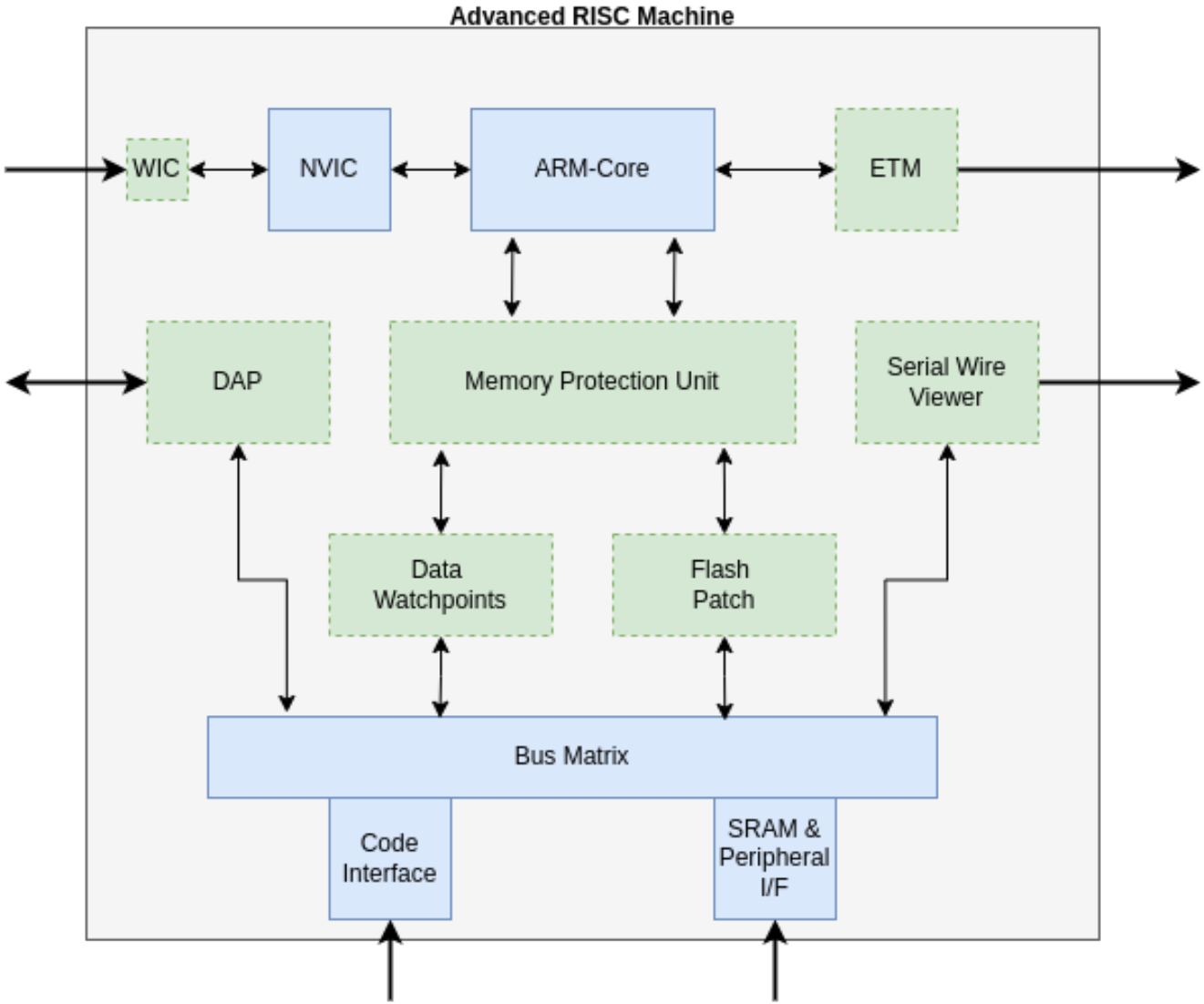
Components:

- ALU (Arithmetic Logic Unit)
- Register set (R0-R15)
- Program Counter, Stack Pointer
- Interrupt Controller
- Bus interfaces (AHB, APB...)

ARM
Architecture Structure

Cortex-M microcontrollers typically implement a modified Harvard architecture, where instruction and data buses are separate internally, but share a unified memory space.

ARM
ARM - Cortex M Block-Diagram



ARM
WIC - Wake-up Interrupt Controller

In deep sleep (core clock and NVIC logic powered down), the WIC acts as a shadow interrupt latch, allowing selected IRQs to wake the system.

Control of WIC behavior through:

- NVIC->ISER (enable/disable interrupts)
- NVIC priorities and BASEPRI/PRIMASK (only unmasked and sufficiently prioritized IRQs can wake the system)
- Sleep depth via SCB->SCR.SLEEPDEEP (Deep Sleep vs. normal Sleep)
- WFI/WFE (how you enter sleep)
- Peripheral wake sources (EXTI edge, RTC alarm, USART-RX, I2C address match, etc.)

NVIC is tightly coupled to the Cortex-M core through the System Control Block (SCB), forming the Exception Model

ARM
NVIC - Nested Vector Interrupt Controller

The Nested Vector Interrupt Controller (NVIC) is the hardware block in the ARM Cortex-M core that:

- Accepts, prioritizes, nests, and forwards interrupts (IRQs) to the CPU,
- Can mask, enable, set/clear pending interrupts,
- Integrates exception handling (Reset, NMI, HardFault, SysTick, etc.) using the same mechanisms.

It is directly integrated into the core, not in the periphery, and coupled with the System Control Block (SCB).

ARM
ARM Core

The ARM core is the actual processing core (CPU core) in the microcontroller - meaning the logical unit that executes code, performs arithmetic operations, processes interrupts, and communicates with memory and peripherals via buses.

In this case (STM32F103), this is an ARM Cortex-M3, based on the ARMv7-M architecture.
This means:

- 32-bit RISC processor
- Harvard architecture (separate buses for code and data)
- Pipeline design
- Thumb-2 instruction set (compact mix of 16- and 32-bit instructions)

ARM
ARM Core: Architectural Features

Harvard Architecture:
Separate buses for code (I-Bus) and data (D-Bus) enables parallel reading of instructions and data

Thumb-2 Instruction Set:
Mix of 16- and 32-bit instructions compact code with full functionality

NVIC Integration:
Interrupt handling directly in the core no external interrupt controllers needed

Sleep and Deep Sleep Modes:
Power saving functions via WFI/WFE instructions

ARM
ARM Core: Architectural Features

Harvard Concept in Action:

- Instructions are fetched via the I-Bus from Flash memory
- Data (variables, peripheral registers) via the D-Bus
- System and debug accesses (DMA, DAP, Trace) via the System bus

This allows the Cortex-M3 to simultaneously read an instruction and access data.

ARM
ARM Core: Conclusion

The ARM Cortex-M3 core is a 32-bit RISC processor with:

- Efficient pipeline design,
- Integrated interrupt controller,
- Memory protection (MPU),
- Integrated debug/trace architecture (CoreSight),
- And ideal for deterministic real-time and embedded applications (e.g., in your STM32F103).

It is the heart of the microcontroller - all other components (Flash, SRAM, Timer, UART, etc.) are built around it as peripherals.

ARM
DAP - Debug Access Port

The DAP (Debug Access Port) is the interface between your debugger (e.g., ST-Link, J-Link) and the internal debug and trace units of your ARM core.

The DAP acts as the "debug router" between the external world and the CoreSight internals.

The DAP consists of an AP (Access Port) and DP (Debug Port) interface e.g. SW-DP for SWD or JTAG-DP for JTAG.

ARM
MPU - Memory Protection Unit

The MPU (Memory Protection Unit) is a hardware unit in the ARM core that divides memory into regions and monitors access rights (read/write/execute) for each region.

It prevents your code from accidentally writing to "forbidden" areas or executing from unauthorized memory.

It is thus a mini memory protection system, similar to an MMU (Memory Management Unit) in a PC - but simpler and without virtual addresses.

ARM
Serial Wire Viewer (SWV)

The Serial Wire Viewer is a real-time trace technology that allows debugging output with minimal overhead.

Key Features:

- Real-time data transmission over SWO (Serial Wire Output) pin
- Low overhead monitoring during program execution
- Part of the CoreSight debug architecture
- Uses ITM (Instrumentation Trace Macrocell) with internal FIFO

Common Use Cases:

- Printf debugging via ITM
- Event counters and timestamps
- Exception trace (HardFault, interrupts)
- PC sampling for profiling

ARM
SWV - Components and Configuration

Hardware Components:

- **ITM:** Instrumentation Trace Macrocell (32 stimulus ports)
- **DWT:** Data Watchpoint and Trace unit (PC sampling, cycle counting)
- **TPIU:** Trace Port Interface Unit (formats and outputs trace data)
- **SWO:** Serial Wire Output pin (shares pin with JTAG TDO)

Important: SWO only works with SWD debug mode, not parallel to JTAG! **Configuration Requirements:**

- Enable trace in DEMCR register
- Configure SWO pin speed (must match debugger)
- Enable desired ITM stimulus ports
- Set up DWT for PC sampling or data trace if needed

ARM
SWV - Practical Printf Example

ITM Printf Implementation:

```c
#include "core_cm3.h"

int _write(int file, char *ptr, int len) {
    for (int i = 0; i < len; i++) {
        // Poll: 0 = FIFO full/port disabled
        //       1 = at least one word fits
        while ((ITM->PORT[0].u32 & 1) == 0);

        // Send character
        ITM->PORT[0].u8 = *ptr++;
    }
    return len;
}

int main(void) {
    // Enable TRCENA in DEMCR
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;

    // Enable ITM stimulus port 0
    ITM->TER |= (1 << 0);

    printf("Hello from SWV!\n");
    while(1);
}
```

ARM
Data Watchpoints (DWT)

The Data Watchpoint and Trace (DWT) unit provides hardware-based memory access monitoring and performance profiling.

Core Capabilities:

- Comparators for watchpoints (implementation dependent)
- Number available in DWT_CTRL.NUMCOMP (typically 4 on Cortex-M3)
- Monitor read, write, or read/write access to memory addresses
- Generate debug events or trigger trace without stopping CPU
- Cycle counter (CYCCNT) for precise timing measurements

Use Cases:

- Detect when a variable is modified (data breakpoint)
- Profile code execution time
- Trace memory access patterns
- Detect buffer overruns or stack corruption

ARM

Data Watchpoints - Configuration

Key DWT Registers (per comparator):

- **DWT_COMPn:** Address to watch
- **DWT_MASKn:** Address range (0 = exact match, 1-31 = range size)
- **DWT_FUNCTIONn:** Function control (read/write, size, action)

FUNCTION Register Options (see ARMv7-M TRM):

- FUNCTION[3:0] = Match mode (data read/write, PC match, etc.)
- DATAVSIZE[1:0] = Data size (byte, halfword, word)
- Action: Generate debug event or emit trace packet

Additional DWT Features:

- **DWT_CYCCNT:** Cycle counter (free-running 32-bit)
- **DWT_CTRL:** Enable cycle counter, read NUMCOMP

ARM
Data Watchpoints - Practical Example

Watch a variable for writes:

```
volatile uint32_t critical_var = 0;

void setup_watchpoint(void) {
    // Enable TRCENA
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;

    // Comparator 0: Watch critical_var
    DWT->COMP0 = (uint32_t)&critical_var;
    DWT->MASK0 = 0;  // Exact match

    // FUNCTION: Use CMSIS symbols or see ARMv7-M TRM Table B3-11
    // Example: Data write match (exact bits depend on implementation)
    // Refer to: interrupt.memfault.com/blog/cortex-m-watchpoints
    DWT->FUNCTION0 = (1 << 4) |  // DATAVMATCH
    (1 << 0);    // FUNCTION bits (consult TRM)
}

int main(void) {
    setup_watchpoint();

    critical_var = 42;  // Triggers watchpoint!
    while(1);
}
```

ARM

Flash Patch and Breakpoint Unit (FPB)

The FPB allows setting breakpoints in code memory and optionally patching code execution to RAM.

Two Main Functions:

- **Breakpoints:** Instruction comparators (implementation dependent)
- **Code Patching:** Remap to RAM (if RMPSPT flag supported)

Implementation Dependent (read FP_CTRL):

- NUM_CODE: Instruction comparators (typically 6 on Cortex-M3)
- NUM_LIT: Literal comparators (typically 2)
- RMPSPT: Remap support flag (not always present)

Note: FPB is a debug feature, not intended for field updates. Code region base depends on MCU memory map (e.g., STM32:

ARM
Flash Patch - Configuration

Key FPB Registers:

- **FP_CTRL:** Enable FPB, read NUM_CODE/NUM_LIT/RMPSPT
- **FP_COMPn:** Comparator n configuration
- **FP_REMAP:** Base address of replacement code in RAM

Comparator Configuration (FP_COMPn):

- Bit [0]: ENABLE (1 = active)
- Bits [1:0]: REPLACE field (2-bit, see ARMv7-M TRM)
  - 00 = Remap to RAM (if RMPSPT supported)
  - 01/10/11 = Breakpoint on instruction/halfword
- Bits [31:2]: Code address bits [31:2] (word-aligned)

References:

- ARMv7-M Architecture Reference Manual (Section C1.11)
- Doulos Application Note on FPB

ARM
Flash Patch - Practical Example

Patch a buggy function in Flash:

```c
// Buggy function in Flash
void buggy_function(void) {
    // Contains a bug
}

// Fixed version in RAM
__attribute__((section(".data")))
void fixed_function(void) {
    // Corrected implementation
}

void setup_flash_patch(void) {
    // Check if remap is supported
    if (!(FP->CTRL & FP_CTRL_KEY_Msk)) return;

    // Enable FPB
    FP->CTRL |= FP_CTRL_ENABLE_Msk | FP_CTRL_KEY_Msk;

    // Set remap base (SRAM start, shifted right by 5)
    FP->REMAP = 0x20000000 >> 5;

    // Comparator 0: Remap buggy_function to fixed_function
    // REPLACE = b00 for remap, ENABLE = 1
    uint32_t flash_addr = (uint32_t)buggy_function;
    FP->COMP[0] = (flash_addr & ~0x3u) | 1u;  // ENABLE=1, REPLACE=00
}
```

ARM

Debug Features - Quick Comparison

Feature	Purpose	Overhead
SWV/ITM	Printf, trace output	Low (FIFO may block)
DWT	Data watchpoints, profiling	Minimal
FPB	Breakpoints, code patching	Debug-only

Resource Limits (implementation dependent):

- ITM: 32 stimulus ports
- DWT: Read DWT_CTRL.NUMCOMP (typically 4)
- FPB: Read FP_CTRL.NUM_CODE/NUM_LIT (typically 6+2)

Common Debugger Support:

- J-Link: Full support for all features
- ST-Link: SWV supported, limited FPB
- OpenOCD: Basic SWV, DWT support

CMSIS vs HAL

CMSIS (Cortex Microcontroller Software Interface Standard)

```
// Direct register access
RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;  // Enable clock
GPIOC->BSRR = GPIO_BSRR_BR13;        // Set pin to LOW
```

- Vendor: ARM (Cortex-M standard)
- Abstraction level: Low (register-level via CMSIS-Device headers)
- What is it?: Standardized core intrinsics + device header mapping (names/addresses)

CMSIS vs HAL

HAL (Hardware Abstraction Layer)

```
// Function calls
__HAL_RCC_GPIOC_CLK_ENABLE();              // Enable clock
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET);
    // Set pin to LOW
```

- Vendor: STMicroelectronics (only for STM32)
- Abstraction level: High (hides registers)
- What is it?: Convenient function library

CMSIS vs HAL

Turning LED on - Both approaches:

CMSIS:

```
RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;
GPIOC->CRH &= ~(GPIO_CRH_CNF13 | GPIO_CRH_MODE13);
GPIOC->CRH |= GPIO_CRH_MODE13_1;
GPIOC->BSRR = GPIO_BSRR_BR13;
```

- 4 lines of code
- You need to understand registers
- Fast (direct)

CMSIS vs HAL

Turning LED on - Both approaches:

HAL:

- 7+ lines of code
- Readable and self-explanatory
- Slower (many function calls)

HAL typically requires more lines of code due to function calls and initializations.

CMSIS vs HAL

Aspect	CMSIS / Register	HAL (STM32)
Performance	typically 3–10× faster*	slower (wrapper calls)
Code size (blink)	∼3–6 KB	∼15–25 KB
Learning curve	steeper	flatter
Readability	terse / technical	very readable
Portability	vendor-agnostic concepts; device headers per family	portable within STM32 family
Debugging	transparent (no hidden layers)	black-boxy at times
Control	100%	limited by API

* Can be much more in tight loops/ISRs, depends on inlining and wait states.

CMSIS vs HAL

Why should you learn CMSIS?

Understand how hardware REALLY works.

```
// HAL hides the magic
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET);

// CMSIS shows you the reality
GPIOC->BSRR = GPIO_BSRR_BR13;  // Set bit 29 in register
    0x4001100C
```

With CMSIS you understand that you're setting a specific bit in a hardware register. **With HAL you're just calling a function.**

CMSIS vs HAL

Performance in Time-Critical Applications

```
// HAL: ~50-100 CPU cycles
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);

// CMSIS: ~2-3 CPU cycles
GPIOC->BSRR = GPIO_BSRR_BS13;
```

At 72 MHz: HAL takes 1.4 ţs, CMSIS only 0.04 ţs - 35x faster!

Important for: Fast PWM, Bit-banging (WS2812 LEDs, OneWire), Interrupt Service Routines and real-time protocols

Exact cycle counts depend on compiler optimization, inlining and bus wait states. Rule of thumb: CMSIS/register-level calls are typically an order of magnitude faster than HAL wrappers.

CMSIS vs HAL

Smaller Code = More Space for Your Program

```
STM32F103C6: 32 KB Flash

HAL project:    HAL library  1525 KB  Leaves ~717 KB for
    your code
CMSIS project:  CMSIS only  36 KB    Leaves ~2629 KB for
    your code
```

Note: The exact size depends on which HAL modules are linked. Even small HAL-based projects often use significantly more Flash due to abstraction layers and initialization code.

CMSIS vs HAL
Understanding Other MCUs

When you switch to other manufacturers (ESP32, Nordic nRF, Raspberry Pi Pico), there is no STM32 HAL. Each vendor provides its own SDK (e.g., ESP-IDF, nRF5). But the register-level principle remains the same.

```
// STM32 (CMSIS)
GPIOC->BSRR = GPIO_BSRR_BS13;
// ESP32 (IDF)
GPIO.out_w1ts = (1 << 13);
// nRF52 (Nordic)
NRF_GPIO->OUTSET = (1 << 13);
```

CMSIS vs HAL
Jobs and Industry

In industry among professional embedded developers:

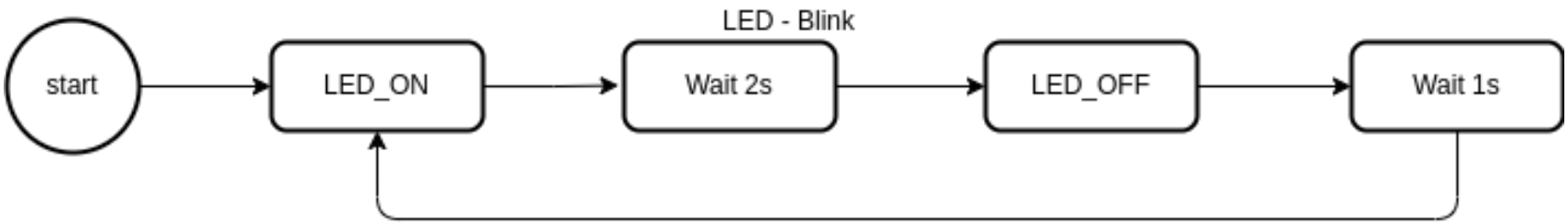
HAL: 20% (prototyping, quick projects)
CMSIS/Register-Level: 80% (production, performance)

Why? Because:

- Firmware must be small (cheaper MCUs)
- Firmware must be fast (real-time requirements)
- Developers must understand hardware (troubleshooting)

Practice
Blink Test

Just as the "Hello World" project is commonly used in pure software programming, here a Blink project is described. This project makes an arbitrary LED blink using an ARM microcontroller. In my case, a **STM32F103C6T6A**.

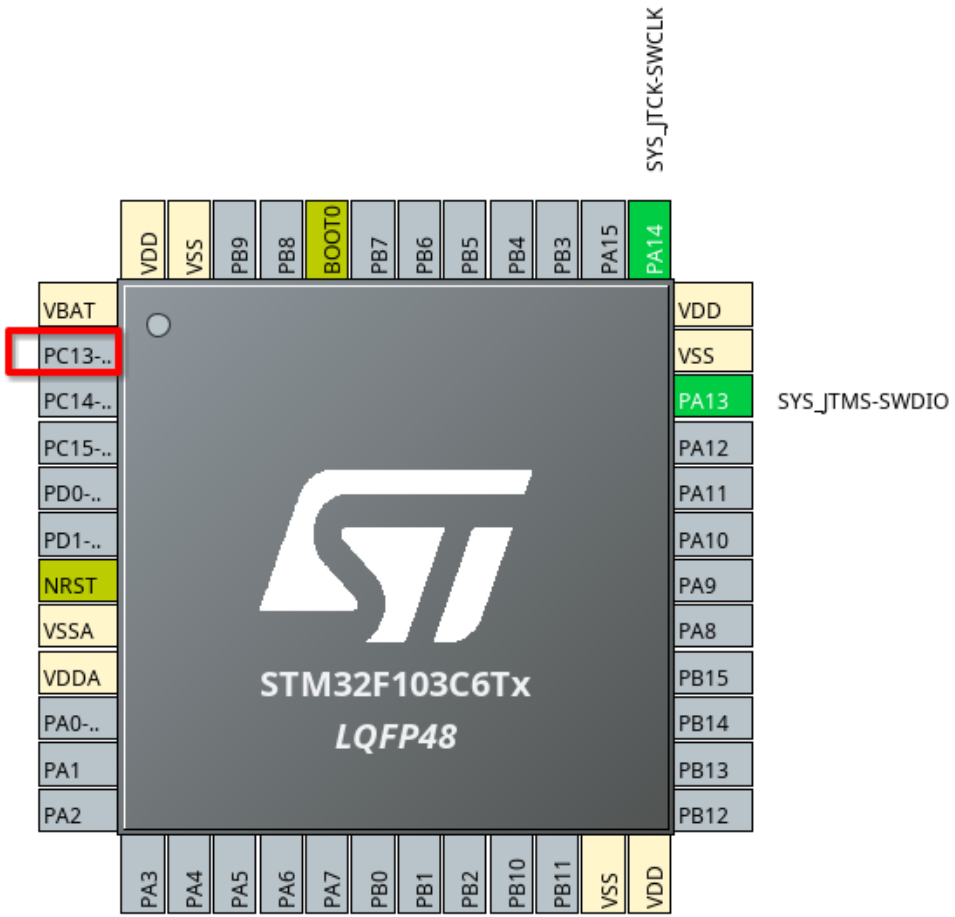


Practice
Blink Test

The program flow is simple: The LED is turned on, this state is maintained for 2 seconds, then the LED is turned off and the processor waits for 1 second before the LED turns on again. This sequence is then executed continuously through a loop.

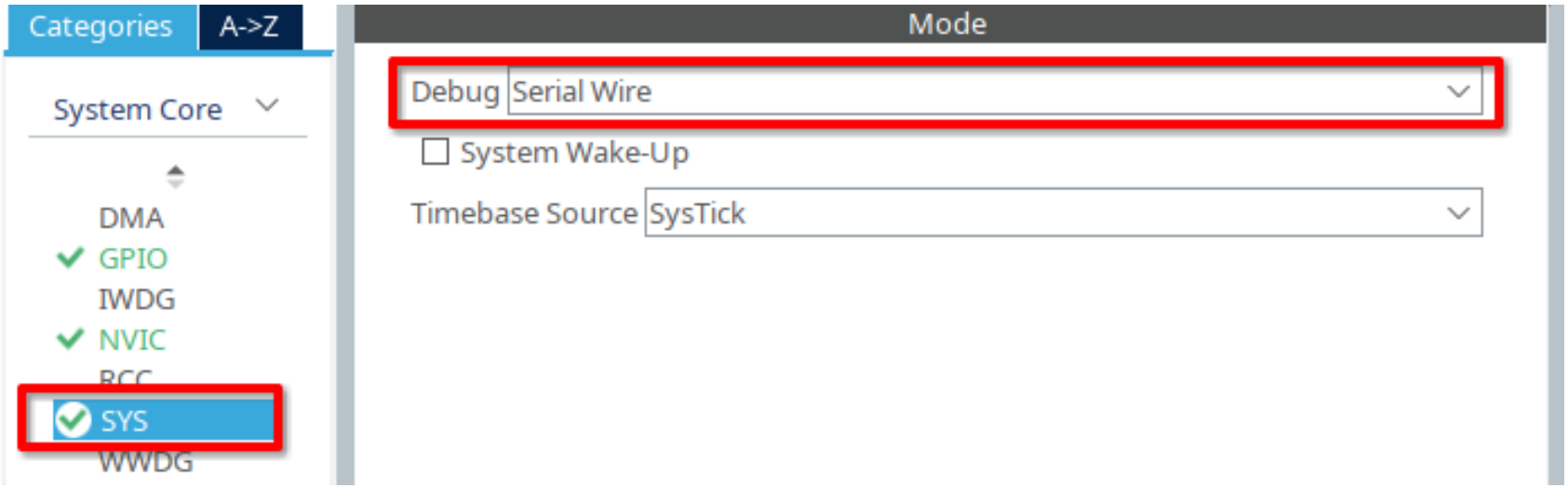
In my case, the LED is controlled via pin PC13. Flashing and debugging is done using an ST-Link, therefore in CubeIDE the debug parameter must be set to Serial Wire.

Practice
CubeIDE Configuration



The pin **PC13** is configured directly in the code.

Practice
CubeIDE Configuration



Practice
Useing HAL

Although HAL is not the main topic here, its interesting to see how the same project looks using the HAL library.

Implementation with HAL

```
int main(void)
    {
        HAL_Init();
        SystemClock_Config();
        MX_GPIO_Init();
        while (1)
        {
            ....
        }
    }
```

The GPIO configuration is implemented directly by the HAL library, whereas in CMSIS you have to configure the registers yourself.

Implementation with HAL

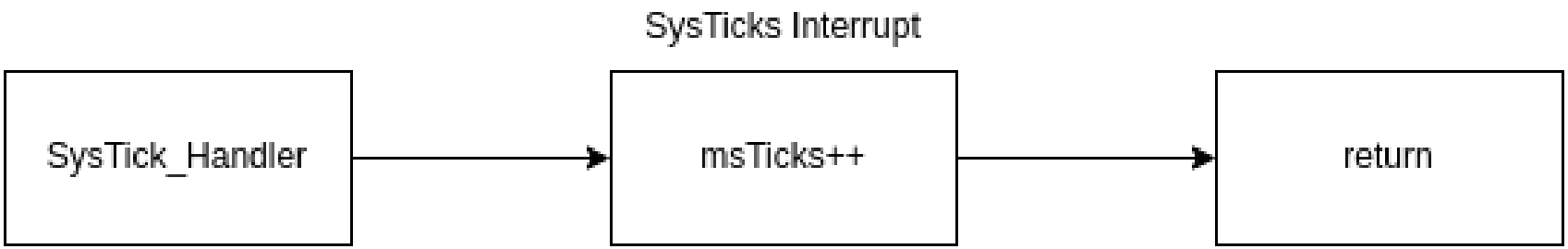
```
while (1)
    {
        // LED ON for 2 seconds (active-low:
    GPIO_PIN_RESET)
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13,
    GPIO_PIN_RESET);
        HAL_Delay(2000);

        // LED OFF for 1 second (active-high:
    GPIO_PIN_SET)
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13,
    GPIO_PIN_SET);
        HAL_Delay(1000);
    }
```

Delay and WritePin functions are also provided by the HAL library, the code closely resembles Arduino code.

Practice
Functional Architecture and Logic

For the LED to blink (toggle), the MCU must know exactly when it should be ON, how long it should be ON, and when the LED must be turned OFF. For this, a delay function implemented with the SysTick_Handler is used.

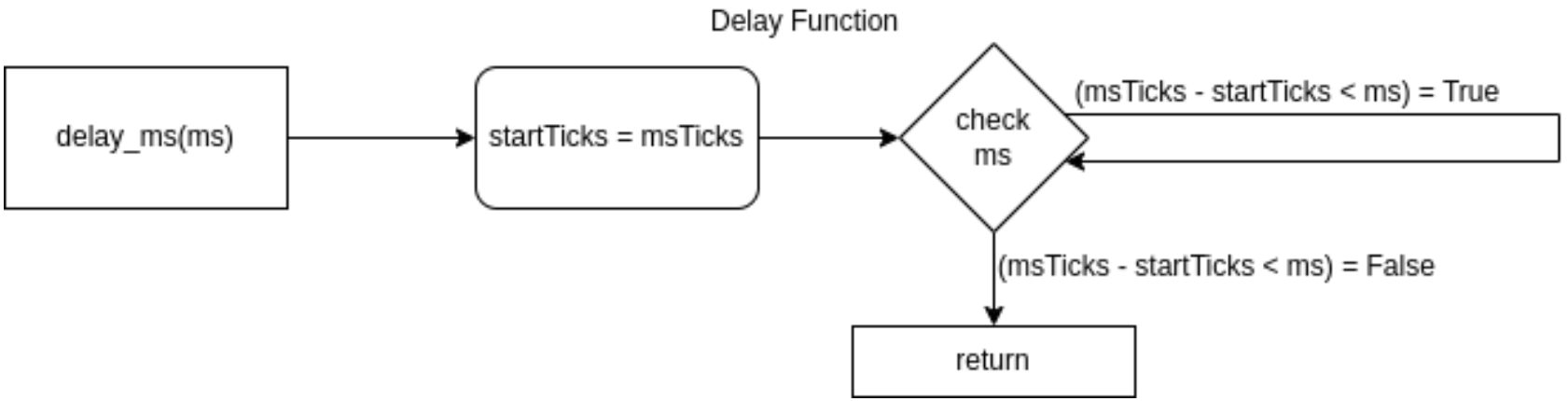


Practice
Functional Architecture and Logic

SysTick_Handler: The SysTick is a timer in the processor that triggers an interrupt at regular intervals (here every millisecond). It's like an alarm clock that rings every millisecond. When the alarm rings, the **SysTick_Handler** function is called. This function does only one thing: it increments the counter **msTicks** by 1.

Interrupt: Imagine you are a teacher grading exams (that's your main program). Suddenly the telephone rings (that's the interrupt). You interrupt the grading, answer the phone and talk to the caller (that's the interrupt handler). When the call is finished, you return to grading and continue exactly where you left off.

Practice
Functional Architecture and Logic



The system utilizes a delay function that waits for milliseconds. It is based on a global counter **msTicks** that is incremented every millisecond by the SysTick interrupt.
The function stores the current value of **msTicks** at the beginning (**startTicks**). It then waits in a loop until the difference between the current **msTicks** and **startTicks** becomes greater than or equal to the desired **ms** value.

Practice
Functional Architecture and Logic

```
void delay_ms(uint32_t ms){
    uint32_t startTicks = msTicks;
    while ((msTicks - startTicks) < ms);
}
```

The msTicks counter overflows after approximately 49 days from 4,294,967,295 to 0, but the calculation (msTicks - startTicks) still functions correctly because subtraction with unsigned integers always produces the correct result.

This approach utilizes the hardware timer (not "estimated" waiting), ensuring that 1000ms are exactly 1000ms.

Practice
Functional Architecture and Logic

Why uint32_t:
System independence: If one uses int, the compiler might allocate different memory sizes (e.g., 2 bytes on a 16-bit system vs. 4 bytes on a 32-bit system). uint32_t ensures that the data type is always exactly 32 bits, regardless of the architecture.

Hardware registers: When programming microcontrollers such as an STM32, one often needs to work with hardware registers that have a fixed bit width (e.g., 32 bits). uint32_t fits perfectly and facilitates bit manipulation.

Practice
Functional Architecture and Logic

This example utilizes the internal clock source.

```
void SystemClock_Config(void){
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    // HSI as oscillator
    RCC_OscInitStruct.OscillatorType =
    RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    // No PLL used
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
}
```

Practice
Functional Architecture and Logic

- Configures the internal 8MHz oscillator (HSI)
- No clock division for buses
- Simplest clock configuration

```
SysTick_Config(SystemCoreClock / 1000);
```

Configures the SysTick timer for an interrupt every 1ms. SystemCoreClock / 1000 divides the CPU clock frequency by 1000 for 1ms intervals. During each interrupt, msTicks is incremented.

Practice
Functional Architecture and Logic

What happens if the clock configuration is not performed or is performed incorrectly?
SystemClock_Config():
This function configures the system clock. Without it, the microcontroller runs with the default clock (e.g., internal HSI oscillator), but potentially not at the expected frequency.

Specifically, it enables the HSI (Internal High-Speed Clock) and sets the clock frequency for the CPU and peripherals.

Without clock configuration, the clock might be too slow or might not run at all, causing the microcontroller to malfunction.

Practice
Functional Architecture and Logic

SysTick_Config():
This function configures the SysTick timer, which is required for the delay_ms function. Without it, msTicks is not incremented, and the delay function would wait indefinitely.

GPIO_LED_Init():
This function activates the clock for GPIOC via RCC->APB2ENR |= RCC_APB2ENR_IOPCEN. For the GPIOC clock to be activated, the RCC (Reset and Clock Control) module itself must be properly clocked. This is ensured by SystemClock_Config(). Without the system clock, setting the RCC_APB2ENR_IOPCEN bit might have no effect because the RCC module is not operational.

Practice
Functional Architecture and Logic

Summary

Without clock configuration, the microcontroller remains in a reset state or operates with an unconfigured clock, resulting in no peripherals (including GPIO) functioning. The GPIO initialization assumes that the system clock has already been configured.

Practice
Functional Architecture and Logic

```
void GPIO_LED_Init(void){
// Enable clock
RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;

// Configure pin
GPIOC->CRH &= ~(GPIO_CRH_CNF13 | GPIO_CRH_MODE13);
GPIOC->CRH |= GPIO_CRH_MODE13_1;

// Turn off LED initially
GPIOC->BSRR = GPIO_BSRR_BS13;
}
```

Practice
Functional Architecture and Logic

Enable GPIOC clock (RCC_APB2ENR)

`RCC->APB2ENR |= RCC_APB2ENR_IOPCEN`

Enables the peripheral clock for **GPIO port C**. The bit lives in the STM32 reference manual under *RCC_APB2ENR APB2 peripheral clock enable register* (field **IOPCEN**).

The compound operator `|=` performs a *readmodifywrite*:

1. Read the current value of `RCC->APB2ENR`.
2. OR it with the mask `RCC_APB2ENR_IOPCEN`.
3. Write the result back to `RCC->APB2ENR`.

Note: `RCC_APB2ENR_IOPCEN` equals `(1u << 4)` (i.e., only bit 4 set).

Practice
Functional Architecture and Logic

Why not use =?

The advantage of `|=` is that it **sets exactly one bit** and **leaves all other bits untouched**.

If you wrote `RCC->APB2ENR = RCC_APB2ENR_IOPCEN` you would:

- enable GPIOC,
- **but clear every other bit** in the register disabling any APB2 peripheral that was previously enabled (AFIO, GPIOA/B, ADC1, ...).

Conclusion:

- use `|=` to **set** bits,
- use `&= ~mask` to **clear** bits (e.g., `RCC->APB2ENR &= ~RCC_APB2ENR_IOPCEN;`).

Practice
Functional Architecture and Logic

About the macro

`RCC_APB2ENR_IOPCEN` is a descriptive macro provided by STMicroelectronics in the device header. It expands to the bit mask `(1u << 4)`, i.e., only bit 4 is set.

Therefore the following are **equivalent**:

- `RCC->APB2ENR |= (1u << 4);`
- `RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;`

Using the named macro is **clearer** and often more **portable** across STM32 families, where bit positions may differ.

Good practice: enable the port clock *before* writing any GPIO registers for that port.

Practice
Bit-Masking: Clear Bits Operation

Why &= ~(...)?
This is a "clear bits" step:

- `GPIO_CRH_CNF13 | GPIO_CRH_MODE13` creates a mask covering all 4 bits of Pin 13 (CNF + MODE).
- `~(...)` inverts the mask → ones everywhere, except these 4 bits (there 0).
- `&=` with this inverted mask sets exactly these 4 bits to 0, leaving all other bits unchanged.

Purpose: Clear the 4 configuration bits of PC13 so no old state remains.

```
GPIOC->CRH &= ~(0xFu << 20);
```

Practice

Pin Configuration: CNF and MODE Bits

Each GPIO pin uses 4 bits: [CNF1 CNF0 MODE1 MODE0]

MODE[1:0] (outputs speed / input select):

- 00 = Input mode
- 01 = Output 10 MHz
- 10 = Output 2 MHz
- 11 = Output 50 MHz

CNF[1:0] (depends on MODE):

- **If MODE=00 (input):** 00=Analog, 01=Floating, 10=Pull-up/Down, 11=Reserved
- **If MODE≠00 (output):** 00=General-Purpose Push-Pull, 01=General-Purpose Open-Drain, 10=Alternate-Function Push-Pull, 11=Alternate-Function Open-Drain

Practice
GPIO Reset State & Safe Pattern

After reset:

- All GPIOs: `MODE=00` (Input), `CNF=01` (Floating input)
- Debug pins (JTAG/SWD) are mapped to debug by default (free via `AFIO_MAPR`)

Why this matters:

- Setting only `MODE` leaves `CNF` as-is.
- Example: `MODE=10` with leftover `CNF=01` ⇒ **general-purpose open-drain output** (not invalid, but often unintended for LEDs; cannot actively drive HIGH without pull-up).

Safe approach (clean-then-set):

- `GPIOC->CRH &= ~(0xFu << 20);` // clear CNF:MODE for PC13
- `GPIOC->CRH |= (0x2u << 20);` // CNF=00, MODE=10 ⇒ Output 2 MHz push-pull

Key takeaway: Always `Clock` → `Clear` → `Set`.

Practice
Setting MODE Bits Risky Approach

Make PC13 an Output @ 2 MHz (keep CNF unchanged)

```
GPIOC->CRH |= GPIO_CRH_MODE13_1;
```

What this does:

- `GPIO_CRH_MODE13_1 = 0x00200000` sets bit 21
- Result: `MODE13 = 10b` Output 2 MHz
- **Problem:** CNF bits are NOT modified!

Why is this risky?

- After reset: CNF=01, MODE=00
- After this line: CNF=01, MODE=10 invalid config!

Safe only if: CNF was already set correctly before.

Practice

Best Practice: Clean-then-Set Pattern

Recommended approach (always safe):

```
// Clear all 4 bits (CNF + MODE)
GPIOC->CRH &= ~(0xFu << 20);

// Set new configuration
GPIOC->CRH |= (0x2u << 20);
```

What happens:

- `0xFu << 20 = 0x00F00000` masks bits 23..20
- Clear these 4 bits, keep all others
- `0x2u << 20 = 0x00200000` sets CNF=00, MODE=10

Calculation: PC13 in CRH at $(13 - 8) \times 4 = 20$

Practice

Complete Example: PC13 Configuration

Configure PC13 as Output Push-Pull, 2 MHz

```
// 1. Enable GPIO Clock (CRITICAL!)
RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;

// 2. Clear PC13 config (bits 23..20)
GPIOC->CRH &= ~(0xFu << 20);

// 3. Set: CNF=00, MODE=10 (2 MHz)
GPIOC->CRH |= (0x2u << 20);
```

Three essential steps:

1. Enable Clock without this, writes are ignored!
2. Clear remove old configuration
3. Set apply new configuration

Practice
Bit Position Calculation

General Formula:

`value = ((CNF « 2) | MODE) « offset`

Offset Calculation:

- **CRL (pins 0..7):** `offset = pin_number * 4`
- **CRH (pins 8..15):** `offset = (pin_number - 8) * 4`

Examples:

- PA3: CRL, offset $= 3 \times 4 = 12$
- PC13: CRH, offset $= (13 - 8) \times 4 = 20$
- PB15: CRH, offset $= (15 - 8) \times 4 = 28$

PA3 Output PP, 50 MHz:

```
GPIOA->CRL &= ~(0xFu « 12);
GPIOA->CRL |= (0x3u « 12);
```

Reference
Quick Reference

CNF	Output (MODE≠00)	Input (MODE=00)
00	Push-Pull	Analog
01	Open-Drain	Floating
10	AF Push-Pull	Pull-up/-down
11	AF Open-Drain	Reserved

Common Pitfalls:

- Forgetting to enable GPIO clock
- Not clearing CNF+MODE before setting
- Assuming pins are in known state after reset

Key Takeaway: Always `Clock  Clear  Set`

Practice
GPIO Bit Set/Reset Register (BSRR)

What is BSRR?

- **B**it **S**et/**R**eset **R**egister
- 32-bit write-only register
- Allows atomic set and reset of GPIO pins
- No read-modify-write needed interrupt-safe!

Register Layout (32 bits):

```
[BR15..BR0][BS15..BS0]
 [31....16][15.....0]
```

Two sections:

- **Bits 0-15 (BSy):** Set bit y to HIGH (1)
- **Bits 16-31 (BRy):** Reset bit y to LOW (0)

Practice
BSRR Operation Principle

Key concept: Write 1 to trigger action, 0 is ignored

To SET a pin HIGH:

- Write 1 to corresponding BS bit (bits 0-15)
- Example: Set PC13 write 1 to bit 13

To RESET a pin LOW:

- Write 1 to corresponding BR bit (bits 16-31)
- Example: Reset PC13 write 1 to bit 29 (16+13)

Multiple pins simultaneously:

- Can set and reset different pins in same write
- Example: `GPIOC->BSRR = (1<<13) | (1<<(16+5));`
- Sets PC13 HIGH, resets PC5 LOW atomically!

Practice
Setting PC13 to HIGH

Set PC13 to logical 1 (HIGH):

```
GPIOC->BSRR = GPIO_BSRR_BS13;
```

What happens:

- `GPIO_BSRR_BS13 = 0x00002000` (bit 13 set)
- Writes 1 to bit 13 of BSRR
- Hardware sets PC13 output to HIGH
- All other pins remain unchanged

Bit calculation:

- Pin 13 BS13 bit position 13
- $2^{13} = 8192 = 0x2000$
- Binary: `0000_0000_0000_0000_0010_0000_0000_0000`

Practice
Resetting PC13 to LOW

Reset PC13 to logical 0 (LOW):

```
GPIOC->BSRR = GPIO_BSRR_BR13;
```

What happens:

- `GPIO_BSRR_BR13 = 0x20000000` (bit 29 set)
- Writes 1 to bit 29 of BSRR (16 + 13)
- Hardware resets PC13 output to LOW
- All other pins remain unchanged

Bit calculation:

- Pin 13 BR13 bit position 16+13 = 29
- $2^{29} = 536870912 = 0x20000000$
- Binary: 0010_0000_0000_0000_0000_0000_0000_0000

Practice
BSRR vs ODR: Why use BSRR?

Alternative: Using ODR (Output Data Register)

```
GPIOC->ODR |= (1 << 13);   // Set HIGH
GPIOC->ODR &= ~(1 << 13);  // Reset LOW
```

Problem with ODR: Read-Modify-Write (RMW)

- Read Modify Write (3 steps)
- Not atomic race condition risk!

Advantage of BSRR: Single write operation

- Atomic interrupt-safe
- Faster (1 instruction vs 3+)
- No critical section needed

Practice
Practical BSRR Examples

Toggle LED on PC13:

```
// LED ON (active-low)
GPIOC->BSRR = GPIO_BSRR_BR13;

// LED OFF
GPIOC->BSRR = GPIO_BSRR_BS13;
```

Multiple pins simultaneously:

```
// Set PC13 HIGH, reset PC14 LOW
GPIOC->BSRR = GPIO_BSRR_BS13 | GPIO_BSRR_BR14;
```

Why this is powerful:

- Both operations in single write cycle
- No intermediate states
- Perfect for synchronized outputs

Practice
Common BSRR Usage Patterns

Pattern 1: Simple pin control

```
#define LED_ON()  GPIOC->BSRR = GPIO_BSRR_BR13
#define LED_OFF() GPIOC->BSRR = GPIO_BSRR_BS13
```

Pattern 2: Conditional set/reset

```
if (condition) {
    GPIOC->BSRR = GPIO_BSRR_BS13;
} else {
    GPIOC->BSRR = GPIO_BSRR_BR13;
}
```

Pattern 3: Clock pulse

```
GPIOC->BSRR = GPIO_BSRR_BS13;  // HIGH
GPIOC->BSRR = GPIO_BSRR_BR13;  // LOW
```

Reference
BSRR Quick Reference

BSRR Bit Layout:

Bits	Name	Function
0-15	BSy	Set pin y to HIGH (write 1)
16-31	BRy	Reset pin y to LOW (write 1)

Quick formulas:

- Set pin n HIGH: `BSRR = (1 « n)`
- Reset pin n LOW: `BSRR = (1 « (16+n))`

Best Practices:

- Always prefer BSRR over ODR for pin control
- Use CMSIS macros (GPIO_BSRR_BSn/BRn) for clarity
- BSRR is write-only reading returns 0
- Setting both BS and BR for same pin: BR has priority

When to use ODR: Only when reading current output state is needed

Practice
Complete LED Blink Example

Goal: Blink LED on PC13 with 1 second intervals

```
        while (1) {
            // LED On (active-low)
            GPIOC->BSRR = (1u << (13+16));
            delay_ms(1000);

            // LED Off
            GPIOC->BSRR = (1u << 13);
            delay_ms(1000);
        }
```

What this does:

- Toggles PC13 between LOW and HIGH
- 1 second ON, 1 second OFF
- Assumes LED is active-low (common on STM32 boards)

Practice
Understanding the Bit Calculations

LED On: Reset PC13 to LOW

```
GPIOC->BSRR = (1u << (13+16));
```

Calculation:

- 13+16 = 29 → BR13 bit position
- 1u « 29 = 0x20000000
- Sets bit 29 in BSRR → resets PC13 to LOW

LED Off: Set PC13 to HIGH

```
GPIOC->BSRR = (1u << 13);
```

Calculation:

- 1u « 13 = 0x00002000
- Sets bit 13 in BSRR → sets PC13 to HIGH

Practice

Manual Calculation vs CMSIS Macros

The code uses manual bit shifts:

```
GPIOC->BSRR = (1u << (13+16));  // Reset
GPIOC->BSRR = (1u << 13);       // Set
```

Equivalent using CMSIS macros:

```
GPIOC->BSRR = GPIO_BSRR_BR13;  // Reset
GPIOC->BSRR = GPIO_BSRR_BS13;  // Set
```

Which is better?

- **CMSIS macros:** More readable, self-documenting
- **Manual shifts:** Shows understanding of register layout
- **Recommendation:** Use CMSIS macros in production code

Both produce identical machine code!

Practice
Understanding Active-Low LEDs

Why reset (LOW) turns LED ON?

Active-Low Configuration:

- LED cathode connected to PC13
- LED anode connected to VCC (3.3V)
- Current flows when pin is LOW

`VCC (3.3V)  LED  PC13  GND`

Logic:

- PC13 = LOW (0V) Current flows LED **ON**
- PC13 = HIGH (3.3V) No voltage difference LED **OFF**

Common on: Blue Pill, Black Pill, STM32 Discovery boards

Practice

The delay_ms() Function

Purpose: Create blocking delay in milliseconds

Typical implementation (busy-wait):

```
void delay_ms(uint32_t ms) {
    for (uint32_t i = 0; i < ms; i++) {
        for (volatile uint32_t j = 0; j < 8000; j++) {
            // Empty loop (CPU frequency dependent)
        }
    }
}
```

Characteristics:

- Blocks CPU (wastes power)
- Accuracy depends on CPU frequency
- Simple but inefficient

Better alternatives: SysTick timer, HAL_Delay(), RTOS delays

Practice
Complete Blink Program

```c
#include "stm32f1xx.h"

void delay_ms(uint32_t ms);

int main(void) {
    // Enable GPIOC clock
    RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;

    // Configure PC13: Output Push-Pull, 2MHz
    GPIOC->CRH &= ~(0xFu << 20);
    GPIOC->CRH |= (0x2u << 20);

    while (1) {
        GPIOC->BSRR = (1u << (13+16));  // LED On
        delay_ms(1000);

        GPIOC->BSRR = (1u << 13);       // LED Off
        delay_ms(1000);
    }
}

void delay_ms(uint32_t ms) {
    for (uint32_t i = 0; i < ms; i++) {
        for (volatile uint32_t j = 0; j < 8000; j++);
    }
}
```

Practice
Program Flow Analysis

Initialization Phase:

1. Enable GPIOC clock via RCC
2. Configure PC13 as output (Push-Pull, 2MHz)

Main Loop (infinite):

1. Write to BSRR Reset PC13 (LED ON)
2. Wait 1000ms
3. Write to BSRR Set PC13 (LED OFF)
4. Wait 1000ms
5. Repeat forever

Timing:

- Total period: 2000ms (2 seconds)
- Frequency: 0.5 Hz
- Duty cycle: 50% (equal ON/OFF time)

Practice
Common Mistakes & Improvements

Common Mistakes:

```
// Mistake 1: No clock enable
GPIOC->CRH |= (0x2u << 20);  // Ignored!

// Mistake 2: Using ODR (not optimal)
GPIOC->ODR &= ~(1 << 13);  // RMW operation
```

Improvements:

```
// Better: Use HAL_Delay() or SysTick
HAL_Delay(1000);

// Best: Non-blocking with timer ISR
void TIM2_IRQHandler(void) {
    GPIOC->ODR ^= (1 << 13);
}
```

Practice

Debugging: LED Not Blinking?

Checklist when LED doesn't work:

1. **Clock Configuration:**
   - Is GPIOC clock enabled? Check `RCC->APB2ENR`
   - Read back register to verify
2. **Pin Configuration:**
   - Is PC13 configured as output? Check `GPIOC->CRH`
   - Correct MODE bits (not 00)?
   - Correct CNF bits (00 for Push-Pull)?
3. **Pin State:**
   - Read `GPIOC->ODR` to check output state
   - Is pin toggling? Use debugger or logic analyzer
4. **Hardware:**
   - LED polarity correct?
   - Current-limiting resistor present?
   - Pin not used by debugger (SWD/JTAG)?