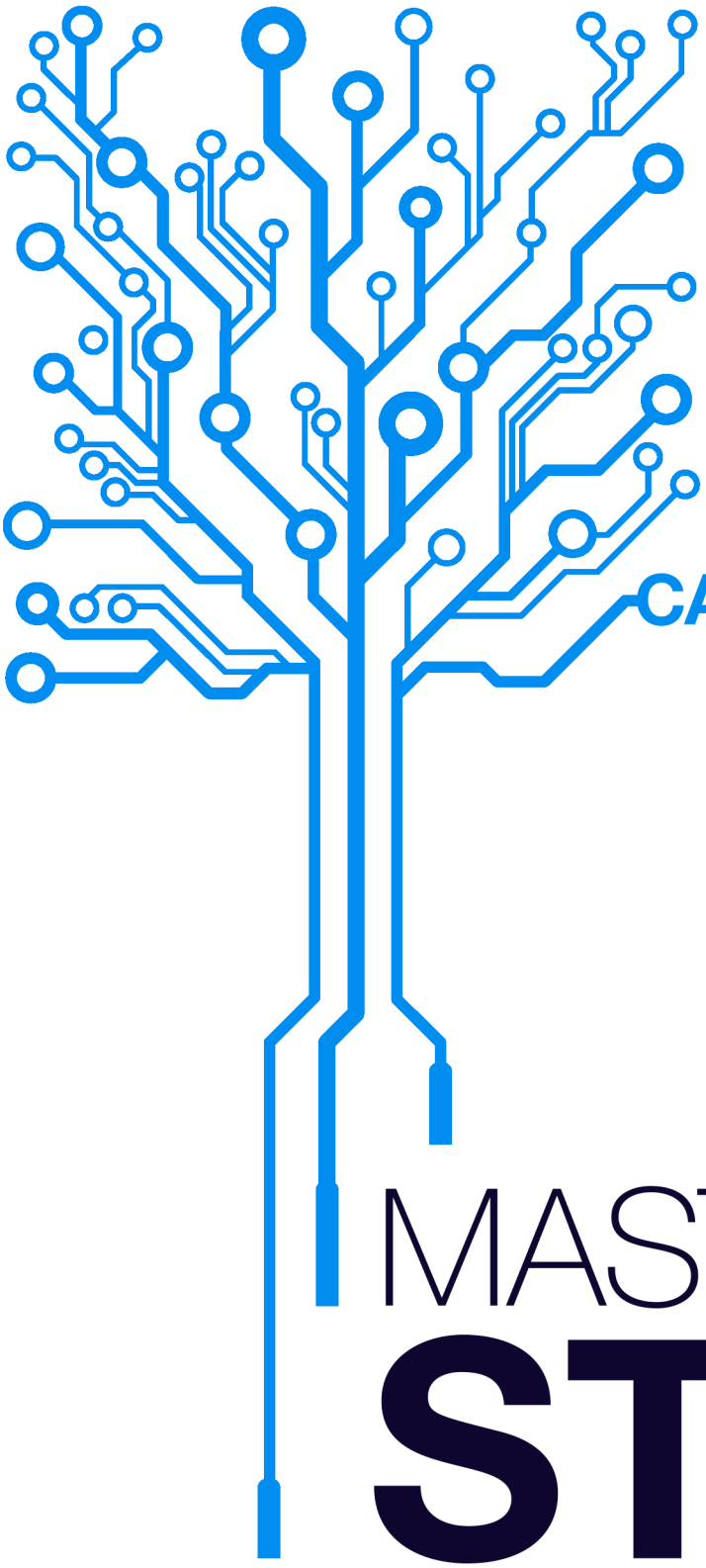
RELEASE **2.0.1**



CARMINE NOVIELLO

MASTERING STM32

A step-by-step guide to the most complete ARM Cortex-M platform, using the official STM32Cube development environment

SECOND EDITION

Contents

I Introduction

1. Introduction to STM32 MCU Portfolio

This chapter gives a brief introduction to the entire STM32 portfolio. Its goal is to introduce the reader to this rather complex family of microcontrollers subdivided in seventeen distinct sub-families. These share a set of characteristics and present features specific to the given series. Moreover, a quick introduction to the Cortex-M architecture is presented. Far from wanting to be a complete reference to either the Cortex-M architecture or STM32 microcontrollers, it aims at being a guide for the readers in choosing the microcontroller that best suits their development needs, considering that, with more than 1200 MCUs to choose from, it is not easy to decide which one fits the bill.

1.1 Introduction to ARM Based Processors

With the term *ARM* we nowadays refer to both a multitude of families of *Reduced Instruction Set Computing* (RISC) architectures and several families of complete *cores* which are the building blocks (hence the term *core*) of CPUs produced by many silicon manufacturers. When dealing with ARM based processors, a lot of confusion may arise since there are many different ARM architecture revisions (ARMv6, ATMv6-M, ARMv7-M, ARMv7-A, ARMv8-M and so on) and many *core* architectures, which are in turn based on an ARM architecture revision. For the sake of clarity, for example, a processor based on the Cortex-M4 core is designed on the ARMv7-M architecture.

An ARM architecture is a set of specifications regarding the instruction set, the execution model, the memory organization and layout, the instruction cycles and more, which precisely describes a *machine* that will implement said architecture. If your compiler is able to generate assembly instructions for that architecture, it is able to generate machine code for all those *actual* machines (aka, processors) implementing that given architecture.

Cortex-M is a family of *physical cores* designed to be further integrated with vendor-specific silicon devices to form a finished microcontroller. The way a core works is not only defined by its related ARM architecture (eg. ARMv7-M), but also by the integrated peripherals and hardware capabilities defined by the silicon manufacturer. For example, the Cortex-M4 core architecture is designed to support bit-data access operations in two specific memory regions using a feature called *bit-banding*, but it is up to the *actual* implementation to add such feature or not. The STM32F1 is a family of MCUs based on the Cortex-M3 core that implements this bit-banding feature. **Figure 1.1** clearly shows the relation between a Cortex-M3 based MCU and its Cortex-M3 core.

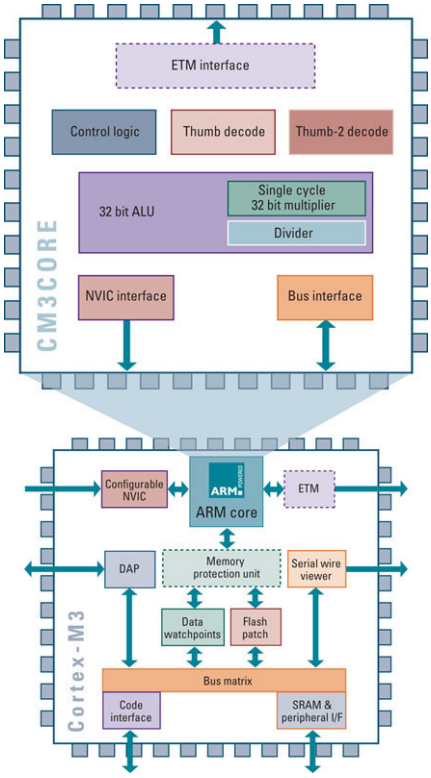


Figure 1.1: The relation between a Cortex-M3 core and a Cortex-M3 based MCU

ARM Holdings is a British company, subsidiary of the *Softbank* Japanese holding, that develops the instruction set and architecture for ARM-based products but does not manufacture devices. This is an important aspect of the ARM world, and the reason why there are many manufacturers of silicon that develop, produce and sell microcontrollers based on the ARM architectures and cores. ST Microelectronics is one of them, and it is currently one of few manufacturers selling a complete portfolio of Cortex-M based processors.

ARM Holdings neither manufactures nor sells CPU devices based on its own designs, but rather licenses the processor architecture to interested parties. ARM offers a variety of licensing terms, varying in cost and deliverables. When referring to Cortex-M cores, it is also common to talk about Intellectual Property (IP) cores, meaning a chip design layout which is considered the intellectual property of one party, namely *ARM Holdings*.

Thanks to this business model and to important features such as low power capabilities, low production costs of some architectures and so on, ARM is the most widely used instruction set architecture in terms of quantity. ARM based products have become extremely popular. More than 160 billion ARM processors have been produced as of 2020. ARM based processors equip about 95% of the world's mobile devices. A lot of mainstream and popular 64-bit and multi-cores CPUs, used in devices that have become icons in the electronic industry (i.e.: Apple's iPhone), are based on an ARM core. In recent years, Apple announced the Apple M1, which is an ARM-based SoC (based on ARMv8.5-A architecture) designed by Apple itself as a *Central Processing Unit* (CPU) and *Graphics Processing Unit* (GPU) for its Macintosh computers and iPad Pro tablets.

Being a sort of widespread standard, there are a lot of compilers and tools, as well as Operating Systems (Linux is the most used OS on Cortex-A processors) which support these architectures, offering developers plenty of opportunities to build their applications.

1.1.1 Cortex and Cortex-M Based Processors

ARM Cortex is a wide set of 32/64-bit *architectures* and *cores* really popular in the embedded world. Cortex microcontrollers are divided into three main subfamilies:

- **Cortex-A**, which stands for **A**pplication, is a series of processors providing a range of solutions for devices undertaking complex computing tasks, such as hosting a rich Operating System (OS) platform (Linux and its derivative Android are the most common ones), and supporting multiple software applications. Cortex-A cores equip the processors found in most of mobile devices, like phones and tablets. In this market segment we can find several silicon manufacturers ranging from those who sell catalogue parts (TI, Freescale and STM with the STM32MP1) to those who produce processors for other licensees. Among the most common cores in this segment, we can find popular Cortex-A7 and Cortex-A9 32-bit processors (they are still common on several cheap *Single Board Computers* (SBC)), as well as the latest ultra-performance 64-bit Cortex-A77 and Cortex-A78 cores.
- **Cortex-M**, which stands for e**M**bedded, is a range of scalable, compatible, energy efficient and easy to use processors designed for the low-cost embedded market. The Cortex-M family is optimized for cost and power sensitive MCUs suitable for applications such as Internet of Things, connectivity, motor control, smart metering, human interface devices, automotive and industrial control systems, domestic household appliances, consumer products and medical instruments. In this market segment, we can find many silicon manufacturers who produce Cortex-M processors: ST Microelectronics is one of them.
- **Cortex-R**, which stand for **R**eal-Time, is a series of processors offering high-performance computing solutions for embedded systems where reliability, high availability, fault tolerance, maintainability and deterministic real-time response are essential. Cortex-R series processors deliver fast and deterministic processing and high performance, while meeting challenging real-time constraints. They combine these features in a performance, power and area optimized package, making them the trusted choice in reliable systems demanding fault tolerance.

The next sections will introduce the main features of Cortex-M processors, especially from the embedded developer point of view.

1.1.1.1 Core Registers

Like all RISC architectures, Cortex-M processors are *load/store* machines, which perform operations only on CPU registers except[1] for two categories of instructions: `load` and `store`, used to transfer data between CPU registers and memory locations.

[1]This is not entirely true, since there are other instructions available in the ARMv6/7 architecture that access memory locations, but for the purpose of this discussion it is best to consider that sentence to be true.

Figure 1.2 shows the core Cortex-M registers. Some of them are available only in the higher performance series like M3, M4 and M7. `R0-R12` are general-purpose registers and can be used as operands for ARM instructions. Some general-purpose registers, however, can be used by the compiler as registers with *special functions*. `R13` is the *Stack Pointer* (SP) register, which is also said to be *banked*. This means that the register content changes according to the current CPU mode (privileged or unprivileged). This function is typically used by Real Time Operating Systems (RTOS) to do context switching.

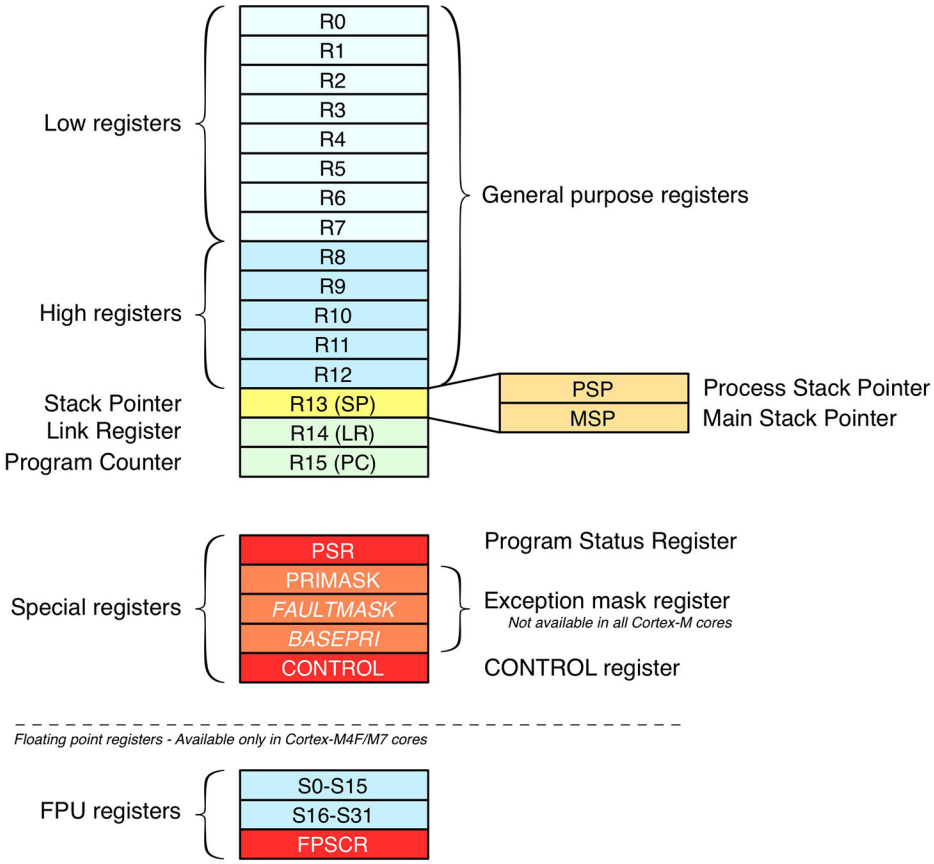


Figure 1.2: ARM Cortex-M core registers

For example, consider the following C code using the local variables “a”, “b”, “c”:

```
...
        uint8_t a,b,c;

        a = 3;
        b = 2;
        c = a * b;
...
```

Compiler will generate the following ARM assembly code[2]:

```
1  movs     r3, #3       ;move "3" in register r3
2  strb     r3, [r7, #7] ;store the content of r3 in "a"
3  movs     r3, #2       ;move "2" in register r3
4  strb     r3, [r7, #6] ;store the content of r3 in "b"
5  ldrb     r2, [r7, #7] ;load the content of "a" in r2
6  ldrb     r3, [r7, #6] ;load the content of "b" in r3
7  smulbb   r3, r2, r3   ;multiply "a" with "b" and store result in r3
8  strb     r3, [r7, #5] ;store the result in "c"
```

As we can see, all the operations always involve a register. Instructions at lines 1-2 move the number `3` into the register `r3` and then store its content (that is, the number `3`) inside the memory location given by the register `r7` plus an offset of `7` memory locations - that is the place where `a` variable is stored. The same happens for the variable `b` at lines 3-4. Then lines 5-7 load the content of variables `a` and `b` and perform the multiplication. Finally, line `8` stores the result in the memory location of variable `c`.

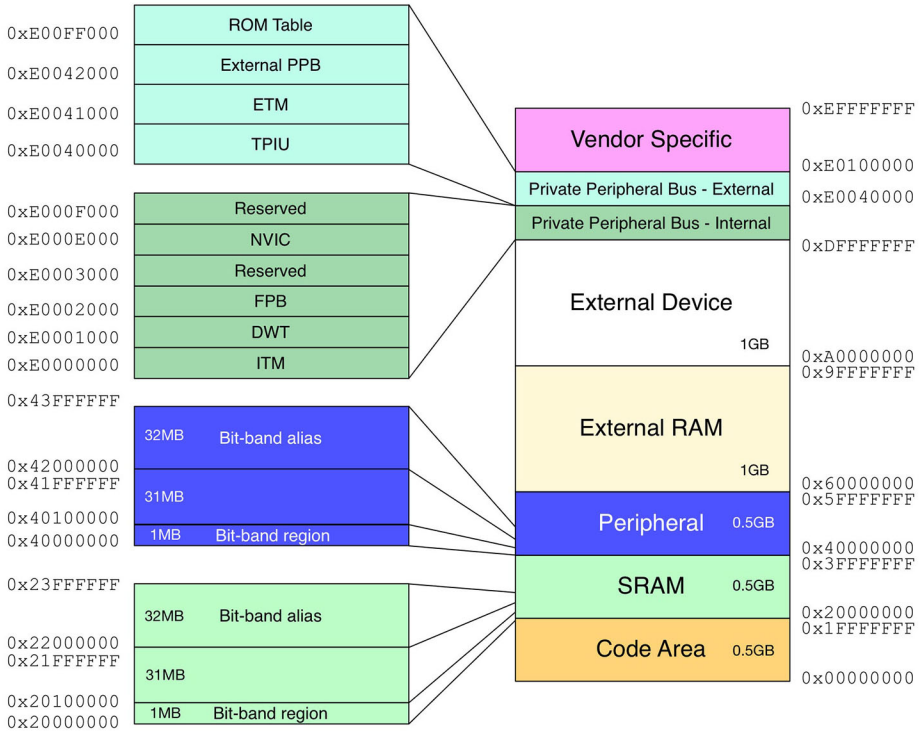


Figure 1.3: Cortex-M fixed memory address space

[2]That assembly code was generated compiling in `thumb` mode with any optimization disabled, invoking GCC in the following way: `$ arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -fverbose-asm -save-temps -O0 -g -c file.c`

1.1.1.2 Memory Map

ARM defines a standardized memory address space common to all Cortex-M cores, which ensures code portability among different silicon manufacturers. The address space is 4GB wide, and it is organized in several sub-regions with different logical functionalities. **Figure 1.3** shows the memory layout of a Cortex-M processor [3].

The first 512MB are dedicated to code area. STM32 devices further divide this area in some sub-regions as shown in **Figure 1.4**. Let us briefly introduce them.

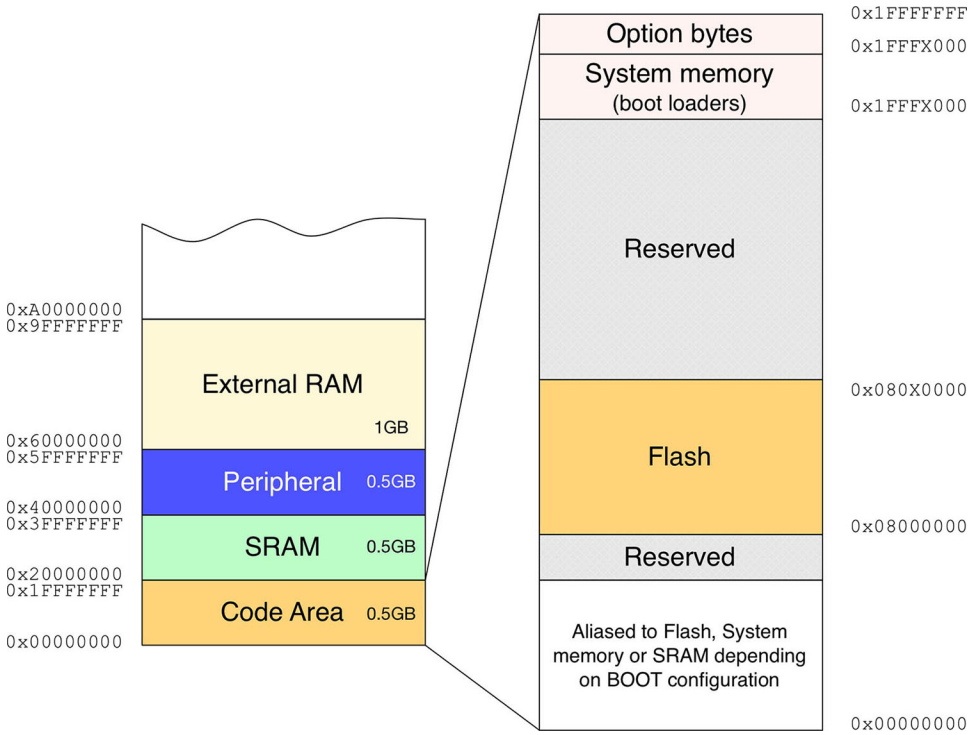


Figure 1.4: Memory layout of Code Area on STM32 MCUs

All Cortex-M processors map the code area starting at address `0x0000 0000`[4]. This area also includes the pointer to the beginning of the stack (usually placed in SRAM) and the *vector table*, as we will see in Chapter 7. The position of the code area is standardized among all other Cortex-M vendors, even if the core architecture is sufficiently flexible to allow manufacturers to arrange this area in a different way. In fact, for all STM32 devices an area starting at address `0x0800 0000` is bound to the internal MCU flash memory, and it is the area where program code resides. However, thanks to a specific boot configuration we will explore in Chapter 22, this area is also *aliased* from address `0x0000 0000`. This means that it is perfectly possible to refer to the content of the flash memory both starting at address `0x0800 0000` and `0x0000 0000` (for example, a routine located at address `0x0800`

[3] Although the memory layout and the size of sub-regions (and therefore also their addresses) are standardized between all Cortex-M cores, some functionalities may differ. For example, Cortex-M7 does not provide bit-band regions, and some peripherals in the *Private Peripheral Bus* region differ. Always consult the reference manual for the architecture you are considering.

[4] To increase readability, all 32-bit addresses in this book are written splitting the upper two bytes from the lower ones. So, every time you see an address expressed in this way (`0x0000 0000`) you have to interpret it just as one common 32-bit address (`0x00000000`). This rule does not apply to C and assembly source code.

`16DC` can also be accessed from `0x0000 16DC`).

The last two sections are dedicated to *System memory* and *Option bytes*. The first one is a ROM region reserved to bootloaders. Each STM32 family (and their sub-families - *low density*, *medium density*, and so on) provides a bootloader pre-programmed into the chip during production. As we will see in Chapter 22, this bootloader can be used to load code from several peripherals, including USARTs, USB and CAN bus. The *Option bytes* region contains a series of bit flags which can be used to configure several aspects of the MCU (such as flash read protection, hardware watchdog, boot mode and so on) and are related to the specific STM32 microcontroller.

Going back to the whole 4GB address space, the next main region is the one bounded to the internal MCU SRAM. It starts at address `0x2000 0000` and can potentially extend to `0x3FFF FFFF`. However, the actual end address depends on the effective amount of internal SRAM. For example, in the case of an STM32F103RB MCU with 20KB of SRAM, we have a final address of **0x2000 4FFF**[5]. Trying to access a location outside of this area will cause a *Bus Fault* exception (more about this later).

The next 0.5GB of memory is dedicated to the mapping of peripherals. Every peripheral provided by the MCU (timers, I²C and SPI interfaces, USARTs, and so on) has an alias in this region. It is up to the specific MCU to organize this memory space.

The next 2GB area is dedicated to external SRAM or flash. Cortex-M devices can execute code and load/store data from external memory, which extend the internal memory resources, through the EMI/FSMC interface. Some STM32 devices, like the STM32F7, are able to execute code from external memory without performance bottlenecks, thanks to an L1 cache and the ART™ Accelerator.

The final 0.5 GB of memory is allocated to the internal (core) Cortex processor peripherals, plus a reserved area for future enhancements to Cortex processors. All Cortex processor registers are at fixed locations for all Cortex-based microcontrollers. This allows code to be more easily ported between different STM32 variants and indeed other vendors' Cortex-based microcontrollers.

1.1.1.3 Bit-Banding

In embedded applications, it is quite common to work with single bits of a word using bit masking. For example, suppose that we want to set or clear the 3rd bit (bit 2) of an unsigned byte. We can simply do this using the following C code:

```
...
uint8_t temp = 0;

temp |= 0x4;
temp &= ~0x4;
...
```

Bit masking is used when we want to save space in memory (using one single variable and assigning a different meaning to each of its bits) or we have to deal with internal MCU registers and peripherals.

[5]The final address is computed in the following way: 20K is equal to 20 * 1024 bytes, which in base 16 is `0x5000`. But addresses start from 0, hence the final address is `0x2000 0000` + `0x4FFF`.

Considering the previous C code, we can see that the compiler will generate the following ARM assembly code[6]:

```
#temp |= 0x4;
a:          79fb              ldrb   r3, [r7, #7]
c:          f043 0304         orr.w  r3, r3, #4
10:         71fb              strb   r3, [r7, #7]
#temp &= ~0x4;
12:         79fb              ldrb   r3, [r7, #7]
14:         f023 0304         bic.w  r3, r3, #4
18:         71fb              strb   r3, [r7, #7]
```

As we can see, such a simple operation requires three assembly instructions (fetch, modify, save). This leads to two types of problems. First of all, there is a waste of CPU cycles related to those three instructions. Second, that code works fine if the CPU is working in single task mode, and we have just one execution stream, but, if we are dealing with concurrent execution, another task (or simply an interrupt routine) may affect the content of the memory before we complete the "bit mask" operation (that is, for example, an interrupt occurs between instructions at lines 0xC-0x10 or 0x14-0x18 in the above assembly code).

Bit-banding is the ability to map each bit of a given area of memory to a whole word in the aliased bit-banding memory region, allowing atomic access to such bit. **Figure 1.5** shows how the Cortex CPU aliases the content of memory address `0x2000 0000` to the bit-banding region `0x2200 0000-1c`. For example, if we want to modify (bit 2) of `0x2000 0000` memory location we can simply access to `0x2200 0008` memory location.

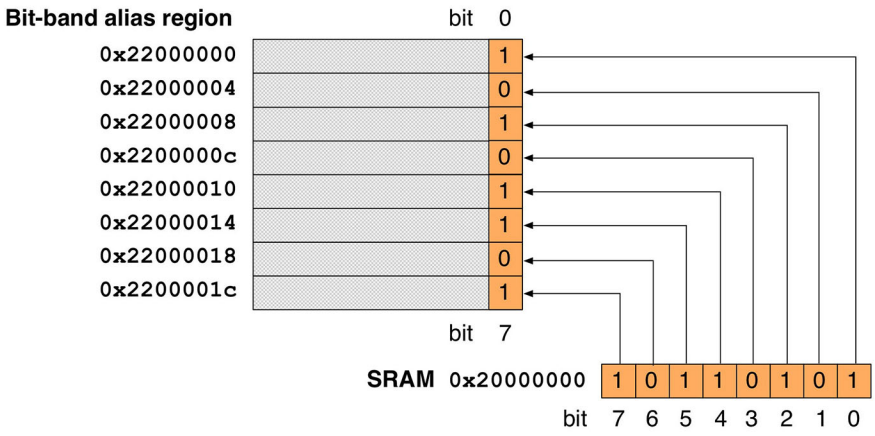


Figure 1.5: Memory mapping of SRAM address `0x2000 0000` in bit-banding region (first 8 of 32 bits shown)

This is the formula to compute the addresses for alias regions:

```
bit_band_address = alias_region_base + (region_base_offset x 32) + (bit_number x 4)
```

[6]That assembly code was generated compiling in `thumb` mode with any optimization disabled, invoking GCC in the following way: `$ arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -fverbose-asm -save-temps -O0 -g -c file.c`

For example, considering the memory address of **Figure 1.5**, to access bit 2 :

```
alias_region_base = 0x22000000
region_base_offset = 0x20000000 - 0x20000000 = 0
bit_band_address = 0x22000000 + 0*32 + (0x2 x 0x4) = 0x22000008
```

ARM defines two bit-band regions for Cortex-M3/4 based MCUs, each one is 1MB wide and mapped to a 32Mbit bit-band alias region. Each consecutive 32-bit word in the "alias" memory region refers to each consecutive bit in the "bit-band" region (which explains that size relationship: 1Mbit <-> 32Mbit). The first one starts at `0x2000 0000` and ends at `0x200F FFFF`, and it is aliased from `0x2200 0000` to `0x23FF FFFF`. It is dedicated to the bit access of SRAM memory locations. Another bit-banding region starts at `0x4000 0000` and ends at `0x400F FFFF`, as shown in **Figure 1.6**.

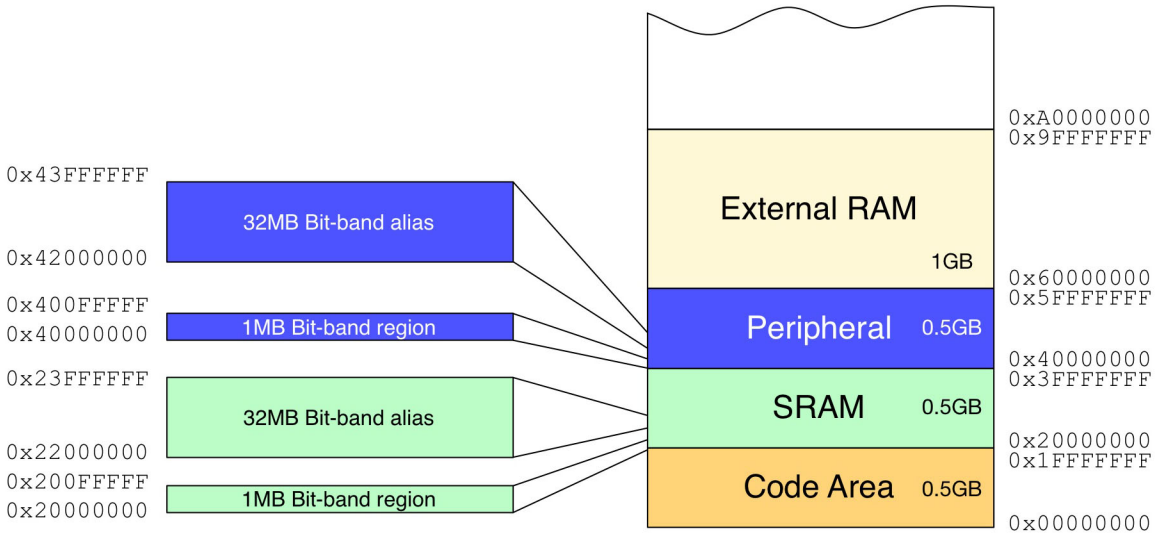


Figure 1.6: Memory map and bit-banding regions

This other region is dedicated to the memory mapping of peripherals. For example, ST maps the GPIO Output Data Register (GPIO->ODR) of GPIOA peripheral from `0x4002 0014`. This means that each bit of the word addressed at `0x4002 0014` allows modifying the output state of a GPIO (from LOW to HIGH and vice versa). So if we want to modify the status of PIN5 of GPIOA port[7], using the previous formula we have:

```
alias_region_base = 0x42000000
region_base_offset = 0x40020014 - 0x40000000 = 0x20014
bit_band_address = 0x42000000 + 0x20014*32 + (0x5 x 0x4) = 0x42400294
```

We can define two macros in C that allow to easily compute bit-band alias addresses:

[7]Anyone who has already played with Nucleo boards, knows that user LED LD2 (the green one) is connected to that port pin.

```
// Define base address of bit-band
#define BITBAND_SRAM_BASE 0x20000000
// Define base address of alias band
#define ALIAS_SRAM_BASE 0x22000000
// Convert SRAM address to alias region
#define BITBAND_SRAM(a,b) ((ALIAS_SRAM_BASE + ((uint32_t)&(a)-BITBAND_SRAM_BASE)*32 + (b*4)))

// Define base address of peripheral bit-band
#define BITBAND_PERI_BASE 0x40000000
// Define base address of peripheral alias band
#define ALIAS_PERI_BASE 0x42000000
// Convert PERI address to alias region
#define BITBAND_PERI(a,b) ((ALIAS_PERI_BASE + ((uint32_t)a-BITBAND_PERI_BASE)*32 + (b*4)))
```

Still using the above example, we can quickly modify the state of PIN5 of the GPIOA port as follows:

```
#define GPIOA_PERH_ADDR 0x40020000
#define ODR_ADDR_OFF    0x14

uint32_t *GPIOA_ODR = GPIOA_PERH_ADDR + ODR_ADDR_OFF;
uint32_t *GPIOA_PIN5 = BITBAND_PERI(GPIOA_ODR, 5);

*GPIOA_PIN5 = 0x1; // Turns GPIO HIGH
```

1.1.1.4 Thumb-2 and Memory Alignment

Historically, ARM processors provide 32-bit instructions set. This not only allows for a rich set of instructions, but also guarantees the best performance during the execution of instructions involving arithmetic operations and memory transfers between core registers and SRAM. However, a 32-bit instruction set has a cost in terms of memory footprint of the firmware. This means that a program written with a 32-bit *Instruction Set Architecture* (ISA) requires a higher amount of bytes of flash storage, which impacts on power consumption and overall costs of the MCU (silicon wafers are expensive, and manufacturers constantly *shrink* chips size to reduce their cost).

To address such issues, ARM introduced the *Thumb* 16-bit instruction set, which is a subset of the most commonly used 32-bit one. Thumb instructions are each 16 bits long and are automatically "translated" to the corresponding 32-bit ARM instruction that has the same effect on the processor model. This means that 16-bit Thumb instructions are transparently expanded (from the developer point of view) to full 32-bit ARM instructions in real time, without performance loss. Thumb code is typically 65% the size of ARM code and provides 160% the performance of the latter when running from a 16-bit memory system; however, in Thumb, the 16-bit opcodes have less functionality. For example, only branches can be conditional, and many opcodes are restricted to accessing only half of all of the CPU's general-purpose registers.
Afterwards, ARM introduced the **Thumb-2** instruction set, which is a mix of 16 and 32-bit instruction

sets in one operation state. *Thumb-2* is a variable length instruction set and offers a lot more instructions compared to the *Thumb* one, achieving similar code density.

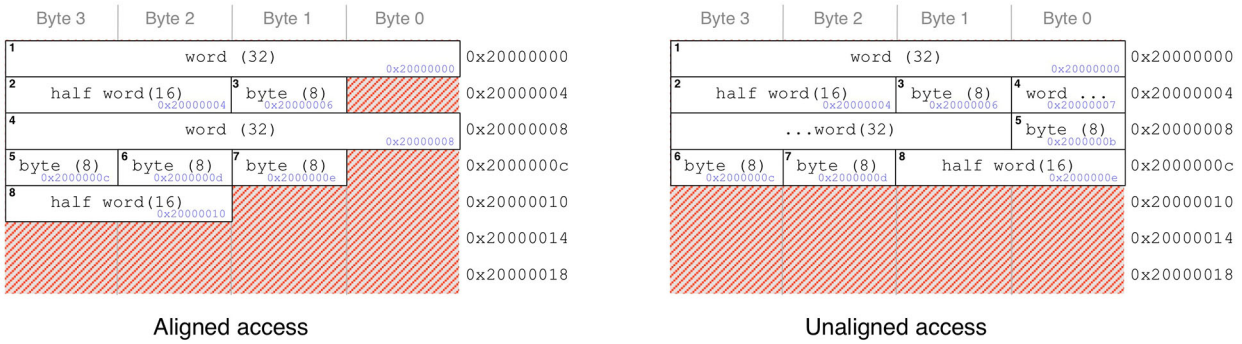


Figure 1.7: Difference between aligned and unaligned memory access

Cortex-M3/4/7 were designed to support the full *Thumb* and *Thumb-2* instruction sets, and some of them support other instruction sets dedicated to Floating Point operations (Cortex-M4/7) and *Single Instruction Multiple Data* (SIMD) operations (also known as NEON instructions).

Another interesting feature of Cortex-M3/4/7 cores is the ability to do unaligned access to memory. ARM based CPUs are traditionally capable of accessing byte (8-bit), half word (16-bit) and word (32-bit) signed and unsigned variables, without increasing the number of assembly instructions as it happens on 8-bit MCU architectures. However, early ARM architectures were unable to perform unaligned memory access, causing a waste of memory locations.

To understand the problem, consider the left diagram in **Figure 1.7**. Here we have eight variables. With memory aligned access we mean that to access the word variables (**1** and **4** in the diagram), we need to access addresses which are multiples of 32-bits (4 bytes). That is, a word variable can be stored only in `0x2000 0000`, `0x2000 0004`, `0x2000 0008` and so on. Every attempt to access a location which is not a multiple of 4 causes a *UsageFaults* exception. So, the following ARM pseudo-instruction is not correct:

```
STR R2, 0x20000002
```

The same applies for half word access: it is possible to access to memory locations stored at multiple of 2 bytes: `0x2000 0000`, `0x2000 0002`, `0x2000 0004` and so on. This limitation causes fragmentation inside the RAM memory. To solve this issue, Cortex-M3/4/7 based MCUs are able to perform unaligned memory access, as shown in the right diagram in **Figure 1.7**. As we can see, variable 4 is stored starting at address `0x2000 0007` (in early ARM architectures this was only possible with single byte variables). This allows us to store variable 5 in memory location `0x2000 000b`, causing variable 8 to be stored in `0x2000 000e`. Memory is now packed, and we have saved 4 bytes of SRAM.

However, unaligned access is restricted to the following ARM instructions:

- LDR, LDRT
- LDRH, LDRHT
- LDRSH, LDRSHT

- STR, STRT
- STRH, STRHT

1.1.1.5 Pipeline

Whenever we talk about *instructions execution*, we are making a series of non-trivial assumptions. Before an instruction is executed, the CPU has to fetch it from memory and decode it. This procedure consumes several CPU cycles, depending on the memory and core CPU architecture, which is added to the actual instruction cost (that is, the number of cycles required to execute the given instruction).

Modern CPUs introduce a way to parallelize these operations in order to increase their instructions throughput (the number of instructions which can be executed in a unit of time). The basic instruction cycle is broken up into a series of steps, as if the instructions traveled along a *pipeline*. Rather than processing each instruction sequentially (one at a time, finishing one instruction before starting with the next one), each instruction is split into a sequence of stages so that different steps can be executed in parallel.

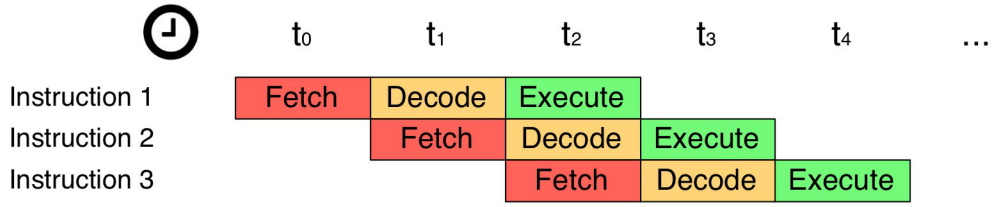


Figure 1.8: Three stage instruction pipeline

All Cortex-M based microcontrollers introduce a form of pipelining. The most common one is the *3-stage pipeline*, as shown in **Figure 1.8**. *3-stage pipeline* is supported by Cortex-M0/3/4. Cortex-M0+ cores, which are dedicated to low-power MCUs, provide a *2-stage pipeline* (although pipelining helps reducing the time cost related to the instruction's fetch/decode/execution cycle, it introduces an energy cost which must be minimized in low-power applications). Cortex-M7 cores provide a *6-stage pipeline*.

When dealing with pipelines, branching is an issue to be addressed. Program execution is all about taking different paths; this is achieved through branching (`if equal goto`). Unfortunately, branching causes the invalidation of pipeline streams, as shown in **Figure 1.9**. The last two instructions have been loaded into the pipeline, but they are discarded due to the optional branch path being taken (we usually refer to them as *branch shadows*)

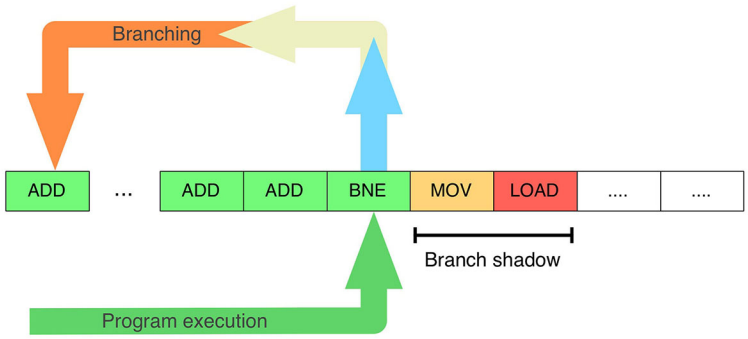


Figure 1.9: Branching in program execution related to pipelining

Even in this case there are several techniques to minimize the impact of branching. They are often referred as *branching prediction techniques*. The idea behind these techniques is that the CPU starts fetching and decoding both the instructions following the branching and the ones that would be reached if the branch were to happen (in **Figure 1.9** both `MOV` and `ADD` instructions). There are, however, other ways to implement a branch prediction scheme. If you want to look deeper into this subject, this post[8] from the official ARM support forum is a good starting point.

1.1.1.6 Interrupts and Exceptions Handling

Interrupts and exception management is one of the most powerful features of Cortex-M based processors. Interrupts and exceptions are asynchronous events that alter the program flow. When an exception or an interrupt occurs, the CPU suspends the execution of the current task, saves its context (that is, its stack pointer) and starts the execution of a routine designed to handle the interrupting event. This routine is called *Exception Handler* in case of exceptions and *Interrupt Service Routine* (ISR) in case of an interrupt. After the exception or interrupt has been handled, the CPU resumes the previous execution flow, and the previous task can continue its execution[9].

[8] https://bit.ly/1k7ggh6

[9] With the term *task* we refer to a series of instructions which constitute the main flow of execution. If our firmware is based on an OS, the scenario could be a bit more articulated. Moreover, in case of low-power sleep mode, the CPU may be configured to go back to sleep after an interrupt management routine is executed. We will analyse these more complex scenarios in following chapters.

Number	Exception type	Priority[a]	Function
1	Reset	-3	Reset
2	NMI	-2	Non-Maskable Interrupt
3	Hard Fault	-1	All classes of Fault, when the fault cannot activate because of priority or the Configurable Fault handler has been disabled.
4	Memory Management[c]	Configurable[b]	MPU mismatch, including access violation and no match. This is used even if the MPU is disabled or not present.
5	Bus Fault[c]	Configurable	Pre-fetch fault, memory access fault, and other address/memory related.
6	Usage Fault[c]	Configurable	Usage fault, such as Undefined instruction executed or illegal state transition attempt.
7	SecureFault[d]	Configurable	SecureFault is available when the CPU runs in *Secure state*. It is triggered by the various security checks that are performed. For example, when jumping from Non-secure code to an address in Secure code that is not marked as a valid entry point.
8-10	-	-	RESERVED
11	SVCall	Configurable	System service call with SVC instruction.
12	Debug Monitor[c]	Configurable	Debug monitor – for software based debug.
13	-	-	RESERVED
14	PendSV	Configurable	Pending request for system service.
15	SysTick	Configurable	System tick timer has fired.
16-[47/239/479][e]	IRQ	Configurable	IRQ Input

[a]The lower the priority number is, the higher the priority is.

[b]It is possible to change priority of exception assigning a different number. For Cortex-M0/0+ processors this number ranges from 0 to 192 in steps of 64 (that is 4 priority levels available). For Cortex-M3/4/7/33 ranges from 8 to 256.

[c]These exceptions are not available in Cortex-M0/0+.

[d]This exception is available just in Cortex-M33.

[e]Cortex-M0/0+ allow 32 external configurable interrupts. Cortex-M3/4/7 allow 240 external configurable interrupts. Cortex-M33 allows 480 external configurable interrupts. However, in practice the number of interrupt inputs implemented in the real MCU is far less.

Table 1.1: Cortex-M exception types

In the ARM architecture, interrupts are one type of exception. Interrupts are usually generated from on-chip peripherals (e.g., a timer) or external inputs (e.g., a tactile switch connected to a GPIO), and in some cases they can be triggered by software. Exceptions are, instead, related to software execution, and the CPU itself can be a source of exceptions. These could be fault events such as an attempt to access an invalid memory location, or events generated by the Operating System, if any.

Each exception (and hence interrupt) has a number which uniquely identifies it. **Table 1.1** shows the predefined exceptions common to all Cortex-M cores, plus a variable number of user-defined ones related to interrupts management. This number reflects the position of the exception handler routine inside the vector table, where the actual address of the routine is stored. For example, position

15 contains the memory address of a code area containing the exception handler for the *SysTick* interrupt, generated when the *SysTick* timer reaches zero.

Other than the first three, each exception can be assigned a priority level, which defines the processing order in case of concurrent interrupts: the lower the number, the higher the priority. For example, suppose we have two interrupt routines related to external inputs A and B. We can assign a higher-priority interrupt (lower number) to input A. If the interrupt related to A arrives while the processor is serving the interrupt from input B the execution of B is suspended, allowing the higher priority interrupt service routine to be executed immediately.

Both exceptions and interrupts are processed by a dedicated unit called *Nested Vectored Interrupt Controller* (NVIC). The NVIC has the following features:

- **Flexible exception and interrupt management**: NVIC is able to process both interrupt signals/requests coming from peripherals and exceptions coming from the processor core, allowing us to enable/disable them in software (except for NMI[10]).
- **Nested exception/interrupt support**: NVIC allows the assignment of priority levels to exceptions and interrupts (except for the first three exception types), giving the possibility to categorize interrupts based on user needs.
- **Vectored exception/interrupt entry**: NVIC automatically locates the position of the exception handler related to an exception/interrupt, without need of additional code.
- **Interrupt masking**: developers are free to suspend the execution of all exception handlers (except for NMI), or to suspend some of them on a priority level basis, thanks to a set of dedicated registers. This allows the execution of critical tasks in a safe way, without dealing with asynchronous interruptions.
- **Deterministic interrupt latency**: one interesting feature of NVIC is the deterministic latency of interrupt processing, which is equal to 12 cycles for all Cortex-M3/4 cores, 15 cycles for Cortex-M0, 16 cycles for Cortex-M0+, regardless of the processor's current status.
- **Relocation of exception handlers**: as we will later in the book, exception handlers can be relocated to other flash memory locations as well as totally different - even external - non-read-only memory. This offers a great degree of flexibility for advanced applications.

1.1.1.7 SysTimer

Cortex-M based processors can optionally provide a System Timer, also known as *SysTick*. The good news is that all STM32 devices provide one, as shown in **Table 1.3**.
SysTick is a 24-bit down-counting timer used to provide a system tick for *Real Time Operating Systems* (RTOS) like FreeRTOS. It is used to generate periodic interrupts to scheduled tasks. Programmers can define the update frequency of *SysTick* timer by setting its registers. SysTick timer is also used by the STM32 HAL to generate precise delays, even if we aren't using an RTOS. More about this timer in Chapter 11.

[10]Also the *Reset exception* cannot be disabled, even if it is improper to talk about the Reset exception disabling, since it is the first exception generated after the MCU resets. As we will see in Chapter 7, the Reset exception is the actual entry point of every STM32 application.

1.1.1.8 Power Modes

The current trend in the electronics industry, especially when it comes to mobile devices design, is all about power management. Reducing power consumption to minimum is the main goal of all hardware designers and programmers involved in the development of battery-powered devices. Cortex-M processors provide several levels of power management, which can be divided into two main groups: *intrinsic features* and *user-defined power modes*.

With *intrinsic features* we refer to those native capabilities related to power consumption defined during the design of both the Cortex-M core and the whole MCU. For example, Cortex-M0+ cores only define two pipeline stages in order to reduce power consumption during instructions prefetch. Another native behavior related to power management is the high code density of the Thumb-2 instruction set, which allows developers to choose MCUs with smaller flash memory to lower power needs.

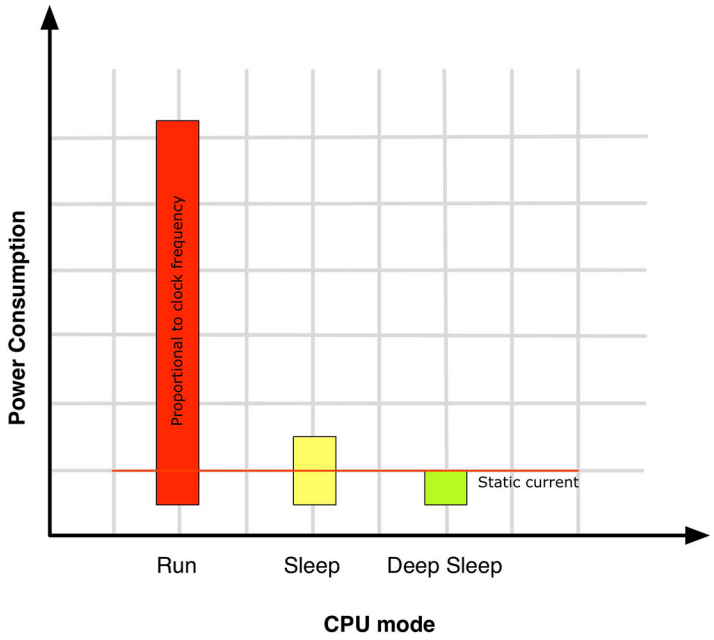


Figure 1.10: Cortex-M power consumption at different power modes

Traditionally, Cortex-M processors provide *user-defined power modes* via *System Control Register* (SCR). The first one is the *Run* mode (see **Figure 1.10**), which has the CPU running at its full capabilities. In *Run* mode, power consumption depends on clock frequency and used peripherals. *Sleep* mode is the first low-power mode available to reduce power consumption. When activated, most functionalities are suspended, CPU frequency is lowered, and its activities are reduced to those necessary for it to wake up. In *Deep sleep* mode all clock signals are stopped, and the CPU needs an external event to wake up from this state.

However, these power modes are only general models, which are further implemented in the actual MCU. For example, consider **Figure 1.11** displaying the power consumption of an STM32F2 MCU

running at 80MHZ @30°C[11]. As we can see, the maximum power consumption is reached in *Run-mode* with the ART™ accelerator disabled. Enabling the ART™ accelerator we can save up to 10mAh while also achieving better computing performances. This clearly shows that the real MCU implementation can introduce different power levels.

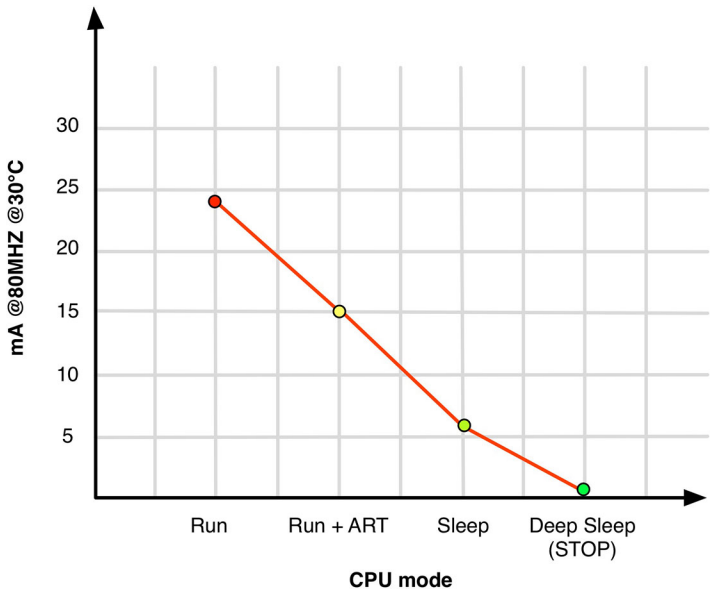


Figure 1.11: STM32F2 power consumption at different power modes

STM32Lx families provide several further intermediate power levels, allowing to precisely select the preferred power mode and hence MCU performance and power consumption.
We will go in more depth about this topic in Chapter 19.

1.1.1.9 TrustZone™

ARM recently introduced two new Cortex-M processor cores named Cortex-M23 and Cortex-M33, both based on the ARMv8-M architecture. These new cores inherit a feature already present on the majority of Cortex-A processors: the ARM TrustZone™. The TrustZone™ is an optional Security Extension that is designed to provide a foundation for improved system security in a wide range of embedded applications. The concept of TrustZone™ technology is not new. The processor can run in *Secure* and *Non-secure* states, with *Non-secure* software able to access to *Non-secure* memory region only. By acting on the memory mapping configuration, it is possible to define regions of memory address space where the access to that region (both when executing code and when accessing to data) is possible just when the processor runs in *Secure* mode. ARM TrustZone™ technology enables the system and the software to be partitioned into *Secure* and *Normal* **worlds**. Secure software can access both *Secure* and *Non-secure* memories and hardware resources, while *Normal* software can only access *Non-secure* memories and resources. These security states are orthogonal to the existing Thread and Handler modes (more about these two running modes later in the text), enabling both a Thread and Handler mode in both Secure and Non-secure states.

[11]Source ST AN3430

ARM TrustZone technology does not cover all aspects of security. For example, it does not include cryptography. In designs with the ARMv8-M architecture with TrustZone™, components that are critical to the security of the system can be placed in the *Secure* world. For example, these critical components may include:

- A Secure boot loader
- Secret keys
- Flash programming support
- High value assets

The remaining applications are usually placed in the *Normal* world.

1.1.1.10 CMSIS

One of the key advantages of the ARM platform (both for silicon vendors and application developers) is the existence of a complete set of development tools (compilers, *run-time* libraries, debuggers, and so on) which are reusable across several vendors.

ARM is also actively working on a way to standardize the software infrastructure among the MCUs vendors. Cortex Microcontroller Software Interface Standard (CMSIS) is a vendor-independent hardware abstraction layer for the Cortex-M processor series and specifies debugger interfaces. The CMSIS consists of the following components:

- **CMSIS-CORE**: API for the Cortex-M processor core and peripherals. It provides a standardized interface for Cortex-M0/0+/3/4/7/23/33.
- **CMSIS-Driver**: defines generic peripheral driver interfaces for middleware making them reusable across supported devices. The API is RTOS independent and connects microcontroller peripherals to middleware which implements, amongst other things, communication stacks, file systems or graphical user interfaces.
- **CMSIS-DSP**: DSP Library Collection with over 60 Functions for various data types: fixed-point (fractional q7, q15, q31) and single precision floating-point (32-bit). The library is available for Cortex-M0, Cortex-M3, and Cortex-M4. The Cortex-M4 implementation is optimized for the SIMD instruction set.
- **CMSIS-RTOS API**: Common API for Real-Time Operating Systems. It provides a standardized programming interface which is portable to many RTOS and therefore enables software templates, middleware, libraries, and other components which can work across supported RTOS systems. We will talk about this API layer in Chapter 23.
- **CMSIS-Pack**: describes, using an XML based package description file named “PDSC”, the user and device relevant parts of a file collection (namely “software pack”) which includes source, header, library files, documentation, flash programming algorithms, source code templates and example projects. Development tools and web infrastructures use the PDSC file to extract device parameters, software components, and evaluation board configurations.

- **CMSIS-SVD**: *System View Description* (SVD) for Peripherals. Describes the peripherals of a device in an XML file and can be used to create peripheral awareness in debuggers or header files with peripheral registers and interrupt definitions.
- **CMSIS-DAP**: Debug Access Port. Standardized firmware for a Debug Unit that connects to the CoreSight Debug Access Port. CMSIS-DAP is distributed as a separate package and well suited for integration on evaluation boards.
- **CMSIS-NN**: Neural Networks. Due to the increase of power computing in latest Cortex-M4/7 cores, neural network-based solutions are becoming increasingly popular even for embedded machine learning applications. CMSIS-NN is a collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint of neural networks on Cortex-M processor cores.

However, this initiative from ARM is evolving by its own, and it is mostly related to the evolving of the ARM Keil tool-chain. The support to all components from ST is still limited to some APIs, as we will see in following chapters. The official ST HAL is the main way to develop applications for the STM32 platform, which presents a lot of peculiarities between MCUs of different families. Moreover, it is quite clear that the main objective of silicon vendors is to retain their customers and avoid their migration to other MCUs platform (even if based on the same ARM Cortex core). So, we are really far from having a complete and portable layer that works on all ARM based MCUs available on the market.

1.1.1.11 Effective Implementation of Cortex-M Features in the STM32 Portfolio

Some of the features presented in the previous paragraphs are optional and may not be available in a given MCU. **Tables 1.2** and **3** summarize the Cortex-M instructions and components available in the STM32 Portfolio. These could be useful during the selection of an STM32 MCU.

STM32 Family	Cortex-M	SysTick Timer	Bit-Banding	Memory Protection Unit (MPU)	Trust Zone	CPU Cache	OS Support	Memory Architecture
F0	M0	Yes	Yes	No	No	No	Yes	Von Neumann
L0, G0	M0+	Yes	Yes	Yes	No	No	Yes	Von Neumann
F1, F2, L1	M3	Yes	Yes	Yes	No	No	Yes	Harvard
F3, F4, L4, L4+, G4, WB	M4	Yes	Yes	Yes	No	No	Yes	Harvard
F7, H7	M7	Yes	No	Yes	No	Yes	Yes	Harvard
L5, U5	M33	Yes	Yes	Yes	Yes	Yes	Yes	Harvard

Optional in ARM specification

Table 1.2: ARM Cortex-M instruction variations

STM32 Family	Cortex-M	Thumb	Thumb-2	Multiply in Hardware	Divide in Hardware	Saturated math	DSP	FPU	ARM Architecture
F0	M0	Most	Some	32-bit result	No	No	No	No	ARMv6-M
L0, G0	M0+	Most	Some	32-bit result	No	No	No	No	ARMv6-M
F1, F2, L1	M3	Entire	Entire	32/64-bit result	Yes	Yes	No	No	ARMv7-M
F3, F4, L4, L4+, G4, WB	M4	Entire	Entire	32/64-bit result	Yes	Yes	Yes	Yes SP	ARMv7E-M
F7, H7	M7	Entire	Entire	32/64-bit result	Yes	Yes	Yes	Yes SP & DP	ARMv7E-M
L5, U5	M33	Entire	Entire	32/64-bit result	Yes	Yes	Yes	Yes SP & DP	ARMv8-M

Optional in ARM specification

Table 1.3: ARM Cortex-M optional components

1.2 Introduction to STM32 Microcontrollers

STM32 is a broad range of microcontrollers divided in seventeen sub-families, each one with its features. ST started the market production of this portfolio in 2007, beginning with the STM32F1 series, which is still in production. **Figure 1.12** shows the internal die of an STM32F103 MCU, one of the most widespread STM32 MCUs[12]. All STM32 microcontrollers have a Cortex-M core, plus some distinctive ST features (like the ART™ accelerator). Internally, each microcontroller consists of the processor core, static RAM, flash memory, debugging interface, and various other peripherals. Some MCUs provide additional types of memory (EEPROM, CCM, etc.), and a whole line of devices targeting low-power applications is continuously growing.

[12]This picture is taken from Zeptobars.ru, a really fantastic blog. Its authors decap (that is, remove the protective casing) integrated circuits in acid and publish images of what's inside the chip. I love those images, because they show what humans were able to achieve in electronics.

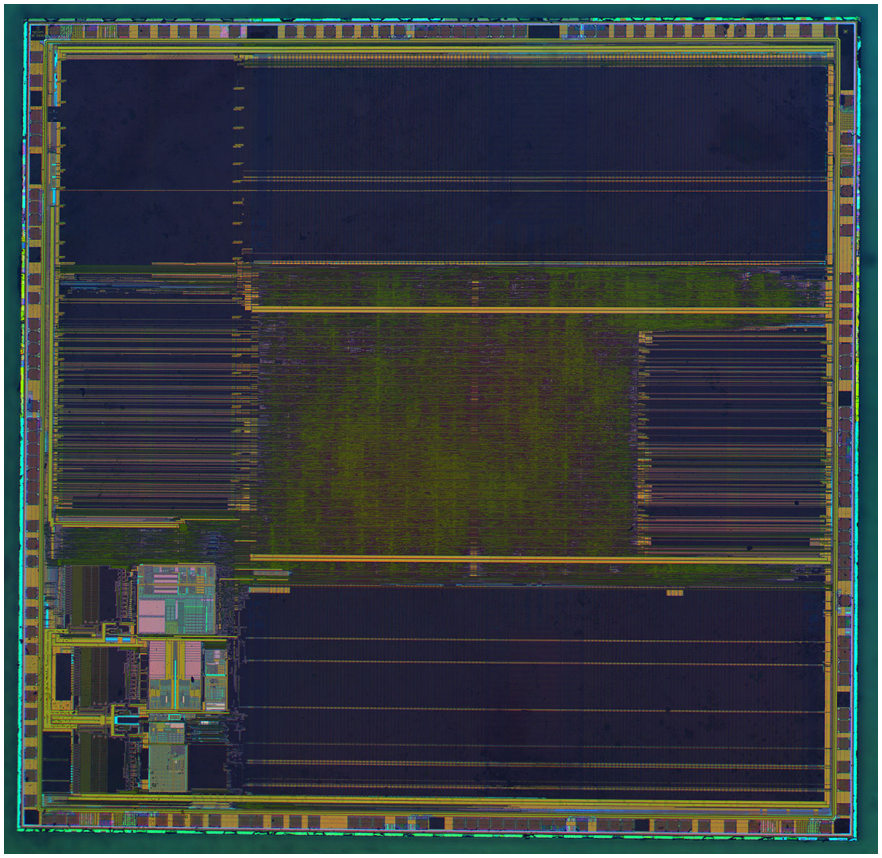


Figure 1.12: Internal die of an STM32F103 MCU

The remaining paragraphs in this chapter will introduce the reader to STM32 microcontrollers, giving a complete overview of all STM32 subfamilies.

1.2.1 Advantages of the STM32 Portfolio....

The STM32 platform provides several advantages for embedded developers. This paragraph tries to summarize the relevant ones.

- **They are Cortex-M based MCUs**: this could still not be clear to those of you who are new to this platform. Being Cortex-M based microcontrollers ensures that you have several tools available on the market to develop your applications. ARM has become a sort of standard in the embedded world (this is especially true for Cortex-A processors; in the Cortex-M market segment there are still several good alternatives: PIC, MSP430, etc.) and 160 billions of devices sold by 2020 is a strong guarantee that investing on this platform is a good choice.
- **Free ARM based tool-chain**: thanks to the diffusion of ARM based processors, it is possible to work with completely free tool-chains, without investing a lot of money to start working with this platform, which is extremely important if you are a hobbyist or a student.
- **Know-how reuse**: STM32 is a quite extensive portfolio, which is based on a common denominator: their main CPU platform. This ensures, for example, that know-how acquired

working on a given STM32Fx CPU can easily be applied to other devices from the same family. Moreover, working with Cortex-M processors allows you to reuse much of the acquired skills if you (or your purchase team) decide to switch to Cortex-M MCUs from other vendors (in theory).

- **Official development environment**: ST invested a lot in recent years in building-up a complete development environment. They made too many mistakes in the past, by funding several projects around that ended in nothing. CooCox IDE and SW4STM32 where two previous attempts by ST to support open-source communities in growing-up a complete *tool-chain* for its microcontrollers, but they failed miserably. Finally, ST understood that this was discouraging a lot of people from adopting the STM32 portfolio, and so they decided to acquire Atollic in order to use their TrueSTUDIO IDE as the official *tool-chain* for the STM32 family of microcontrollers.
- **Pin-to-pin compatibility**: most of STM32 MCUs are designed to be pin-to-pin compatible inside the extensive STM32 portfolio. This is especially true for LQFP64-100 packages, and it is a big plus. You will have less responsibility in the initial choice of the right microcontroller for your application, knowing that you can eventually jump to another family in case you find it does not fit your needs.
- **5V tolerant**: Most STM32 pins are 5V tolerant. This means that you can interface other devices that do not provide 3.3V I/O without using level shifters (unless speed is key to your application, a level shifter always introduce a parasitic capacitance that reduced the commutation frequency).
- ***32 cents for 32 bit***[13]: STM32F0 is the right choice if you want to migrate from 8/16-bit MCUs to a powerful and coherent platform, while keeping a comparable target price. You can use an RTOS to boost your application and write much better code.
- **Integrated bootloader**: STM32 MCUs are shipped with an integrated bootloader, which allows to reprogram the internal flash memory using some communication peripherals (USART, I²C, etc.). For some of you this will not be a killer feature, but it can dramatically simplify the work of people developing devices as professionals.

1.2.2 ....And Its Drawbacks

This book is not a brochure, or a document made by marketing people. Nor is the author an ST employee or is he having business with ST. So, it is right to say that there are some pitfalls regarding this platform.

- **Learning curve**: STM32's learning curve can be quite steep, especially for inexperienced users. If you are completely new to embedded development, the process of learning how to develop STM32 applications can be frustrating. Even if ST is doing a great job at trying to improve the overall documentation and the official libraries, it is still hard to deal with this platform, and this is a shame. Historically, ST documentation has not been the best one for inexperienced people, being too cryptic and lacking clear examples.

[13] Due to the silicon market crisis in the twenties, this slogan is no longer valid. The crazy situation of the IC industry pushed prices of low-cost ICs (which are the most affected ones) by more than 25%. However, misery loves company and for low-cost applications STM32F0 family is still an interesting series to evaluate, unless a 8-bit solution is suitable for you.

- **Fragmented and dispersive documentation**: ST is actively working on improving its official documentation for the STM32 platform. You can find a lot of huge datasheets on ST's website, but there is still a lack of good documentation especially for its HAL. Recent versions of the CubeHAL provide one or more "CHM" files[14], which are automatically generated from the documentation inside the CubeHAL source code. However, those files are not sufficient to start programming with this framework, especially if you are new to the STM32 ecosystem and the Cortex-M world.
- **Buggy and non-performing HAL**: frankly speaking, the official HAL from ST improved a lot over the years but it is still evolving, and it is quite common to find some severe bugs especially in advanced and less widespread modules of the HAL, like it happened to this author with the `HAL_PCD` module (I spent two months to identify a nasty bug affecting the USB HAL library on an STM32L052 MCU). ST is actively working on fixing the HAL bugs, but it seems we are still far from a "stable release". Moreover, their software release lifecycle is too old and not appropriate for the times we live in: bug fixes are released after several months, and sometimes the fix bares more issues than the broken code itself. Finally, the CubeHAL is far from being considered an optimized library. The HAL is designed to be abstracted from the underlying family and the specific MCU and this requires that HAL routines are full of `if-then-else` statement. For time-critical applications ST released the CubeHAL-LL library, which is essentially a set of macros to simplify the manipulation of peripherals' registers. Again, do not forget that you are the pilot and you have to rule all those tools.

1.3 A Quick Look at the STM32 Subfamilies

If you are new to the STM32 world and if you already own an STM32 development kit or a custom device that you are impatient to start using, then it is safe to skip this long (and quite tedious :-)) paragraph and to jump to the next one.

As you read, the STM32 is a rather complex product lineup, spanning over seventeen product sub-families. **Figure 1.13** summarizes the current STM32 portfolio. The diagrams aggregate the subfamilies in four macro groups: *High-performance*, *Mainstream*, *Ultra Low-Power* and *Wireless* MCUs.

High-performance microcontrollers are those STM32 MCUs dedicated to CPU-intensive and multimedia applications. They are Cortex-M3/4F/7 based MCUs, with maximum clock frequencies ranging from 120MHz (F2) up to 550MHz (H7). Some MCUs in this group provide ART™ Accelerator, an ST technology that allows *0-wait* execution from flash memory.

[14] a CHM file is a typical Microsoft file format used to distribute documentation in HTML format in just one file. It is really common on the Windows OS, and you can find several good free tools on MacOS and Linux to read them.

2. Get In Touch With SM32CubeIDE

Before we can start developing applications for the STM32 platform, we need a complete *tool-chain*. A tool-chain is a set of programs, compilers and tools that allows us:

- to write down our code and to navigate inside source files of our application;
- to navigate inside the application code, allowing us to inspect variables, function definition-s/declarations, and so on;
- to compile the source code using a cross-platform compiler;
- to upload and debug our application on the target development board (or a custom board we have made).

To accomplish these activities, we essentially need:

- an IDE with integrated source editor and navigator;
- a cross-platform compiler able to compile source code for the ARM Cortex-M platform;
- a debugger that allows us to execute step by step debugging of firmware on the target board;
- a tool that allows to interact with the integrated hardware debugger of our Nucleo board (the ST-LINK interface) or the dedicated programmer (e.g., a JTAG adapter).

In this chapter I will show how to install and to run the STM32CubeIDE tool-chain on Windows, Mac OS and Linux and I will provide to you an essential overview of the main functionalities of the Eclipse IDE.

2.1 Why Choose STM32CubeIDE as Tool-Chain for STM32

It has been a long time since the first edition of this book, and a lot of things are changed. Traditionally, STM32 lacked an official development environment fully, directly and actively maintained by ST. In the past years, ST tried to support several open source and community-based projects, all based on the free Eclipse/GCC tool-chains. However, none of these projects (CooCox, AC6) reached a real maturity level and this represented one of the major roadblocks in starting to work with STM32 microcontrollers.

The first edition of this book was characterized by the fact that it showed how to setup a complete, cross-platform and totally free tool-chain from scratch. It was an Eclipse-based tool-chain with the

addition of a set of plug-ins, named Eclipse Embedded CDT[1], developed and maintained by Liviu Ionescu, who did a really excellent work in providing support for the GCC ARM tool-chain. Without those plug-ins it was almost impossible to develop and run code with Eclipse for the STM32 platform. This project is now one of the official Eclipse Foundation projects, and it is still a good development environment to work with, especially if you are used to work with different Cortex-M platform.

At the end of 2017 STM decided to acquire Atollic[2], the company behind the TrueStudio IDE, a commercial distribution of Eclipse CDT and ARM GCC with the addition of dedicated plug-ins to develop embedded applications for ARM Cortex-M microcontrollers. After the acquisition of Atollic, ST decided to release the TrueStudio IDE for free for all STM32 developers, and the IDE was renamed in *STM32CubeIDE*. As we will see in this book, STM32CubeIDE is much more than a flavor of Eclipse CDT. ST invested a lot in integrating all the STM32-related tools inside just one piece of software, without requiring to developers to deal with the installation of several non-integrated tools scattered around the STM website. Moreover, STM finally completed the porting of all fundamental development tools to Linux and MacOS, allowing programmers to work with their favorite OS. This represents a true quantum leap for the STM32 platform, and nowadays I cannot see any real reason to use other development environments, unless you have strong requirements related to your very specific application (for example, to develop electronics for the automotive/aerospace industry). For this and other reasons better explained next, this edition of the book will be entirely based on the STM32CubeIDE.

However, despite the fact that STM32CubeIDE is now the official development environment by ST, there are several additional considerations to take in account while evaluating your tool-chain if you are in doubt about which to choose. Here you can find just a few considerations:

- **It is GCC based**: GCC is probably the best compiler on the earth, and it gives excellent results even with ARM based processors. ARM is nowadays the most widespread architecture (thanks to the embedded systems becoming widespread in the recent years), and many hardware and software manufacturers use GCC as the base tool for their platform.
- **It is cross-platform**: if you have a Windows PC, the latest sexy Mac or a Linux server you will be able to successfully develop, compile and upload the firmware on your development board with no difference. Nowadays, this is a mandatory requirement.
- **Eclipse diffusion**: a lot of IDEs for STM32 are also based on Eclipse, which has become a sort of standard. There are a lot of useful plug-ins for Eclipse that you can download with just one click. And it is a product that evolves day by day.
- **Eclipse It is Open Source**: ok. I agree. For such giant pieces of software, it is really hard to try to understand their internals and modify the code, especially if you are a hardware engineer committed to transistors and interrupts management. But if you get in trouble with your tool, it is simpler to try to understand what goes wrong with an open source tool than a closed one.
- **Large and growing community**: these tools have by now a great international community, which continuously develops new features and fixes bugs. You will find tons of examples and blogs, which can help you during your work. Moreover, many companies, which have adopted

[1] https://eclipse-embed-cdt.github.io/
[2] https://www.st.com/content/st_com/en/about/media-center/press-item.html/c2839.html

this software as official tools, give economical contribution to the main development. This guarantees that the software will not suddenly disappear.
- **It is free**: Yep. I placed this as the last point, but it is not the least. As said before, a commercial IDE can cost a fortune for a small company or a hobbyist/student. And the availability of free tools is one of the key advantages of the STM32 platform.

If you are completely new to Eclipse and/or GCC, here are some more specific considerations regarding these two products.

2.1.1 Two Words About Eclipse...

Eclipse[3] is an Open Source and a free Java based IDE. Despite this fact (unfortunately, Java programs tend to eat a lot of machine resources and to slow down your PC), Eclipse is one of the most widespread and complete development environments. Eclipse comes in several pre-configured versions, customized for specific uses. For example, the *Eclipse IDE for Java Developers* comes preconfigured to work with Java and with all those tools used in this development platform (Ant, Maven, and so on). In our case, the STM32CubeIDE is essentially based on the *Eclipse IDE for C/C++ Developers*.

Eclipse is designed to be expandable thanks to plug-ins. There are several plug-ins available in Eclipse Marketplace useful for software development for embedded systems. We will install and use most of them in this book. Moreover, Eclipse is highly customizable. I strongly suggest you to take a look at its settings, which allow you to adapt it to your needs and flavor.

2.1.2 ... and GCC

The GNU Compiler Collection[4] (GCC) is a complete and widespread compiler suite. It is the only development tool able to compile several programming languages (front-end) to tens of hardware architectures that come in several variants. GCC is a really complex piece of software. It provides several tools to accomplish compilation tasks. These include, in addition to the compiler itself, an assembler, a linker, a debugger (known as *GNU Debugger* - GDB), several tools for binary files inspection, disassembly and optimization. Moreover, GCC is also equipped with the *run-time* environment for the C language, customized for the target architecture.

In recent years, several companies, even in the embedded world, have adopted GCC as their official compiler. For example, NXP uses GCC as cross-compiler for its LPC family of Cortex microcontrollers.

[3] http://www.eclipse.org
[4] https://gcc.gnu.org/

What Is a Cross-Compiler?

We usually refer to term *compiler* as a tool able to generate machine code for the processor in our PC. A compiler is just a "language translator" from a given programming language (C in our case) to a low-level machine language, also known as *assembly*. For example, if we are working on Intel x86 machine, we use a compiler to generate x86 assembly code from the C programming language. For the sake of completeness, we have to say that nowadays a compiler is a more complex tool that addresses both the specific target hardware processor and the Operating System we are using (e.g., Windows 7).

A *cross-platform compiler* is a compiler able to generate machine code for a hardware machine **different** from the one we are using to develop our applications. In our case, the GCC ARM Embedded compiler generates machine code for Cortex-M processors while compiling on an x86 machine with a given OS (e.g., Windows or Mac OSX).

In the ARM world, GCC is the most used compiler especially due the fact that it is used as main development tool for Linux based Operating Systems for ARM Cortex-A processors (ARM microcontrollers that equip almost every mobile device). ARM engineers actively maintain the ARM GCC branch of GCC. STM32CubeIDE uses one of the most recent GCC based tool-chains. Finally, consider that acquiring knowledge about this suite of compilers can be also useful in future: it is a skill that can be reused also for other embedded architectures.

2.2 Downloading and Installing the STM32CubeIDE

The STM32CubeIDE can be freely downloaded from the official STM website[5]. The only requirement is that you register on the STM website providing a valid email address.

In the same webpage you can find the installation packages for all operating systems. You will find five links, as shown in **Figure 2.1**[6]. Three links are related to Linux, while the other two are for Windows and MacOS:

- **STM32CubeIDE-Win**: this executable package contains the installer for Windows.
- **STM32CubeIDE-Mac**: this ZIP file contains the DMG file (Apple Disk Image) with the installer for Mac OSX.
- **STM32CubeIDE-DEB**: this package contains a Linux Debian installation package (`.deb` package). This is for Linux Debian distributions and derived (notably Ubuntu).
- **STM32CubeIDE-RPM**: this package contains a Linux RedHat installation package (`.rpm` package). This is for Linux RedHat distributions and derived (notably CentOS).
- **STM32CubeIDE-Lnx**: this is a generic Linux tarball containing the STM32CubeIDE and all necessary tools and libraries. This package is for *advanced* Linux users, who usually know how to install by themselves custom applications.

[5] https://www.st.com/en/development-tools/stm32cubeide.html

[6] In this book all screen captures, unless differently required, are based on Mac OS, because it is the OS the author uses to develop STM32 applications (and to write this book). However, they also apply to other Operating Systems.

Select the porting of STM32CubeIDE for your Operating System by clicking on the pinky icon "**Get Software**" in the download section. Once the software is downloaded, follow the instructions in the next sections.

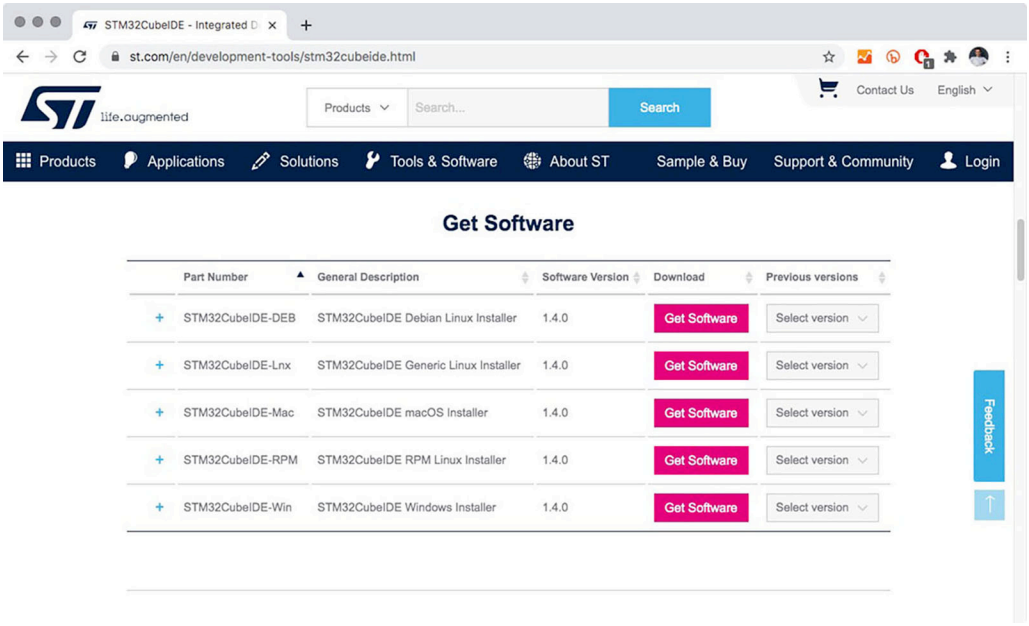


Figure 2.1: The STM32CubeIDE download page on the official STM website

The next three paragraphs, and their sub-paragraphs, are almost identical. They only differ on those parts specific for the given OS (Windows, Linux or Mac OS). So, jump to the paragraph you are interested in, and skip the remaining ones.

2.2.1 Windows - Installing the Tool-Chain

The Windows installation package is contained inside a ZIP file. Once the download has completed, extract the ZIP archive and run the contained executable file (the executable filename has this structure: `st-stm32cubeide_VERSION_x86_64.exe`, where `VERSION` corresponds to the latest release of the IDE).
During the installation process, the Windows may display a dialog stating: "*Do you want to allow this app to make changes to your device?*" with info "*Verified publisher: STMicroelectronics Software AB*". Accept by clicking on "YES" to let the installer continue.

- `libusb-1.0.0-dev`

The installation of this packaged changes according to the specific Linux distribution.
In Ubuntu they can be installed with the following commands at the terminal prompt:

```
$ sudo apt-get install libusb libusb-1.0.0-dev
```

You can download STM32CubeProgrammer from the official ST page[8] (the download link is at the bottom of the page in the "**Get Software**" section). Once the download is completed, extract the `.zip` package. You will find the `SetupSTM32CubeProgrammer-2.9.0.linux` file. Run it and follow the installation instructions.

Jump to the STM32CubeIDE overview paragraph if you are totally new to the Eclipse IDE.

2.2.3 Mac - Installing the Tool-Chain

The MacOS installation package is contained inside a ZIP file. Once the download has completed, extract the ZIP archive and run the contained DMG file (the filename has this structure: `st-stm32cubeide_VERSION_x86_64.dmg`, where `VERSION` corresponds to the latest release of the IDE). Double-click on the DMG file to let MacOS mount the Apple disk image. The *License Agreement* dialog appears, as shown in **Figure 2.5**.

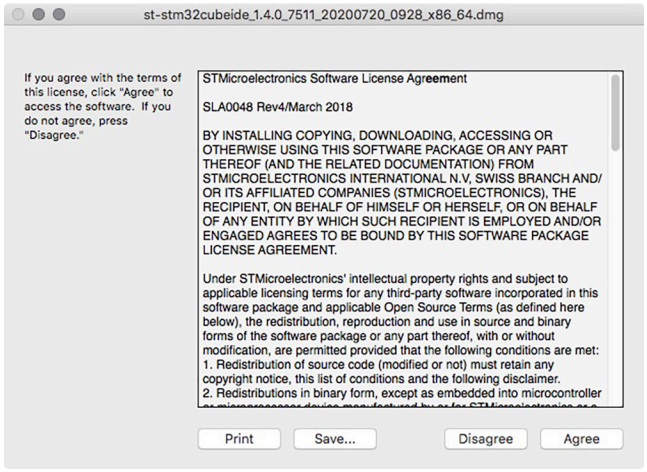


Figure 2.5: ***License Agreement*** **dialog**

Read the license agreement and click "Agree" to accept the terms of the agreement and to move on with the software installation. The *Install page* appears, as shown in **Figure 2.6**. Before we drag the large IDE icon inside the `Application` folder it is important to install the ST-LINK server package first. So, click on the *Install me 1st* icon (the classical icon representing a software package in MacOS) and follow the installation instructions.

[8] http://bit.ly/2CK4aFa

MacOS will prevent you from installing the software, being it download from an untrusted website and not from the official App Store. However, I assume that you are sufficiently familiar with MacOS and able to bypass this restriction by going in the MacOS **System Preferences->Security and Privacy settings**. Alternatively, you can completely disable the MacOS Gatekeeper (the component that enforces code signing and verifies downloaded applications before allowing them to run in recent MacOS releases) by running the following command at MacOS terminal:

```
$ sudo spctl --master-disable
```

Once completed, you can safely drag the IDE icon inside the `Application` folder and wait until the operation completes.

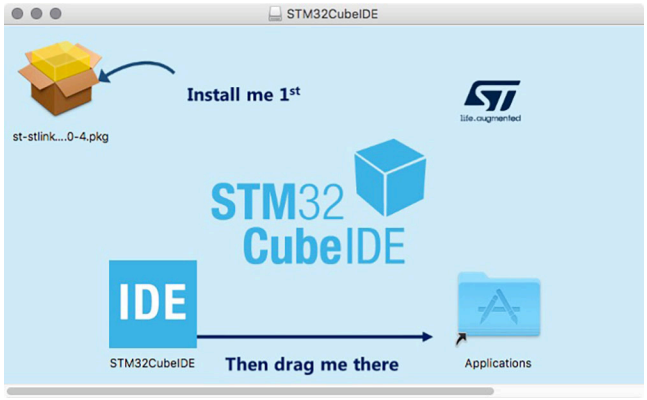
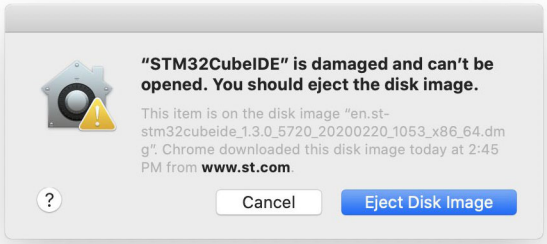


Figure 2.6: ***Install page*** dialog

Read Carefully!

It is very common for MacOS users to fail to run the STM32CubeIDE the first time they launch the application. The following system warning is shown:



Unfortunately, this error is due to wrong extended attribute permissions, and it is mostly related to the new path of MacOS X, which is driving MacOS toward a sort of more advanced iOS. Honestly speaking, being a really old and advanced MacOS user, I cannot see anything good with this drift.
However, the good news is that you can easily get rid of this issue by opening the MacOS Terminal and executing the following command at the prompt:

```
$ xattr -c /Applications/STM32CubeIDE.app
```

This should fix the issue and you should be able to run the STM32CubeIDE.

The next tool to install is the STM32CubeProgrammer. It is a software that uploads the firmware on the MCU using the ST-LINK interface of our Nucleo, or a dedicated ST-LINK programmer. We will not use this tool too much in the book, apart for the next chapter. However, this tool comes in handy very often during the common development life cycle, especially if for small production lots. So, I think that it is ok to get familiar with this tool.

You can download STM32CubeProgrammer from the official ST page[9] (the download link is at the bottom of the page in the "**Get Software**" section). Once the download is completed, extract the `.zip` package. You will find the `SetupSTM32CubeProgrammer-2.9.0.app` file. Run it and follow the installation instructions.

The tool-chain installation is complete, and you can jump to the next paragraph if you are totally new to the Eclipse IDE.

2.3 STM32CubeIDE overview

Now that we have completed the installation of the tool-chain, we can have a first look to the main interface and functionalities.

[9] http://bit.ly/2CK4aFa

When you start Eclipse you are asked to indicate a workspace directory, as shown in **Figure 2.7**. You are free to point this folder in any location of your hard drive.

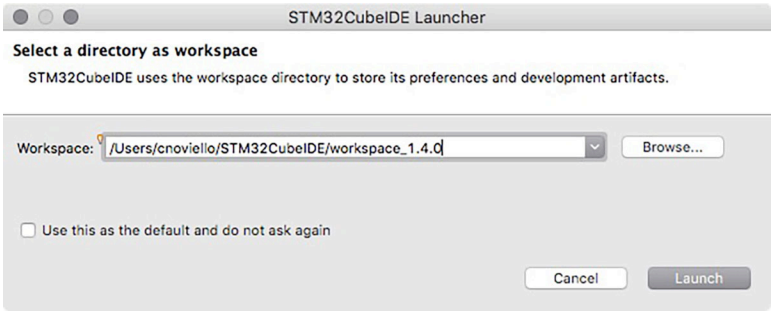


Figure 2.7: Eclipse workspace selection dialog

The workspace is a directory on the disk where the Eclipse platform and all the installed plug-ins store preferences, configurations and temporary information. Subsequent Eclipse invocations will use this storage to restore the previous state. As the name suggests, it is your "space of work". It defines your area of interest during an Eclipse session. Apart from IDE configuration parameters, a workspace is also a repository for all projects belonging to a given workspace.

Having more than a workspace is mostly a programmer's choice, who can organize projects and IDE configurations according to personal needs. If you do not plan to have multiple workspaces, you can check the flag *Use this as the default and do not ask again*. Eclipse will automatically open the workspace during startup.

Anytime you decide to change your mind and want to switch to a new workspace, you can overwrite the default configuration by clicking on **File->Switch workspace->Other**.... The dialog in **Figure 2.7** will appear again and you will be able to select a different workspace.

Once the default workspace location is set, click on the **Launch** button, and wait for the complete Eclipse startup.

When you start STM32CubeIDE, you might be a bit puzzled by its interface if you are new to Eclipse. **Figure 2.8** shows how Eclipse appears when started for the first time.

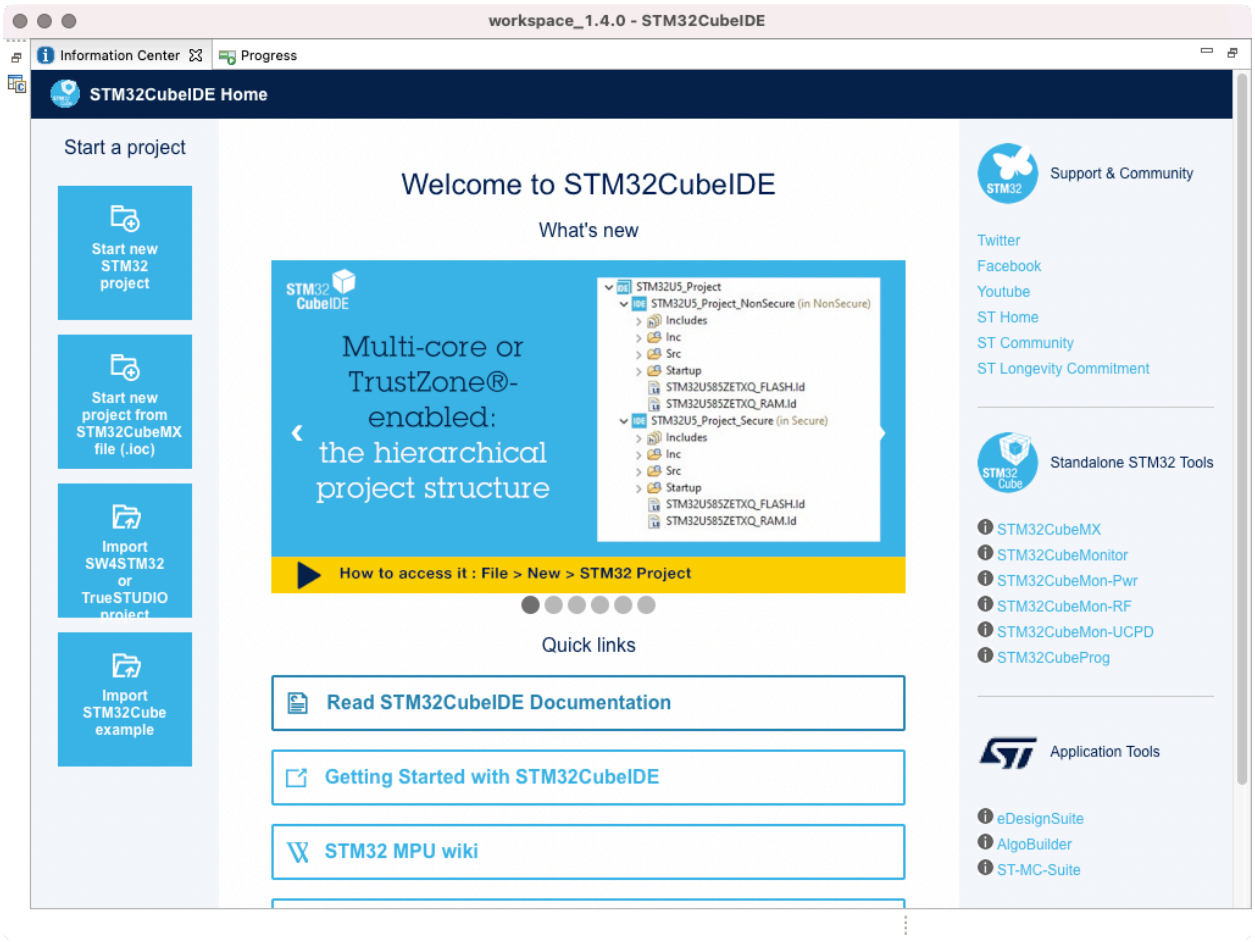


Figure 2.8: The Eclipse interface once started for the first time

Eclipse is a multi-view IDE, organized so that all the functionalities are displayed in one window, but the user is free to arrange the interface at its needs. When Eclipse starts, a welcome screen is presented. The content of that *Welcome Tab* is called *view*.

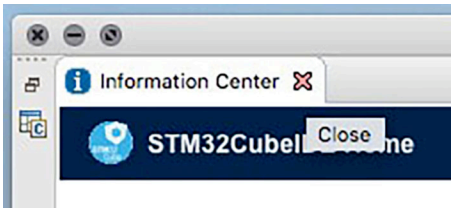


Figure 2.9: How to close the *Welcome view* by clicking on the *X*.

To close the *Welcome view*, click on the cross icon, as shown in **Figure 2.9**. Once the *Welcome view* goes away, the *C/C++ perspective* appears, as shown in **Figure 2.10**.

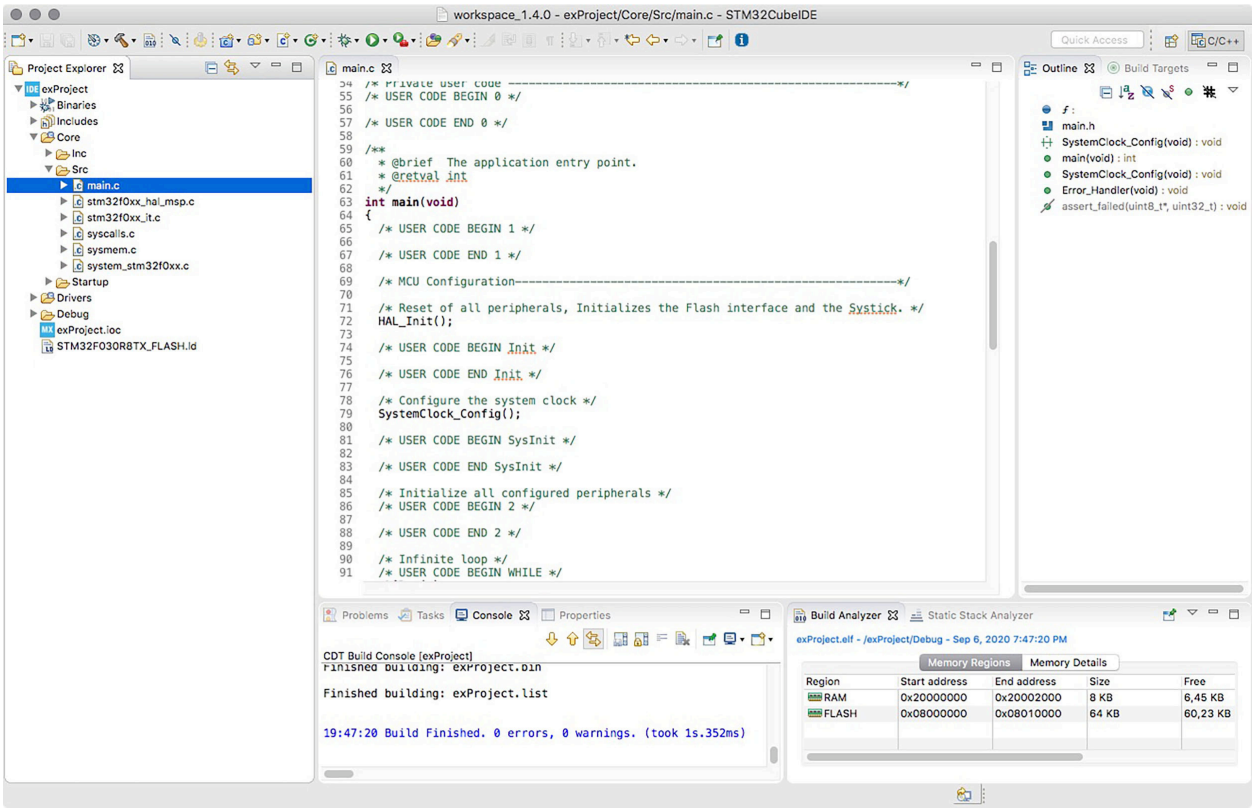
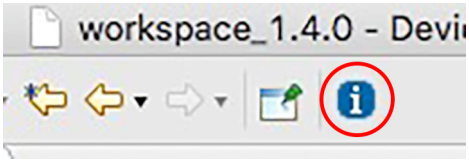


Figure 2.10: The C/C++ perspective view in eclipse (with a main.c file loaded later)

In case you want to show back the welcome view, click on the icon circled in red on the main toolbar and shown in the picture below.



In Eclipse a *perspective* is a way to arrange views in a manner that is related to the functionalities of the perspective. The *C/C++ perspective* is dedicated to coding, and it presents all aspects related to the editing of the source code and its compiling. It is divided into four views.

The view on the left, named *Project Explorer*, shows all projects inside the workspace. The centered view, that is the larger one, is the C/C++ editor. Each source file is shown as a tab, and it is possible to have many tabs opened at the same time.

The views in the bottom of Eclipse window are dedicated to several activities related to compiling, and they are subdivided into tabs. For example, the *Console* tab shows the output from the compiler;

the *Problems* tab organizes all messages coming from the compiler in a convenient way to inspect them; the *Search* tab contains the search results.

The view on the right contains several other tabs. For example, the *Outline* tab shows the symbols contained in each source file (functions, variables, and so on), allowing quickly navigation inside the file content.

There are other views available (and many other ones that are provided by custom plug-ins). Users can see them by going inside the **Window->Show View->Other**... menu. Some of them will be analyzed in later chapters.

Sometimes it happens that a view is "minimized" and it seems to disappear from the IDE. When you are new to Eclipse, this might lead to frustration trying to understand where it went.
For example, looking at **Figure 2.11** it seems that the *Project Explorer* view has disappeared, but it is simply minimized and you can restore it clicking on the icon circled in red.
However, sometimes the view has really been closed. This happens when there is only one tab active in that view and we close it. In this case you can enable the view again going in the **Window->Show View->Other**... menu.

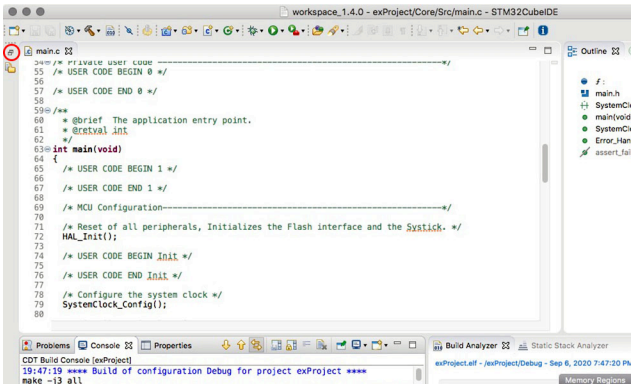


Figure 2.11: *Project Explorer* view minimized

To switch between different perspectives, you can use the specific toolbar available in the top-right side of Eclipse (see **Figure 2.12**)

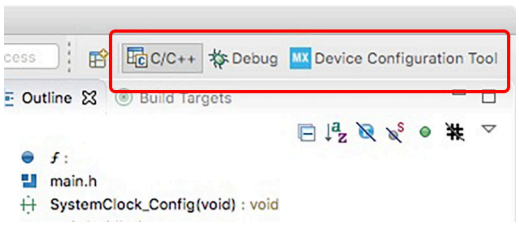
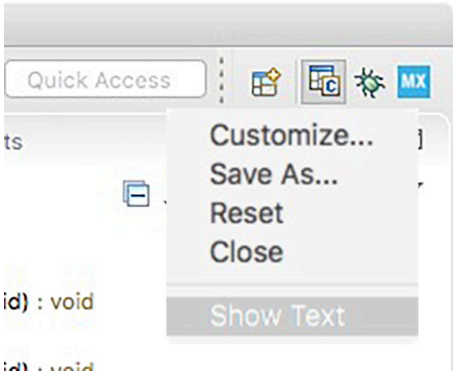


Figure 2.12: Perspective switcher toolbar

By default, the other available perspective is *Debug*, which we will see in more depth later. You can enable other perspectives by going to **Window->Perspective->Open Perspective->Other**... menu.

Starting from Eclipse 4.6 (aka Neon), the perspective switcher toolbar no longer shows the perspective name by default, but only the icon associated to the perspective. This tends to confuse novice users. You can show the perspective name near its icon by clicking with the right button of the mouse on the toolbar and selecting the **Show Text** entry, as shown below.



Eclipse IDE is designed with a main toolbar that adapts its content according to the type of files selected inside the main perspective view. The most common and relevant icons in the toolbar are explained in the **Table 2.1**.

Icon	Description
	This icon corresponds to the File->New menu. It allows to quick creating a new Project or adding a source file to the project.
	This icon allows to switch between different Project Build Configurations. By default every Project has two configurations, **Debug** and **Release**, used to generate a binary file equipped with all necessary Debug information or ready for the production. You are free to create as many configurations you want. This comes really useful for real-life projects.
	This icon perform a Project build for the selected Project Configuration.
	This icon perform a Project build for all currently active projects.
	This icon forces a project generation. It's useful when we change a configuration inside the Device Configuration Tool (STM32CubeMX) and we need to generate the updated C code.
	These icons are related to the creation of new C files, new project and new classes for C++.
	These icons are related to debugging. We'll analyze them in a later chapter.

Table 2.1: Main Eclipse's toolbar icons

As we go forward with the topics of this book, we will have a chance to see other features of Eclipse.

3. Hello, Nucleo!

There is no programming book that does not begin with the classic "Hello world!" program. And this book will follow the tradition. In the previous chapter we have configured the STM32CubeIDE needed to develop STM32 embedded applications. So, we are now ready to start coding.

In this chapter we will create a really basic program: a blinking LED. We will use the STM32CubeIDE to create a complete application in a few steps without dealing, in this phase, with aspects related to the ST *Hardware Abstraction Layer* (HAL) and the MCU graphical configurator (better known as STM32CubeMX). I am aware that not all details presented in this chapter will be clear from the beginning, especially if you are totally new to embedded programming.
However, this first example will allow us to become familiar with the development environment. Following chapters, especially the next one, will clarify a lot of obscure things. So, I suggest you to be patient and try to take the best from the following paragraphs.

3.1 Create a Project

Let us create our first project. We will create a simple application that makes the Nucleo `LD2` LED (the green one) blink.
Go to **File->New->STM32 Project**. Depending on your current IDE configuration, the **Target Selection** wizard (shown in **Figure 3.1**) will appear in a while.

The first time you run the Target Selection wizard it may require several seconds to show. A progress indicator will prompt you about the fetching of MCUs and boards specifications from ST's servers. New STM32 microcontrollers or development kits are released monthly, and the tool is designed so that it is not wired with the complete list of part numbers. So, be patient and let the software to complete the operations.

As we will learn later, the Target Selection wizard is part of a more complex piece of software formerly known as STM32CubeMX. This tool is designed to dramatically streamline the configuration process of hardware peripherals of a given STM32 MCU, as well as the generation of all necessary library files to drive those peripherals. The next chapter will be entirely based on the explanation of the most relevant STM32CubeMX functionalities. So, we will not spend too much time on this matter now.

Now click on the **Board Selector** tab (highlighted in light blue in **Figure 3.1**), expand the menu **Type** and check the **Nucleo-64** family of boards. Dig inside the list of boards to find your specific Nucleo-64 board.

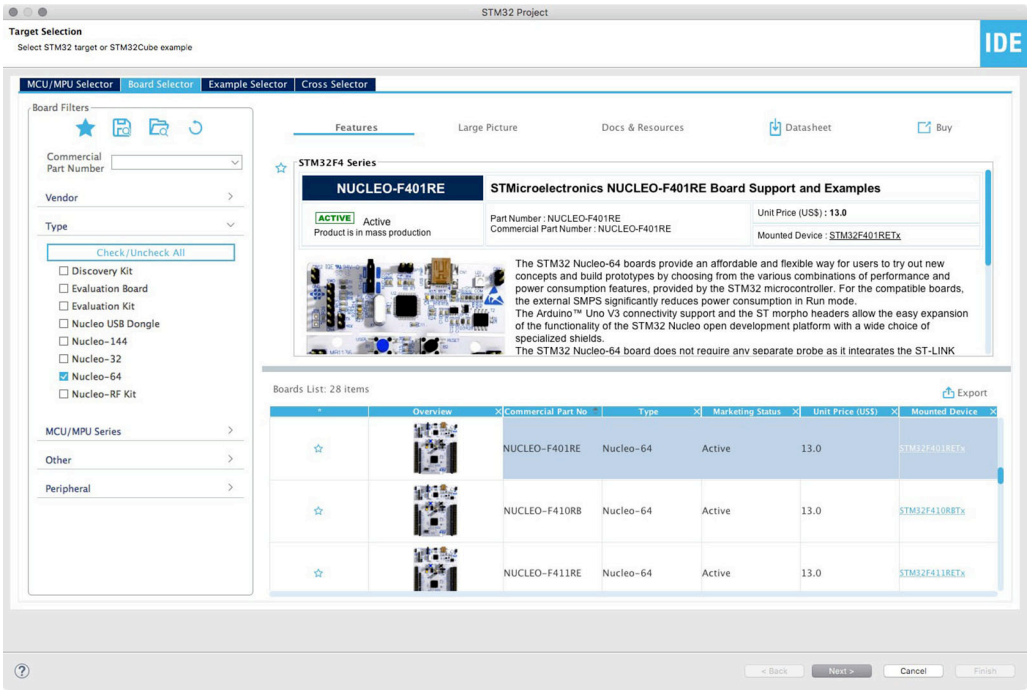


Figure 3.1: The Target Selection wizard - STEP 1

As mentioned in Chapter 1, this book is entirely based on the Nucleo-64 board, and the examples in the text are tested for the boards listed in **Table 1.18**. However, the aim of this book is to teach the foundation concepts of STM32 programming. My opinion is that it should be relatively easy to adapt this and all other examples in the text to any other development board (Nucleo-32, Nucleo-144, Discovery, and so on).

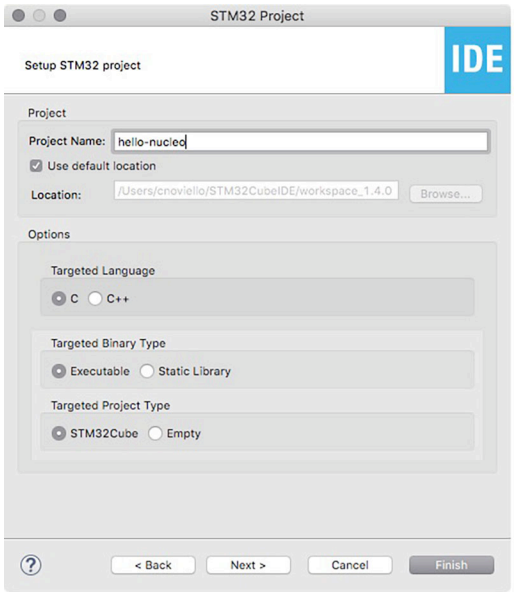


Figure 3.2: Project wizard - STEP 2

Once you select the board in the list, click on **Next** button. Specify a Project Name in the relative field (*hello-nucleo* in our case, but feel free to choose what it is best for you) and leave all other flags with the default configuration, as shown in **Figure 3.1**. Click on **Finish** to start the project generation.
The STM32CubeIDE will ask you if you want to initialize all peripherals with their default mode, as shown in **Figure 3.3**. What does this mean?

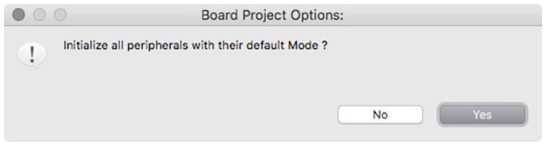


Figure 3.3: Project wizard - STEP 3

In every development board the target MCU (that is, the main MCU where the firmware is uploaded) is both equipped with internal peripherals (e.g., the *Real Time Clock* - RTC) and wired to several external other devices through individual GPIO pins. For example, in every Nucleo-64 board one GPIO is directly connected to the green USER LED, marked as `LD2` on the PCB. To use those "default" peripherals, we have to properly configure both the internal peripheral and the corresponding GPIOs. By answering **Yes** to this question, we instruct the IDE to automatically perform all the necessary configurations to use all internal and external default peripherals for the Nucleo-64 board.
As soon as we will dive into the book, you will learn how to configure every peripheral by yourself. However, in this phase, to keep things simple, click on the **Yes** and let the IDE do the magic.

The STM32CubeIDE will start the generation of the new project. If this is the first time you create a project for the STM32 family of your Nucleo board (that is, STM32F0, STM32F4, etc.), the IDE needs to download the corresponding *Cube Firmware Package* (for example, if your board is a Nucleo-F401RE, it needs to download the `stm32cube_fw_f4_v1XXX.zip` package for the STM32F4 family).

These packages contain several relevant components:

- **The complete HAL for the given STM32 family**: the *Hardware Abstraction Layer* (HAL) is the set of libraries that allow to drive the microcontroller peripherals and core features without dealing with the details of the given MCU. This book is entirely based on the STM32CubeHAL and we will learn a lot regarding this quite complex library.
- **Additional Middleware packages**: some STM32 MCUs integrate advanced peripherals that require additional libraries to be used (either developed by ST or by third parties) . For example, to program the USB controller integrated in some STM32 MCUs it is necessary to use the complete USB stack freely provided by ST. Every Cube Firmware package comes with a set of Middleware libraries, and we will analyze several of them in the third part of this book.
- **Examples projects for development boards**: ST provides a lot of complete and working examples for its development boards. Every example is made to show how-to use a given feature of the board. Every Cube Firmware package integrates several examples, and you can browse the complete example list by clicking on the **Example Selector** tab in the **Target Selection** wizard (see **Figure 3.1**).

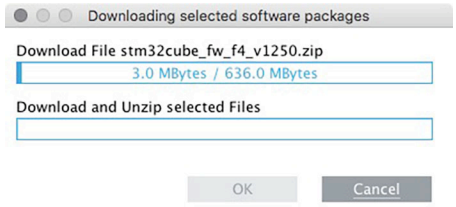


Figure 3.4: Cube Firmware Package download dialog

Depending on the STM32 family, the size of a Cube Firmware package can be quite relevant. The basic rule is that the more powerful the STM32 family is the larger the corresponding Cube Firmware is. So, be patient and wait for the complete download, as shown by the specific dialog (see **Figure 3.4**).

3.2 Adding Something Useful to the Generated Code

Figure 3.5 shows what appears in the STM32CubeIDE after the project has been generated. The Project Explorer view shows the project structure. This is the content of the first-level folders (going from top to bottom)[1]:

`Includes`
: This folder shows all folders that are part of the *GCC Include Folders*[2].

[1] We are not going to describe now all generated files and folders. This is just a brief introduction. In later chapters we will have the opportunity to better analyze them. Do not spend too much time on them now.

[2] Every C/C++ compiler needs to be aware of where to look for include files (files ending with **.h**). These folders are called *include folders* (or, more properly, *include paths*) and their paths must be specified to GCC using the **-I** parameter. However, Eclipse can do this for us automatically and the *include folder* visually shows the search paths where GCC will look for include files.

`Core`
: This Eclipse folder contains the generated project, made of several `.c` files that make up our application. One of these files is `src/main.c`, which contains the `int main(void)` routine that we are going to customize in a while.

`Drivers`
: This Eclipse folder usually contains header and source files of many relevant libraries (like, among the other, the ST CubeHAL and the CMSIS package). We will see them more in depth in the next chapter.

`hello-nucleo.ioc`
: This file is the STM32CubeMX project graphically shown in the **Device Configuration Tool** view (the default main view after project generation and shown in **Figure 3.5**). We will analyze deeply the STM32CubeMX interface and functionalities in the next chapter.

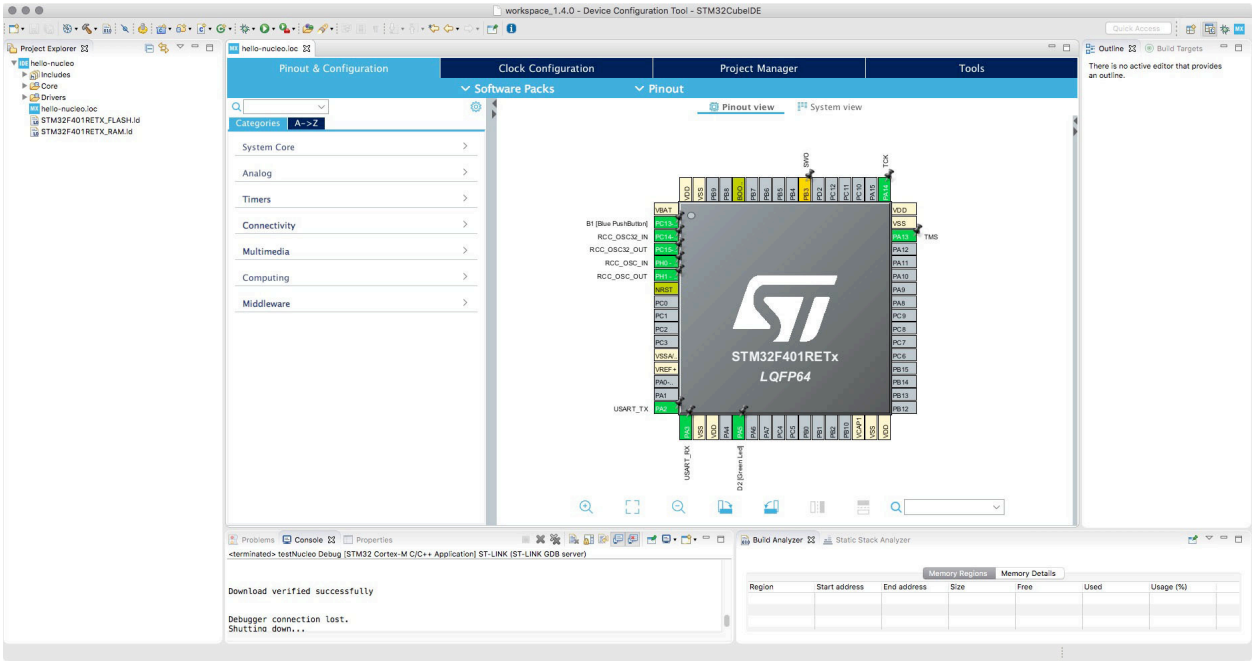


Figure 3.5: The STM32CubeIDE interface after complete project generation

We are now ready to start working at the real core application. The project generated by the IDE automatically contains all the necessary code to build a self-consistent application. To make the `LD2` LED blink we need to add just two lines of code to the `main()` function, the entry point[3] of our custom application.

So, in the **Project explorer** pane, expand the folders **Core->Src** and double click on the `main.c` file. Go at around line 66, where you can find the definition of the `main()` function. Here you can find

[3]Experienced STM32 programmers know that it is improper to say that the `main()` function is the entry point of an STM32 application. The execution of the firmware begins much earlier, with the calling of some important setup routines that create the execution environment for the firmware. However, from the *application point of view*, its start is inside the `main()` function. A following chapter will show in detail the bootstrap process of an STM32 microcontroller.

the invocation of four routines[4]:

- `HAL_Init()`: this function initializes the CubeHAL framework. It is responsible of the very first MCU initialization. We will analyze this function later in the book.
- `SystemClock_Config()`: this function plays a really paramount role, since it configure the MCU to work with one of the possible clock sources. STM32 MCUs offer the possibility to use several different clock sources. This is an advanced topic that we will deepen in Chapter 10.
- `MX_GPIO_Init()`: this function initializes I/O pins, according to the graphical configuration done in STM32CubeMX. In our case it is going to configure the GPIO pin associated to the `LD2` LED. More about these topics in Chapter 6.
- `MX_USART2_UART_Init()`: this function initializes the UART2 peripheral, which is wired to the ST-LINK interface in all Nucleo boards. More about this in Chapter 8.

Right after the invocation of those four initialization routines, you can find a `while` loop, an infinite loop where all the firmware activities take place. Here you can add two function calls, as shown next at lines 101-102.

Filename: `src/main.c`

```
66  int main(void)
67  {
68    /* USER CODE BEGIN 1 */
69
70    /* USER CODE END 1 */
71
72    /* MCU Configuration--------------------------------------------------------*/
73
74    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
75    HAL_Init();
76
77    /* USER CODE BEGIN Init */
78
79    /* USER CODE END Init */
80
81    /* Configure the system clock */
82    SystemClock_Config();
83
84    /* USER CODE BEGIN SysInit */
85
86    /* USER CODE END SysInit */
87
88    /* Initialize all configured peripherals */
89    MX_GPIO_Init();
```

[4]Depending on your development board, especially if you are not using a Nucleo-64, you may find additional routines invoked inside the `main()` function. This means that your development board provides additional peripherals compared to the standard Nucleo-64 board (remember that we answered "Yes" when STM32CubeIDE asked us to automatically initialize all peripherals in their default mode. So, do not care too much if your `main()` differs a little from the one shown here.

```
90    MX_USART2_UART_Init();
91    /* USER CODE BEGIN 2 */
92
93    /* USER CODE END 2 */
94
95    /* Infinite loop */
96    /* USER CODE BEGIN WHILE */
97    while (1)
98    {
99      /* USER CODE END WHILE */
100     /* USER CODE BEGIN 3 */
101     HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
102     HAL_Delay(500);
103   }
104   /* USER CODE END 3 */
```

You will have noticed that the code generated by CubeMX is full of these commented regions:

```
/* USER CODE BEGIN 1 */
...
/* USER CODE END 1 */
```

What are those comments for? CubeMX is designed so that if you change the hardware configuration you can regenerate the project code without losing the pieces of code you have added. Placing your code inside those "guarded regions" should guarantee that you will not lose your work. However, I have to admit that sometimes CubeMX can make a mess with generated files, and the user code goes lost. So, I suggest always generating another separated project and doing a copy and paste of the changed code inside the application files. This also gives you the full control over your code.

The code should be self-explanatory. The `HAL_GPIO_TogglePin()` function simply inverts the logical state of the PIN connected to the LD2 LED (which corresponds to PIN 5 of the GPIO port A in all Nucleo-64 boards), while the function `HAL_Delay()` is just a busy wait spin that lasts 500ms: the `LD2` so will blink at 1HZ rate.

How can we know to which pin the LED is connected? ST provides schematics[5] of the Nucleo board. Schematics are made using the *Altium Designer* CAD, a quite expensive piece of software used in the professional world. However, luckily for us, ST provides a convenient PDF with schematics. Looking at page 4, we can see that the LED is connected to the PA5 pin[6], as shown in **Figure 3.6**.

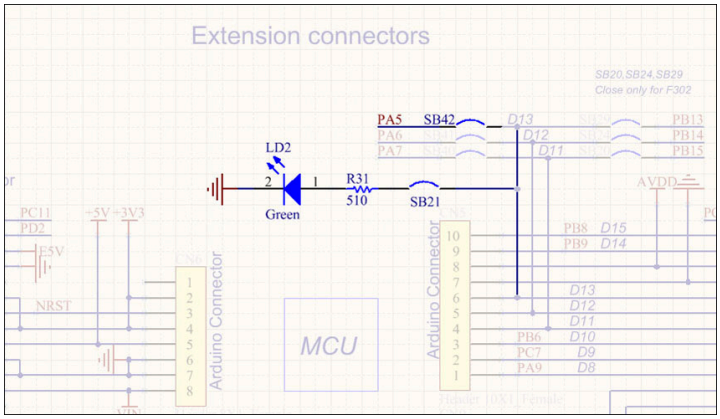


Figure 3.6: `LD2` connection to PA5

PA5 is shorthand for PIN5 of GPIOA port, which is the standard way to indicate a GPIO in the STM32 world. Finally, STM32CubeMX automatically defines the macro `LD2_GPIO_Port` and `LD2_Pin` so that their expansion corresponds to GPIOA port and PIN5.

We can now compile the project. Go to menu **Project->Build Project**. After a while, we should see something similar to this in the output console[7].

```
arm-none-eabi-size   hello-nucleo.elf
arm-none-eabi-objdump -h -S  hello-nucleo.elf  > "hello-nucleo.list"
arm-none-eabi-objcopy  -O binary  hello-nucleo.elf  "hello-nucleo.bin"
   text           data            bss            dec            hex        filename
   8224             20           1636           9880           2698        hello-nucleo.elf
Finished building: default.size.stdout
Finished building: hello-nucleo.bin
Finished building: hello-nucleo.list

15:22:52 Build Finished. 0 errors, 0 warnings. (took 5s.769ms)
```

[5] http://bit.ly/1FAVXSw

[6] Except for the Nucleo-F302RB, where `LD2` is connected to PB13 port. More about this next.

[7] The number of bytes required for each binary section (`text`, `bss`, and so on) may differ from the yours. This happens because the HALs differ between each STM32 series and due to different compiler optimization levels. Do not pay attention to these details that will be much clearer later.

3.3 Connecting the Nucleo to the PC

Once we have compiled our test project, you can connect the Nucleo board to your computer using a USB cable connected to micro-USB port (called **VCP** in **Figure 3.7**). You should see at least two LEDs turning ON.

Read Carefully

Please, ensure that the USB port is able to provide sufficient power to the board. It is strongly suggested to use a USB port able to provide at least 500mAh or a self-powered external hub.

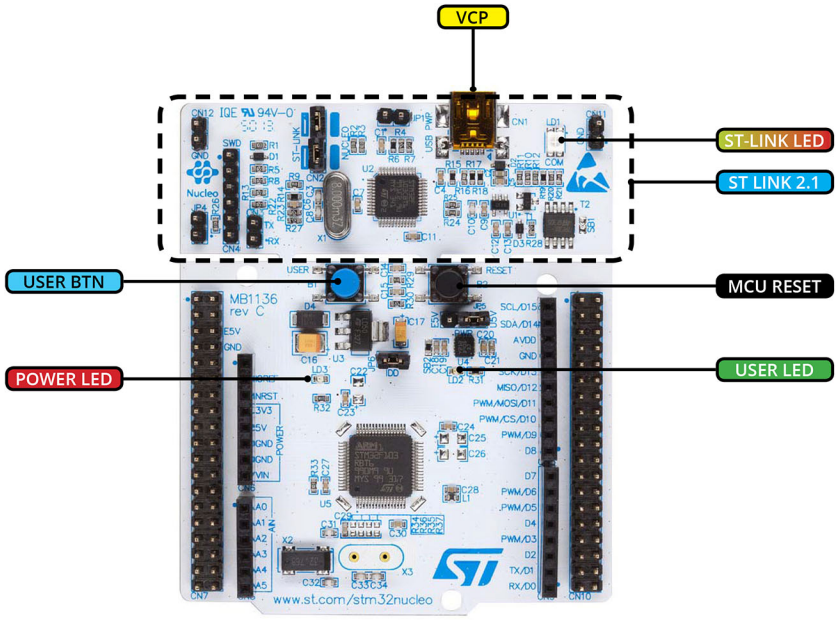


Figure 3.7: A Nucleo board and its main interfaces

The first one is the `LD1` LED, which in **Figure 3.7** is labeled **ST-LINK LED**. It is a red/green LED, and it is used to signal the ST-LINK activity: once the board is connected to the computer, that LED is green; during a debug session or while uploading the firmware on the MCU it blinks green and red alternatively.

Another LED that turns ON when the board is connected to the computer is the LED `LD3`, which is labeled **POWER LED** in **Figure 3.7**. It is a red LED that turns ON when the USB port ends *enumeration*, that is the ST-LINK interface is properly recognized by the computer OS as a USB peripheral. The target MCU on the board is powered only when that LED is ON (this means that the ST-LINK interface also manages the powering of the target MCU).

Finally, if you have not still flashed your board with a custom firmware, you will see that the `LD2` LED, the green LED labeled **USER LED** in **Figure 3.7**, also blinks: this happens because ST preloads

the board with a firmware that makes the LD2 LED blinking. You can change the blinking frequency by pressing the switch label **USER BTN** in **Figure 3.7** (the blue one).

We are going to replace the on-board firmware with the one made by us before in a while. Before we move on with this step, it is important to be sure that the ST-LINK debugger in our Nucleo board is equipped with the latest ST-LINK 2.1 firmware.

3.3.1 ST-LINK Firmware Upgrade

Warning

Read this paragraph carefully. Do not skip this step!

I bought several Nucleo boards, and I saw that all boards usually come with a quite old ST-LINK firmware. To use the Nucleo with latest STM32CubeIDE, the firmware must be updated at least to the V2J37.x version.

The upgrade procedure can be easily carried on with the STM32CubeIDE. Connect your Nucleo board using a USB cable and go to **Help->ST-LINK Upgrade**. The **ST-LINK Upgrade** program appears, as shown in **Figure 3.8**

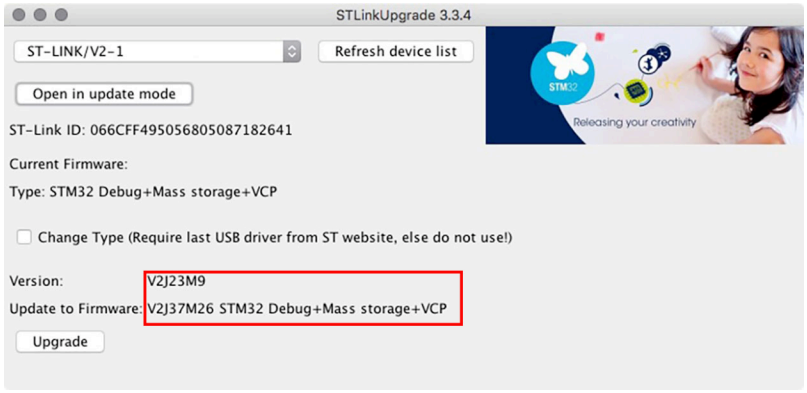


Figure 3.8: The ST-LINK Upgrade program

Click on **Refresh device list**: the connected board should be identified as **ST-LINK/V2-1**. Click on **Open in update mode**. ST-LINK Upgrade will show if your Nucleo firmware needs to be updated (pointing out a different version, as shown in **Figure 3.8**). If so, click on the **Upgrade** button and wait till the firmware is updated.

Error in upgrading the ST-LINK firmware

The above procedure may fail when you click on **Open in update mode** button. The STLinkUpgrade tool may show the error massage *Error connecting to device ST-LINK/V2-1 (error 0x1); check the USB connection and refresh device list* even if the board is properly connected to the PC. This usually happens due to an insufficient powering of the board. Try to use a self-powered USB HUB or use another USB port. This error is quite common on recent iMac when connecting the board to an Apple keyboard if use a keyboard with integrated USB port.

3.4 Flashing the Nucleo using STM32CubeProgrammer

We installed STM32CubeProgrammer in Chapter 2 and now we are going to use it. Launch the program and connect your Nucleo to the PC using the USB cable. Click the refresh button, circled in red in **Figure 3.9**.
Once STM32CubeProgrammer has identified the board, its serial number will appear in the **Serial number** box, as shown in **Figure 3.9**.

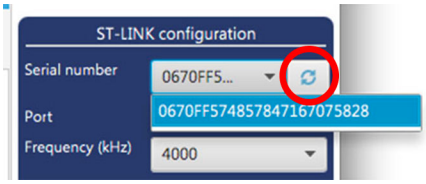


Figure 3.9: The ST-LINK interface serial number as shown by STM32CubeProgrammer tool

Read Carefully

If the label "*Old ST-LINK Firmware*" appears instead of the ST-LINK interface serial number, then you need to update the ST-LINK firmware to the latest version. Click on the **Firmware upgrade** button at the bottom of the **ST-LINK Configuration** pane and follow the same instructions reported in the previous paragraph.

Once the ST-LINK board has been identified, click the **Connect** button. After a while you will see the content of flash memory, as shown in **Figure 3.10**. Ok, let us finally upload the example firmware to the board.
Click on the **Erase & programming** icon (the second green icon on the left). Then, click on the **Browse** button in the **File programming**. Go in your Eclipse workspace directory (by default, the path is `%HOMEPATH%\STM32CubeIDE\workspace_1.X.0.` in Windows or `~/STM32CubeIDE7workspace_-1.X.0` in Linux and Mac OS, where `1.X.0` corresponds to the exact release of your STM32CubeIDE). Then move inside the `hello_nucleo\Debug` sub-folder and choose the file named `hello_nucleo.elf`. Check the **Verify programming** and **Run after programming** flags and click on **Start Programming** button to start flashing. At the end of flashing procedure your Nucleo green LED will start blinking.

Congratulations: welcome to the STM32 world ;-)

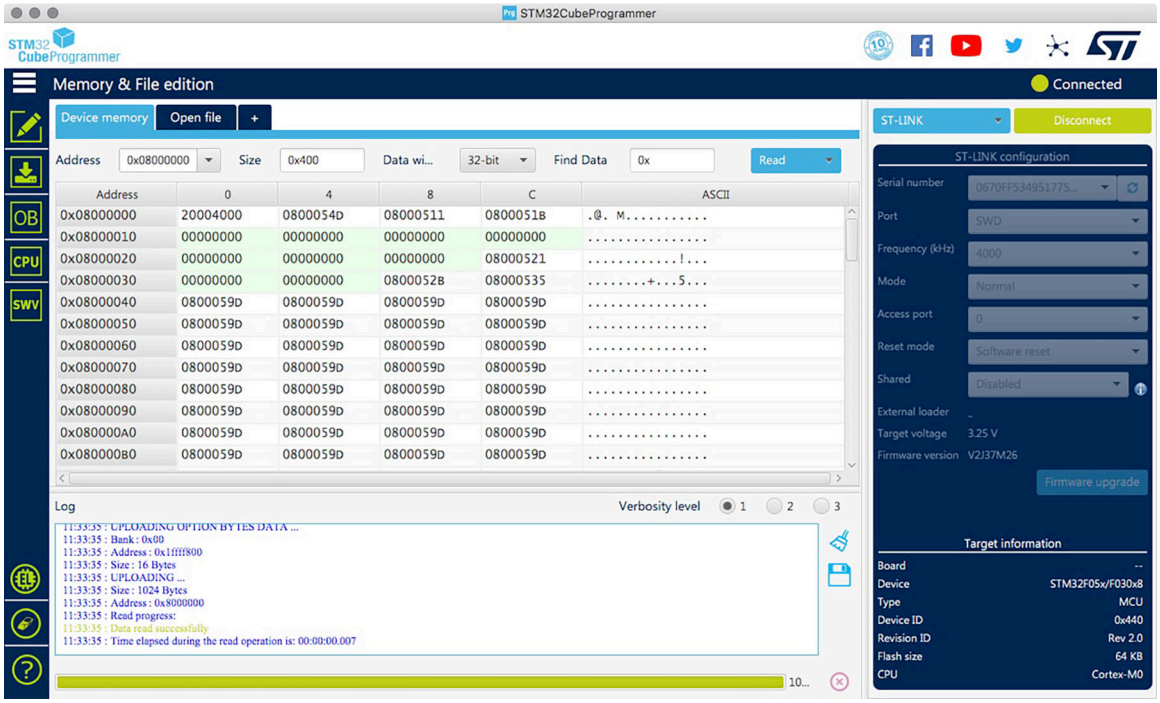
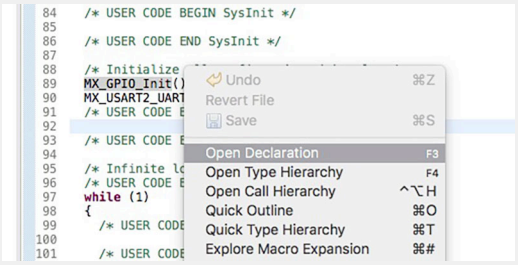


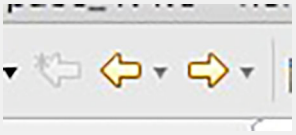
Figure 3.10: The STM32CubeProgrammer interface when connected to the Nucleo board

Eclipse intermezzo

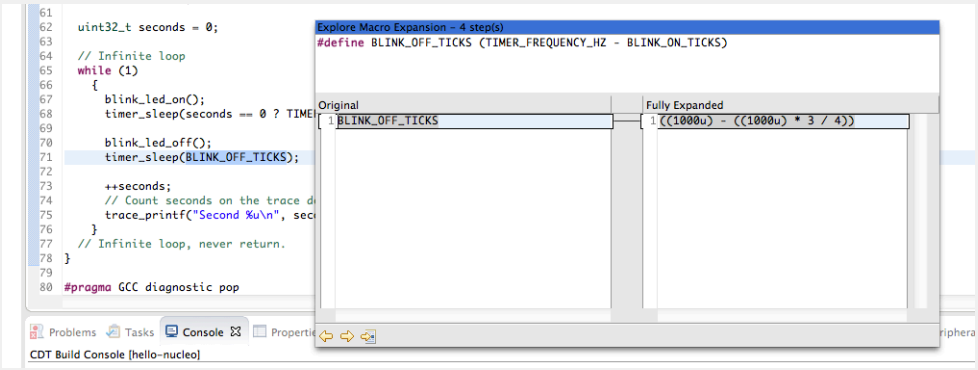
Eclipse allows us to easily navigate inside the source code, without jumping between source files manually looking for where a function is defined. For example, let us suppose that we want to see how the function `MX_GPIO_Init()` is coded. To go to its definition, highlight the function call, click with the right mouse button and select **Open declaration** entry, as shown in the following image.



Alternatively, you can hold the `Ctrl` key down (`CMD⌘` in MacOS) while clicking on the given symbol. Moreover, it is possible to navigate through the opened source files during symbol navigation by using the two dedicated buttons on the Eclipse Tool-bar, as shown below.



Another interesting Eclipse feature is the ability to expand complex macros. For example, click with right mouse button on a macro and choose the entry **Explore macro expansion**. The following contextual window will appear.



Sometimes, it happens that Eclipse makes a mess of its index files, and it is impossible to navigate inside the source code. To address this issue, you can force Eclipse to rebuild its index going to **Project->C/C++ Index->Rebuild** menu.

4. STM32CubeMX Tool

The times when the configuration of an 8-bit microcontroller peripheral could be performed with a few assembly instructions are far away. Although there is a well-established group of people that still develops embedded software in pure assembly code[1], time is the most expensive thing during project development nowadays, and it is important to receive as much help as possible for a quite complex hardware platform like the STM32. Moreover, in modern 32-bit MCUs, especially those with a very high number of I/Os, to drive even a simple GPIO could require to you digging inside tens of pages of a one thousand pages datasheet. Believe me or not, it could be really frustrating to simply initialize an advanced peripheral like `DCMI` or `ETH` interface in an STM32H7 without dedicated support by the ST HAL. Finally, to know all the possible configuration alternatives of a GPIO requires that you have a complete overview of all supported peripherals by the very specific STM32 part number, with its specific pinout and configuration. Lucky for us, ST provides a powerful and convenient tool that avoid us to simply ignore all the specific implementation details underling a peripheral configuration: STM32CubeMX.

STM32CubeMX[2] is the Swiss army knife of every STM32 developer, and it is a fundamental tool especially if you are new to the STM32 platform. It is a quite complex piece of software distributed freely by ST, and it is available both as a stand-alone tool downloadable from the ST website and as integrated component inside the STM32CubeIDE.

In this chapter we will see how CubeMX works, and how to generate working projects from scratch using the code generated by it. This will allow us to create better code and ready to be integrated with the rest of STM32Cube HAL. However, this chapter is not a substitute for the official ST documentation for CubeMX tool[3], a document made of more than 350 pages that explains in depth all its functionalities.

4.1 Introduction to CubeMX Tool

CubeMX is the tool used to configure the microcontroller chosen for our project. It is used both to choose the right hardware connections and to generate the code necessary to configure the ST HAL.

CubeMX is an *MCU-centric* application. This means that all activities performed by the tool are based on:

- The family of the STM32 MCU (F0, F1, and so on).

[1] Probably, one day someone will explain them that, except for rare and specific cases, a modern compiler can generate better assembly code from C than could be written directly in assembly by hand. However, we have to say that these habits are limited to ultra low-cost 8-bit MCUs like PIC12 and similar.

[2] STM32CubeMX name will be simplified in *CubeMX* in the rest of the book.

[3] https://bit.ly/3k8HeE2

- The type of package chosen for our device (LQFP48, BGA144, and so on).
- The hardware peripherals we need in our project (USART, SPI, etc.).
  - How selected peripherals are mapped to microcontroller pins.
- MCU general configurations (like clock, power management, NVIC controller, and so on)

In addition to features related to the hardware, CubeMX is also able to deal with the following software aspects:

- Management of the ST HAL for the given MCU family (CubeF0, CubeF1, and so on).
- Configuration of additional software libraries, named *Middlewares* libraries, we may need in our project to drive specific peripherals or to manage complex software stacks (FatFs, LwIP, FreeRTOS, etc.).
- The development environment we will use to build the firmware (IAR EWARM, Keil MDK-ARM, STM32CubeIDE)[4].

A project generation in CubeMX consists essentially of a two-phases activity. The first step consists in selecting the right STM32 MCU or development board by using the **Target Selection** wizard. The second step consists in configuring the MCU and any needed Middleware library according to your project needs. Let us study these two phases in depth.

4.1.1 Target Selection Wizard

We already used CubeMX in Chapter 3[5] to generate the *hello-nucleo*, our first STM32 project. We saw that the project generation starts with the **Target selection** view. The view is a tab-based window, with four main tabs (see **Figure 4.1**): *MCU/MPU Selector*, *Board Selector*, *Example Selector* and *Cross Selector*. Let us analyze them more in depth.

[4]The standalone version of CubeMX can generate project code and configuration not only for the official STM32CubeIDE, but also for other commercial IDEs. However, in this book we will focus exclusively on the aspects related to the usage of CubeMX inside the STM32CubeIDE.

[5]ch3-hello-nucleo-project

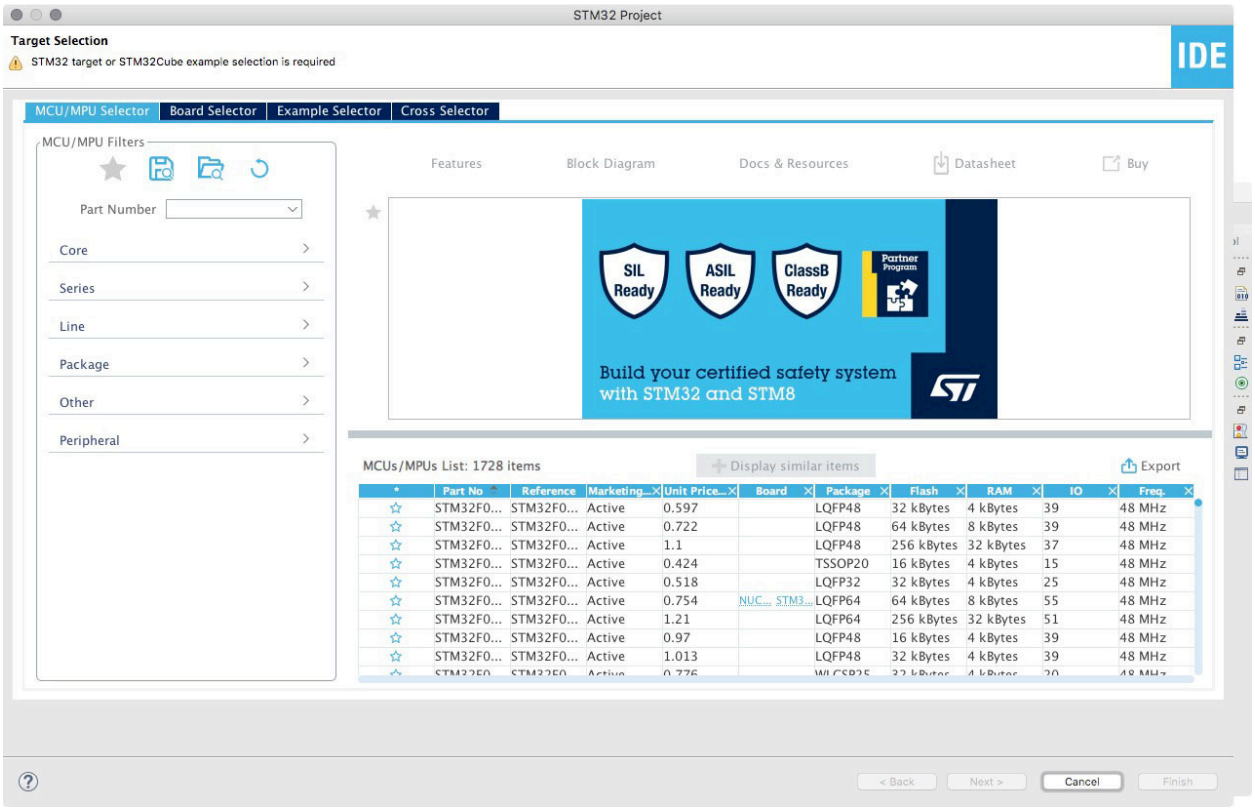


Figure 4.1: CubeMX *MCU/MPU Selector*

4.1.1.1 MCU/MPU Selector

The first tab allows to choose a microcontroller from the whole STM32 portfolio. Several filters help to identify the right microcontroller for the user application.

- **Core**: this filter allows to select just those MCUs belonging to selected Cortex-M cores (-M0, M4, etc.).
- **Series**: with this filter, we can show just those MCUs belonging to selected STM32 series (F0, F4, etc.).
- **Line**: this filter allows to further select the MCUs belonging to a sub-family (*Value line*, etc.).
- **Package**: this filter allows to select all MCUs having the desired physical package (LQFP, WLCSP, etc.).
- **Other** this section offers several filters to allow limiting the MCUs according to budgetary price, number of I/Os, dimensions of FLASH, SRAM and EEPROM memories.
- **Peripheral**: this section allows to select just those MCUs having a wanted integrated peripheral.

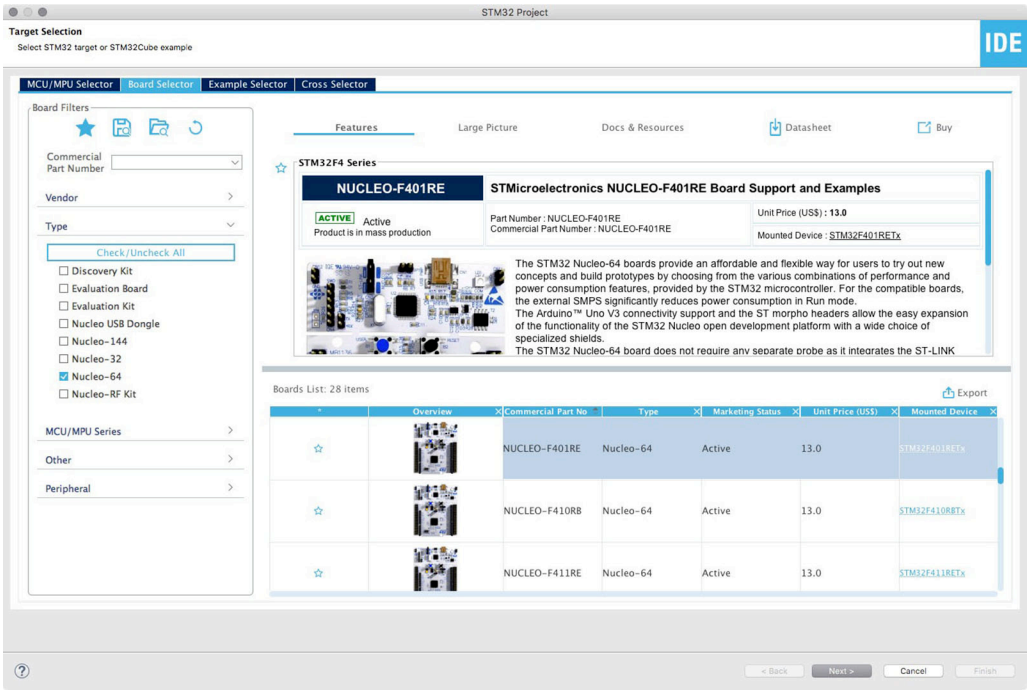


Figure 4.2: CubeMX *Board Selector*

4.1.1.2 Board Selector

The *Board Selector* tab allows to filter among all the official ST development boards (see **Figure 4.2**). Several filters help to identify the right development board.

- **Type**: this filter allows to restrict the selection to just those boards belonging to a given family (Nucleo-64, Discovery, Evaluation Board, etc.).
- **MCU/MPU Series**: with this filter, we can show just those board with a target MCU belonging to selected STM32 series (F0, F4, etc.).
- **Other** this section offers two filters to allow limiting the MCUs according to budgetary price or oscillator frequency (not that useful filter, according to this author).
- **Peripheral**: this section allows to select the development board according to wanted integrated peripherals.

4.1.1.3 Example Selector

During the years, ST developed thousands of examples to show how to use individual peripherals or extensions Middleware for the STM32 lineup. The *Example Selector* tab allows to filter among more than 5.000 examples (see **Figure 4.3**).
Several filters help to identify the right example

- **Name**: this filter allows to restrict examples list to those with a given project name (every example is usually available to several MCUs and/or development boards).

- **Keyword**: this filter restricts the examples list according to a given search keyword.
- **Board**: this filter selects all the examples for a given target development board.
- **MCU/MPU**: this filter selects all the examples for a given target MCU.
- **Project Type**: this filter allows to select among *Application*, *Demonstration* and *Example* project; well, in my humble opinion they are just splitting hairs with that field.
- **Based on driver**: with this filter it is possible to select those examples made with the CubeHAL, the CubeHAL-LL or a mix of the twos; more about this later.
- **Middleware**: this filter allows to select all those examples showing the usage of a given Middleware library.
- **MCU/MPU Library**: this filter may be a little misleading, since its usage is to select all those examples showing how to program a given peripheral.
- **Board Support Package Library**: a lot of STM32 development kits integrates additional peripherals like, for example, LCD displays, DCMI cameras, MEMS sensors and so on. To test those additional peripherals, ST provides useful *Board Support Package* (shortened **BSP**) libraries. These free libraries come in handy when you have to drive a similar peripheral on your custom board. This filter allows to select among the examples using a given BSP library.

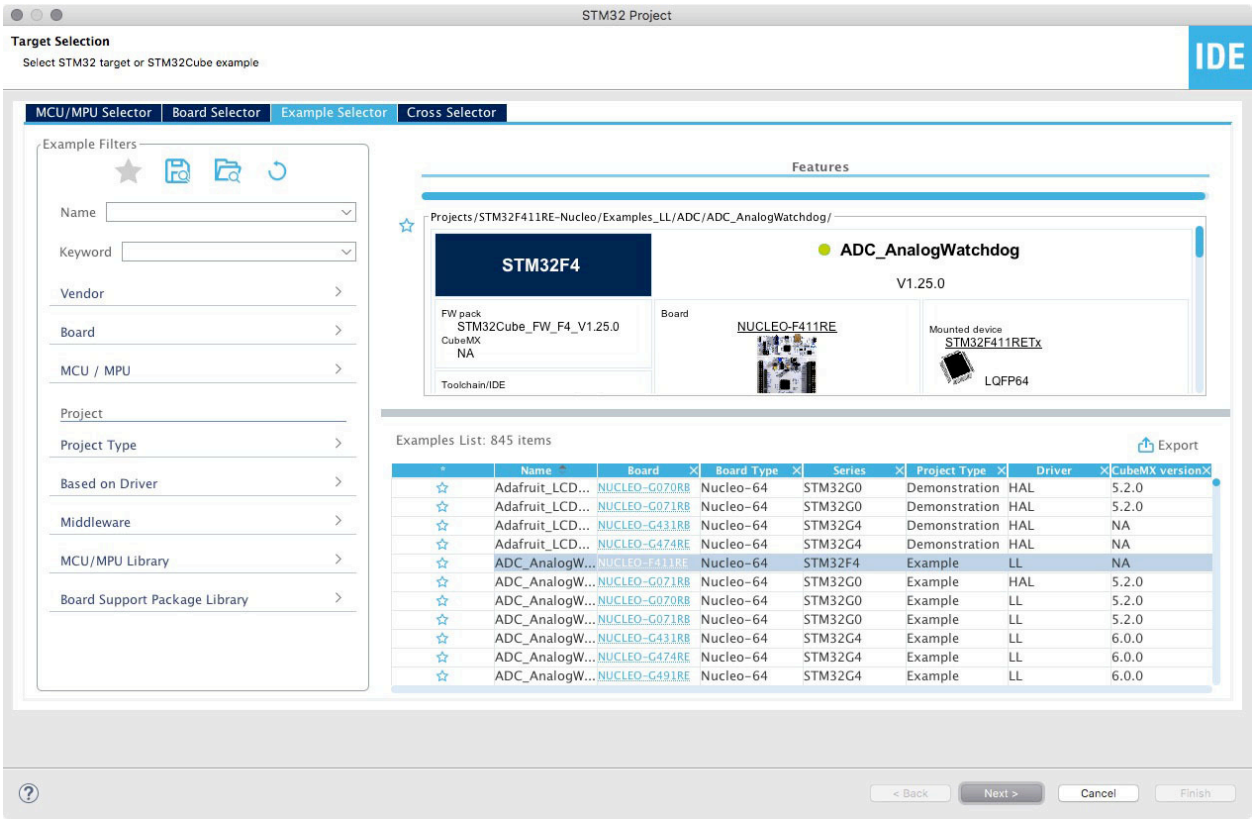


Figure 4.3: CubeMX *Example Selector*

4.1.1.4 Cross Selector

If you are used to work with an MCU from another supplier (Microchip, Renesas, etc.), and if you are considering porting an electronic board to an STM32 microcontroller, the *Cross Selector* section may help you in identifying the right alternative. However, in my opinion this tool works well in finding an alternative to a given STM32 MCU. And this could come in handy especially if an STM32 MCU is not available on the market (an event that is anything but rare these times). The *Cross Selector* tool (shown in **Figure 4.4**) provides a *match percentage*, which gives you an idea of how different is the suggested alternative from your current MCU. However, always keep the datasheet at your hands and check carefully even non-primary specifications like, for example, physical behavior of GPIOs (voltage tolerance, slew rate, etc.).

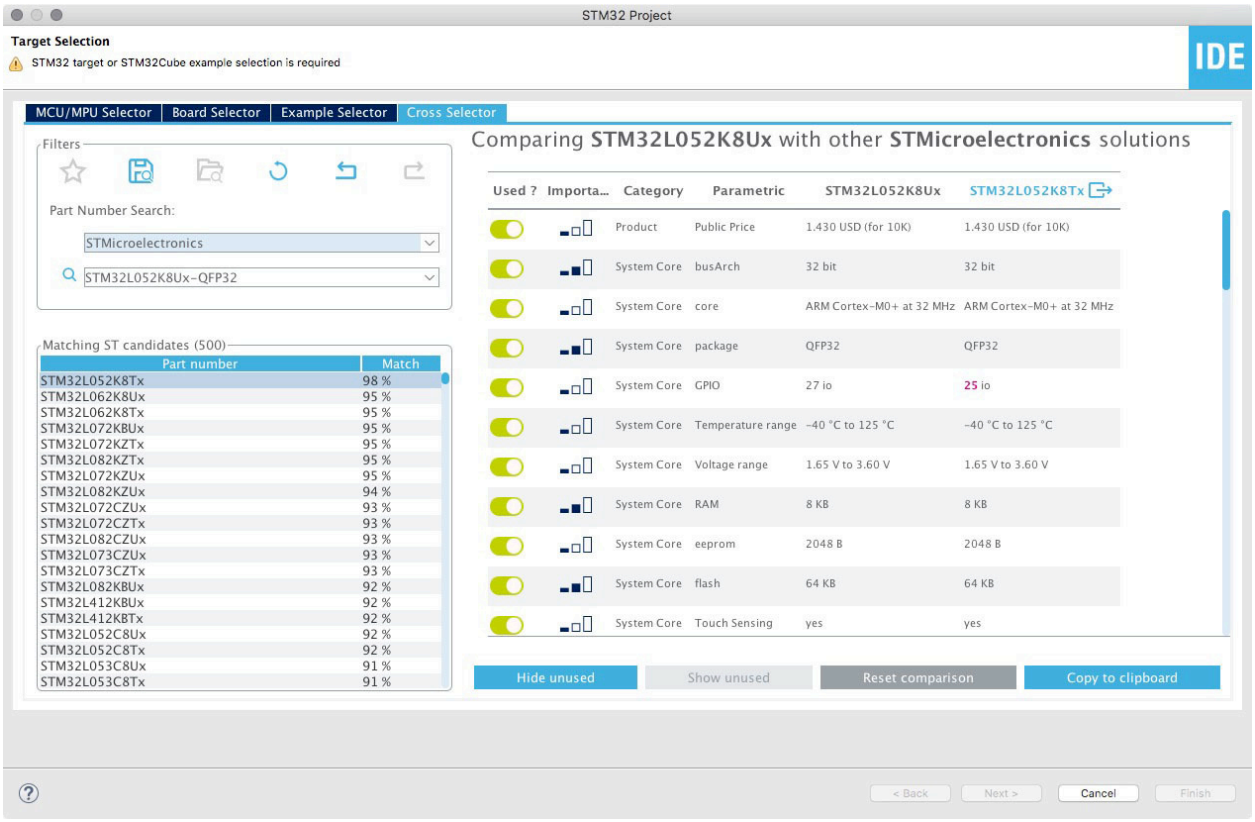


Figure 4.4: CubeMX *Cross Selector*

4.1.2 MCU and Middleware Configuration

Once the project has been generated, the STM32CubeIDE automatically opens a file in the project folder named `<project-name>.ioc`. This file is the CubeMX main project file, containing all the configurations performed in CubeMX. Starting from them, CubeMX will generate the corresponding project structure, with all source files and libraries needed to use the selected peripherals and Middleware components.

When the `.ioc` file is opened, CubeMX shows the *Device Configuration Tool*, as shown in **Figure 4.5**.

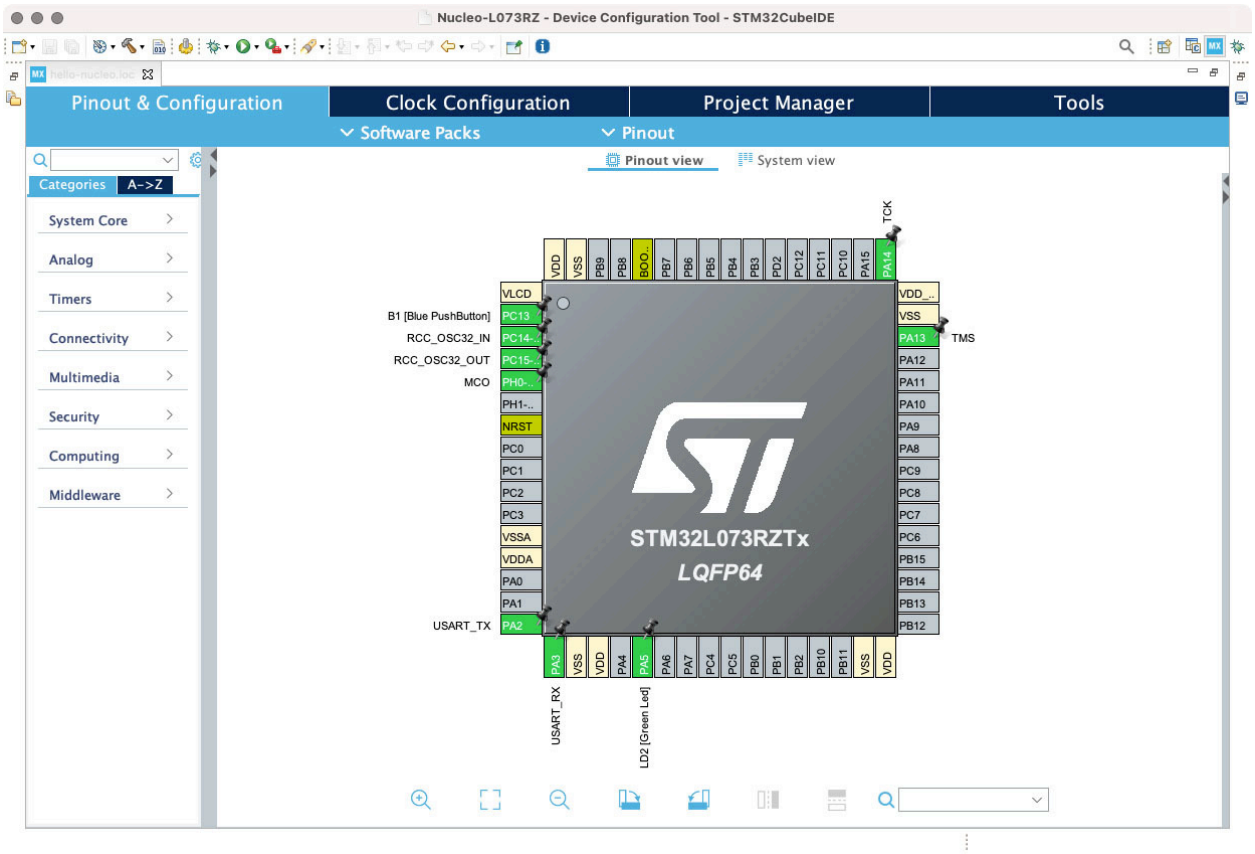


Figure 4.5: CubeMX *Device Configuration Tool* view

In the top part of the CubeMX view you can see a contextual menu, in light and dark blue. The menu is divided in four main tabs, each one with its dedicated view. Let us briefly introduce them.

4.1.2.1 Pinout View & Configuration

The *Pinout & Configuration* view is the first one, and it is in turn divided in sub-parts.
The right side contains the MCU representation with the selected peripherals and GPIOs, and it is called by ST *Pinout view*. It allows to easily navigate inside the MCU configuration, and it is a convenient way to configure the microcontroller.
Pins[6] colored in bright green are *enabled*. This means that CubeMX will generate the necessary code to configure that pin according to the bound peripherals. For example, considering the project configuration in **Figure 4.5**, for pin `PA5` CubeMX will generate the C code needed to setup it as generic output pin to drive LD2 LED[7]. Instead, for `PA2` pin CubeMX will generate the code to configure it as USART TX pin.

[6] In this context, *pin* and *signal* can be used as synonyms.
[7] In some Nucleo boards the LD2 LED is connected to different pins. For example, in the Nucleo-F302 LD2 is wired to PB13 pin. Consult the manual for your very specific board before configuring it.

A pin is colored in orange when **the corresponding peripheral** is not enabled. For example, in **Figure 4.6** PA2[8] and PA3 pins are enabled and CubeMX will generate corresponding C code to initialize them, but the associated peripheral (USART2) is not enabled and no USART related code to setup the peripheral will be automatically generated.
Light-yellow pins are power source pins, and their configuration cannot be changed.
BOOT and RESET pins are colored in khaki, and their configuration cannot be changed.

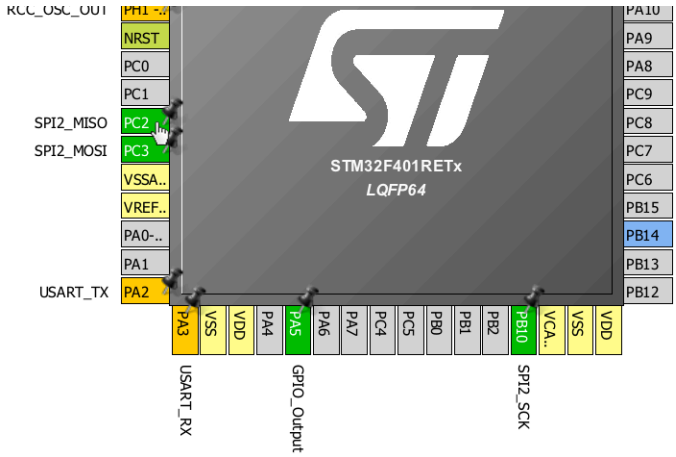


Figure 4.6: Alternate mapping of peripherals

A contextual tool-tip is showed when moving the mouse pointer over the MCU pins (see **Figure 4.7**). For example, contextual tool-tip for pin PB3 says to us that the signal is mapped to *Serial Wire Debug* (SWD) interface and it acts as *Serial Wire Output* (SWO) pin. Moreover, the pin number (55 in this case) is also shown.

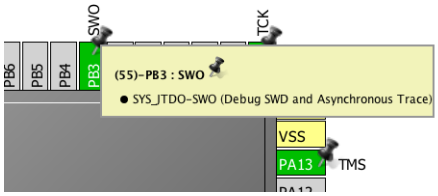


Figure 4.7: Contextual tool-tips help understanding signal usage

STM32 MCUs with high pin count allow mapping a peripheral to different pins. For example, in an STM32F401xE MCU, SPI2 MOSI signal can be mapped to pins PC2 or PB14. CubeMX makes it easy to see the allowed alternatives with a Ctrl+click on pin. If an alternate pin exists, it is shown in blue (the alternative is shown only if the pin is not in reset state - that is, it is enabled). For example, in **Figure 4.6** we can see that, if we do a Ctrl+click on PC2 pin, the PB14 signal is highlighted in blue[9]. This comes really in handy during the layout of a board. If it is impossible on not convenient to route a signal to that pin, or if that pin is needed for some other functionality, the usage of an alternate

[8] The pin configurations shown in this section are referred to the STM32L073RZ MCU.

[9] Take note that in recent CubeMX version the Ctrl+click behavior has been slightly changed. To show the alternative pin you need to keep pressed the mouse after a Ctrl+click until the alternative pin starts blinking. Not that intuitive the first time you do it.

pin may simplify the board.

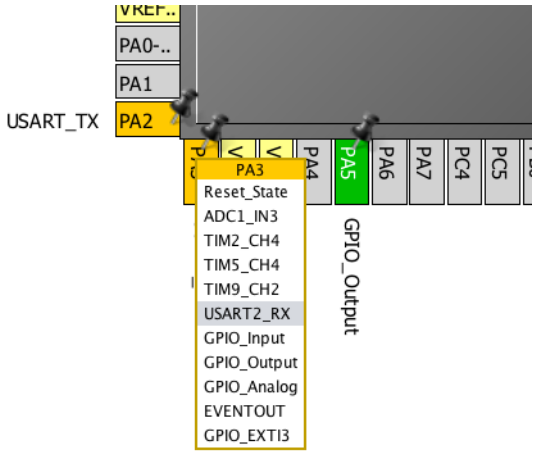


Figure 4.8: Alternate function of a pin

In the same way, most of MCU pins can have alternate functionalities. A contextual menu is shown when clicking on a pin. This allows us to select the function we are interested to enable for that signal.

Such flexibility leads to the generation of conflicts between signal functions. CubeMX tries to resolve these conflicts automatically, assigning the signal to another pin. Pinned signals are those pins whose functionality is locked to a specific pin, preventing CubeMX to choose an alternate pin. When a conflict prevents a peripheral to be used, the pin mode in *Chip View* is disabled, and the pin is colored in orange. To mark an I/O as pinned, right click on a pin and choose **Pin locking** entry.

CubeMX provides also a very convenient feature: the possibility to define custom labels for every individual MCU signal. By right-clicking on an enabled PIN you can select the entry **Enter User Label**. A contextual pop-up will appear, as shown in **Figure 4.9**. The label can have the form `LABEL [Comment]`. The `LABEL` part will be used to generate a corresponding macro inside the `main.h` file, while the `[Comment]` part is just a comment for the developer shown in the *Pinout View*.

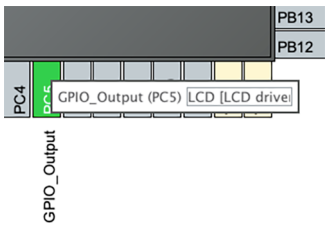


Figure 4.9: How to add custom label to MCU I/Os

As alternative view to the *Pinout view*, the *System view* gives an overview of all components configurable in software: GPIOs, peripherals, DMA, NVIC, Middleware and additional software components. Clickable buttons allow opening the configuration options for the given component (*Mode* and *Configuration panels*). The button icon color reflects the status of the configuration status.

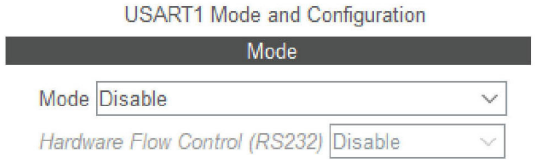
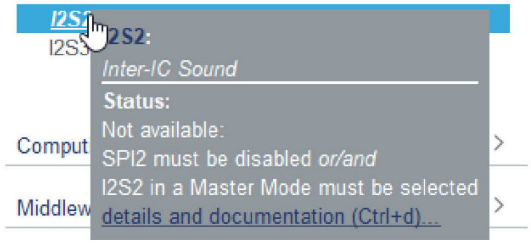
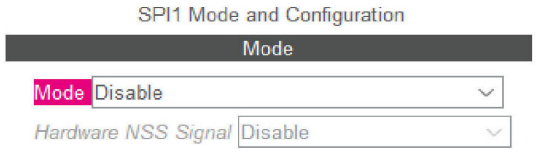
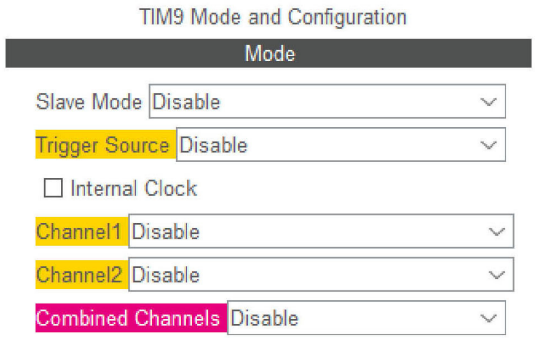
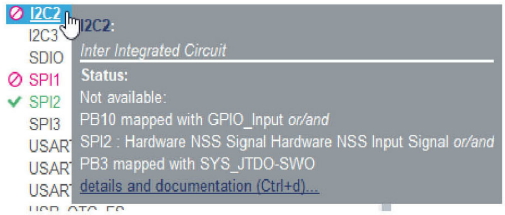
Case	Display	Component status	Corresponding Mode view / Tooltips
1	USART1 (Plain Black Text)	The peripheral is not configured (no mode is set) and all modes are available.	
2	*I2S2* (Gray Italic Text)	Peripheral is not available because some constraints are not solved. See tooltip.	
3	✔ CAN1 ⊗ CAN2	The peripheral is configured (at least one mode is set) and all other modes are available. The green check mark indicates that all parameters are properly configured, a cross indicates they are not.	
4	⚠ TIM9	The peripheral is not configured (no mode is set) and at least one of its modes is unavailable.	
5	⊘ I2C2	The peripheral is not configured (no mode is set) and no mode is available. Move the mouse over the peripheral name to display the tooltip describing the conflict.	

Table 4.1: CubeMX way to show components list in the Configuration Pane

In the left side of the *Pinout & Configuration* view we have the *Categories list* (also called *Components list* in the official ST documentation), that can be visualized both in alphabetical order and per categories. By default, it consists of the list of peripherals and Middleware components that the target MCU supports, and it is a convenient way to enable/disable and to configure the desired peripherals and software Middleware. Selecting an entry from that list opens two additional panels (*Mode* and *Configuration*) that allow the user to set its functional mode and configure the

initialization parameters that will be included in the generated code.

Table 4.1 shows the icons and color scheme used in the component list view and the corresponding color scheme in the Mode panel.

- **Case 1**: indicates that the peripheral is available and currently disabled, and all its possible modes can be used. For example, in case of a USART interface, all possible modes for this peripheral (*Asynchronous*, *Synchronous*, *IrDA*, etc.) are available.
- **Case 2**: shows that the peripheral is disabled due to a conflict with another peripheral. This means that both the peripherals use the same GPIOs, and it is not possible to use them simultaneously. Passing the mouse over it will show the other peripheral involved in conflict. For example, for an STM32F401RE MCU it is impossible to use I2S2 and SPI2 pins at the same time.
- **Case 3**: indicates that the peripheral is configured (at least one mode is set) and all other modes are available. The green check mark indicates that all parameters are properly configured, a fuchsia cross indicates they are not.
- **Case 4**: shows that the peripheral is not configured (no mode is set) and at least one of its modes is unavailable.
- **Case 5**: indicates that the peripheral is not configured (no mode is set) and no mode is available. By moving the mouse over the peripheral name is possible to display the tooltip describing the conflict.

4.1.2.2 Clock Configuration View

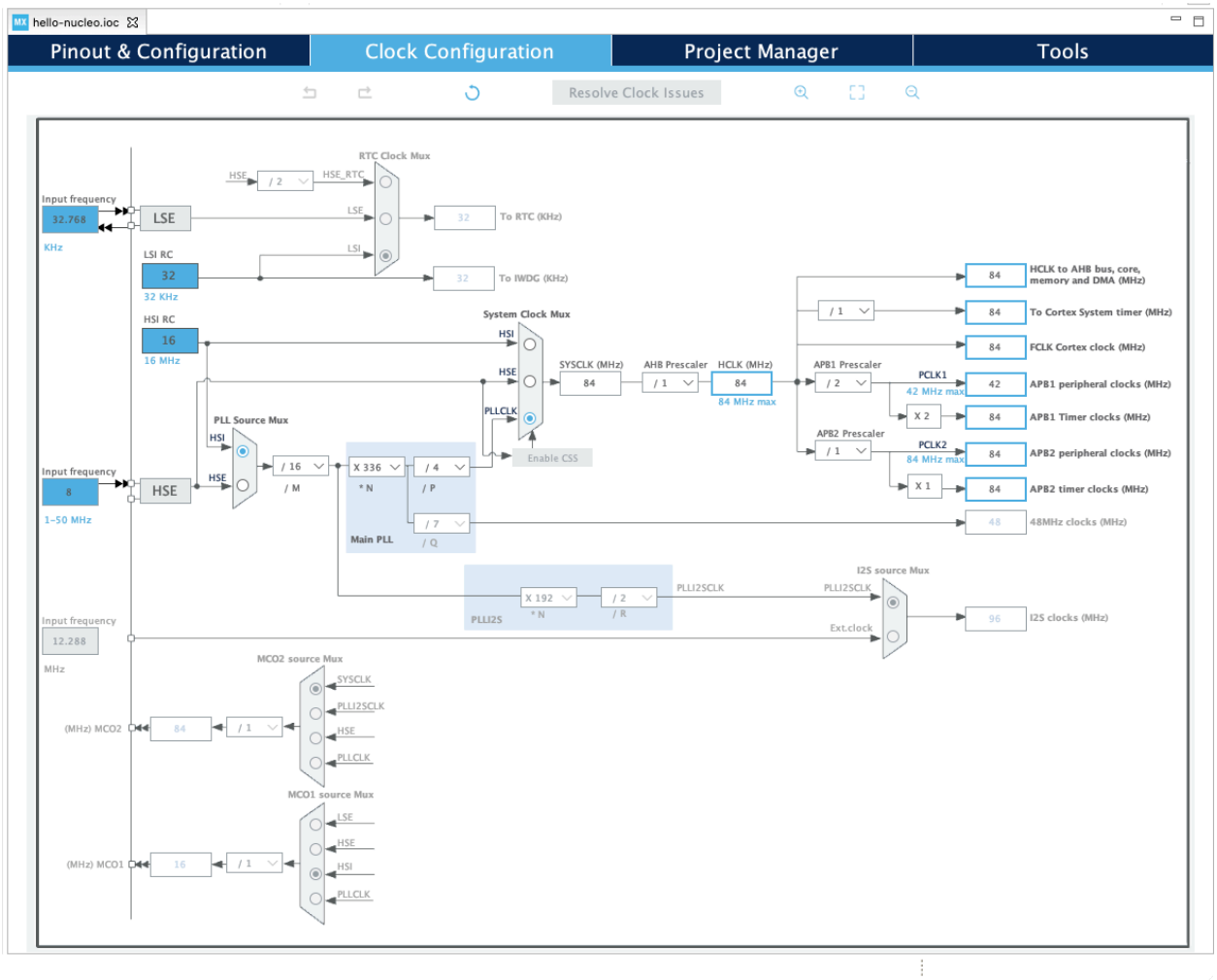


Figure 4.10: The CubeMX clock view

Clock Configuration view is the pane where all configurations related to clocks management take place. Here we can set both the main core and the peripherals clocks. All clock sources and PLLs configurations are presented in a graphical way (see **Figure 4.10**). The first times the user sees this view, he could be puzzled by the amount of configuration options. However, with a little bit of practice, this is the simplest way to deal with the STM32 clock configuration (which is quite complex if compared to 8-bit MCUs).

If your board design needs an external source for the High Speed clock (HSE), the Low Speed clock (LSE) or both, you have to first enable it in the *Pinout* view in the *System Core->RCC* section, as shown in **Figure 4.11**.

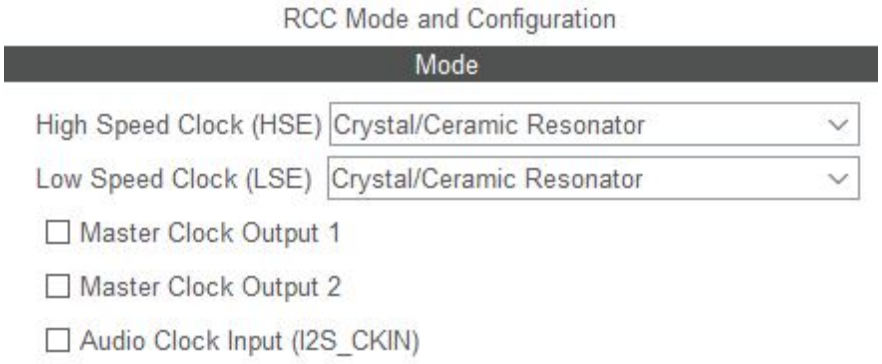


Figure 4.11: HSE and LSE enabling in CubeMX

Once this is accomplished, you will be able to change clock sources in clock view.

Clock tree configuration will be explored in Chapter 10. To avoid confusion in this phase, leave all parameters as automatically configured by CubeMX.

Overclocking

A common hacking practice is to overclock the MCU core, changing the PLL configuration so that it can run at a higher frequency. This author strongly discourages this practice, which not only could seriously damage the microcontroller, but it may result in abnormal behavior difficult to debug.

Do not change anything unless you are absolutely sure of what you are doing.

4.1.3 Project Manager

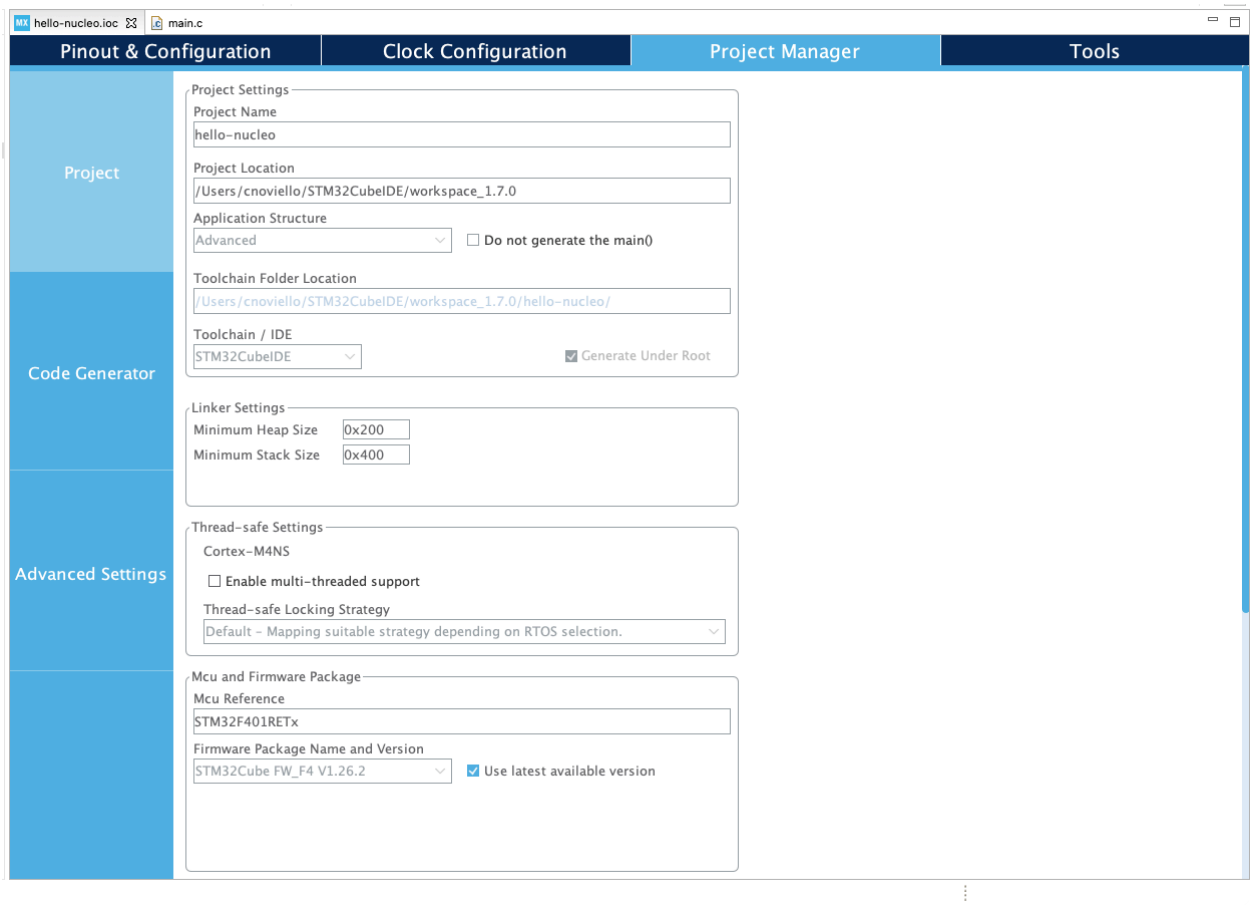


Figure 4.12: The Project Manager view

The *Project Manager* view contains project-wide configurations related to the workspace, the tool-chain, source code generation and type of HAL library used. The view is in turn divided in three sections:

- **Project**: this section contains general project setting such as the project name, project location on the filesystem, tool-chain, and CubeHAL libraries version.
- **Code Generator**: this section contains additional options related to the CubeMX code generation, such as how HAL `*.c/h` files are included inside the project, how template file structure is kept when new changed are applied to the project settings, and so on.
- **Advanced Settings**: this section includes more advanced project options mostly related to the type of CubeHAL used to generate the initialization code for a given peripheral. It is possible to choose between the CubeHAL and the more optimized Cube-LL library. At the same time, it is possible to choose not to generate code for some peripherals or middleware components, so that the programmer can decide to add his own code.

What is the Cube Low-Layer API?

With the advent of the STCube initiative, ST completely redesigned the SDK for the STM32 series introducing the *Hardware Abstraction Layer* (HAL) library and throwing out of the windows the old *Standard Peripheral Library* (SPL), which was very popular in the ST community despite of the fact it lacked many features related to more recent and powerful STM32 MCUs. However, the HAL library has attracted a lot of criticism over the years, both because it suffered of too many bugs during the first years and - more important - because it does not represent a good example of well optimized code for the development of embedded applications.

The CubeHAL is really a not-performant library, but for one simple reason: it is designed to be abstract and to simplify the porting of user code between MCUs of the same series and MCUs of different STM32 series. This led to a library full of `if` and `then` and full of unnecessary code when working with a very specific microcontroller. But this is the price to pay when you want to streamline the development process and - more important - the adoption of a given complex microcontroller architecture like the STM32 portfolio. The HAL APIs are split into two categories: generic APIs, which provide common and generic functions for all the STM32 series, and extension APIs, which include specific and customized functions for a given line or part number. The HAL drivers include a complete set of ready-to-use APIs that simplify the user application implementation. The HAL drivers are feature-oriented instead of IP-oriented. For example, the timer APIs are split into several categories following the IP functions, such as basic timer, capture and pulse width modulation (PWM). The HAL driver layer implements run-time failure detection by checking the input values of all functions. Such dynamic checking enhances the firmware robustness.

In the recent years, ST answered to strong criticism of the library's performances by introducing the *Cube Low-Layer* (shortened **LL**) set of drivers. As the name suggest, the LL library is born to be very optimized, leaving to the programmer the responsibility to deal with very specific characteristics of the given STM32 series and the given P/N. The LL drivers offer hardware services based on the available features of the STM32 peripherals. These services reflect exactly the hardware capabilities and provide atomic operations that must be called by following the programming model described in the product line reference manual. As a result, the LL services are not based on standalone processes and do not require any additional memory resources to save their states, counter or data pointers. All operations are performed by changing the content of the associated peripheral registers. Unlike the HAL, LL APIs are not provided for peripherals for which optimized access is not a key feature, or for those requiring heavy software configuration and/or a complex upper-level stack (such as USB). LL-based code is essentially a sequence of C macros that will be expanded in a series of statements with a very limited usage of branches and unpredictable statements from the performance point-of-view.

This book will not cover topics related to the LL library. It would require a completely different approach to the text and a strong focus on just few STM32 P/N. This book aims to be generic and to provide an overview of the most relevant features to start designing powerful and complex electronic boards. If you need to control every single aspect of a given peripheral to reach the most optimized code, then the LL library is what you need. But, at the first instance, I suggest you start designing the firmware by using the CubeHAL and then moving to the next step, unless you are a very experienced firmware developer.

4.1.4 Tools View

The *Tools* view contains other relevant configuration panes, some of which are available on more advanced STM32 MCUs like the STM32MP1 series. Instead, for all STM32 microcontrollers it is available the *Power Consumption Calculator* (PCC), which is a feature of CubeMX that, given a microcontroller, a battery model and a user-defined power sequence, provides an estimation of the following parameters:

- Average power consumption.
- Battery life.
- Average DMIPS.

It is possible to add user-defined batteries through a dedicated interface.
For each step, the user can choose VBUS as possible power source instead of the battery. This will impact the battery life estimation. If power consumption measurements are available at different voltage levels, CubeMX will also propose a choice of voltage values.

PCC view will be analyzed in a following chapter.

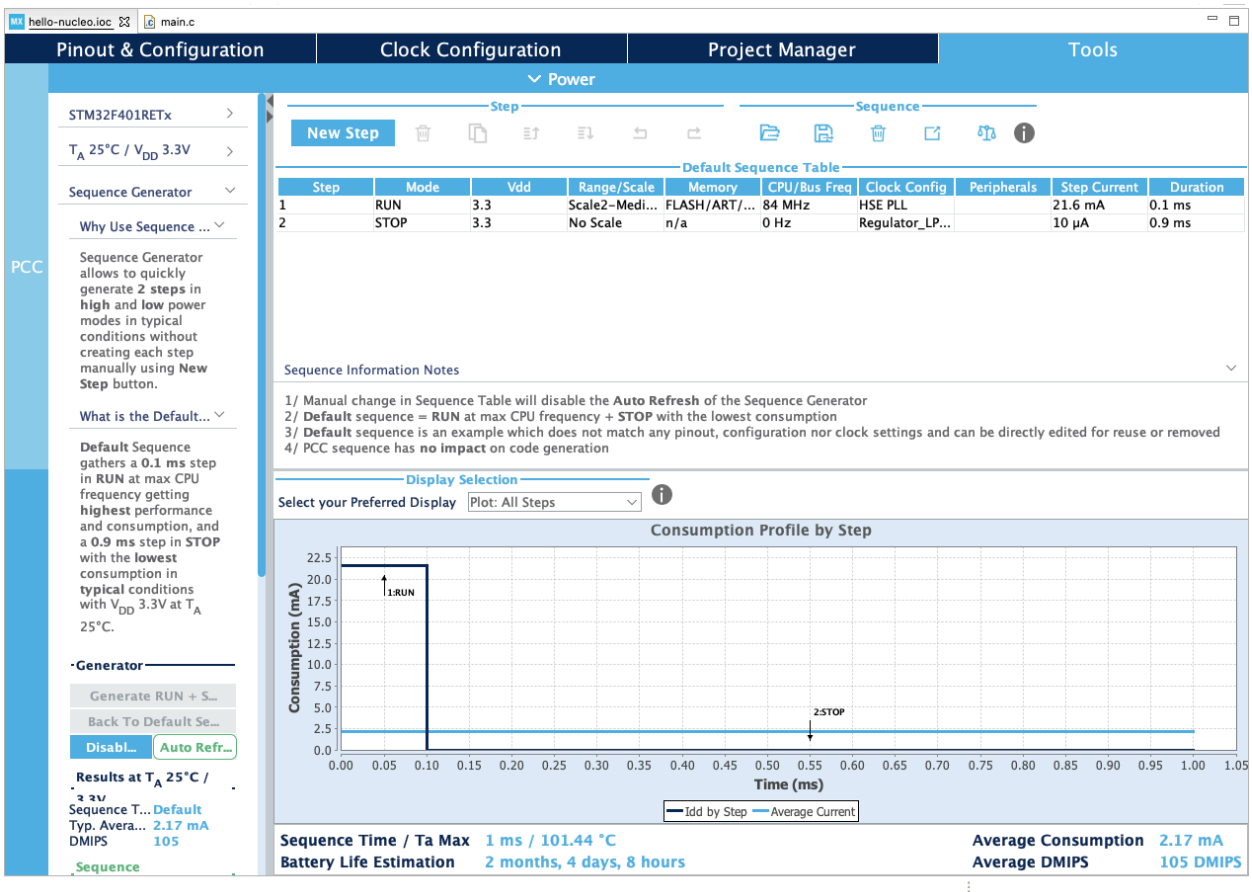


Figure 4.13: The CubeMX Tools view

4.2 Understanding Project Structure

Once the configuration of the MCU, of its peripherals and Middleware components is completed, we can use CubeMX to generate the C project skeleton for us. The code generation can be started in two ways:

- by saving the CubeMX project (the `.ioc` file) and letting CubeMX automatically perform the generation;
- by going to **Project->Generate Code** menu (while the `.ioc` file is the one selected in the main perspective) or by clicking on the corresponding icon on the Eclipse toolbar (see **Table 2.1**).

CubeMX will generate all the necessary files and will arrange them according to the structure shown in **Figure 4.14**.

At first you could be puzzled by all that complexity, especially if you are new to the embedded programming or such complex architectures[10]. But do not worry: in the next chapters we will deal with all details contained in every one of those auto-generated source files. However, to start programming without the fear of not understanding relevant topics, it is best if we take a quick look at the main project folders and the files contained in them. The **Figure 4.14** is a good reference in this quick walkthrough.

`Binaries`
: This folder contains the final binary file generated by the compiler when the Build process ends. It is a file in the *Executable and Linkable Format* (ELF), an object file typical in Linux-based Operating Systems. That folder is part of the Eclipse CDT's project organization, and its content is redundant: the same binary file is contained inside the `Debug` folder.

`Includes`
: This folder plays two roles. It is a graphical representation of all `Include` paths for the compiler (that is, all directories on the filesystem where GCC will look for C include files (`.h`)). But it is also a place to add additional references to other include files without dealing with compiler specific arguments and include paths. For more information regarding this topic, please refer to Eclipse CDT documentation.

`Core`
: This folder contains all application specific files. The files in this folder and its sub-folder are specific of the project settings and MCU configuration in CubeMX. `Core` folder should contain all other files you need to develop your application, but Eclipse is sufficiently smart allowing you to rearrange those files in different folders at your needs. But there is a very high price to pay for this: if you change the `Core` folder structure you will not be able to update source code if you introduce any modification to the CubeMX project. For this reason, my suggest is to leave it as-is at least in the early stages of a project.
Some files in the `Core` folder play a very special role in the project. Let us analyze them.

[10] Well... It is never a good idea to start learning embedding programming with an STM32H7 ;-P.

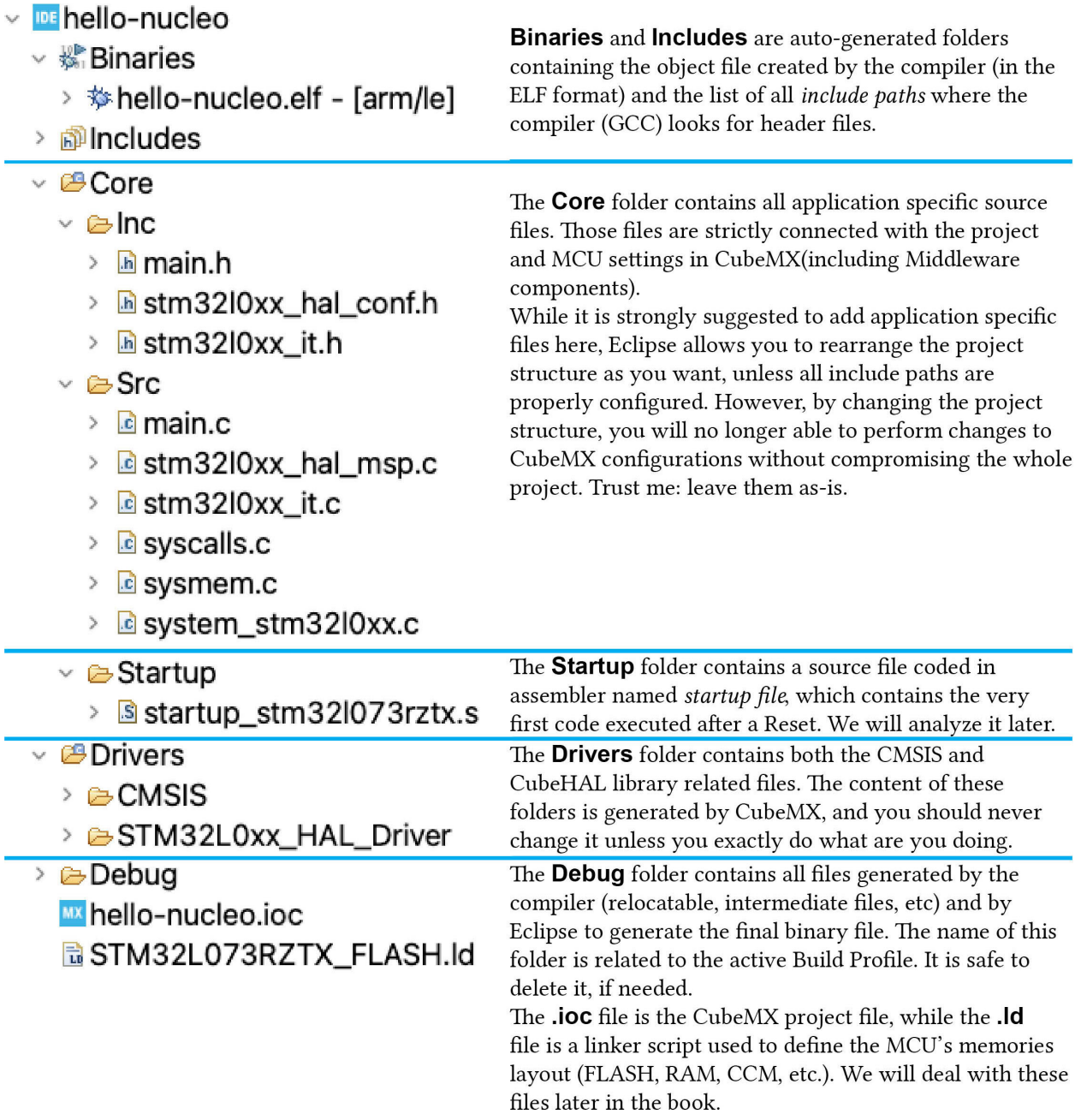


Figure 4.14: The typical structure of CubeMX project

`Core/Inc/main.h`
: This file is the companion header file for the `main.c` one. It contains, among the other, the macro declarations for all labels associated to individual peripherals using CubeMX.

`Core/Inc/stm32XXxx_hal_conf.h`
: This is the file where the HAL configurations are translated into C code, using several macro definitions. These macros are used to "instruct" the HAL about enabled MCU functionalities. You will find a lot of commented macros, like the ones shown below.

Filename: `Core/Inc/stm32XXxx_hal_conf.h`

```
55 #define HAL_UART_MODULE_ENABLED
56 /**define HAL_USART_MODULE_ENABLED   */
57 /**define HAL_IRDA_MODULE_ENABLED   */
58 /**define HAL_SMARTCARD_MODULE_ENABLED   */
59 /**define HAL_SMBUS_MODULE_ENABLED   */
60 /**define HAL_WWDG_MODULE_ENABLED   */
61 /**define HAL_PCD_MODULE_ENABLED   */
62 #define HAL_GPIO_MODULE_ENABLED
63 #define HAL_EXTI_MODULE_ENABLED
64 #define HAL_DMA_MODULE_ENABLED
65 #define HAL_I2C_MODULE_ENABLED
66 #define HAL_RCC_MODULE_ENABLED
67 #define HAL_FLASH_MODULE_ENABLED
68 #define HAL_PWR_MODULE_ENABLED
69 #define HAL_CORTEX_MODULE_ENABLED
```

They are used to selectively include HAL modules at compile time. When you need a module, you can simply uncomment the corresponding macro, if the corresponding `.c/.h` files are already included in the project. We will have the opportunity to see all the other macros defined in this file throughout the rest of the book.

`Core/Inc/stm32XXxx_it.h` and `Core/Src/stm32XXxx_it.c`

These two files are another fundamental part of our project. It is where all the *Interrupt Service Routines* (ISR) generated by CubeMX are stored. Given the CubeMX configuration we have chosen, the file contains the definition of several function. Considering the the *hello-nucleo* project in Chapter 3[11], there is only one useful function: `void SysTick_Handler(void)`. This function is the ISR of the *SysTick* timer, that is the routine invoked when the SysTick timer reaches 0. But where is this ISR invoked?

Filename: `Core/Src/stm32XXxx_it.c`

```
123 /**
124   * @brief This function handles System tick timer.
125   */
126 void SysTick_Handler(void)
127 {
128   /* USER CODE BEGIN SysTick_IRQn 0 */
129
130   /* USER CODE END SysTick_IRQn 0 */
131   HAL_IncTick();
132   /* USER CODE BEGIN SysTick_IRQn 1 */
133
134   /* USER CODE END SysTick_IRQn 1 */
135 }
```

[11] ch3-hello-nucleo-project

The answer to this question gives us the opportunity to start dealing with one of the most interesting features of Cortex-M processors: the *Nested Vectored Interrupt Controller* (NVIC). Table 1.1 in Chapter 1 shows the Cortex-M exception types. If you remember, we have said that in Cortex-M CPU interrupts are a special type of exceptions. Cortex-M defines the `SysTick_Handler` to be the fifteenth exception in the NVIC vector array. But where is this array defined? Inside the `Core/Startup` folder there is a special file written in assembly, called *startup file*. By opening this file, we can see the minimal vector table for a Cortex processor, as shown below:

Filename: `Core/Startup/startup_stmXXxx.s`

```
116 /******************************************************************************
117 * The minimal vector table for a Cortex M4. Note that the proper constructs
118 * must be placed on this to ensure that it ends up at physical address
119 * 0x0000.0000.
120 ******************************************************************************/
121 .section  .isr_vector,"a",%progbits
122   .type  g_pfnVectors, %object
123   .size  g_pfnVectors, .-g_pfnVectors
124
125
126 g_pfnVectors:
127   .word  _estack
128   .word  Reset_Handler
129
130   .word  NMI_Handler
131   .word  HardFault_Handler
132   .word  MemManage_Handler
133   .word  BusFault_Handler
134   .word  UsageFault_Handler
135   .word  0
136   .word  0
137   .word  0
138   .word  0
139   .word  SVC_Handler
140   .word  DebugMon_Handler
141   .word  0
142   .word  PendSV_Handler
143   .word  SysTick_Handler
144
145   /* External Interrupts */
```

Line 145 is where the `SysTick_Handler()` is defined as the ISR for the SysTick timer.

Please, consider that *startup files* have minor modifications between the ST HALs. Line numbers reported here could differ a little bit from the startup file for your MCU. Moreover, the `MemManage Fault`, `Bus Fault`, `Usage Fault` and `Debug Monitor` exceptions are not available (and hence the corresponding vector entry is RESERVED - see the Table 1.1 in Chapter 1) in Cortex-M0/0+ based processors. However, the first fifteen exceptions in NVIC are always the same for all Cortex-M0/0+ based processors and all Cortex-M3/4/7 based MCUs.

`Core/Src/stm32XXxx_hal_msp.c`

This is another relevant file to analyze. First, it is important to clarify the meaning of "MSP". It stands for *MCU Support Package*, and it defines all the initialization functions used to configure the on-chip peripherals according to the user configuration (PIN allocation, enabling of clock, use of DMA and Interrupts). Let us explain this in depth with an example. A peripheral is essentially composed of two things: the peripherals itself (for example, the SPI2 interface) and the hardware pins associated with this peripheral.

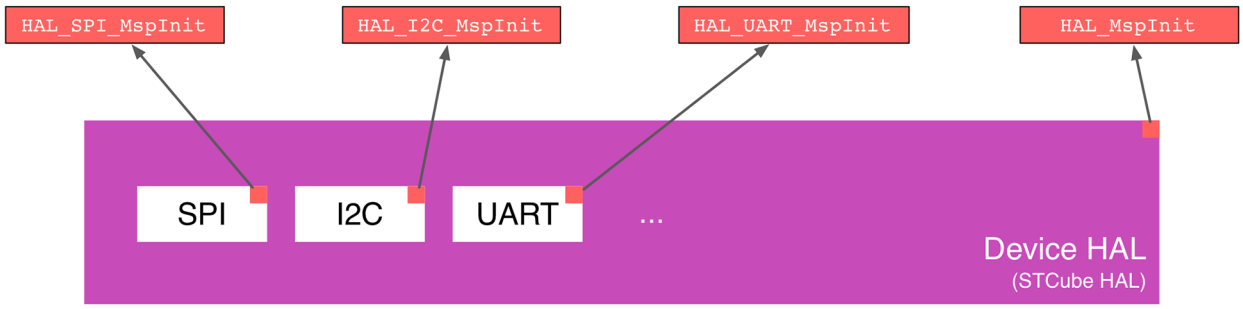


Figure 4.15: The relation between MSP files and the HAL

The ST HAL is designed so that the SPI module of the HAL is generic and abstracted from the specific I/O settings, which may differ due to the MCU package and the user-defined hardware configuration. So, ST developers have leaved to the user the responsibility to "fill" this piece of the HAL with the code necessary to configure the peripheral, using a sort of *callback* routines, and this code resides inside the **Core/Src/stm32XXxx_hal_msp.c** file (see **Figure 4.15**).

Let us open it. Here we can find the definition of the function `void HAL_UART_MspInit()`:

Filename: `Core/Src/stm32XXxx_hal_msp.c`

```
86  void HAL_UART_MspInit(UART_HandleTypeDef* huart)
87  {
88    GPIO_InitTypeDef GPIO_InitStruct = {0};
89    if(huart->Instance==USART2) {
90    /* USER CODE BEGIN USART2_MspInit 0 */
91
92    /* USER CODE END USART2_MspInit 0 */
93      /* Peripheral clock enable */
94      __HAL_RCC_USART2_CLK_ENABLE();
95
96      __HAL_RCC_GPIOA_CLK_ENABLE();
97      /**USART2 GPIO Configuration
98      PA2     ------> USART2_TX
99      PA3     ------> USART2_RX
100     */
101     GPIO_InitStruct.Pin = USART_TX_Pin|USART_RX_Pin;
102     GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
103     GPIO_InitStruct.Pull = GPIO_NOPULL;
104     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
105     GPIO_InitStruct.Alternate = GPIO_AF4_USART2;
106     HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
107   }
108 }
```

As you can see, `HAL_UART_MspInit()` is responsible of the actual configuration of the pin couples associated to the USART peripheral (that is, `PA2` and `PA3`). **Figure 4.16** shows the call hierarchy of the function`HAL_UART_MspInit()`: as you can see, it is called by the generic HAL function `HAL_UART_Init()`, which is in turn called in the `main.c` by the function `MX_USART2_UART_Init()`.

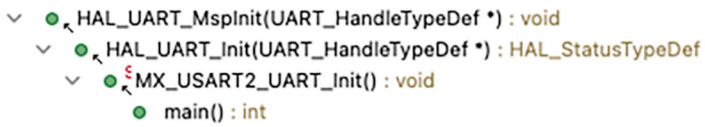


Figure 4.16: The Call Hierarchy of the function `HAL_UART_MspInit()`

The last file we have to analyze is `Core/Src/main.c`. It essentially contains four routines: `SystemClock_Config(void)`, `MX_GPIO_Init(void)`, `MX_USART2_UART_Init(void)` and `int main(void)`. The first function is used to initialize core and peripheral clocks. Its explanation is outside the scope of this chapter, but its code is not so much complicated to understand if you are not new to this matter. `MX_GPIO_Init(void)` is the function that configures the GPIOs connected to `LD2` pin and `B1` pin, the one connected to the blue switch on the Nucleo board. Chapter 6 will explain this matter in depth.

Finally, we have the `main(void)` function, as shown below.

Filename: `Core/Src/main.c`

```
66  int main(void) {
67    /* MCU Configuration--------------------------------------------------------*/
68
69    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
70    HAL_Init();
71
72    /* Configure the system clock */
73    SystemClock_Config();
74
75    /* Initialize all configured peripherals */
76    MX_GPIO_Init();
77    MX_USART2_UART_Init();
78
79    /* Infinite loop */
80    while (1)
81    {
82      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
83      HAL_Delay(500);
84    }
85  }
```

The code is self-explaining. First, the HAL is initialized by calling the function `HAL_Init()`. Then, clocks and GPIOs and USART2 are initialized. Finally, the application enters an infinite loop: that is the place where our code was placed.

Peripheral Initialization and Deinitialization

If you look at the file `Core/Src/stm32XXxx_hal_msp.c` you can find the definition of the function `HAL_UART_MspDeInit()`. Similarly to the `HAL_UART_MspInit()` function, the `HAL_UART_MspDeInit()` is invoked by the HAL function `HAL_UART_DeInit()`. It should be a good programming practice to always de-initialize a peripheral when no longer used. For the majority of peripherals, the deinitialization procedure consists in putting the associated GPIOs in high impedance state (to avoid any type of power leakage) and in stopping any peripheral's activities by shutting down its clock source. The CubeHAL is so designed to properly handle the peripheral deinitialization.

To keep things simple, you will not find proper deinitialization procedures for the majority of the examples shown in this text. But keep in mind that in embedded programming the more you can control the less risk of unwanted behaviors you will have.

`Core/Src/syscalls.c`

ARM GCC, and consequently the whole STM32 development environment, relies on a reduced version of the standard C run-time library for embedded systems (the *newlib* library): the so

called *newlib nano*. Every time we compile our firmware for an STM32 microcontroller, some pieces of the `newlib nano` are glued with the CubeHAL and our source code. `newlib nano` offers the possibility to use some traditional C library functions in embedded applications. For example, it is perfectly possible to use the classical `printf()`/`scanf()` I/O manipulator functions even if our board does not provide any screen or keyboard. However, some of the `newlib nano` functions relies on more "low-level" routines (called *system calls* or *syscalls*) that knows how to deal with the very specific hardware features. The file `Core/Src/syscalls.c` contains a dummy implementation for such routines: without them, the linking process would fail.
In next chapters will see how to customize some *syscalls* to implement more powerful and advanced debugging capabilities. We will analyze several alternative ways to establish a communication between the board and our host PC, by using the *Instrumentation Trace Macrocell* (ITM), the so-called *ARM Semihosting* or even a simple UART communication.

`Core/Src/sysmem.c`

Similarly to the `syscalls.c` file, this file contains a fully working implementation for the `_sbrk()` routine. This routine is a syscall in UNIX based environments to control the amount of memory allocated to the data segment of the process (that is, the *heap*). In Chapter 20 we will study the memory layout of a typical STM32 application in depth. This will allow us to understand the logic behind the `_sbrk()` routine and how to customize it at our needs.

`Drivers`

This folder contains both the CMSIS-CORE package and the CubeHAL library. Regarding the HAL, by default CubeMX will put in this folder and its subfolder just the file needed to use the peripherals enabled by using the *Device Configuration Tool*. Instead, the CMSIS-CORE package implements the basic run-time system for a Cortex-M device and gives the user access to the processor core and the device peripherals with convenient C macros. In detail it defines:

- **HAL** for Cortex-M processor registers, with standardized definitions for the *SysTick*, NVIC, System Control Block registers, MPU registers, FPU registers, and core access functions.
- **System exception names** to interface to system exceptions without having compatibility issues.
- **Methods to organize header files** that make it easy to learn new Cortex-M microcontroller products and improve software portability. This includes naming conventions for device-specific interrupts.
- **Methods for system initialization** to be used by each MCU vendor. For example, the standardized `SystemInit()` function is essential for configuring the clock system when device starts.
- **Intrinsic functions** used to generate CPU instructions that are not supported by standard C functions.
- **A global variable**, named `SystemCoreClock`, to easily determine the system clock frequency.

The most relevant subfolder in the CMSIS-CORE package is `CMSIS/Include`. It contains several `core_<cpu>.h` files (where `<cpu>` is replaced by `cm0`, `cm3`, etc). These files define the core peripherals and provide helper functions that access the core registers (*SysTick*, NVIC, ITM, DWT etc.). These files are generic for all Cortex-M based MCUs.

Debug

This folder contains all the intermediate files (relocatable files, map files, etc.) generated by Eclipse and the GCC compiler to obtain the final binary file in ELF or other binary format. The name `Debug` comes from the name of the active *Build Configuration*. *Build configurations* is a feature that all modern IDEs support. It allows having several project configurations inside the same project. Every Eclipse project has at least two build configurations: *Debug* and *Release*. The former is used to generate a binary file suitable to be debugged. The latter is used to generate optimized firmware for production.
It is safe to delete this folder entirely, if needed.

`STM32XXxx_FLASH.ld` and `STM32XXxx_RAM.ld`

These files (the `_RAM.ld` may not be present in projects generated for some STM32 MCUs) are linker script describing the memory layout of our application. They define the amount of FLASH and RAM memory and - more important - they define how these memories are organized at run-time. In Chapter 20 we will study the memory layout of a typical STM32 application in depth. This will allow us to understand the content of these files and how to customize them at our needs.

4.3 Downloading Book Source Code Examples

All examples presented in this book are available for downloading from its GitHub repository: http://github.com/cnoviello/mastering-stm32-2nd[12].

The examples are divided for each Nucleo model, as you can see in **Figure 4.17**. You can clone the whole repository using `git` command:

```
$ git clone https://github.com/cnoviello/mastering-stm32-2nd.git
```

or you can download only the repository content as a .zip package following this link[13]. The repository is divided in nine subfolders, each one related to one of the Nucleo boards used to build examples in this book. Now you have to import all the Eclipse projects for your Nucleo into the Eclipse workspace.

Open Eclipse and switch to a new Workspace. Go to **File->Import**…. The Import dialog appears. Select the entry **General->Existing Project into Workspace** and click on the **Next** button. Now browse to the folder containing the example projects for your Nucleo by clicking on the **Browse** button. Once selected the main folder, a list of the contained projects appears. Check all the projects you are interested in and check the entries **Search for nested projects** and **Copy projects into workspace**, as shown in **Figure 4.18** and click the **Finish** button.

[12] http://github.com/cnoviello/mastering-stm32-2nd
[13] https://github.com/cnoviello/mastering-stm32-2nd/archive/refs/heads/main.zip

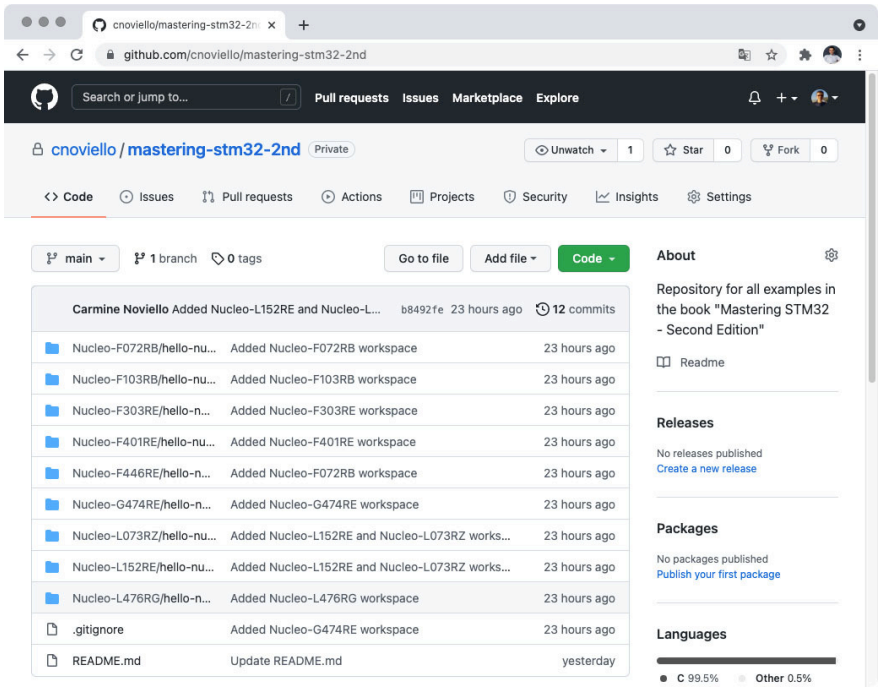


Figure 4.17: The content of the GitHub repository containing all the book examples

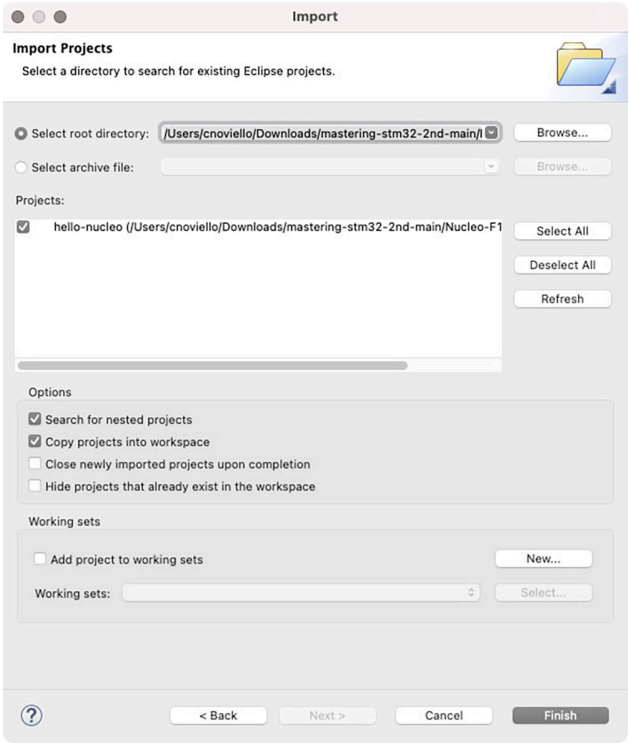


Figure 4.18: Eclipse project import wizard

Now you can see all imported projects inside the *Project Explorer* pane.

Every project is related to a given chapter and all examples shown in that chapter are available in the same project. To switch between different examples, select the corresponding *Build Configuration* as shown in **Figure 4.19**

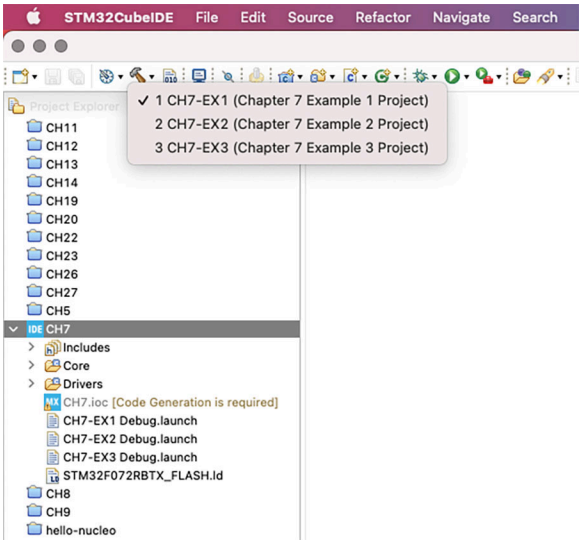


Figure 4.19: How to switch to a different project configuration to select other chapter's examples

4.4 Management of STM32Cube Packages

STM32CubeIDE allows to manage CubeHAL packages and additional Cube Extension Packs directly from the IDE. Going to **Help->Manage Embedded Software Packages** it is possible to launch the *Embedded Software Package Manager* as shown in **Figure 4.20**. This tool also allows to install packages manually, by specifying a local ZIP file or a remote URL.

Please, take note that all Cube packages ends in `C:\Users\{USERNAME}\STM32Cube\Repository` folder on Windows or `~/STM32Cube/Repository` directory in MacOS and Linux. Keep the content of this folder under control because it can eat a lot of HDD space.

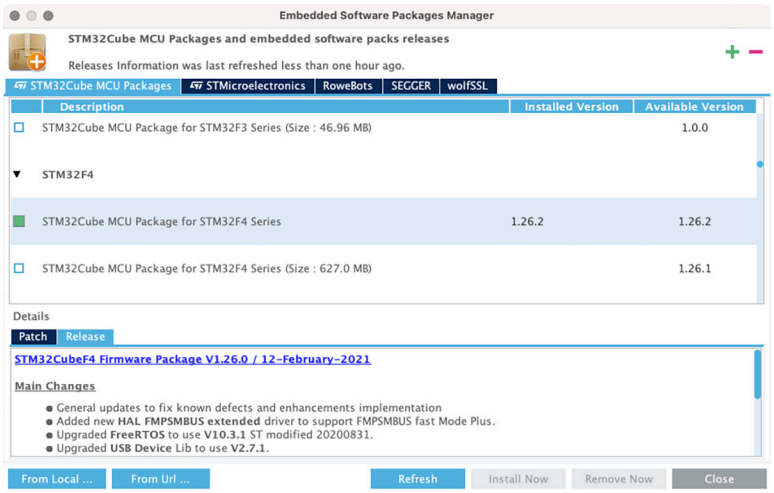


Figure 4.20: The Embedded Software Package Manager

5. Introduction to Debugging

Coding is all about debugging, said a friend of mine one day. And this is dramatically true. We can do all the best writing great code, but sooner or later we have to deal with software bugs (hardware bugs are another terrible beast to fight). And a good debugging of embedded software is all about to be a happy embedded developer.

In this chapter we will start working with the most basic debugging functionalities offered by the STM32CubeIDE. As we will see, STM32CubeIDE offers a powerful set of debugging tools that allow us to easily investigate bugs and unwanted behaviors. Moreover, ST did a great job in integrating debugging tools in Eclipse: using them is simple and natural, and there is no need to use external programs or additional hardware tools like happened in the past. Nowadays all you need is just a cheap ST-LINK debug probe and the STM32CubeIDE.

This chapter is a preliminary view of the debugging process, which would require a separate book even for simpler architectures like the STM32. Chapter 24 will give a close look at other debugging tools, and it will focus on Cortex-M exception mechanism, which is a distinctive feature of this platform.

5.1 What is Behind a Debug Session

Before we see how to start a debug session and how to perform the typical debug operations (adding breakpoints, step-by-step execution, step-into, etc.), it is better if we give a quick to look to the software and hardware tools involved. **Figure 5.1** tries to provide an overview of the debug setup behind the scenes.

In a GCC based development environment, the fundamental tool to perform debugging operations is the *GNU Debugger* (GDB). GDB is a command-line tool with an integrated shell and a quite large set of commands and options. GDB is designed with the same philosophy of GCC: it is abstract from the specific target architecture (x86, MIPS, ARM, etc.), from the specific programming language (C, C++, etc.), from the specific host Operating System (Windows, Linux, MacOS, etc.).

To enforce this portability among so different target architectures, GDB is designed with a strong separation between the frontend part (that is the real GDB's core, responsible of the binary file manipulation, interpretation of debug information inside the object file, etc.) and the backend part, which knows all the details related to the target hardware and software architecture. So, it is common to say that GDB has a client and a server part that communicate through a network connection by using a well-defined protocol. Cleary, this connection can be established between two separated machine or on the same machine as well.

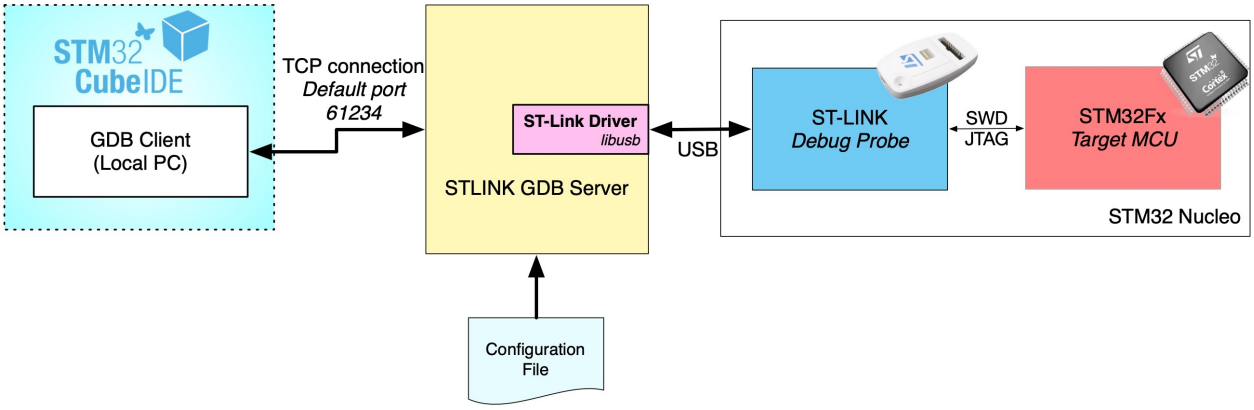


Figure 5.1: How OpenOCD interacts with a Nucleo board

For embedded architectures like the Cortex-M, it is common that the server part is not provided with the ARM-GCC distribution. This because in a debug session is always involved a dedicated debug adapter, a piece of hardware that translates (both from the physical and logical point-of-view) "high level" commands in JTAG or SWD signals and instructions. For all the Nucleo boards, this adapter is the integrated ST-LINK interface[1].

For this reason, ST provides a dedicated backend server for GDB, called *ST-LINK GDB Server*, which communicates with the ST-LINK adapter through the USB connection by using *libusb* or any API-compatible library able to allow user-space applications to interface USB devices. Thanks to a set of configuration files included in the STM32CubeIDE distributions, *ST-LINK GDB Server* knows how to deal with the specific target MCU (STM32F030, STM32F401, etc.), its specific *Debug Access Port* (DAB), it is specific FLASH memory[2], bus architecture, and so on.

When a debug session starts, the following main operations take place[3]:

1. STM32CubeIDE executes in background the *ST-LINK GDB Server* by passing several command line arguments that specify things like the path to the *STM32CubeProgrammer*, the TCP/IP port used to accept connection from the GDB client, the type of debug mode to use (SWD, JTAG), speed of debug port, etc[4]. If the *ST-LINK GDB Server* can communicate with the ST-LINK probe and the target board correctly, it starts waiting for command on the given TCP/IP port (by default, 61234 port).
2. STM32CubeIDE then executes the *GDB client*, using the above TCP/IP port to connect to the remote GDB server (that is, the *ST-LINK GDB Server*).
3. STM32CubeIDE then loads the binary file inside the target MCU's FLASH memory and starts the firmware execution.

[1]The Nucleo ST-LINK debugger is designed so that it can be used as standalone adapter to debug an external device (e.g., a board designed by you equipping an STM32 MCU). Consult the documentation of your Nucleo board to configure it accordingly.

[2]One common misunderstanding about the STM32 platform is that all STM32 devices have a common and standardized way to access to their internal FLASH. This is not true, since every STM32 family has specific capabilities regarding their peripherals, including the internal flash. This requires *ST-LINK GDB Server* to provide drivers to handle all STM32 devices.

[3]Take note that the following steps are a very simplified view of the actual operations carried out in a debug session. A lot of details are omitted, because a detailed description would require a deep knowledge of several Cortex-M details, several STM32 technicalities and a decent knowledge of the GDB framework.

[4]The complete set of command-line arguments is well documented in the related user manual available at this address: https://bit.ly/3EfV2aH

Finally, Eclipse-CDT contains all necessary logic to drive GDB in background while the user is free to use the GUI to perform typical debugging operations without knowing any GDB shell command.

5.2 Debugging With STM32CubeIDE

Eclipse provides a separated perspective dedicated to debugging. It is designed to offer the most of required tools during the debugging process, and it can be customized at need with additional plug-ins (more about this later).

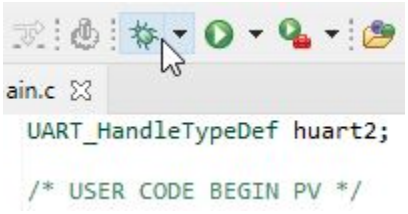


Figure 5.2: The *Debug* icon to start debugging in Eclipse

To start a new debug session, you can simply click on the **Debug** icon on the Eclipse toolbar, as shown in **Figure 5.2**. Eclipse will ask you if you want to switch to the *Debug Perspective*. Click on the **Yes** button (it is strongly suggested to flag the **Remember my decision** checkbox). Eclipse switches to the *Debug Perspective*, as shown in **Figure 5.3**.

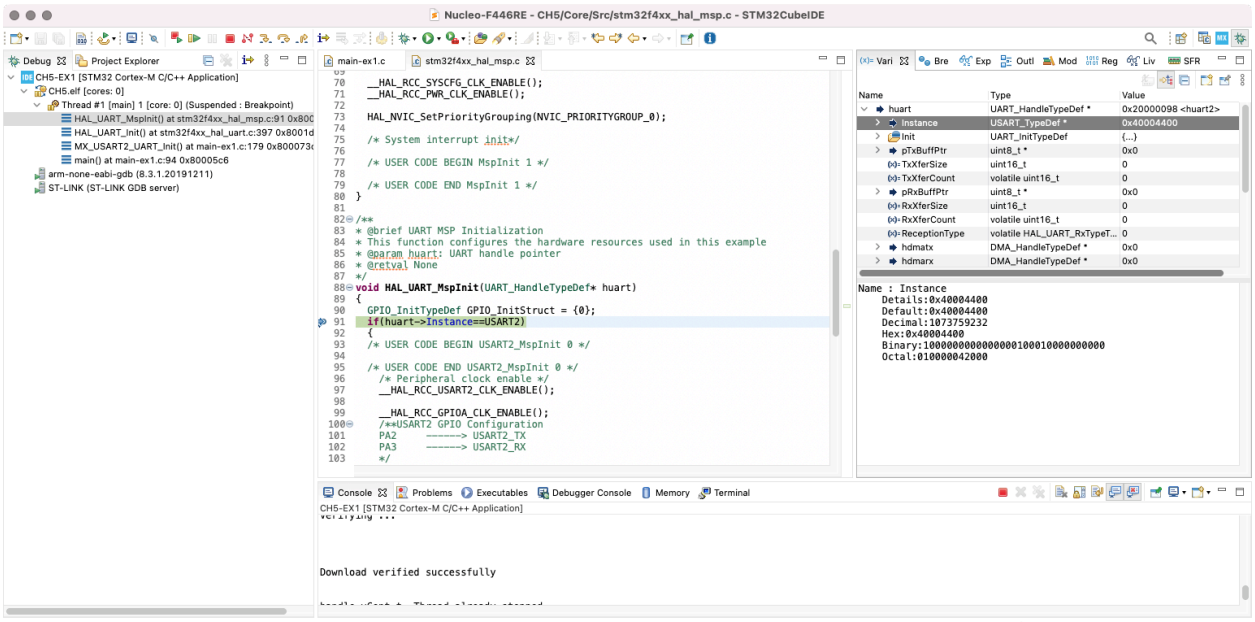


Figure 5.3: The *Debug Perspective*

Let us see what each view is used for. The top-left view is called **Debug** and it shows all the running debug activities. This is a tree-view and, when the firmware execution is halted, it shows the complete call stack offering a quick way to navigate inside the call stack.

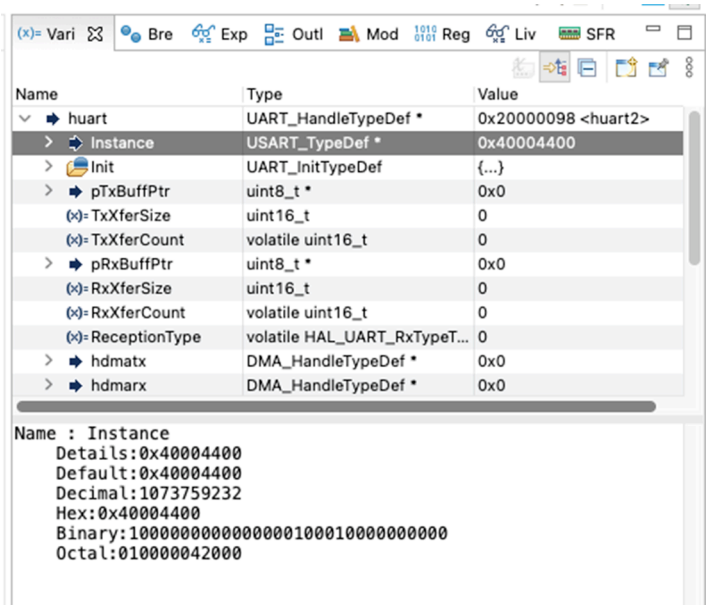


Figure 5.4: The variables inspection pane in the *debug perspective*

The top-right view contains several sub-panes. The **Variables** one offers the ability to inspect the content of variables defined in the current stack frame (that is, the selected procedure in the call stack). Clicking on an inspected variable with the right button of mouse, we can further customize the way the variable is shown. For example, we can change its numeric representation, from decimal (the default one) to hexadecimal or binary form. We can also cast it to a different datatype (this is really useful when we are dealing with raw amount of data that we know to be of a given type - for example, a bunch of bytes coming from a stream file). We can also go to the memory address where the variable is stored clicking on the **View Memory**... entry in the contextual menu.
The **Breakpoint** pane lists all the used breakpoints in the application. A *breakpoint* is a hardware primitive that allows to stop the execution of the firmware when the *Program Counter*(PC) reaches a given instruction. When this happens, the debugger is stopped, and Eclipse will show the context of the halted instruction. Every Cortex-M base MCU has a limited number of hardware breakpoints. **Table 5.1** summarizes the maximum breakpoints and watchpoints[5] for a given Cortex-M family.

Table 5.1: Available breakpoints/watchpoints in Cortex-M cores

Cortex-M	Breakpoints	Watchpoints
M0/0+	4	2
M3/4/7/33	8	4

[5]A watchpoint, indeed, is a more advanced debugging primitive that allows to define conditional breakpoints over data and peripheral registers, that is the MCU halts its execution only if a variable satisfies an expression (e.g. `var == 10`). We will analyze watchpoints in Chapter 24.

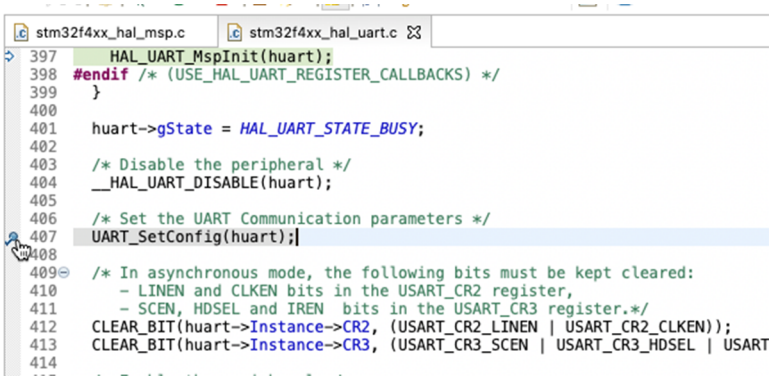


Figure 5.5: How to add a breakpoint at a given line number

Eclipse allows to easily setup breakpoints inside the code from the editor view in the center of **Debug perspective**. To place a breakpoint, simply double-click on the greyish stripe on the left of the editor, near to the instruction where we want to halt the MCU execution. A blue bullet will appear, as shown in **Figure 5.5**.

When the program counter reaches the first assembly instruction constituting that line of code, the execution is halted, and Eclipse shows the corresponding line of code as shown in **Figure 5.3**. Once we have inspected the code, we have several options to resume the execution. **Table 5.2** explain the usage of the most relevant icons on the Eclipse debug toolbar.

Table 5.2: Most relevant icons on the Eclipse debug toolbar

Icon	Description
	This icon is used ignore all breakpoints and continue the execution without interruptions.
	This icon is used to do a soft reset of MCU, without stopping the debug and relaunch it again.
	This icon terminates the debug session, starts a build of the project and restart debug session.
	This icon resumes the debug session after the MCU reached a breakpoint or an explicit pause by the user.
	This icon halts the code execution to the next C statement.
	This icon causes the end of the debug session. GDB is terminated and the target board is halted.
	This icon is the first one of two icons used to do step-by-step debugging. When we execute the firmware line-by-line, it could be important to enter inside a called routine. This icon allows to do this, otherwise the next icon is what needed to execute the next instruction inside the current stack frame.

Table 5.2: Most relevant icons on the Eclipse debug toolbar

Icon	Description
	This icon has - unfortunately - a counterintuitive name. It is called *step over*, and its name might suggest "skip the next instruction" (that is, **go over**). But this icon is the one used to execute the next instruction. Its name comes from the fact that, unlike the previous icon, it executes a called routine without entering inside it.
	By clicking on this icon, the execution will resume and the MCU will keep running till the exit (that is, the `return`) from the current routine. The execution will stop exactly to next instruction in the calling function.

Finally, in the views on the right you can find another two interesting views: **SFR** and **Registers**. They display both the content of all hardware registers in the given STM32 MCU and the content of all Cortex-M core registers. They can be really useful to understand the current state of a peripheral or the Cortex-M core. In Chapter 24 about debugging we will see how to deal with Cortex-M exceptions, and we will learn how to interpret the content of some important Cortex-M registers.

5.2.1 Debug Configurations

Eclipse is a generic and high configurable IDE, and it allows to create multiple debug configurations that easily fit our development scenario.
So far, we started the debug session by simply clicking on the corresponding icon on the toolbar (see **Figure 5.2**). However, the first time we click on that icon, we ask STM32CubeIDE to automatically configure the debug operations on behalf of us.

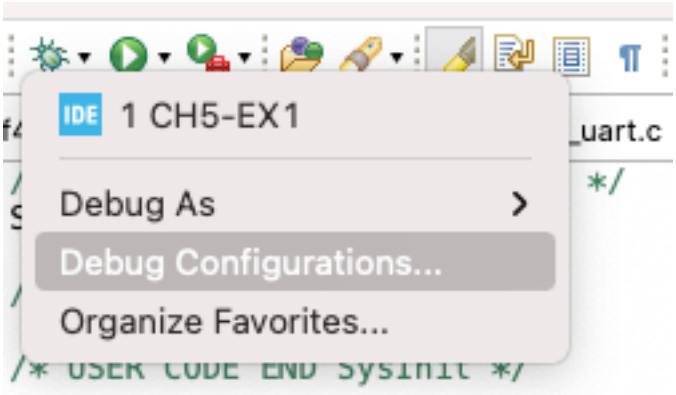


Figure 5.6: Debug Contextual Menu

By clicking on the down arrow close to the debug icon we can access to debug contextual menu (see **Figure 5.6**). Selecting the **Debug Configurations**... we can access to all debug configurations, as shown in **Figure 5.7**.

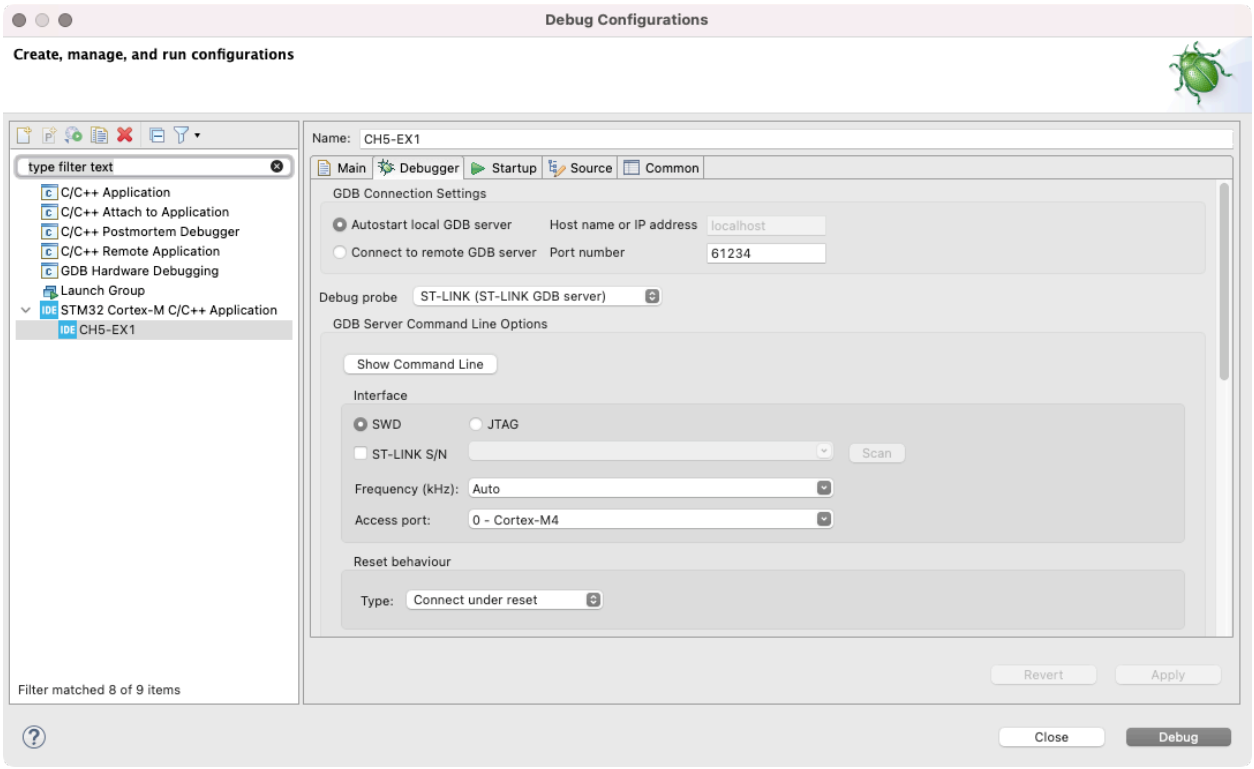


Figure 5.7: Debug Configurations Dialog

The view is divided in two main panes. On the left you can see a tree pane containing several configuration types. We are interested to the **STM32 Cortex-M C/C++ Application**. By expanding the entry, you can see the debug configuration created for us (the name of the configuration corresponds to the project name).
On the right you can find a tabbed pane with several tabs. The most notably ones are **Main**, **Debugger** and **Startup**.

The **Main** tab essentially contains the name of the project and which binary file to load on the target MCU to start a debug session. The **Debugger** view contains several relevant options to configure the debug session. Some of those options are advanced topics we will discuss in a later chapter. Here we are going to describe the most important ones.

- **GDB Connection Settings**: this group of settings is related to the configuration of the GDB Server. We can choose if to connect to a local or a remote server, its IP address and port number. It is strongly suggested to leave all the options as-is.
- **Debug Probe**: STM32CubeIDE supports three different debug probes (the standard ST-LINK, the SEGGER J-Link and OpenOCD). In this text is assumed the usage of the ST-LINK debug probe integrated in the Nucleo board. However, we will discuss about the other twos in a following chapter.
- **Interface**: with these settings you can choose which MCU debug port to use. The majority STM32 MCUs support both JTAG and SWD interfaces. In this book, we assume the usage of SWD interface.

The **Reset behavior** section requires a deeper explanation. Sometimes it happens that it is not possible to flash the MCU or to debug it using ST-LINK. Another recognizable symptom is that the ST-LINK LD1 LED (the one that blinks red and green alternatively while the board is under debugging) stops blinking and remains frozen with both the LEDs ON. When this happens, it means that the ST-LINK debugger cannot access to the debug port (through SWD interface) of the target MCU or the flash is locked preventing its access to the debugger.

There are usually two reasons that leads to this faulty condition:

- SWD pins have been configured as general-purpose GPIOs (this often happens if we perform a reset of pins configuration in CubeMX).
- The MCU is in a deep *low-power* mode that turns off the debug port.
- There is something wrong with the option bytes configuration (probably the flash has been write-protected or read protection level 1 is turned ON).

To address this issue, we have to force ST-LINK debugger to connect to the target MCU while keeping its `nRST` pin low. This operation is called *connection under reset*, and it can be performed by selecting one **Reset behavior**, which are described next.

- **Connect under reset** (default): ST-LINK reset line is activated and ST-LINK connects in the SWD or JTAG mode while reset is active. Then the reset line is deactivated.
- **Software system reset**: System reset is activated by software writing the in `RCC` register. This resets the core and peripherals and can reset the whole system as the reset pin of the target is asserted by itself.
- **Hardware reset**: ST-LINK reset line is activated and deactivated (pulse on reset line), then ST-LINK connects in the SWD or JTAG mode.
- **Core reset**: Core reset is activated by software writing in a Cortex-M register (not possible on Cortex®-M0/0+/33 cores). This only resets the core, not the peripherals nor the reset pin.
- **None**: For attachment to a running target where the program is downloaded into the device already. There must not be any file program command in the **Startup** tab.

The **Startup** tab configures how to start a debug session. The **Initialization Commands** edit field can be updated with any kind of GDB or GDB server `monitor` commands if there is any special need to send some commands to the GDB server before load commands are sent. For instance, when using ST-LINK GDB server a `monitor flash mass_erase` command can be entered here if a full FLASH memory erase is needed before load.
The **Load Image and Symbols** list box must contain the file(s) to debug. The **Runtime Options** section contains checkboxes to set the start address and breakpoint and enable exception handling and resume. The **Set breakpoint at** checkbox is enabled by default and the edit field displays `main`. It means that, by default, a breakpoint is set at `main()` routine when the program is debugged. This is the reason why the execution halts at `main()` at the beginning of every debug session.
Three exception checkboxes, are used to make it easier to find problems when debugging an application:

- **Exception on divide by zero**: it is enabled by default to make it easier to trap a divide-by-zero error when debugging.
- **Exception on unaligned access**: it can be enabled to get exceptions if there are any unaligned memory accesses
- **Halt on exception**: it is enabled by default so that program execution halts when an exception error occurs during debugging.

5.3 I/O Retargeting

In Chapter 4 we talked about the possibility to use standard C I/O primitives like `printf()`/`scanf()` to exchange data between the target MCU and the outside world. Often, the usage of breakpoints is not possible while debugging, because this would cause the loss of relevant events. At the same time, to print on a serial console[6] a few messages could help a lot in understanding what's going wrong with our firmware[7]. Finally, thanks to the string formatting capabilities of the `printf()` function, we do not have to deal to datatype conversion while printing a simple integer.

The simplest and effective solution is to redefine the needed system calls (`_write()`, `_read()`, `_isatty()`, `_close()`, `_fstat()`) to retarget the `STDIN`, `STDOUT` and `STDERR` standard streams to the Nucleo USART2. This can be easily done in the following way:

Filename: `CH5-EX1/Core/Src/retarget.c`

```
15 UART_HandleTypeDef *gHuart;
16
17 void RetargetInit(UART_HandleTypeDef *huart) {
18   gHuart = huart;
19
20   /* Disable I/O buffering for STDOUT stream, so that
21    * chars are sent out as soon as they are printed. */
22   setvbuf(stdout, NULL, _IONBF, 0);
23 }
24
25 int _isatty(int fd) {
26   if (fd >= STDIN_FILENO && fd <= STDERR_FILENO)
27     return 1;
28
29   errno = EBADF;
30   return 0;
31 }
32
33 int _write(int fd, char* ptr, int len) {
```

[6] To interact with the serial console you need a terminal emulator. For more information, follow the instructions in Chapter 8.

[7] For the sake of completeness, the solution implemented in the `CH5-EX1` is not that fast. This for two reasons. First of all, it uses the CubeHAL that is not implemented considering the code speed as a fundamental requirement. Secondly, it uses the UART in *polling mode*: the UART itself is not a high-speed peripheral, and driving it in polling mode makes the code calling the `HAL_UART_Trasmit()` really slow. Even printing a string of few characters will slow down a lot your code. For this reason, consider the `retarget.c` just as a bare-bone example. It should be coded by using the UART in *interrupt mode* or - a lot better - in *DMA mode*. We will study these advanced topics later in text.

```
34    HAL_StatusTypeDef hstatus;
35
36    if (fd == STDOUT_FILENO || fd == STDERR_FILENO) {
37      hstatus = HAL_UART_Transmit(gHuart, (uint8_t *) ptr, len, HAL_MAX_DELAY);
38      if (hstatus == HAL_OK)
39        return len;
40      else
41        return EIO;
42    }
43    errno = EBADF;
44    return -1;
45  }
46
47  int _close(int fd) {
48    if (fd >= STDIN_FILENO && fd <= STDERR_FILENO)
49      return 0;
50
51    errno = EBADF;
52    return -1;
53  }
54
55  int _read(int fd, char* ptr, int len) {
56    HAL_StatusTypeDef hstatus;
57
58    if (fd == STDIN_FILENO) {
59      hstatus = HAL_UART_Receive(gHuart, (uint8_t *) ptr, 1, HAL_MAX_DELAY);
60      if (hstatus == HAL_OK)
61        return 1;
62      else
63        return EIO;
64    }
65    errno = EBADF;
66    return -1;
67  }
68
69  int _fstat(int fd, struct stat* st) {
70    if (fd >= STDIN_FILENO && fd <= STDERR_FILENO) {
71      st->st_mode = S_IFCHR;
72      return 0;
73    }
74
75    errno = EBADF;
76    return 0;
77  }
```

A part for the usage of the USART peripheral, that we will study in Chapter 8, the code is quite self explanatory. The most relevant functions are `_write()` and `_read()`, which makes usage of the

`HAL_UART_*` routines to exchange data over the UART.

To retarget the standard streams in your firmware, just initialize the library calling the `RetargetInit()` and passing the pointer to the `UART_HandleTypeDef` instance of the UART2 (line 46). For example, the following code shows how to use `printf()`/`scanf()` functions in your firmware:

Filename: `CH5-EX1/Core/Src/main.c`

```
1  #include "main.h"
2  #include <retarget.h>
3  #include <stdio.h>
4
5  /* Private variables ---------------------------------------------------------*/
6  UART_HandleTypeDef huart2;
7
8  /* Private function prototypes -----------------------------------------------*/
9  void SystemClock_Config(void);
10 static void MX_GPIO_Init(void);
11 static void MX_USART2_UART_Init(void);
12
13 int main(void) {
14   uint8_t uTimes = 0;
15
16   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
17   HAL_Init();
18   /* Configure the system clock */
19   SystemClock_Config();
20
21   /* Initialize all configured peripherals */
22   MX_GPIO_Init();
23   MX_USART2_UART_Init();
24   /* Enables retarget of standard I/O over the USART2 */
25   RetargetInit(&huart2);
26
27   printf("How many times to print the message?: ");
28   scanf("%hhu", &uTimes);
29   printf("\r\n");
30
31   for(uint8_t i = 0; i < uTimes;) {
32         HAL_Delay(500);
33         printf("Hello, Nucleo: %u \r\n", ++i);
34   }
35   while(1);
36 }
```

Please, take note that this example assumes a project generated by following the same procedure shown in Chapter 3. If not all things are clear now, do not worry: after reading the Chapter 8 you will be able to understand every operations performed.

`printf()` and `float` datatypes.

If you are going to use `printf()`/`scanf()` functions to print/read `float` datatypes on the serial console (but also if you are going to use `sprintf()` and similar routines), you need to explicitly enable `float` support in `newlib-nano`, which is the more compact version of the C *runtime* library for embedded systems. To do this, go to **Project->Properties**... menu, then go to **C/C++ Build->Settings->MCU Settings** and check **Use float with printf from newlib-nano** and **Use float with scanf from newlib-nano** according to the feature you need, as shown in **Figure 5.8**. This will increase the firmware binary size.

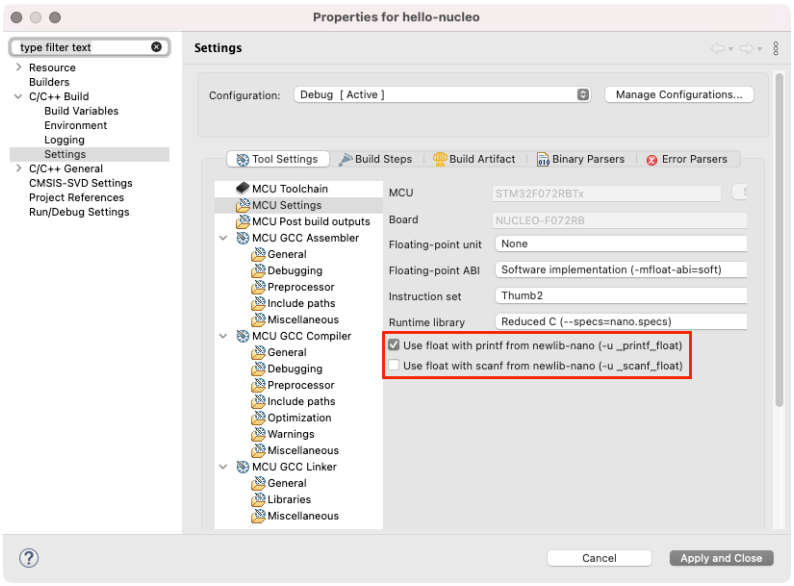


Figure 5.8: How to enable `float` support in `printf()` and `scanf()`