ARM-Embedded-Path

CMSIS-Basics

Pavel Pys

October 24, 2025

Overview

- Introduction
- ARM-Architecture
- Practice

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.
- ST-Microelectronics
- ...

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?

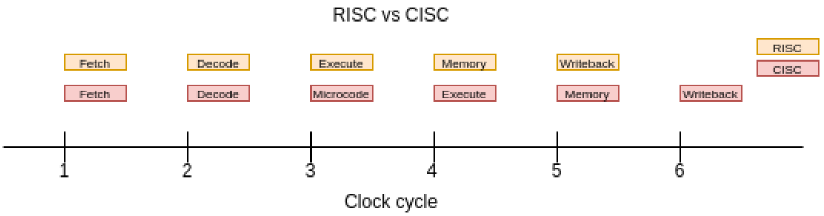
A microcontroller architecture describes the internal structure and functionality of a microcontroller – meaning how the individual components on the chip are interconnected and how they work together.

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:

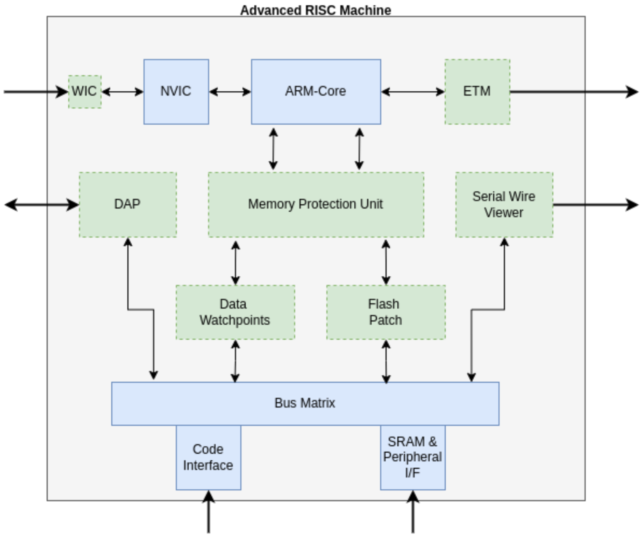
- ALU (Arithmetic Logic Unit)
- Register set (R0-R15)
- Program Counter, Stack Pointer
- Interrupt Controller
- Bus interfaces (AHB, APB...)

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, I^{2}C 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) $\rightarrow$ enables parallel reading of instructions and data

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

NVIC Integration:
Interrupt handling directly in the core $\rightarrow$ 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.

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 (STM**
Performance	typically 3–10$\times$ faster[*]	slower (wrap
Code size (blink)	$\sim$3–6 KB	$\sim$15–25 KB
Learning curve	steeper	flatter
Readability	terse/technical	very readabl
Portability	vendor-agnostic concepts; device headers per family	portable wit
Debugging	transparent (no hidden layers)	black-boxy a
Control	100%	limited by A

Table: CMSIS vs HAL (rule-of-thumb; depends on modules/optimizations)

[*] 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

```c
// 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 µs, CMSIS only 0.04 µs - 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.

```c
// 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:

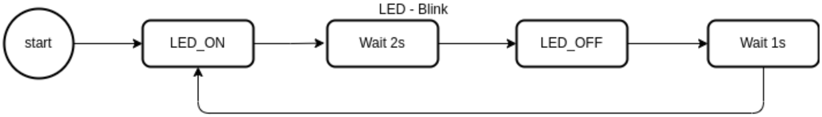
HAL: 20% (prototyping, quick projects)
CMSIS/Register-Level: 80% (production, performance)

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**.

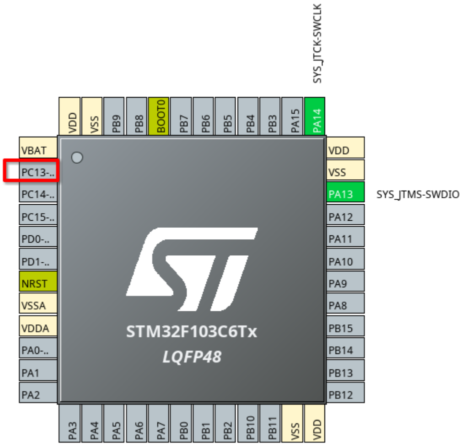


Practice
Blink Test

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.

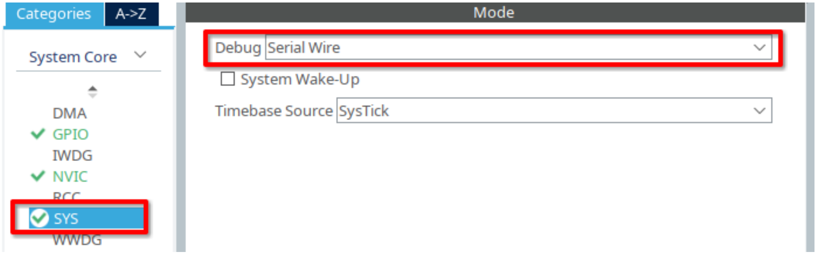
Practice
CubeIDE Configuration



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

Practice

CubeIDE Configuration



Debug: Serial Wire

☐ System Wake-Up

Timebase Source: SysTick

Practice
Useing HAL

Although HAL is not the main topic here, it's 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

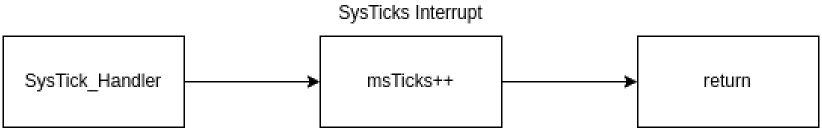
```
while (1)
    {
        // LED ON for 2 seconds (active-low:
    GPIO_PIN_RESET)
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13,
    GPIO_PIN_RESET);
        HAL_Delay(2000);

        // 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.

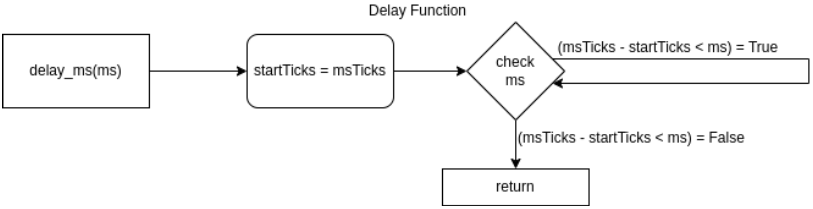


Practice
Functional Architecture and Logic

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


Delay Function
delay_ms(ms)
startTicks = msTicks
check ms
(msTicks - startTicks < ms) = True
(msTicks - startTicks < ms) = False
return

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

```c
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.