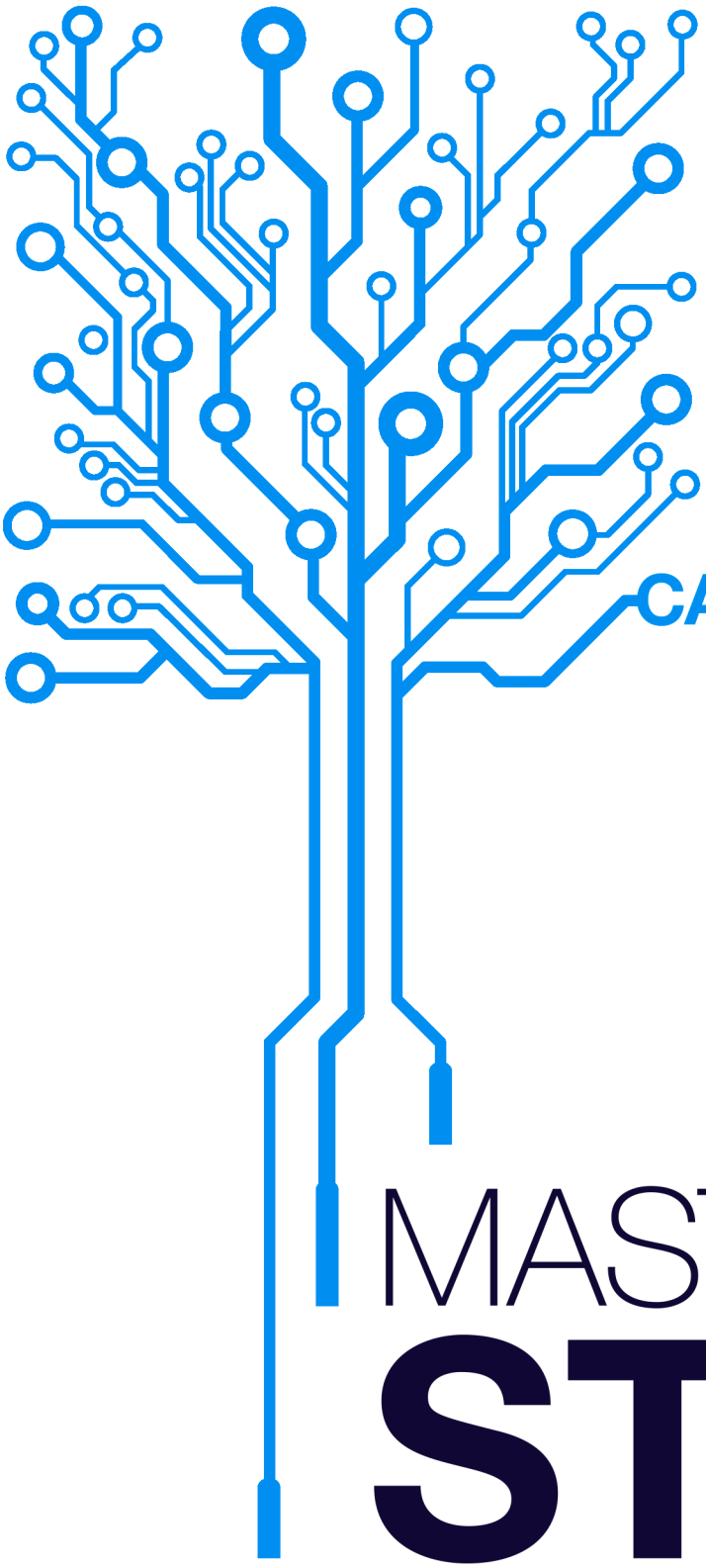
RELEASE **2.2**



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

Mastering STM32 - Second Edition

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

Carmine Noviello

This book is available at http://leanpub.com/mastering-stm32-2nd

This version was published on 2025-02-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2015-2024 Carmine Noviello

Tweet This Book!

Please help Carmine Noviello by spreading the word about this book on Twitter!

The suggested hashtag for this book is #MasteringSTM32.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#MasteringSTM32

To my wife Anna, who has always blindly supported me in all my projects

To my daughter Giulia, who completely upset my projects

Contents

Preface

It was the summer of 2015 when I began to consider the hypothesis of grouping a series of posts on my personal blog to give shape to a more structured guide about the use of STM32 microcontrollers. At that time, it was not trivial to set up a complete tool-chain for the STM32 portfolio unless you could afford a license for ARM Keil. Moreover, STM was migrating from the historical *Standard Peripheral Library* (SPL) to the new CubeHAL SDK, and it was not clear which path to follow to start learning this very interesting product lineup.

I started writing the very first chapters of this book, showing how to set up a complete and free Eclipse tool-chain based on the *GNU MCU Eclipse plug-ins* by Liviu Ionescu (now called *Eclipse Embedded CDT* and officially supported by the Eclipse Foundation). I decided to use the LeanPub platform, which allowed me to publish an in-progress book that I could update as soon as I added a new chapter. From the very first release, many people adopted the text and helped me a lot in shaping the book structure and its contents. It took me two years to complete the first edition, and trust me, it was very hard work, especially because things changed day-by-day. Over the years, the book has been adopted by several universities around the world as an official text in Embedded System classes. A lot of people contacted me to provide feedback, some asking for help with the text and some others with the development of their boards, some asking for a revision of the text, and some others for a revision of the examples, some criticizing the whole book, and others letting me know that they thank me every time they go to sleep.

Seven years later, things have changed. A lot. STM pushed hard the development of both the hardware and software ecosystem. The first release of the book was about nine STM32 families ranging across about 500 P/N. Now, there are eighteen families in the STM32 portfolio, spreading over more than 1200 P/N. But the huge improvement was on the software part. STM decided to fix the main issue with the STM32 portfolio: the lack of an official tool-chain. STM acquired Atollic and its TrueStudio IDE and launched the STM32CubeIDE, which, together with the whole STM32Cube initiative, represents a quantum leap for the development of STM32-based devices.

This required me to make a deep revision of the text. I started working on this second edition in the spring of 2021, and it took me about one year to update the text and add new content that was lacking in the first edition. This is a lot of time, but things changed a lot, even for me, in these years. A totally different job full of too many responsibilities and a daughter came in the middle, and now my free time ranges from 5:00 am to 7:00 am, and you can figure out how hard it is to work on a book with 900 pages in just two hours a day.

Even in the second edition, the book is divided into three parts: an introductory part showing how to set up the STM32CubeIDE and how to work with it; a part that introduces the basics of STM32 programming and the main aspects of the official HAL (Hardware Abstraction Layer); and a more advanced section covering aspects such as the use of a Real-Time Operating System, the boot sequence, and the memory layout of an STM32 application, and advanced peripherals like USB.

However, this book does not aim to replace official datasheets from ST Microelectronics. A datasheet is still the main reference about electronic devices, and it is impossible (as well as making little sense) to arrange the content of tens of datasheets in a book. You have to consider that the official datasheet of one of the latest - and not the most complex in the portfolio - STM32G4 MCU alone is almost three thousand pages! Hence, this text will offer a hint to start diving into the official documentation from ST. Moreover, this book will not focus on low-level topics and questions related to the hardware, leaving this hard work to datasheets. Lastly, this book is not a cookbook about custom and funny projects: you will find several good tutorials on the web.

Who Is This Book For?

This book is addressed to novices of the STM32 platform interested in learning how to program these fantastic microcontrollers in less time. However, *this book is not for people completely new to the C language or embedded programming*. I assume you have a decent knowledge of C and are not new to most fundamental concepts of digital electronics and MCU programming. The perfect reader of this book may be both a hobbyist or a student who is familiar with the Arduino platform and wants to learn a more powerful and comprehensive architecture, or a professional in charge of working with an MCU they do not know yet.

What About Arduino?

I received this question many times from several people in doubt about which MCU platform to learn. The answer is not simple for several reasons.

First of all, Arduino is not a given MCU family or a silicon manufacturer. Arduino[a] is both a *brand* and an *ecosystem*. Today, there are tens[b] of Arduino development boards available on the market, some with an 8-bit MCU and some others with more powerful 32-bit MCUs, even if it is common to refer to the Arduino UNO board as "the Arduino". Arduino UNO is a development board built around the ATMega328, an 8-bit microcontroller designed by Atmel. However, Arduino is not only a cold piece of hardware, but it is also a community built around the Arduino IDE (a derived version of Processing[c]) and the Arduino libraries, which greatly simplify the development process on ATMega MCUs. This large, stable, and continuously growing community has developed hundreds of libraries to interface with as many hardware devices and thousands of examples and applications.

So, the question is: "Is Arduino good for professional applications or for those wanting to develop the next mainstream product on Kickstarter?". The answer is: "YES, definitely." I myself have developed a couple of custom boards for a customer, and since these boards were based on the ATMega328 IC (the SMD version), the firmware was developed using the Arduino IDE. So, it is not true that Arduino is only for hobbyists and students.

However, if you are looking for something more powerful than an 8-bit MCU or if you want to increase your knowledge about firmware programming (the Arduino environment hides too much detail about what's under the hood), the STM32 is probably the best choice for you. Thanks to a development environment based on Eclipse and GCC, you will not have to invest a fortune to start developing STM32 applications. Moreover, if you are building a cost-sensitive device, where each PCB square inch makes a difference for you, consider that the STM32F0 value line is also known as the *32-bit MCU for 32 cents*. This means that the low-cost STM32 line has a price perfectly comparable with 8-bit MCUs but offers a lot more computing power, hardware capabilities, and integrated peripherals.

[a] https://www.arduino.cc/
[b] https://www.arduino.cc/en/Main/Products
[c] https://processing.org/

How to Integrate This Book?

This book does not aim to be a fully comprehensive guide to STM32 microcontrollers but is essentially a guide to developing applications using the official ST HAL. It is strongly suggested to integrate it with a book about the ARM Cortex-M architecture, and the series by Joseph Yiu[1] is the best source for every Cortex-M developer.

[1] http://amzn.to/1P5sZwq

How Is the Book Organized?

The book is divided into twenty-eight chapters covering the following topics.

Chapter 1 gives a brief and preliminary introduction to the STM32 platform. It presents the main aspects of these microcontrollers, introducing the reader to the ARM Cortex-M architecture. Moreover, the key features of each STM32 subfamily (L0, F1, etc.) are briefly explained. The chapter also introduces the development board used throughout this book as the testing board for the presented topics: the Nucleo.

Chapter 2 shows how to set up the STM32CubeIDE to start developing STM32 applications. The chapter is divided into three different branches, each one explaining the tool-chain setup process for the Windows, Linux, and Mac OS X platforms.

Chapter 3 is dedicated to showing how to build the first application for the STM32 Nucleo development board. This is a really simple application: a blinking LED, which is, without a doubt, the Hello World application of hardware.

Chapter 4 is about the STM32CubeMX tool, our main companion every time we need to start a new application based on STM32 MCUs. The chapter gives a hands-on presentation of the tool, explaining its characteristics and how to configure the MCU peripherals according to the features we need.

Chapter 5 introduces the reader to debugging, showing a brief view of STM32CubeIDE's debugging capabilities. Finally, the reader is introduced to an important topic: I/O retargeting.

Chapter 6 gives a quick overview of the ST CubeHAL, explaining how peripherals are mapped inside the HAL using *handlers* to the peripheral memory-mapped region. Next, it presents the `HAL_GPIO` libraries and all the configuration options offered by STM32 GPIOs.

Chapter 7 explains the mechanisms underlying the NVIC controller: the hardware unit integrated in every STM32 MCU, which is responsible for the management of exceptions and interrupts. The `HAL_NVIC` module is introduced extensively, and the differences between Cortex-M0/0+ and Cortex-M3/4/7 are highlighted.

Chapter 8 gives a practical introduction to the `HAL_UART` module used to program the UART interfaces provided by all STM32 microcontrollers. Moreover, a quick introduction to the difference between UART and USART interfaces is given. Two ways to exchange data between devices using a UART are presented: *polling* and *interrupt*-oriented modes. Finally, we present in a practical way how to use the integrated VCP of every Nucleo board and how to retarget the `printf()`/`scanf()` functions using the Nucleo's UART.

Chapter 9 talks about the DMA controller, showing the differences between several STM32 families, like the more powerful and recent DMAMUX available in STM32L4+/L5/Gx/H7 families. A more detailed overview of the internals of an STM32 MCU is presented, describing the relations between the Cortex-M core, DMA controllers, and slave peripherals. Moreover, it shows how to use the `HAL_DMA` module in both *polling* and *interrupt* modes. Finally, a performance analysis of *memory-to-memory* transfers is presented.

Chapter 10 introduces the clock tree of an STM32 microcontroller, showing the main functional blocks and how to configure them using the `HAL_RCC` module. Moreover, the CubeMX *Clock configuration* view is presented, explaining how to change its settings to generate the right clock configuration.

Chapter 11 is a walkthrough into timers, one of the most advanced and highly customizable peripherals implemented in every STM32 microcontroller. The chapter will guide the reader step-by-step through this subject, introducing the most fundamental concepts of *basic*, *general-purpose*, and *advanced* timers. Moreover, several advanced usage modes (master/slave, external trigger, input capture, output compare, PWM, etc.) are illustrated with practical examples.

Chapter 12 provides an overview of the *Analog to Digital* (ADC) peripheral. It introduces the reader to the concepts underlying SAR ADCs and then explains how to program this useful peripheral using the designated CubeHAL module. Moreover, this chapter provides a practical example that shows how to use a hardware timer to drive ADC conversions in DMA mode.

Chapter 13 briefly introduces the *Digital to Analog* (DAC) peripheral. It provides the most fundamental concepts underlying R-2R DACs and how to program this useful peripheral using the designated CubeHAL module. This chapter also shows an example detailing how to use a hardware timer to drive DAC conversions in DMA mode.

Chapter 14 is dedicated to the I²C bus. The chapter starts by introducing the essentials of the I²C protocol and then shows the most relevant routines from the CubeHAL to use this peripheral. Moreover, a complete example that explains how to develop I²C *slave* applications is also shown.

Chapter 15 is dedicated to the SPI bus. The chapter starts by introducing the essentials of the SPI specification and then shows the most relevant routines from the CubeHAL to use this fundamental peripheral.

Chapter 16 talks about the CRC peripheral, briefly introducing the math behind its calculation, and shows the related CubeHAL module used to program it.

Chapter 17 is about IWDT and WWDT timers, and it briefly introduces their role and how to use the related CubeHAL modules to program them.

Chapter 18 talks about the RTC peripheral and its main functionalities. The most relevant CubeHAL routines to program the RTC are also shown.

Chapter 19 introduces the reader to the power management capabilities offered by STM32F and STM32L microcontrollers. It starts by showing how Cortex-M cores handle low-power modes, introducing `WFI` and `WFE` instructions. Then it explains how these modes are implemented in STM32 MCUs. The corresponding `HAL_PWR` module is also described.

Chapter 20 analyzes the activities involved during the compilation and linking processes, which define the memory layout of an STM32 application. A bare-bone application is shown, and a complete and working *linker script* is designed from scratch, showing how to organize the STM32 memory space. Moreover, the usage of CCM RAM is presented, as well as other important Cortex-M functionalities like *vector table* relocation.

Chapter 21 introduces the internal flash memory and its related controller available in all STM32 microcontrollers. It illustrates how to configure and program this peripheral, showing the related CubeHAL routines. Moreover, a walk-through of the STM32F7 bus and memory organization introduces the reader to the architecture of these high-performing MCUs.

Chapter 22 describes the operations performed by STM32 microcontrollers at startup. The whole booting process is described, and some advanced techniques (like the vector table relocation in Cortex-M0 microcontrollers) are explained. Moreover, a custom and secure bootloader is shown, which can upgrade the on-board firmware through the USART peripheral. The bootloader uses the AES algorithm to encrypt the firmware.

Chapter 23 is dedicated to the FreeRTOS Real-Time Operating System. It introduces the reader to the most relevant concepts underlying an RTOS and shows how to use the main FreeRTOS functionalities (like threads, semaphores, mutexes, and so on) using the CMSIS-RTOS v2 layer developed by ST on top of the FreeRTOS API. Moreover, some advanced techniques like the *tickless mode* in low-power design and the handling of concurrency with the C stdlib are shown.

Chapter 24 introduces the reader to some advanced debugging techniques. The chapter starts by explaining the role of the fault-related exceptions in Cortex-M based cores and how to interpret the related hardware registers to go back to the source of fault. Moreover, all STM32CubeIDE advanced debugging tools are presented, such as watchpoints, expressions, and SWV-related tools. Finally, a brief introduction to SEGGER J-LINK professional debuggers is given, and how to use them in the Eclipse tool-chain.

Chapter 25 briefly introduces the reader to the FatFs middleware. This library allows for manipulating structured filesystems created with the widespread FAT12/16/32 filesystem. The chapter also shows how ST engineers have integrated this library into the CubeHAL. Finally, it provides an overview of the most relevant FatFs routines and configuration options.

Chapter 26 describes a solution to interface Nucleo boards to the Internet by using the W5500 network processor. The chapter shows how to develop Internet- and web-based applications using STM32 microcontrollers even if they do not provide a native Ethernet peripheral. Moreover, the chapter introduces the reader to possible strategies to handle dynamic content in static web pages. Finally, an application of the FatFs middleware is shown in order to store web pages and the like on an external SD card.

Chapter 27 introduces one of the widespread communication protocols: USB 2.0. The chapter will guide the reader through the fundamentals of the USB specification, both from the hardware and the communication protocol points of view. Moreover, the STM32 USB Device Stack is deeply explained with practical examples about USB-CDC and USB-HID classes.

Chapter 28 shows how to start a new custom PCB design using an STM32 MCU. This chapter is mainly focused on hardware-related aspects such as decoupling, signal routing techniques, and so on. Moreover, it shows how to use CubeMX during the PCB design process and how to generate the application skeleton when the board design is complete.

F4

During the book, you will find some horizontal rulers with "badges" like the one above. This means that the instructions in that part of the book are specific to a given family of STM32 microcontrollers. Sometimes you could find a badge with a specific MCU type: this means that instructions are exclusively related to that MCU. A black horizontal ruler (like the one below) closes the specific section. This means that the text returns to being generic for the whole STM32 platform.

You will also find several asides, each one starting with an icon on the left. Let us explain them.

This a warning box. The text contained explains important aspects or gives important instructions. It is strongly recommended to read the text carefully and follow the instructions.

This is an information box. The text contained clarifies some concepts introduced before.

This is a tip box. It contains suggestions to the reader that could simplify the learning process.

This a discussion box, and it is used to talk about the subject in a broader way.

This a bug-related box, used to report some specific and/or un-resolved bug (both hardware and software).

Differences With the First Edition

Every next release is never a complete refactoring of the previous one. And this is also true for technical books. This second edition has some major differences from the first edition:

- It is updated to the recent evolutions of the STM32 portfolio.
- It was completely changed to cover the STM32CubeIDE tool-chain, which is different from the one shown in the first edition (even if both are based on Eclipse and GCC).
- It fixes several errors (some really severe).
- It introduces completely new topics not available in the first edition.

If you want to have a detailed list of the differences in each chapter, then you can jump to Appendix D.

About the Author

When someone asks me about my career and my studies, I like to say that I am a high-level programmer that someday started fighting against bits.

I began my career in informatics when I was only a young boy with an 80286 PC, but unlike all those who started programming in BASIC, I decided to learn a quite uncommon language: Clipper. Clipper was a language mostly used to write software for banks, and a lot of people suggested that I should start with this programming language (uh?!?). When visual environments like Windows 3.1 started to become more common, I decided to learn the foundations of Visual Basic and I wrote several programs with it (one of them, a program for patient management for medical doctors, made it to the market) until I began college, where I started programming in Unix environments and programming languages like C/C++. One day, I discovered what would become the programming language of my life: Python. I have written hundreds of thousands of lines of code in Python, ranging from web systems to embedded devices. I think Python is an expressive and productive programming language, and it is always my first choice when I have to code something.

For about eight years, I worked as a research assistant at the National Research Council in Italy (CNR), where I spent my time coding web-based and distributed content management systems. In 2010, my professional life changed dramatically. For several reasons that I will not detail here, I found myself slingshot into a world I had always considered obscure: electronics. I first started developing firmware on low-cost MCUs, then designing custom PCBs. In 2010, I co-founded a company that produced wireless sensors and control boards used for small-scale automation. Unfortunately, this company was unlucky and did not reach the success we wanted.

In 2013, I was introduced to the STM32 world during a presentation day at the ST headquarters in Naples. Since then, I have successfully used STM32 microcontrollers in several products I have designed, ranging from industrial automation to security tokens. Even thanks to the success of this book, I currently work mainly as a full-time hardware consultant for some Italian companies.

In 2016, I joined Bit4id, a worldwide leader in the sector of PKI systems. I started as a hardware consultant to help develop a Bluetooth smart card reader, and I ended up a few years later becoming the head of the most relevant company's Business Unit, managing a total business of ∼€12M, thousands of customers, more than 30 brilliant engineers, salespeople, and a complete production for ∼2M devices every year. To support this switch to a more management role, I attended an MBA in one of the EU business schools. Do I like this new life in a management role? Next question??!?! :-D

In 2021, my first daughter appeared in my life. Well, since then things changed a lot, as you can figure out. My spare time reduced a lot, and this is one of the reasons why the second edition of the book is really late ;-)

Errata and Suggestions

I am aware of the fact that there are several errors in the text. Unfortunately, English is not my mother tongue, and this is one of the main reasons I like lean publishing: being an in-progress book, I have all the time to check and correct them. I have decided that once this book reaches completion, I will look for a professional editor to help me fix all the mistakes in my English. However, feel free to contact me to signal what you find.

On the other hand, I am totally open to suggestions and improvements about the book content. I like to think that this book will save your day every time you need to understand an aspect related to STM32 programming, so feel free to suggest any topic you are interested in or to signal parts of the book which are not clear or well explained.

You can reach me through this book's website: http://www.carminenoviello.com/en/mastering-stm32/[2]

Book Support

I have set up a small forum on my personal website as support site for the topics presented in this book. For any question, please subscribe here: http://www.carminenoviello.com/en/mastering-stm32/[3].

It is impossible for me to answer questions sent privately by e-mail, since they are often variations on the same topic. I hope you understand.

How to Help the Author

If you want to help me, you may consider:

- giving me feedback about unclear things or errors contained both in the text and examples;
- writing a small review about what you think[4] of this book in the feedback section[5];
- using your favorite social network or blog *to spread the word*. The suggested hashtag for this book on Twitter is #MasteringSTM32[6];

[2] http://www.carminenoviello.com/en/mastering-stm32/
[3] http://www.carminenoviello.com/en/mastering-stm32/
[4] Negative feedback is also welcome ;-)
[5] https://leanpub.com/mastering-stm32/feedback
[6] https://twitter.com/search?q=%23MasteringSTM32

Copyright Disclaimer

This book contains references to several products and technologies whose copyright is owned by their respective companies, organizations or individuals.

ART™ Accelerator, STM32, ST-LINK, STM32Cube, STM32CubeIDE, STM32Programmer and the *STM32 logo with the white butterfly on the cover of this book* are copyright ©ST Microelectronics NV.

ARM, Cortex, Cortex-M, CoreSight, CoreLink, Thumb, Thumb-2, TrustZone, AMBA, AHB, APB, Keil are registered trademarks of ARM Holdings.

GCC, GDB and other tools from the GNU Collection Compilers mentioned in this book are copyright © Free Software Foundation.

Eclipse is copyright of the Eclipse community and all its contributors.

During the rest of the book, I will mention the copyright of tools and libraries I will introduce. If I have forgotten to attribute copyrights for products and software used in this book and you think I should add them here, please e-mail me through the LeanPub platform.

Credits

The cover of this book was designed by Alessandro Migliorato (AleMiglio[7])

The rest of the chapter is not available in the book sample

[7] https://99designs.it/profiles/alemiglio

Acknowledgments to the First Edition

Even if there is just my name on the cover, this book would not have been possible without the help of a lot of people who have contributed during its development.

First and foremost, I big thank you to Alan Smith, manager of the ST Microelectronics site in Naples (Arzano - Italy). Alan, with persistence and great determination, came to my office more than three years ago bringing a couple of Nucleo boards with him. He said to me: *You must know STM32!*. This book was born almost that day!

I would like to thank several people that silently and actively contributed to this work. Enrico Colombini (aka Erix[8]) helped me a lot during the early stages of this book, by reviewing several parts of it. Without his initial support and suggestions, probably this book would have never seen the end. For a self-publishing and in-progress author the early feedback is paramount to better understand how to arrange a so complex work.
Ubaldo de Feo (aka @ubi[9] also helped me a lot by providing technical feedback and by performing an excellent proof-reading of some chapters.
Another special thanks goes to Davide Ruggiero, from ST Microelectronics in Naples, who helped me by reviewing several examples and editing the chapter about CRC peripheral (Davide is a mathematician and he better knows how to approach formulas :-)). Davide also actively contributed by donating me some wine bottles: without adequate fuel you cannot write a 900 pages book!
Some english speaking people tried to help me with my poor english, dedicating a lot of time and effort to several parts of the text. So a big thank you to: Omar Shaker, Roger Berger, J. Clarke, William Den Beste, J.Behloul, M.Kaiser. I hope not to forget anyone.

A big thanks also to all early adopters of the book, especially to those ones that bought it when it was made of just few chapters. This fundamental encouragement gave me the necessary energies to complete a so long and hard work.

Regards,
Carmine I.D. Noviello

[8] http://www.erix.it
[9] http://ubidefeo.com

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.

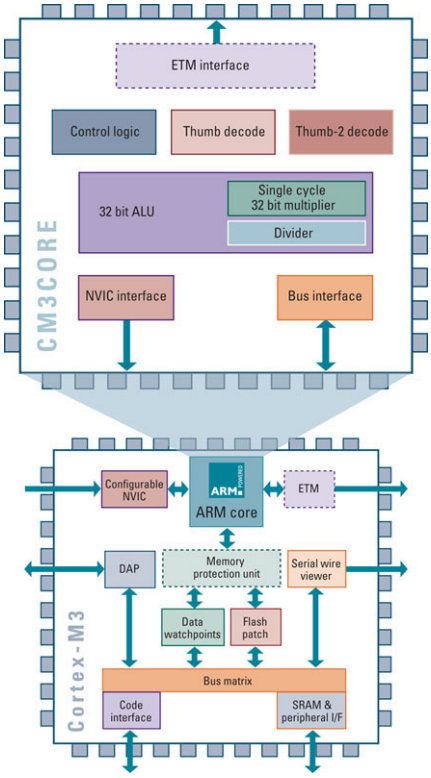


Figure 1.1: The relation between a Cortex-M3 core and a Cortex-M3 based MCU

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.

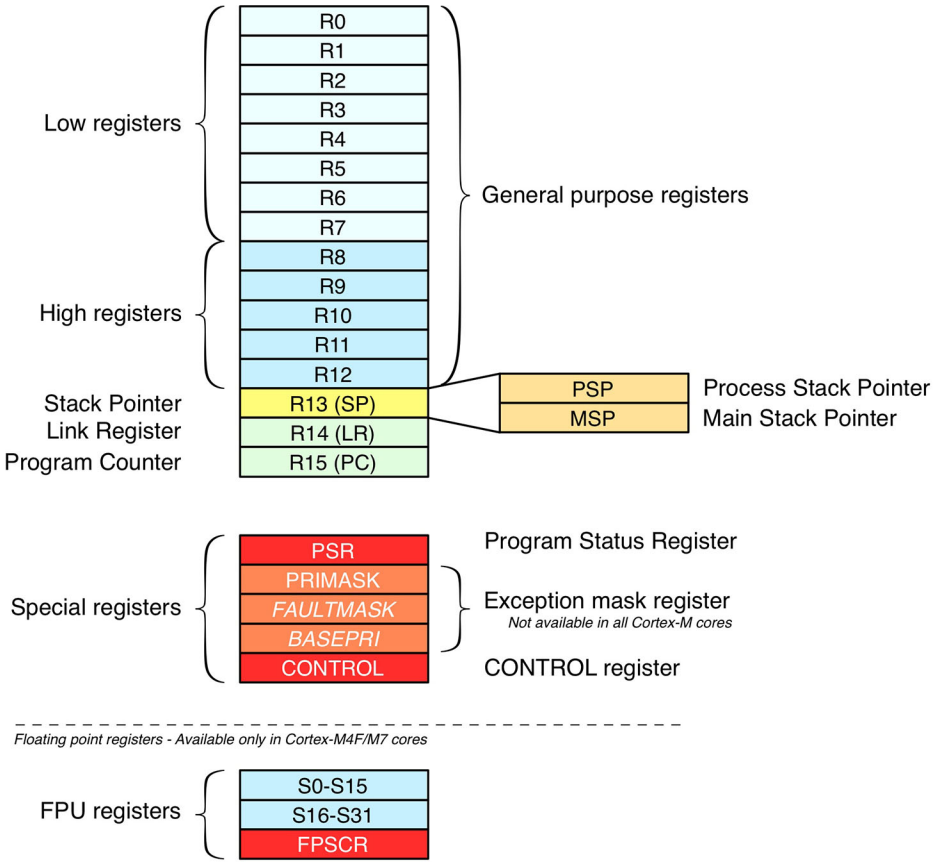


Figure 1.2: ARM Cortex-M core registers

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`.

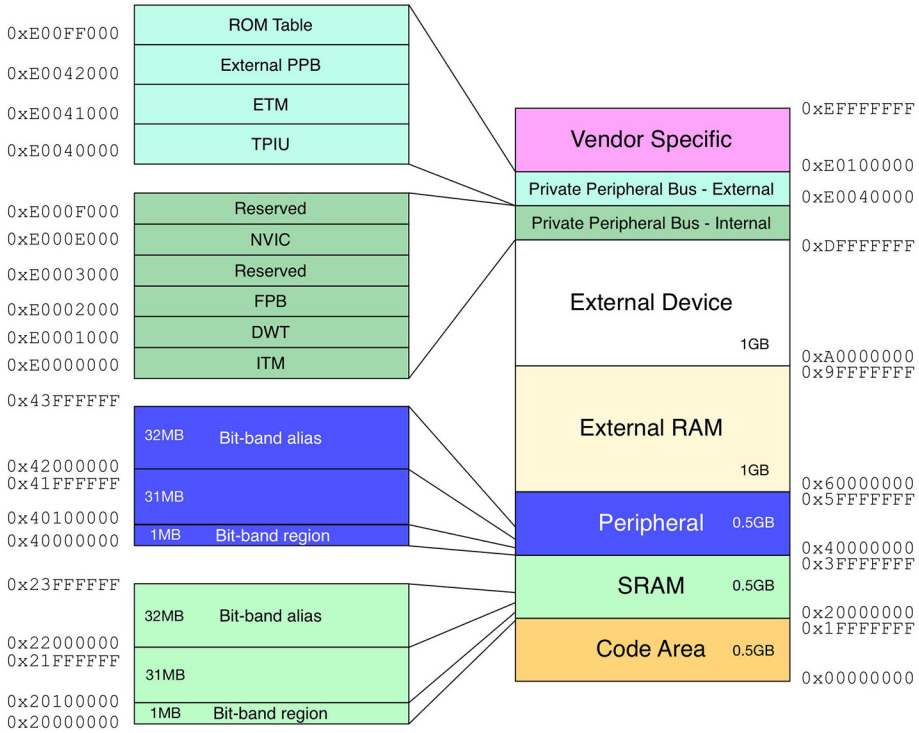


Figure 1.3: Cortex-M fixed memory address space

[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.

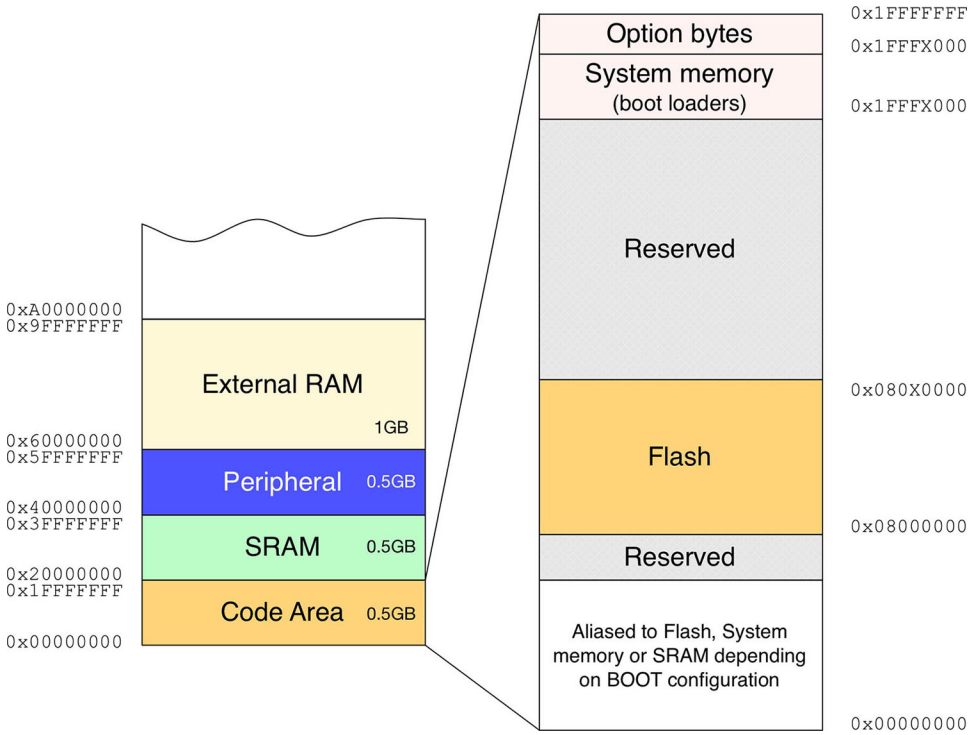


Figure 1.4: Memory layout of Code Area on STM32 MCUs

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.

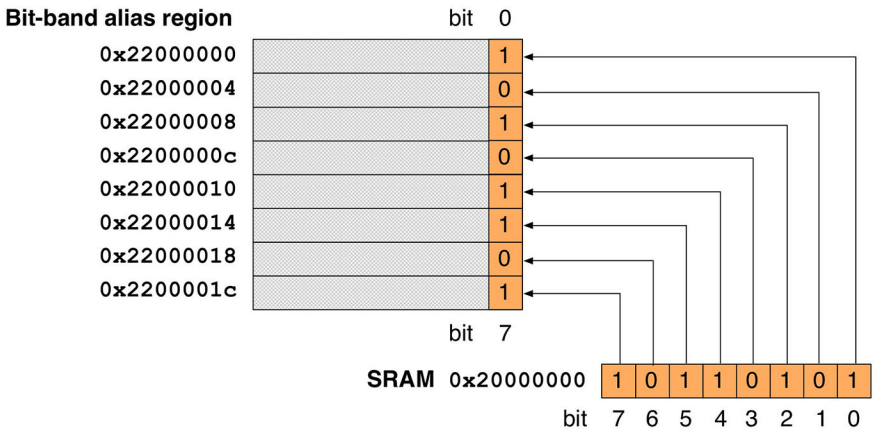


Figure 1.5: Memory mapping of SRAM address `0x2000 0000` in bit-banding region (first 8 of 32 bits shown)

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**.

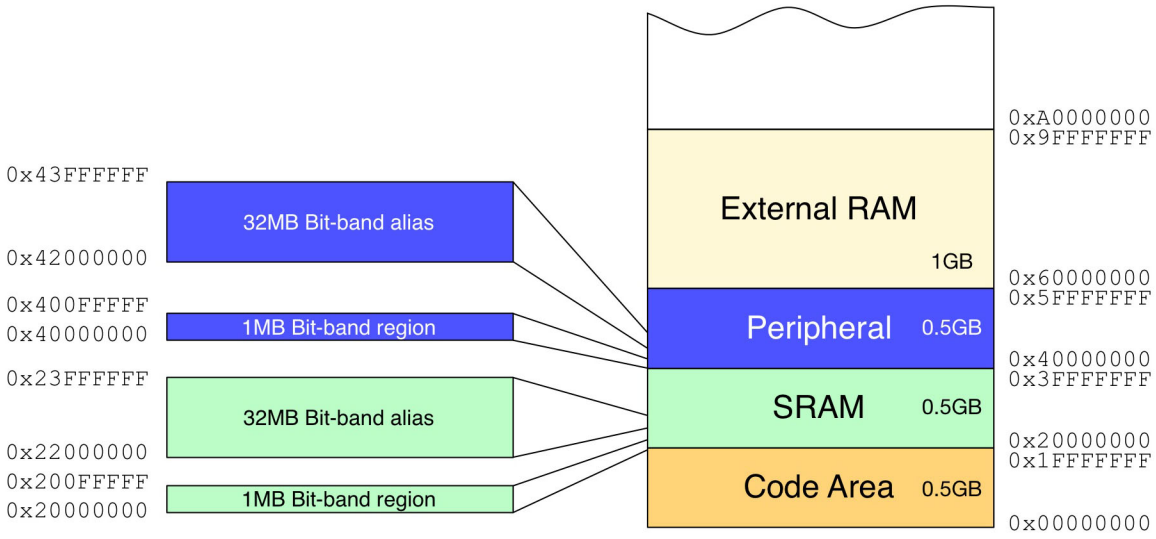


Figure 1.6: Memory map and bit-banding regions

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.

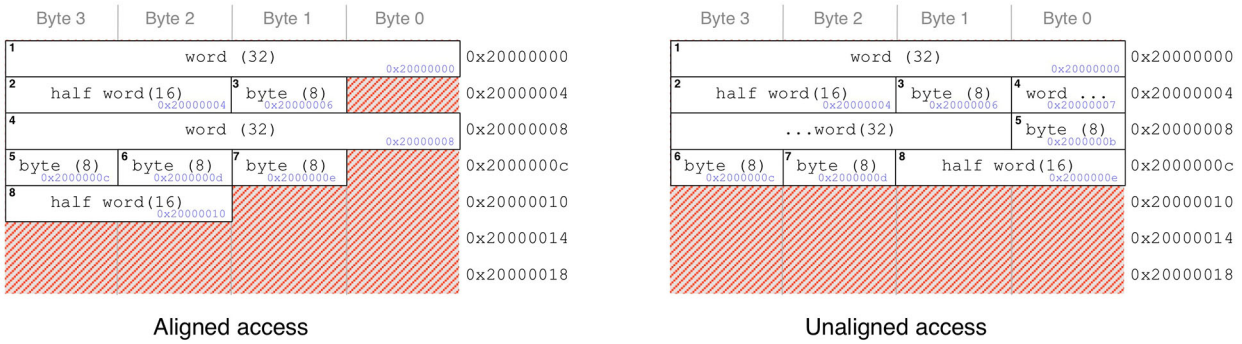


Figure 1.7: Difference between aligned and unaligned memory access

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.

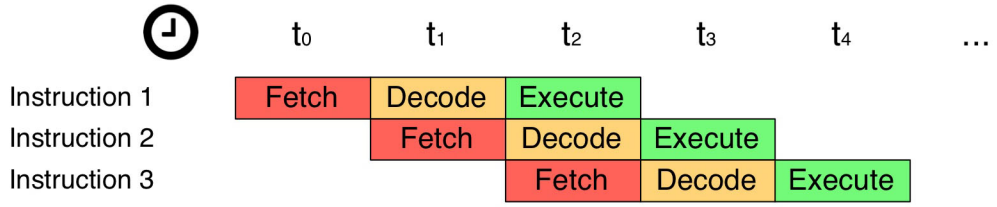


Figure 1.8: Three stage instruction pipeline

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*)

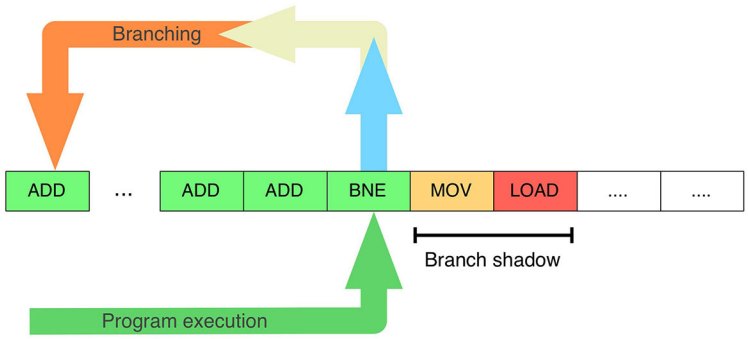


Figure 1.9: Branching in program execution related to pipelining

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.

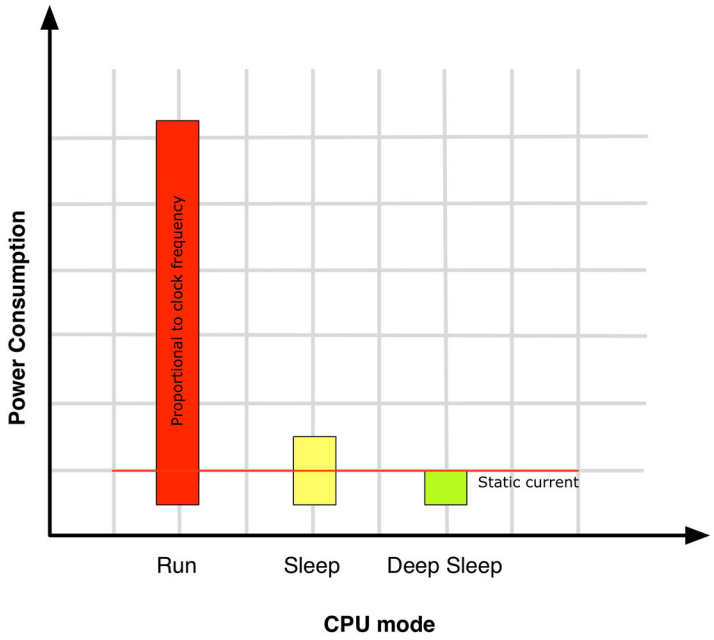


Figure 1.10: Cortex-M power consumption at different power modes

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.

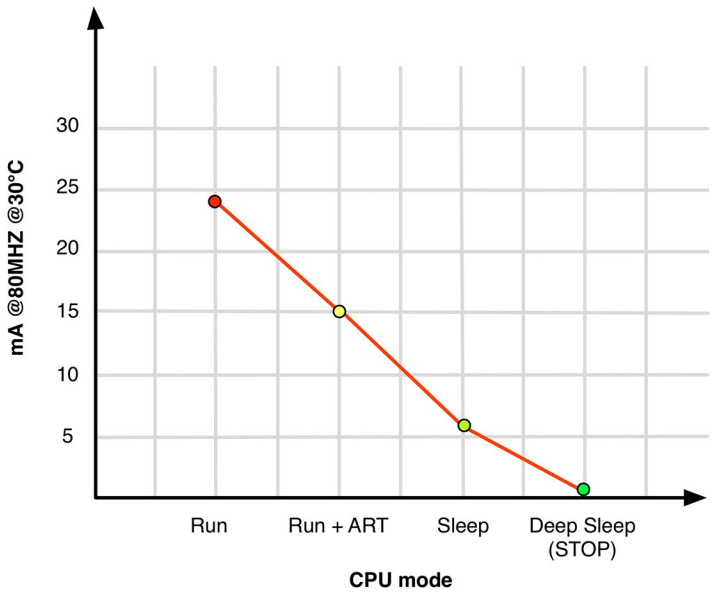


Figure 1.11: STM32F2 power consumption at different power modes

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.

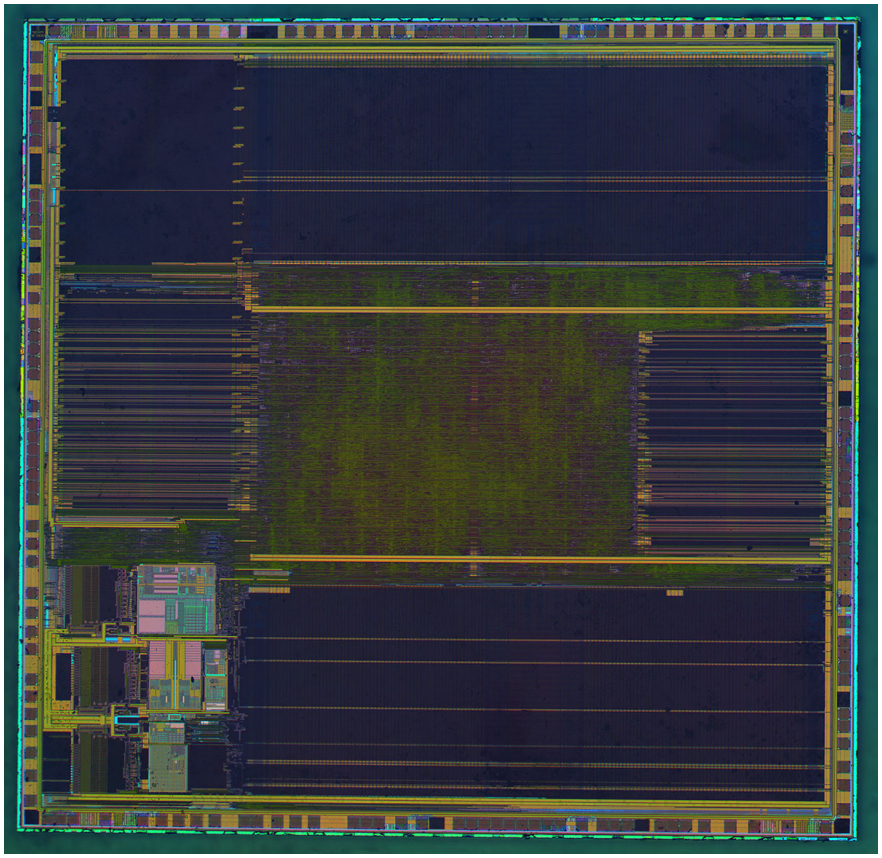


Figure 1.12: Internal die of an STM32F103 MCU

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.

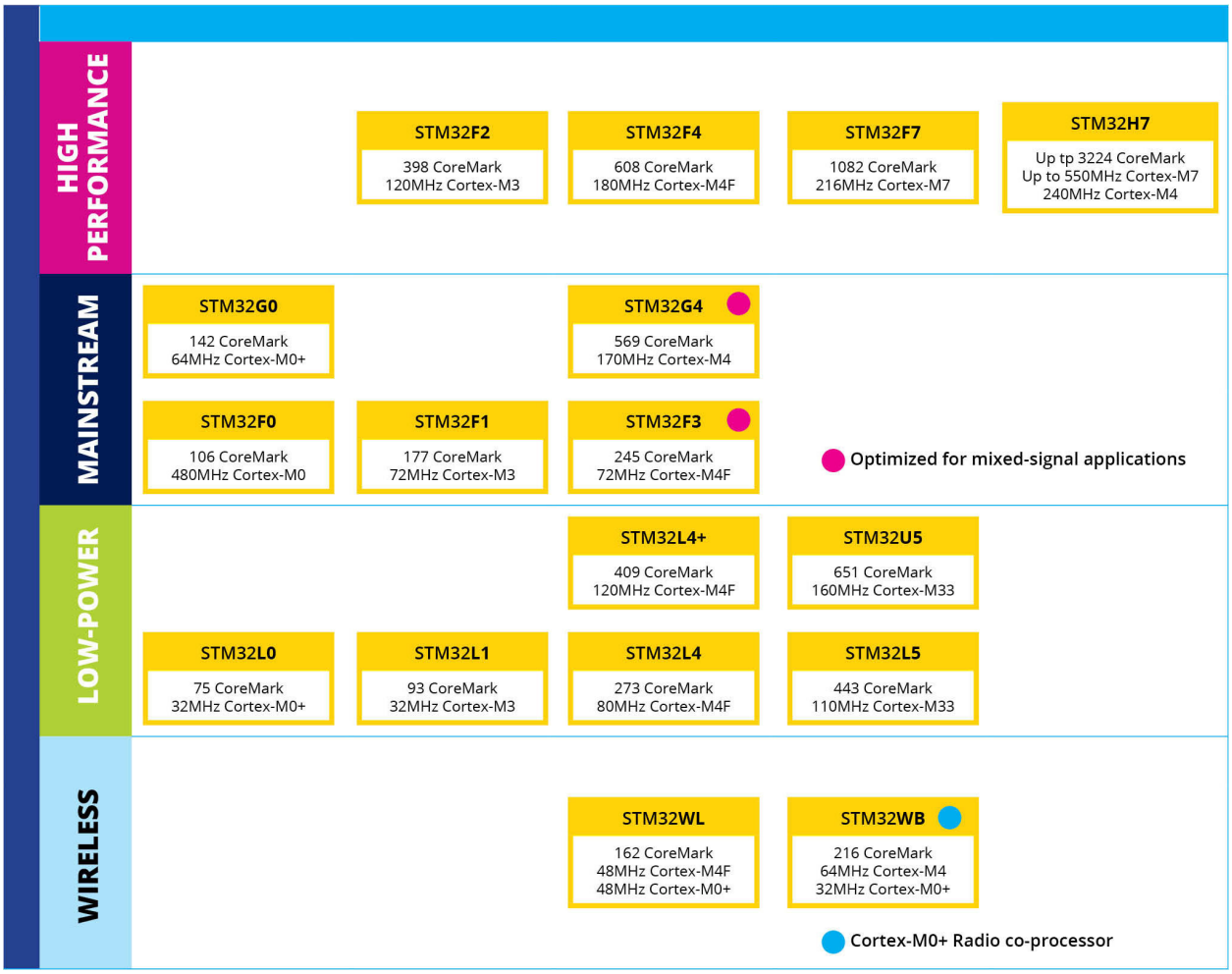


Figure 1.13: STM32 portfolio

Mainstream MCUs are developed for cost-sensitive applications, where the cost of the MCU must be even less than 1$/pc and space is a strong constraint. In this group we can find Cortex-M0/0+/3/4 based MCUs, with maximum clock frequencies ranging from 48MHz (F0) to over 170MHz (G4).

The *Ultra Low-Power* group contains those STM32 families of MCUs addressing low-power applications, used in battery-powered devices which need to reduce total power consumption to low levels ensuring longer battery life. In this group we can find both Cortex-M0+ based MCUs, for cost-sensitive applications, and Cortex-M4F based microcontrollers with *Dynamic Voltage Scaling* (DVS), a technology which allows to optimize the internal CPU voltage according to its frequency. Moreover, ST recently introduced in this category also MCUs with the new Cortex-M33 core, dedicated to application where security is a strong constraint. MCUs from this group range from 32MHz Cortex-M0+ up to 160MHz Cortex-M33.

Wireless MCUs are the brand-new lineup of dual-core STM32 microcontrollers dedicated to wireless connectivity. They cover Sub-GHz as well as 2.4GHz frequency range operation. They are easy to use, reliable and perfectly tailored for a wide range of industrial and consumer applications.These

MCUs feature a Cortex-M0+ core (named *Network Processor*) dedicated to the radio management and a user-programmable Cortex-M4 core (named *Application Processor*) for the main embedded application. STM32Wx solutions are compatible with multiple protocols, from point-to-point & mesh to wide-area networks.

The following paragraphs give a brief description of each STM32 family, introducing its main features. The most important ones will be summarized inside tables. Tables were arranged by the author of this book, inspired by the official ST documentation.

1.3.1 F0

MAINSTREAM Common to all STM32F0 • Reset POR/PDR • 2xWDT • Hardware CRC • Crystal oscillators • PLL • RTC calendar/clock • 16/32-bit timers • 1x12-bit ADC • Up to 12-channel DMA • Temperature sensor • Multiple channel DMA • SWD • Unique ID	STM32 F0	**Core:** Cortex-M0 **Instruction set:** *Thumb* subset, *Thumb-2* subset **Internal RC oscillators:** HSI=8MHz , LSI=40KHz **External clocks:** HSE=4 - 32MHz, LSE=32.768 - 1000 KHz **Maximum Core Frequency:** 48MHz **Low power modes:** Sleep, Stop and Standby **Year of commercialization:** 2012 **Available Packages:** LQFP(32,48,64,100), TSSOP20, UFBGA(64,100), UFQFPN(28,32,48), WLCSP(25,36,49,64)								
Product Line	**FLASH (KB)**	**RAM (KB)**	**Operating Voltage**	**Backup Memory**	**DAC**	**Touch Sense**	**Up to 2xSPI/I²S, 2xI²C**	**USART**	**CAN**	**USB 2.0**
STM32F0x0 Value Line	16 to 256	4 to 32	2.4 to 3.6 V				•	6		•
STM32F0x1 Access Line	16 to 256	4 to 32	2.0 to 3.6 V	•	•	•	•	8	•	
STM32F0x2 USB Line	16 to 128	4 to 32	2.0 to 3.6 V	•	•	•	•	8	•	• (crystal-less)
STM32F0x8 Low-Voltage Line	32 to 256	4 to 32	1.8 V ±8%	•	•	•	•	8	•	• (crystal-less)

Table 1.4: STM32F0 features

The STM32F0 series is the most cost-effective line of MCU from the STM32 portfolio. It is designed to have a street price able to compete with some 8/16-bit MCUs from other vendors, offering a more advanced and powerful platform.
The most important features of this series are:

- **Core**:
  - ARM Cortex-M0 core at a maximum clock rate of 48 MHz.
  - Cortex-M0 options include the SysTick Timer.
- **Memory**:
  - Static RAM from 4 to 32 KB.
  - Flash from 16 to 256 KB.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each F0-series device features a range of peripherals which vary from line to line (see **Table 1.4** for a quick overview).

- Oscillator source consists of internal RC (8 MHz, 40 kHz), optional external HSE (4 to 32 MHz), LSE (32.768 to 1000 kHz).
- IC packages: LQFP, TSSOP20[15], UFBGA, UFQFPN, WLCSP (see **Table 1.4** for more about this).
- Operating voltage range is 2.0V to 3.6V with the possibility to go down to 1.8V ±8%.

The rest of the chapter is not available in the book sample

[15] F0/G0/L0 are the only STM32 families that provides this convenient package.

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.

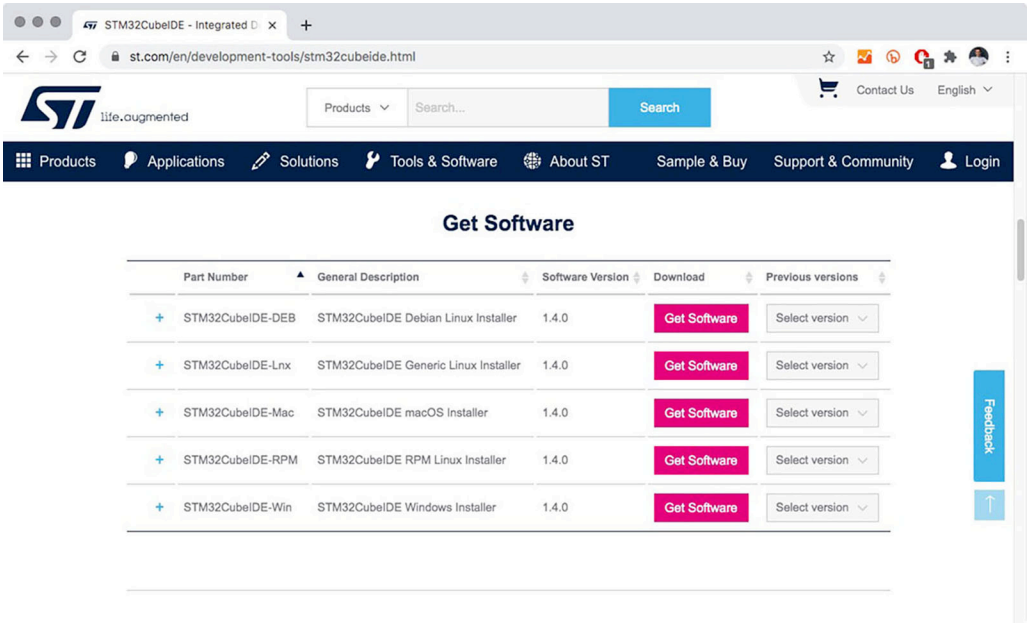


Figure 2.1: The STM32CubeIDE download page on the official STM website

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.

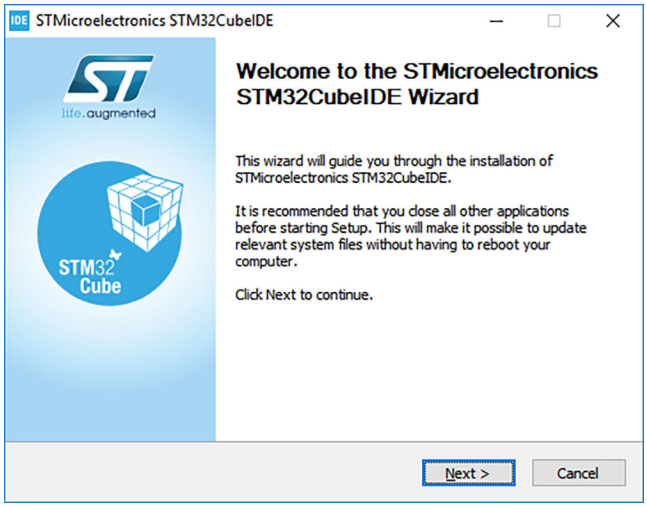


Figure 2.2: Windows installer welcome page

After few seconds the "*Welcome to...*" page of the installer appears, as shown in **Figure 2.2**. Click on "Next", read the license agreement and click on "I Agree" to accept the terms of the agreement.
In the next dialog (see **Figure 2.3**), it is possible to select the location for the installation. It is recommended to choose a short path to avoid facing Windows limitations with too long paths for the workspace. My suggestion is to leave the default path (`C:\ST\STM32CubeIDE`).

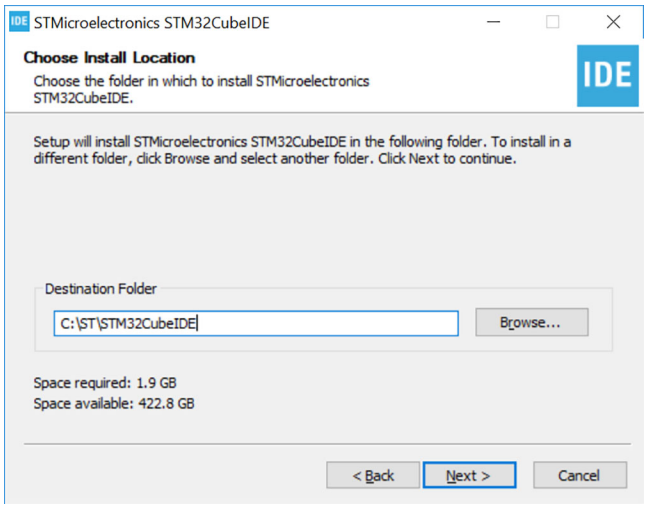


Figure 2.3: *Chose Install Location* dialog

Individual Projects will not be stored inside that path. Instead, they are grouped inside a preferred location named "Workspace". This is the common Eclipse's way to store projects, and the good news is that it is possible to have as many workspaces as you want: each workspace represents a totally independent environment, both from the stored projects point-of-view and the IDE configurations. This means that you can customize each workspace according to your specific needs. Every project, every installed plug-in, every IDE and tools configurations will be local to that specific workspace. This is also a lot useful for beginners: it is not that uncommon to mess with the IDE configuration and to break some relevant configurations. If this the case, you simply need to throw away the current workspace and to make a new one, without affecting the overall system configuration.

Once the installation path is chosen, click on "Next". The "Choose Components" dialog is displayed as shown in **Figure 2.4**. Unless you are confident with those flags, my suggestion is to leave all of them checked. The role of those components will be clearer as we progress through the book.
Click on the "Install" button and wait for the completion of the operations. In this step, the installer will copy in the selected location the IDE and all relevant components: a Java virtual machine, Eclipse and all it is plug-ins, GCC compiler and debuggers, Windows drivers for ST-LINK debuggers.

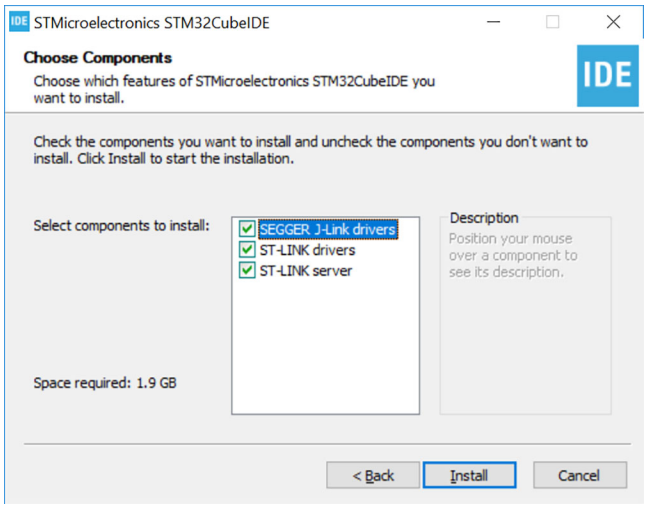


Figure 2.4: *Choose Components* dialog

At the end of the installation step, click on "Finish".

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[7] (the download link is at the bottom of the page in the "**Get Software**" section). Once the download is completed, extract the

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

`.zip` package. You will find the `SetupSTM32CubeProgrammer-2.9.0.exe` file. Run it and follow the installation instructions.

The tool-chain installation is now complete, and you can jump to the STM32CubeIDE overview paragraph if you are totally new to the Eclipse IDE.

2.2.2 Linux - Installing the Tool-Chain

The whole installation procedure will assume these requirements:

- A PC running a recent Linux-64bit version:
  - Ubuntu Linux 20.04 LTS Desktop or later
  - Fedora 29 or later
- Sufficient hardware resources (I suggest having at least 4Gb of RAM and 20Gb of free space on the Hard Disk); the instructions should be easily arranged for other Linux distributions.

The installer comes in different bundles to suit the various Linux distributions. The bundles are named `st-stm32cubeide_VERSION_ARCHITECTURE.PACKAGE` where:

- `VERSION` is the actual product version and build date (for example: 1.0.0_2026_20190221_1309)
- `ARCHITECTURE` is the architecture of the target host computer to run STM32CubeIDE (for example: amd64)
- PACKAGE is the Linux package type to be installed. The supported packages are:
  - `rpm_bundle.sh` for Fedora/CentOS
  - `deb_bundle.sh` for Ubuntu/Debian
  - `.sh` for generic Linux

Proceed as follows:

1. Navigate to the location of the installer file with a command console on the host computer. **2**. Enter the following command in the console window:

```
1 $ sudo sh ./st-stm32cubeide_VERSION_ARCHITECHURE.PACKAGE
```

where `VERSION`, `ARCHITECTURE` and `PACKAGE` must be entered after the selected Linux package.

3. Follow the further instructions provided through the console window.

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.

To execute the STM32CubeProgrammer in Linux, it is required you have some packages already installed on your machine. The needed packages are:

- `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**.

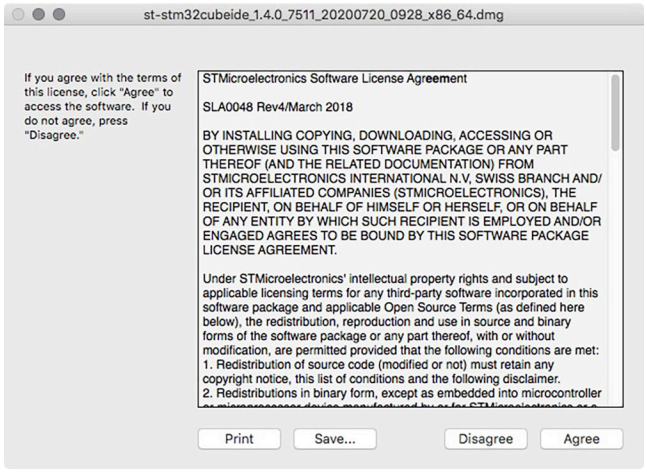


Figure 2.5: ***License Agreement*** **dialog**

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.

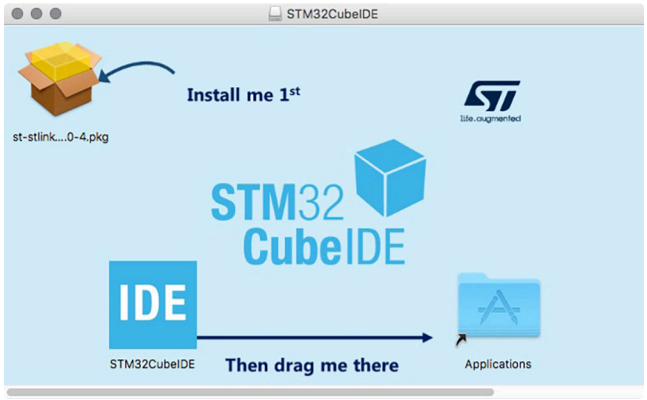
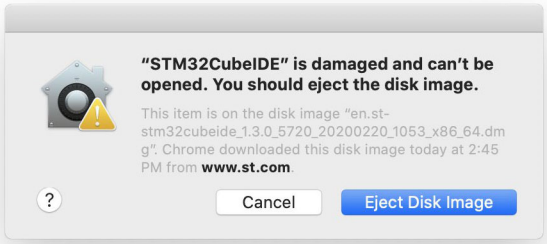


Figure 2.6: ***Install page*** dialog

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.

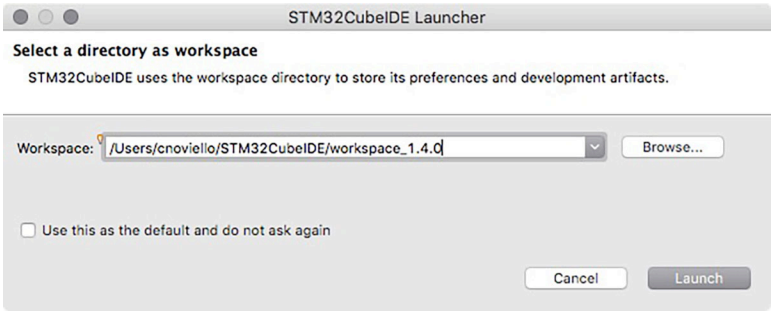


Figure 2.7: Eclipse workspace selection dialog

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.

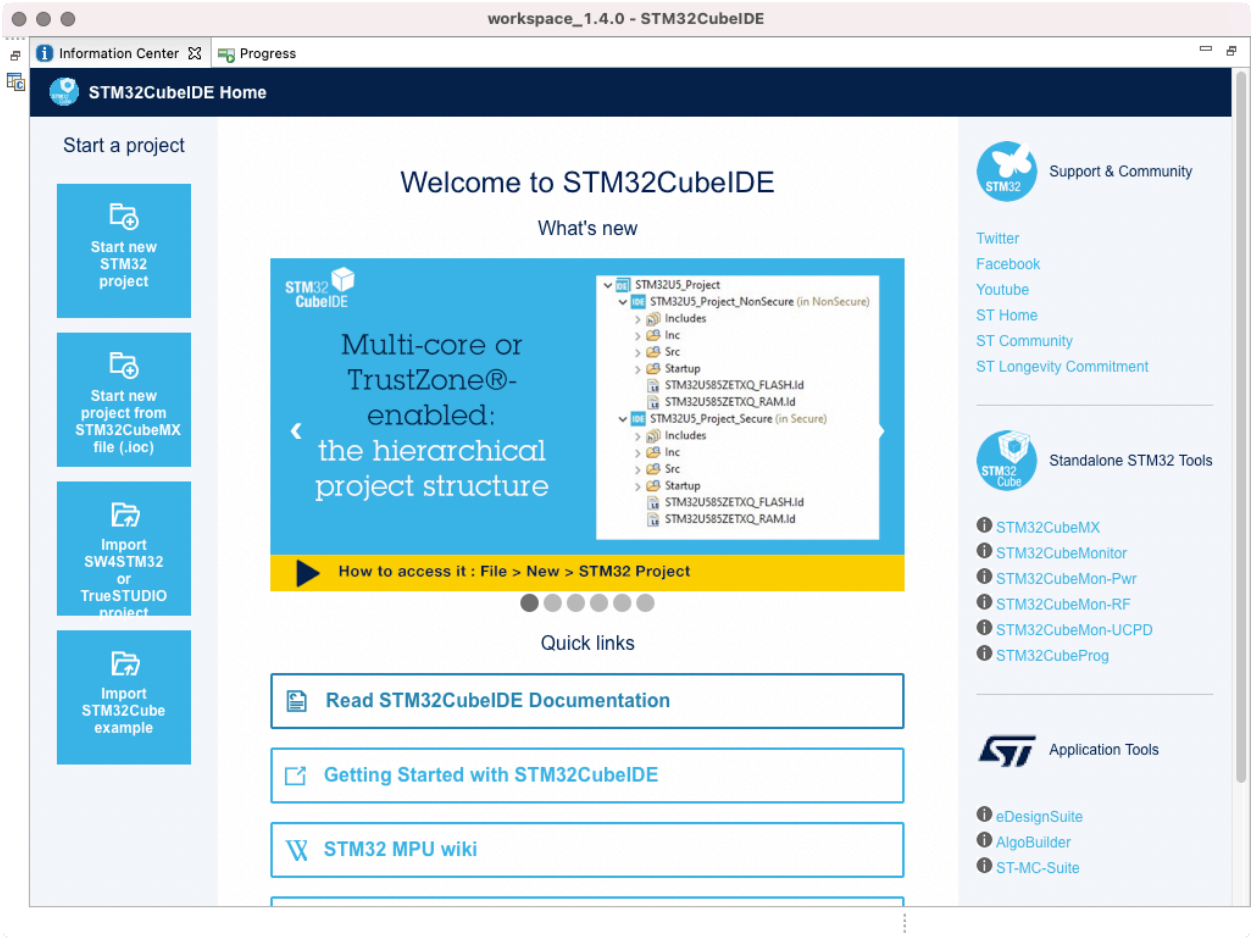


Figure 2.8: The Eclipse interface once started for the first time

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*.

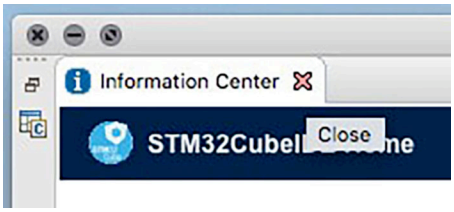


Figure 2.9: How to close the *Welcome view* by clicking on the *X*.

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**.

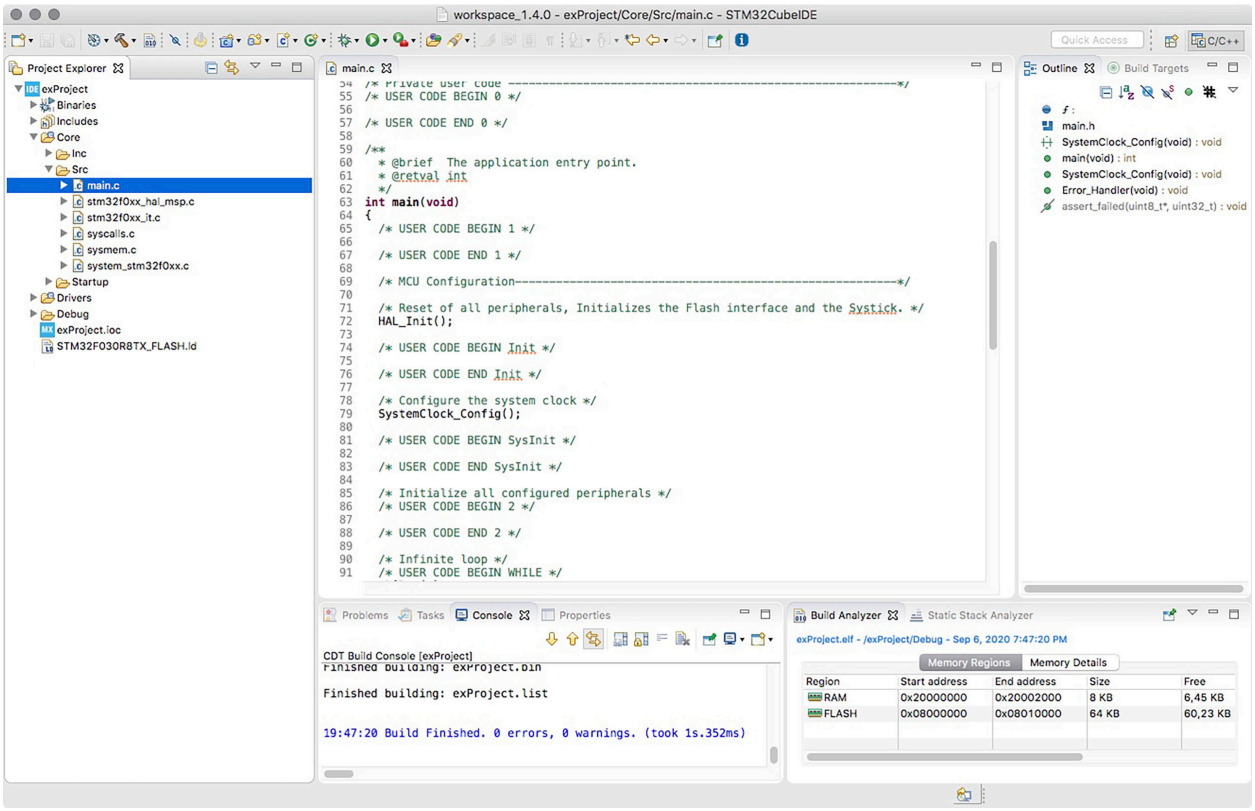
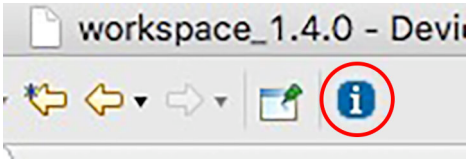


Figure 2.10: The C/C++ perspective view in eclipse (with a main.c file loaded later)

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.

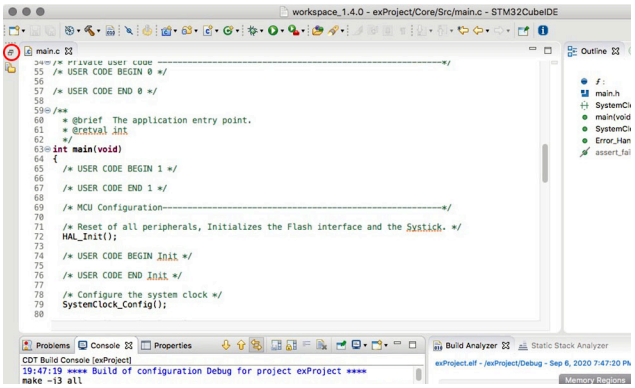


Figure 2.11: *Project Explorer* view minimized

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

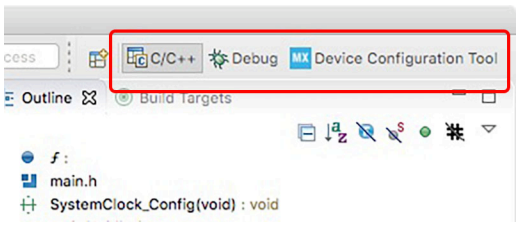
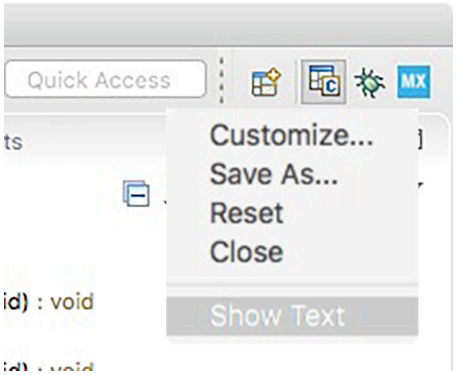


Figure 2.12: Perspective switcher toolbar

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.

The rest of the chapter is not available in the book sample

3. Hello, Nucleo!

No programming book is complete without the classic "Hello, world!" example, and this book is no exception. In the previous chapter, we set up STM32CubeIDE for developing STM32 embedded applications. Now, we are ready to dive into coding.

In this chapter, we will create a simple program: a blinking LED. This is a foundational exercise in embedded programming, and we will use STM32CubeIDE to build a complete application in just a few steps. For now, we will avoid discussing deeper topics such as the ST *Hardware Abstraction Layer* (HAL) and the MCU graphical configurator (commonly known as STM32CubeMX). The focus is on quickly getting hands-on experience with the development environment.

I understand that some of the details in this chapter may not be entirely clear, especially if you are new to embedded programming. However, this example will help you become familiar with the workflow in STM32CubeIDE. In later chapters, particularly the next one, many of these initial concepts will become clearer. For now, I encourage you to be patient and absorb as much as you can from the following sections.

3.1 Create a Project

Let us begin by creating our first project: a simple application that makes the Nucleo's LD2 LED (the green one) blink.
Start by navigating to **File->New->STM32 Project**. After a few moments, depending on your current IDE setup, the **Target Selection** wizard will appear (as shown in **Figure 3.1**).

If this is your first time launching the **Target Selection** wizard, it may take a few seconds to load. During this time, a progress indicator will show as the IDE retrieves MCU and board specifications from STMicroelectronics' servers. Since new STM32 microcontrollers and development boards are frequently released, the tool dynamically updates its database rather than being preloaded with all available part numbers. Be patient while the software completes this process.

As we will cover in more detail later, the **Target Selection** wizard is a component of a larger toolset, formerly known as STM32CubeMX. This tool streamlines the process of configuring hardware peripherals for STM32 microcontrollers and generates the necessary libraries to control them. We will delve deeper into STM32CubeMX's key features in the next chapter, so for now, we will keep the focus on setting up our project.

Next, switch to the **Board Selector** tab (highlighted in light blue in **Figure 3.1**). Expand the **Type** dropdown menu and choose the **Nucleo-64** family of boards. From the resulting list, locate and select your specific Nucleo-64 board model to proceed.

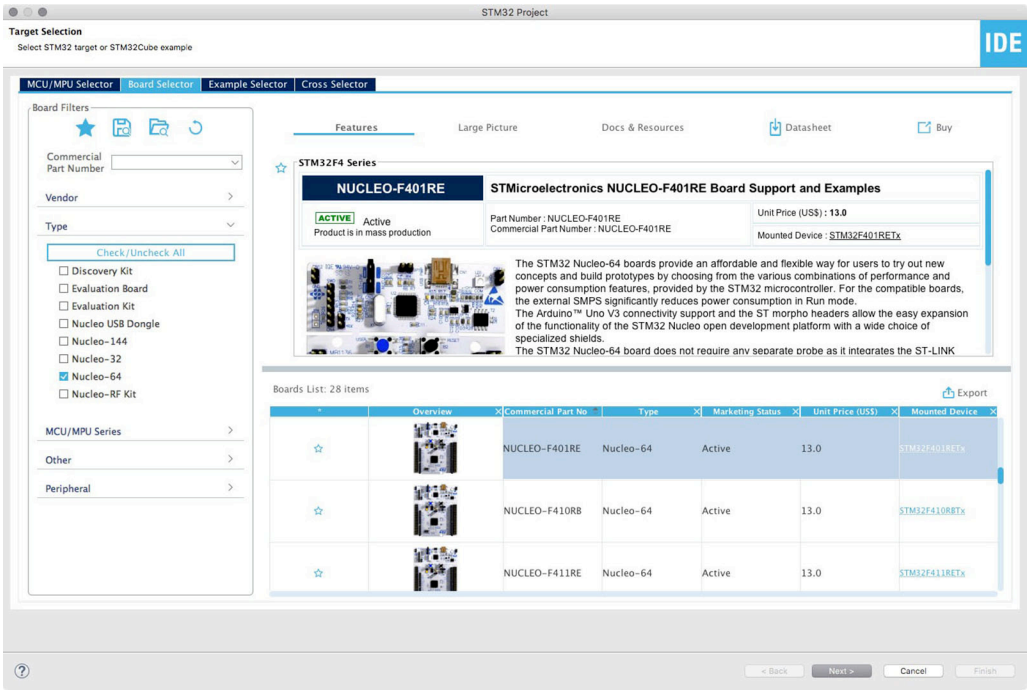


Figure 3.1: The Target Selection wizard - STEP 1

As discussed in Chapter 1, this book is entirely based on the Nucleo-64 board. All examples presented throughout the text have been specifically tested on the boards listed in **Table 1.18**. However, the primary aim of this book is to teach the core concepts of STM32 programming. In my opinion, it should be fairly straightforward to adapt the examples to other development boards, such as the Nucleo-32, Nucleo-144, Discovery, or similar platforms.

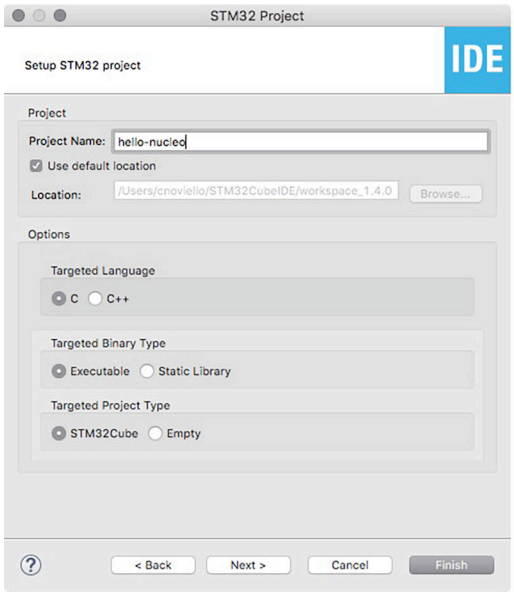


Figure 3.2: Project wizard - STEP 2

Once you have selected your board from the list, click the **Next** button. In the following window, enter a name for your project in the **Project Name** field (for example, *hello-nucleo*, although you can choose any name that fits your project). Leave all other options at their default settings, as shown in **Figure 3.1**, and click **Finish** to begin generating the project.

At this stage, STM32CubeIDE will prompt you to initialize all peripherals in their default mode, as shown in **Figure 3.3**. But what exactly does this mean?

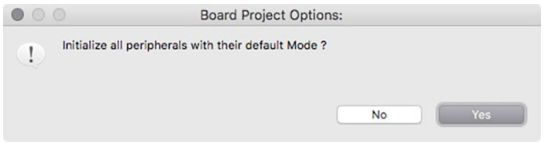


Figure 3.3: Project wizard - STEP 3

Every development board features a target MCU (the microcontroller where the firmware is uploaded), which includes internal peripherals like the *Real Time Clock* (RTC) and is connected to various external devices via individual GPIO pins. For instance, in almost all of Nucleo-64 boards, one GPIO pin is connected to the green USER LED, labeled LD2 on the PCB. To utilize these "default" peripherals, both the internal peripherals and the associated GPIOs need to be correctly configured. When you select **Yes** in response to the prompt, you are instructing the IDE to automatically configure all necessary internal and external peripherals for your Nucleo-64 board. This simplifies the process by setting up everything required to get started. While later in the book you will learn how to manually configure each peripheral, for now, to keep things simple, click **Yes** and allow the IDE to handle the configuration.

After confirming, STM32CubeIDE will begin generating your project. If this is your first time creating a project for your specific STM32 MCU family (e.g., STM32F0, STM32F4, etc.), the IDE may need to download the corresponding *Cube Firmware Package*. For example, if you are using

a Nucleo-F401RE board, the IDE will download the `stm32cube_fw_f4_v1XXX.zip` package for the STM32F4 family. These firmware packages contain several important components:

- **Complete HAL for the STM32 family**: the *Hardware Abstraction Layer* (HAL) is a set of libraries that enable you to control the microcontroller's peripherals and core features without dealing with the MCU's low-level details. This book is built around the STM32CubeHAL, and you will develop a strong understanding of this powerful, albeit complex, library.
- **Additional Middleware packages**: some STM32 MCUs include advanced peripherals that require additional libraries, provided either by ST or third-party developers. For instance, programming the USB controller on certain STM32 MCUs requires using a complete USB stack, which is freely available from ST. Each Cube Firmware package includes a set of middleware libraries, and we will explore several of these in later chapters.
- **Examples projects for development boards**: ST offers a wide range of example projects for its development boards. These projects demonstrate how to use specific features of the board and can serve as valuable references for your own work. To browse these examples, click on the **Example Selector** tab in the **Target Selection** wizard (as shown in **Figure 3.1**). They can be an excellent starting point for your development.

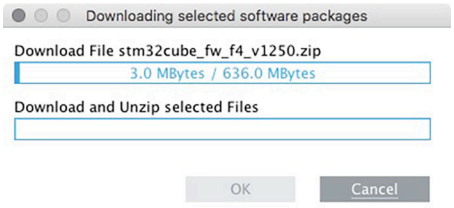


Figure 3.4: Cube Firmware Package download dialog

The size of a Cube Firmware package can vary significantly depending on the STM32 family. As a general rule, the more powerful the STM32 family, the larger the corresponding Cube Firmware package will be. This is because more advanced MCUs come with additional features, peripherals, and libraries, all of which require greater support in the firmware package. Consequently, the download process may take some time. Be patient and let the download complete, as shown by the progress dialog (see **Figure 3.4**).

3.2 Adding Something Useful to the Generated Code

Figure 3.5 displays the STM32CubeIDE interface after generating the project. The Project Explorer view highlights the project structure, and below is a brief overview of the top-level folders, listed from top to bottom[1]:

[1]At this stage, we are providing only a high-level introduction to the generated files and folders. A more detailed analysis will follow in subsequent chapters, so there is no need to delve too deeply into them now.

`Includes`
: This folder contains all directories associated with GCC Include Folders[2]. These include paths are necessary for the compiler to locate the header files required for the project.

`Core`
: This folder, part of the Eclipse project structure, houses the core files of the application, including multiple `.c` source files. Among these, `src/main.c` holds the `int main(void)` function, which we will soon modify to customize the behavior of our application.

`Drivers`
: This folder typically contains the header and source files for essential libraries, such as the ST CubeHAL and CMSIS packages. We will explore these libraries in greater detail in the next chapter.

`hello-nucleo.ioc`
: This file represents the STM32CubeMX project, which is visually displayed in the **Device Configuration Tool** view (the default main view after project generation, as shown in **Figure 3.5**). The STM32CubeMX interface and its functionalities will be thoroughly examined in next chapter.

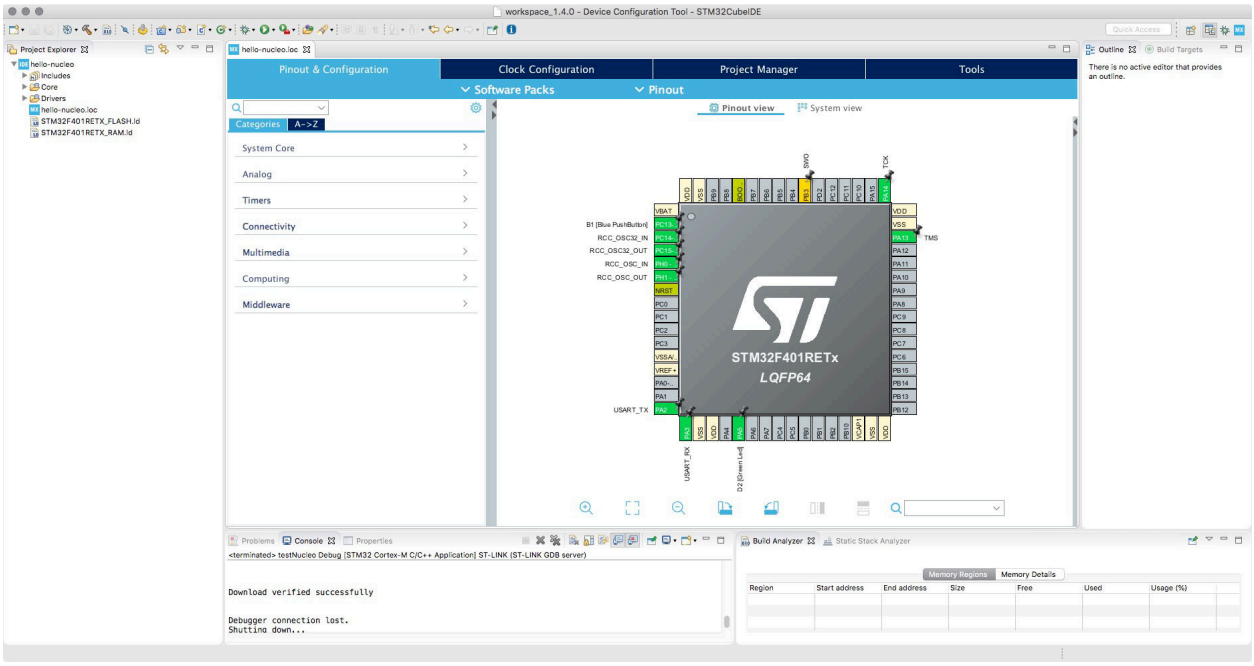


Figure 3.5: The STM32CubeIDE interface after complete project generation

We are now ready to begin working on the core of our application. The project generated by the IDE already includes all the necessary code to create a functional application. To make the `LD2` LED

[2]In C/C++ projects, the compiler needs to know where to find header files (with the extension `.h`). These locations are known as *include paths* and must be provided to the GCC compiler using the `-I` option. Fortunately, Eclipse automates this process by managing the include paths for us, and the **Includes** folder visually displays the directories where GCC will search for header files.

blink, we need to add just two lines of code to the `main()` function, which serves as the entry point[3] for our custom application.
In the **Project Explorer** pane, navigate to **Core->Src** and double-click on the `main.c` file. Scroll to approximately line 66, where the `main()` function is defined. Here, you will see the invocation of four key routines[4]:

- `HAL_Init()`: this function initializes the CubeHAL framework, setting up the microcontroller for basic operation. We will delve into the details of this function later in the book.
- `SystemClock_Config()`: this function configures the microcontroller's clock settings. STM32 MCUs offer various clock sources, and this function selects one for the MCU to use. This is a relevant topic, which will be explored in depth in Chapter 10.
- `MX_GPIO_Init()`: this function initializes the GPIO pins based on the configuration defined in STM32CubeMX. In this case, it configures the GPIO pin connected to the `LD2` LED. We will cover GPIO in more detail in Chapter 6.
- `MX_USART2_UART_Init()`: initializes the UART2 peripheral, which is connected to the ST-LINK interface on all Nucleo boards. More on UART configuration can be found in Chapter 8.

Following these initialization routines, you will find an infinite `while` loop. This loop is where the ongoing operations of the firmware are executed. To make the LED blink, we will add two function calls just after this loop begins, around 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
```

[3]While experienced firmware developers know that the `main()` function is not the true entry point of an STM32 application (the firmware execution starts much earlier with setup routines that establish the execution environment), from an application perspective, the process begins inside the `main()` function. The STM32 boot process will be covered in detail in a later chapter.

[4]If you are using a development board other than a Nucleo-64, you might see additional routines invoked within the `main()` function. This is due to the presence of extra peripherals on your board. When STM32CubeIDE asked if you wanted to initialize all peripherals in their default modes, we selected "**Yes**". Therefore, do not worry if your `main()` function differs slightly from the one shown here.

```
84    /* USER CODE BEGIN SysInit */
85
86    /* USER CODE END SysInit */
87
88    /* Initialize all configured peripherals */
89    MX_GPIO_Init();
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 may have noticed that the code generated by CubeMX includes several commented regions like this:

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

What is the purpose of these comments? CubeMX is designed so that if you make changes to the hardware configuration and regenerate the project code, the custom code you have added will not be overwritten. By placing your code inside these "guarded regions," CubeMX ensures that your modifications are preserved during code regeneration.

However, it is important to mention that CubeMX is not always foolproof. Occasionally, it can mishandle the generated files, causing user-added code to be lost. To avoid this, I recommend generating a separate project whenever you make significant changes and then copying and pasting your modified code into the appropriate sections. This approach gives you complete control over your code and ensures nothing gets unintentionally overwritten.

The code is straightforward. The `HAL_GPIO_TogglePin()` function toggles the logical state of the pin connected to the `LD2` LED (which corresponds to PIN 5 of GPIO port A on all Nucleo-64 boards). The `HAL_Delay()` function creates a delay of 500 milliseconds using a busy-wait loop. As a result, the `LD2` LED will blink at a rate of 1Hz, meaning it will toggle on and off every second.

How do we know which pin the LED is connected to? ST provides schematics[5] for the Nucleo board, which can be found here. These schematics are originally created using *Altium Designer*, a high-end CAD tool commonly used in professional settings. Fortunately for us, ST also provides a convenient PDF version of the schematics. By examining page 4, we can see that the `LD2` LED is connected to the PA5 pin[6], as illustrated in **Figure 3.6**.

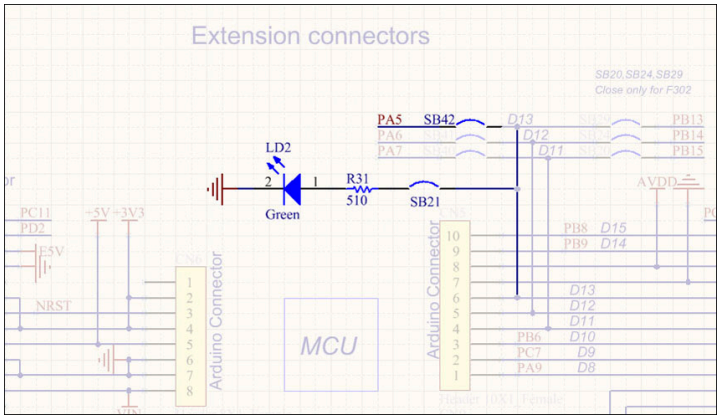


Figure 3.6: `LD2` connection to PA5

PA5 stands for PIN5 of the GPIOA port, which is the standard way to reference a GPIO pin in STM32 nomenclature. Moreover, STM32CubeMX automatically generates macros `LD2_GPIO_Port` and `LD2_Pin`, that resolve to the GPIOA port and pin 5, respectively.

We are now ready to compile the project. To do so, navigate to **Project->Build Project** from the menu. After a short while, the output console should display something similar to the following[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 your output. This variation is due to differences in HAL implementations across different STM32 series and the compiler's optimization settings. Don't worry about these details for now — they will be explained more thoroughly in later chapters.

3.3 Connecting the Nucleo to the PC

Once the test project has been successfully compiled, you can connect the Nucleo board to your computer using a USB cable. Connect the cable to the micro-USB port, labeled **VCP** in **Figure 3.7**. Upon connection, you should see at least two LEDs illuminate.

Read Carefully

Ensure that the USB port you are using can supply sufficient power to the board. It is highly recommended to use a USB port capable of providing at least 500mA or a self-powered external hub to avoid power issues.

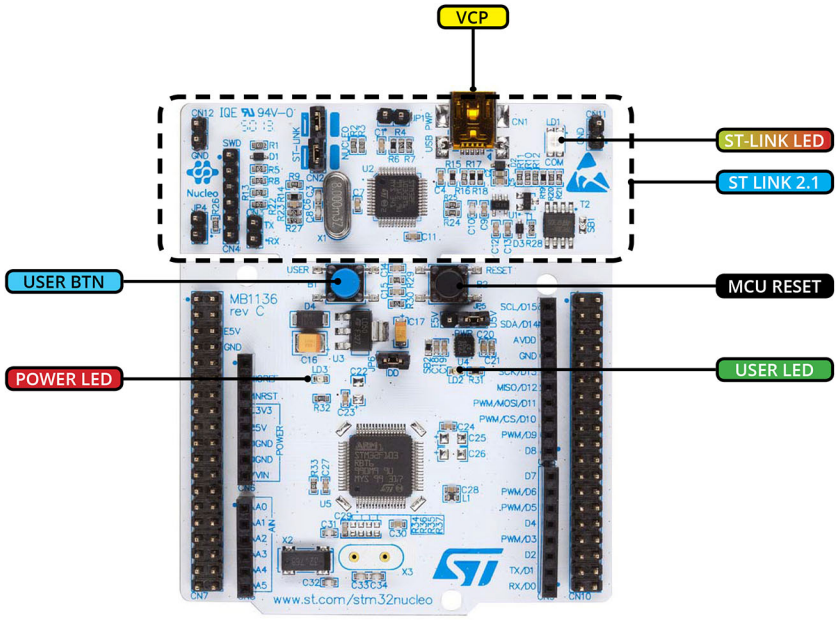


Figure 3.7: A Nucleo board and its main interfaces

The first LED you will notice is `LD1`, labeled ST-LINK LED in **Figure 3.7**. This is a red/green LED that indicates ST-LINK activity. When the board is connected to the computer, the LED glows green. During a debugging session or when uploading firmware to the MCU, it alternates between green and red, signaling activity.

Another LED that turns ON when the board is connected is `LD3`, labeled **POWER LED** in **Figure 3.7**. This red LED indicates that USB *enumeration* has been completed, meaning the ST-LINK interface has been recognized by the computer's operating system as a USB peripheral. The target MCU on the board is powered only when this LED is on, signifying that the ST-LINK interface also controls power to the MCU.

If you have not yet flashed your board with custom firmware, you will also see the `LD2` LED, a green LED labeled USER LED in Figure 3.7, blinking. This happens because ST preloads the board

with firmware that causes `LD2` to blink. You can change the blinking frequency by pressing the blue switch labeled **USER BTN** in **Figure 3.7**.

We are going to replace the preloaded firmware with the custom firmware we have created. However, before proceeding, it is important to ensure that the ST-LINK debugger on your Nucleo board is equipped with the latest ST-LINK 2.1 firmware.

3.3.1 ST-LINK Firmware Upgrade

Warning

Read this section carefully. Do not skip this step!

Based on my experience with multiple Nucleo boards, most of them come preloaded with an outdated ST-LINK firmware. To ensure compatibility with the latest STM32CubeIDE, you must update the ST-LINK firmware to the leastest version.

Updating the firmware is a simple process using STM32CubeIDE. First, connect your Nucleo board via a USB cable. Then, navigate to **Help->ST-LINK Upgrade**. This will launch the **ST-LINK Upgrade** tool, as shown in **Figure 3.8**.

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**

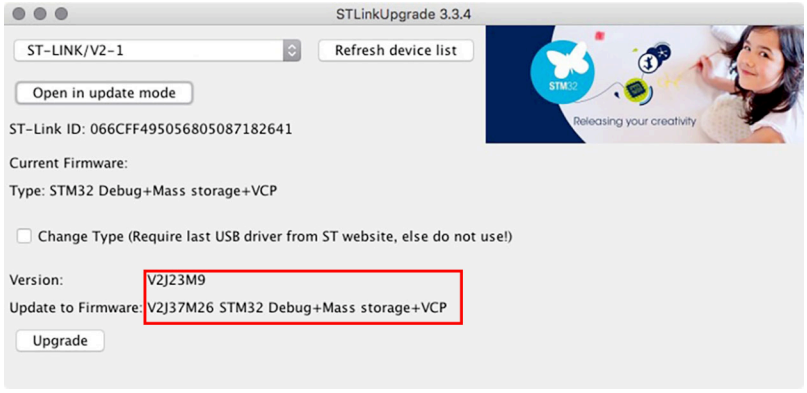


Figure 3.8: The ST-LINK Upgrade program

Click on **Refresh device list**: your connected Nucleo board should appear as ST-LINK/V2-1 or ST-LINK/V3. Next, click on **Open in update mode**. The **ST-LINK Upgrade** tool will then indicate whether your Nucleo board's firmware needs to be updated by displaying the current version, as shown in **Figure 3.8**. If an update is required, click on the **Upgrade** button and wait for the firmware to be updated.

Error in upgrading the ST-LINK firmware

The update process may sometimes fail when you click on the **Open in update mode** button. The **ST-LINK Upgrade** tool may show an error message like *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. This issue often arises due to insufficient power supply to the board. To resolve this, try using a self-powered USB hub or switching to a different USB port. This error is particularly common on recent iMacs when the board is connected through an Apple keyboard's integrated USB port.

3.4 Flashing the Nucleo using STM32CubeProgrammer

In Chapter 2, we installed STM32CubeProgrammer, and now it is time to use it. First, launch the program and connect your Nucleo board to the PC using a USB cable. Click the refresh button, highlighted in red in **Figure 3.9**. Once STM32CubeProgrammer detects the board, its serial number will appear in the **Serial number** field, as shown in the figure.

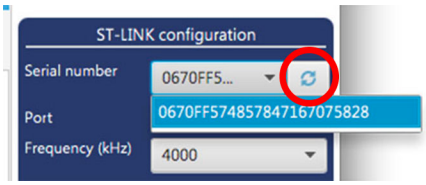


Figure 3.9: The ST-LINK interface serial number as shown by STM32CubeProgrammer tool

Read Carefully

If you see "*Old ST-LINK Firmware*" instead of the serial number, it means that the ST-LINK firmware needs to be updated. Click the **Firmware upgrade** button located at the bottom of the **ST-LINK Configuration** pane, and follow the instructions described in the previous section.

Once the ST-LINK interface has been correctly identified, click the **Connect** button. After a short moment, the contents of the flash memory will be displayed, as shown in **Figure 3.10**.

Now we are ready to upload the example firmware to the board. Click the **Erase & programming** icon (the second green icon on the left). In the **File programming** section, click **Browse**. Navigate to your Eclipse workspace directory (by default, `%HOMEPATH%\STM32CubeIDE\workspace_1.X.0` on Windows, or `~/STM32CubeIDE/workspace_1.X.0` on Linux and macOS, where `1.X.0` corresponds to your specific STM32CubeIDE version). Then, go to the `hello_nucleo\Debug` sub-folder and select the `hello_nucleo.elf` file.

Ensure that both the **Verify programming** and **Run after programming** options are checked, then click **Start Programming** to begin flashing the firmware. Once the process is complete, the green `LD2` LED on your Nucleo board will start blinking.

Congratulations: welcome to the STM32 world ;-)

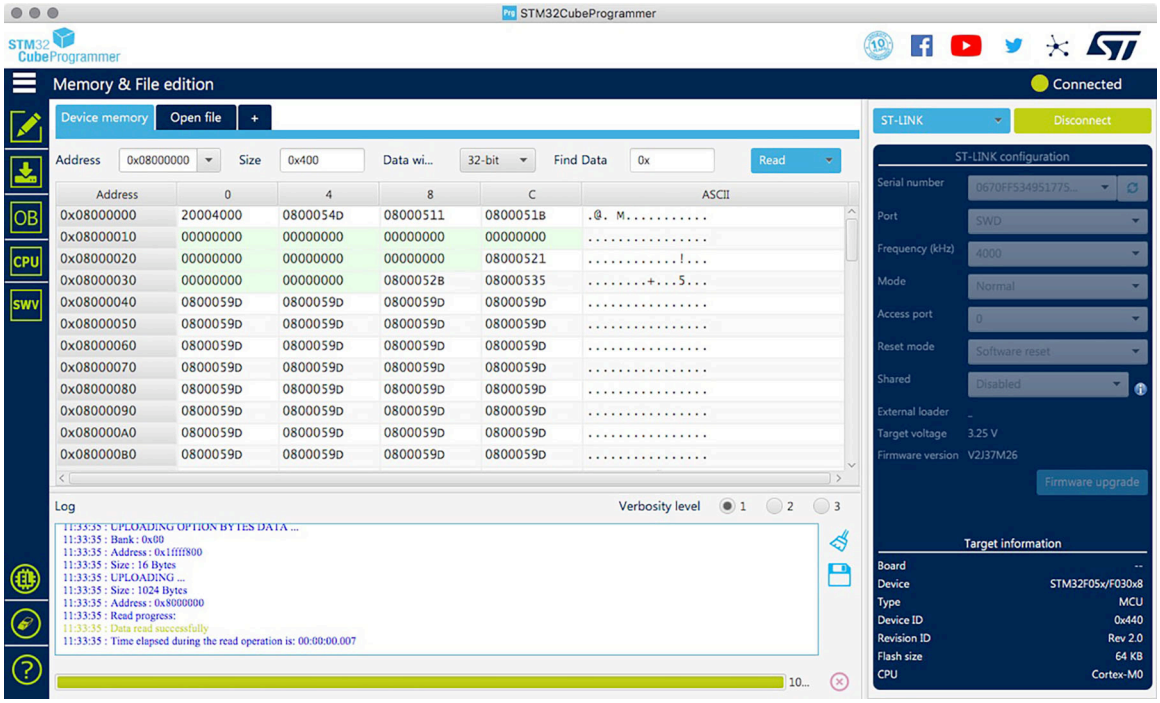
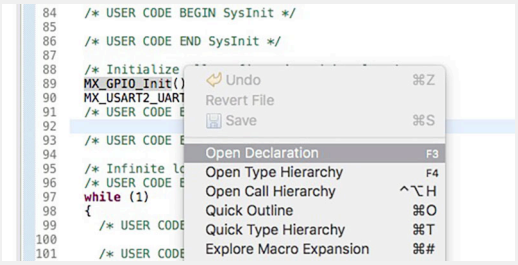


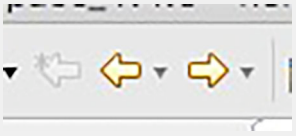
Figure 3.10: The STM32CubeProgrammer interface when connected to the Nucleo board

Eclipse intermezzo

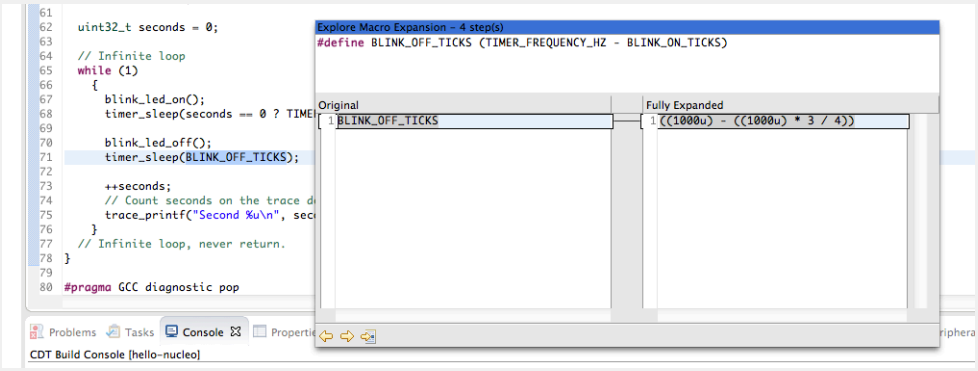
Eclipse provides convenient tools for navigating through source code, eliminating the need to manually jump between files to find function definitions. For instance, if you want to see how the `MX_GPIO_Init()` function is implemented, simply highlight the function call, right-click, and select **Open Declaration** from the menu, as shown in the image below.



Alternatively, you can hold down the `Ctrl` key (or `CMD⌘` on macOS) while clicking on the symbol to navigate to its definition. Additionally, Eclipse allows you to easily navigate through the opened source files during symbol exploration using the dedicated navigation buttons on the toolbar, as illustrated here:



Another useful feature in Eclipse is the ability to expand complex macros. To do this, right-click on the macro and select **Explore macro expansion**. A contextual window like the one below will appear, showing how the macro is expanded.



However, at times, Eclipse may run into issues with its indexing, making it difficult or impossible to navigate the source code. If this happens, you can force Eclipse to rebuild its index by going to **Project->C/C++ Index->Rebuild**.

The rest of the chapter is not available in the book sample

4. STM32CubeMX Tool

Gone are the days when configuring an 8-bit microcontroller peripheral could be accomplished with just a few assembly instructions. While there remains a dedicated group of developers who still write embedded software entirely in assembly[1], the most valuable resource in modern project development is time. Working with complex hardware platforms like the STM32 requires every bit of assistance to streamline the process. In modern 32-bit microcontrollers—especially those with a large number of I/O pins—even controlling something as basic as a GPIO can require sifting through dozens of pages in a thousand-page datasheet. Believe it or not, trying to initialize advanced peripherals like the `DCMI` or `ETH` interface on an STM32H7 without the support of the ST HAL can be incredibly frustrating. Moreover, fully understanding the available configuration options for a GPIO often demands a thorough grasp of all peripherals supported by that specific STM32 part number, including its pinout and configuration. Fortunately, ST provides a powerful and user-friendly tool that abstracts away many of the peripheral configuration complexities: STM32CubeMX.

STM32CubeMX[2] is the Swiss Army knife for every STM32 developer and is particularly indispensable for those new to the STM32 platform. It is a sophisticated software tool, freely provided by ST, and available both as a standalone application and as an integrated component within STM32CubeIDE.

In this chapter, we will explore how CubeMX works and how to generate fully functioning projects from scratch using its code generation capabilities. By doing so, we can produce efficient code that is ready for integration with the rest of the STM32Cube HAL. However, this chapter should not be considered a replacement for the official official ST documentation for CubeMX tool[3], a comprehensive guide exceeding 350 pages that covers its functionalities in depth.

4.1 Introduction to CubeMX Tool

CubeMX is the primary tool for configuring the microcontroller selected for our project. It is used not only to define the appropriate hardware connections but also to generate the code necessary for configuring the ST HAL (Hardware Abstraction Layer).

CubeMX is an *MCU-centric* application, meaning that all actions within the tool revolve around the following key aspects:

- **STM32 MCU family**: The specific family of the microcontroller (e.g., F0, F1, and so on).

[1] One day, someone may explain to these developers that—except for a few rare cases—modern compilers typically generate more optimized assembly code from C than could be written manually. However, these practices seem largely confined to ultra-low-cost 8-bit microcontrollers like the PIC12 and similar devices.

[2] Throughout the rest of this book, the name STM32CubeMX will be referred to simply as *CubeMX*.

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

- **Package type**: The physical package of the MCU (e.g., LQFP48, BGA144, etc.).
- **Hardware peripherals**: The peripherals required for the project (e.g., USART, SPI, etc.) and how these are mapped to the microcontroller pins. **General MCU configuration**: Settings such as clock configuration, power management, and NVIC (Nested Vectored Interrupt Controller) settings.

In addition to these hardware-specific features, CubeMX also handles several software-related aspects:

- **ST HAL management**: It manages the ST HAL specific to the selected MCU family (e.g., CubeF0, CubeF1, etc.).
- ***Middleware* configuration**: It allows the integration of additional software libraries, known as *Middleware*, which might be necessary to manage specific peripherals or complex software stacks (e.g., FatFs, LwIP, FreeRTOS, etc.).
- **Development environment configuration**: CubeMX helps set up the project for the development environment you will be using to build the firmware (e.g., IAR EWARM, Keil MDK-ARM, STM32CubeIDE)[4].

Project generation in CubeMX involves two key phases. The first phase is selecting the correct STM32 MCU or development board using the **Target Selection** wizard. The second phase is configuring the MCU and any required Middleware libraries to meet your project's needs. Let us explore these two phases in detail.

4.1.1 Target Selection Wizard

We first used CubeMX in Chapter 3[5] to generate the *hello-nucleo*, our initial STM32 project. As we saw, project generation begins with the **Target selection** view. The view is organized in a tabbed window, with four main tabs (see **Figure 4.1**): *MCU/MPU Selector*, *Board Selector*, *Example Selector* and *Cross Selector*. Let us take a closer look at each of these tabs and their functions.

[4]The standalone version of CubeMX can generate project code and configuration files for various commercial IDEs, not just the official STM32CubeIDE. However, this book will focus exclusively on using CubeMX within STM32CubeIDE.

[5]ch3-hello-nucleo-project

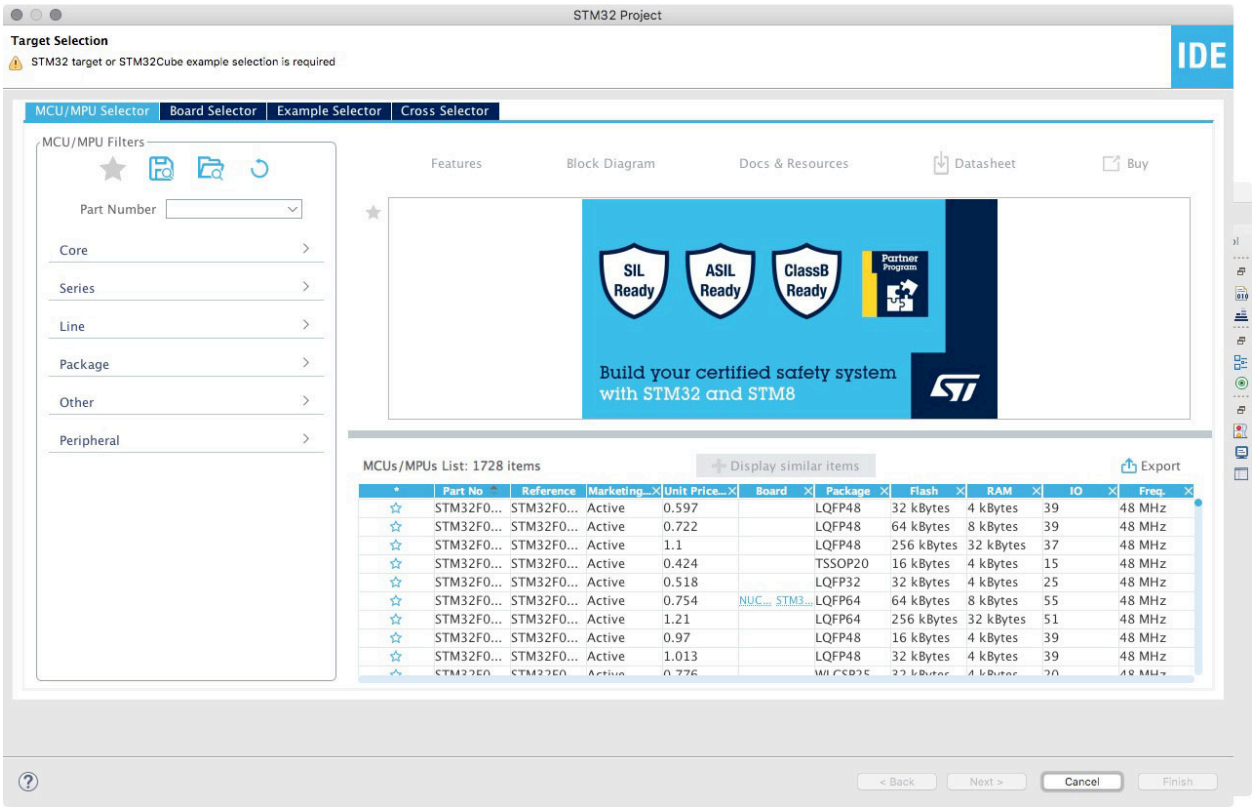


Figure 4.1: CubeMX *MCU/MPU Selector*

4.1.1.1 MCU/MPU Selector

The first tab, *MCU/MPU Selector*, allows users to choose a microcontroller from the entire STM32 portfolio. Several filters are available to help identify the right microcontroller for your application:

- **Core**: This filter narrows the selection to MCUs with specific Cortex-M cores (e.g., M0, M4, etc.).
- **Series**: Filters MCUs by STM32 series (e.g., F0, F4, etc.).
- **Line**: Further refines the selection by sub-family, such as the *Value Line*.
- **Package**: Filters MCUs based on the desired physical package (e.g., LQFP, WLCSP, etc.).
- **Other**: Offers additional filters based on budgetary price, the number of I/O pins, and the size of FLASH, SRAM, and EEPROM memories.
- **Peripheral**: Allows you to filter for MCUs with specific integrated peripherals.

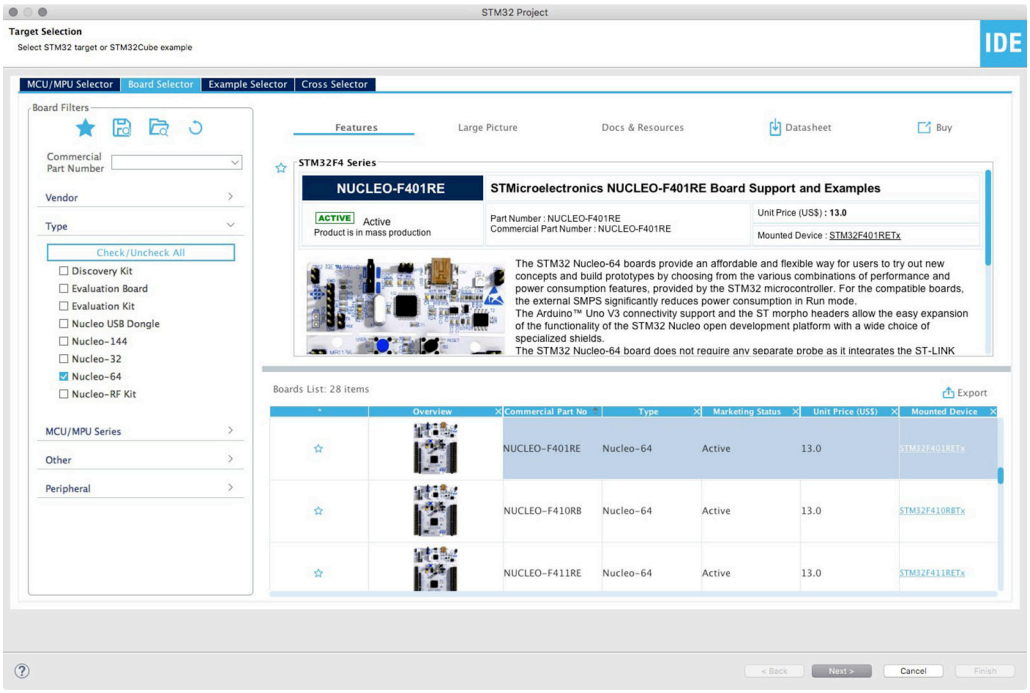


Figure 4.2: CubeMX *Board Selector*

4.1.1.2 Board Selector

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

- **Type**: Restricts the selection to boards from a specific family (e.g., Nucleo-64, Discovery, Evaluation Board, etc.).
- **MCU/MPU Series**: Filters boards based on the MCU series they use (e.g., F0, F4, etc.).
- **Other**: Offers two filters to limit the selection based on budgetary price or oscillator frequency (though the latter is not particularly useful, in the author's opinion).
- **Peripheral**: Filters boards according to the desired integrated peripherals.

4.1.1.3 Example Selector

Over the years, ST has developed thousands of examples demonstrating how to use individual peripherals or Middleware extensions in the STM32 lineup. The *Example Selector* tab allows users to filter through more than 5.000 examples (see **Figure 4.3**). Several filters help to identify the appropriate example:

- **Name**: Restricts the example list to those with a specific project name (each example is typically available for multiple MCUs and/or development boards).
- **Keyword**: Filters the examples list based on a given search keyword.

- **Board**: Selects examples for a specific target development board.
- **MCU/MPU**: Filters examples for a specific target MCU.
- **Project Type**: Allows selection between *Application*, *Demonstration*, and *Example* projects; though in my humble opinion, this distinction is somewhat arbitrary.
- **Based on driver**: Filters examples based on whether they were developed using CubeHAL, CubeHAL-LL, or a combination of both; more on this later.
- **Middleware**: Selects examples that showcase the use of a particular Middleware library.
- **MCU/MPU Library**: This filter can be somewhat confusing, as it is used to select examples demonstrating how to program specific peripherals.
- **Board Support Package Library**: Many STM32 development kits include additional peripherals, such as LCD displays, DCMI cameras, MEMS sensors, etc. To support these, ST provides useful *Board Support Package* (BSP) libraries. These free libraries are handy when you need to control a similar peripheral on a custom board. This filter allows selection of examples that use a specific BSP library.

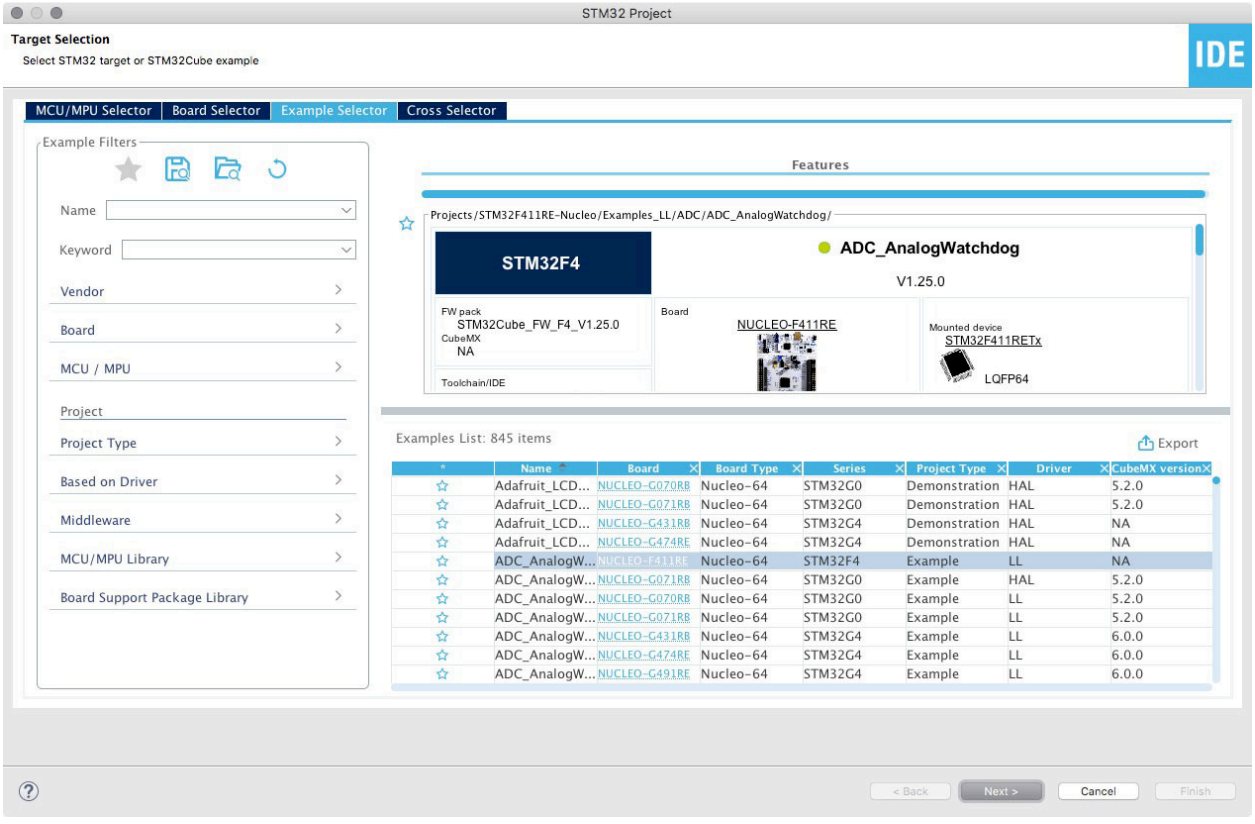


Figure 4.3: CubeMX *Example Selector*

4.1.1.4 Cross Selector

If you are used to working with microcontrollers from other suppliers (e.g., Microchip, Renesas) and are considering porting your design to an STM32 microcontroller, the *Cross Selector* tool can

help you identify a suitable alternative. However, in my opinion, this tool is especially effective when searching for a replacement within the STM32 lineup itself. This can be particularly useful if a specific STM32 MCU is unavailable in the market—an increasingly common issue these days.

The *Cross Selector* tool (as shown in **Figure 4.4**) provides a *match percentage*, indicating how closely the suggested alternative matches your current MCU. Nevertheless, always keep the datasheet on hand and carefully verify even secondary specifications, such as the physical behavior of GPIOs (e.g., voltage tolerance, slew rate, etc.).

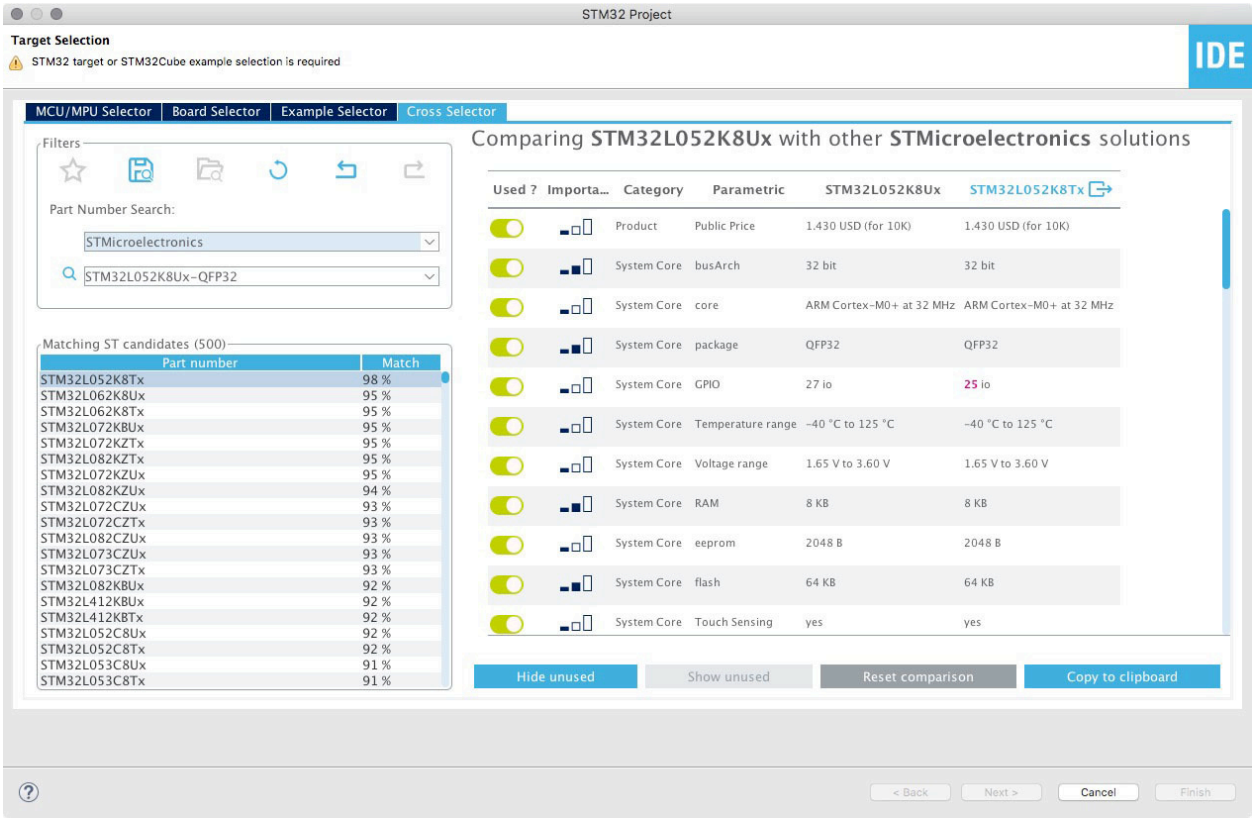


Figure 4.4: CubeMX *Cross Selector*

4.1.2 MCU and Middleware Configuration

Once the project has been generated, STM32CubeIDE automatically opens a file in the project folder named `<project-name>.ioc`. This file is the main CubeMX project file, containing all the configurations made within CubeMX. Based on this file, CubeMX generates the corresponding project structure, including all the source files and libraries required to work with the selected peripherals and Middleware components.

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

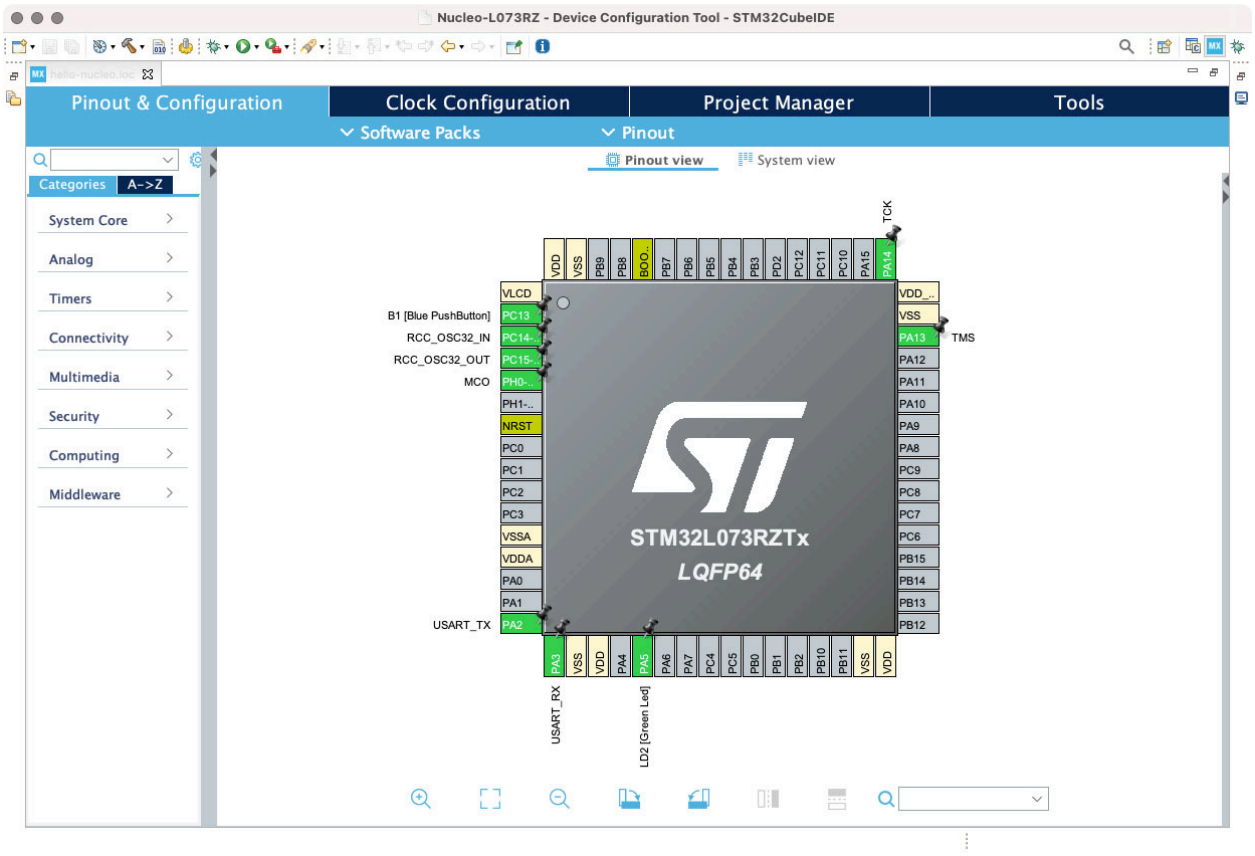


Figure 4.5: CubeMX *Device Configuration Tool* view

At the top of the CubeMX window, you will see a contextual menu with a light and dark blue theme. This menu is divided into four main tabs, each offering a dedicated view. Let us briefly introduce them.

4.1.2.1 Pinout View & Configuration

The *Pinout & Configuration* view is the first tab, and it is further divided into sub-parts.
On the right side, you will find the MCU representation with the selected peripherals and GPIOs, referred to by ST as the *Pinout view*. This view allows for easy navigation within the MCU configuration and provides a convenient way to configure the microcontroller.

Pins[6] colored in bright green are *enabled*. This means CubeMX will generate the necessary code to configure the pin based on the bound peripherals. For example, in the project configuration shown in **Figure 4.5**, for pin `PA5`, CubeMX will generate C code to set it up as a generic output pin to drive the LD2 LED[7]. Meanwhile, for pin `PA2`, CubeMX will generate code to configure it as a USART TX pin.

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

A pin is colored in orange when **the corresponding peripheral** is not enabled. For instance, in **Figure 4.6**, the PA2[^ch4-pb3-pin] and PA3 pins are enabled, and CubeMX will generate C code to initialize them, but since the associated peripheral (USART2) is not enabled, no USART-related code will be generated to configure the peripheral.

Light-yellow pins are power source pins, and their configuration cannot be changed.
BOOT and RESET pins are colored in khaki, and their configuration is fixed as well.

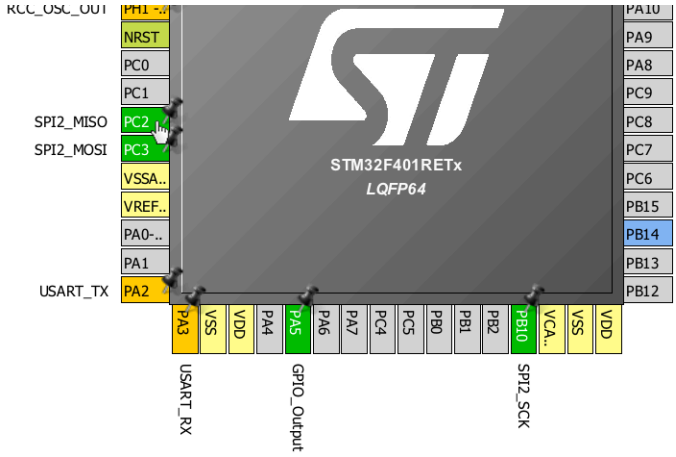


Figure 4.6: Alternate mapping of peripherals

A contextual tool-tip is displayed when hovering the mouse pointer over the MCU pins (see **Figure 4.7**). For example, the contextual tool-tip for pin PB3 informs us that the signal is mapped to the *Serial Wire Debug* (SWD) interface, where it functions as the *Serial Wire Output* (SWO) pin. Additionally, the pin number (55 in this case) is also shown.

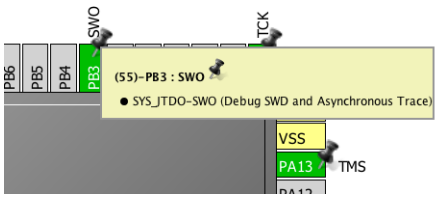


Figure 4.7: Contextual tool-tips help understanding signal usage

STM32 MCUs with a high pin count allow peripheral signals to be mapped to different pins. For example, in an STM32F401xE MCU, the SPI2 MOSI signal can be mapped to either pin PC2 or PB14. CubeMX makes it easy to view these alternative pin mappings with a Ctrl+click on the pin. If an alternative pin exists, it will be highlighted in blue (the alternative is only shown if the pin is not in a reset state—that is, if it is enabled). For instance, in **Figure 4.6**, a Ctrl+click on pin PC2 will highlight the PB14 signal in blue[8].

This feature is particularly useful during the layout of a board. If routing a signal to a specific pin is

[8]Note that in recent CubeMX versions, the Ctrl+click behavior has slightly changed. To display the alternative pin, you need to press and hold the mouse after a Ctrl+click until the alternative pin starts blinking. It is not very intuitive the first time you use it.

either impossible or inconvenient, or if the pin is needed for another function, using an alternative pin can simplify the board design.

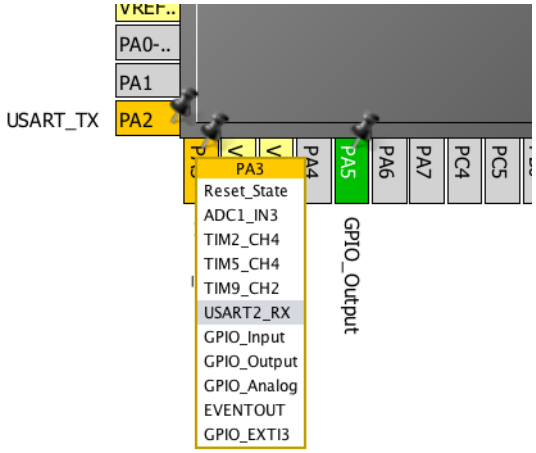


Figure 4.8: Alternate function of a pin

In the same way, most MCU pins can support alternate functionalities. A contextual menu appears when clicking on a pin, allowing you to select the function you want to enable for that signal.

This flexibility, however, can lead to conflicts between signal functions. CubeMX attempts to resolve these conflicts automatically by assigning the signal to a different pin. *Pinned* signals are those whose functionality is locked to a specific pin, preventing CubeMX from choosing an alternative. If a conflict prevents a peripheral from being used, the pin mode in the *Chip View* is disabled, and the pin is colored orange. To mark an I/O as pinned, right-click on a pin and select the **Pin locking** option.

CubeMX also offers a convenient feature: the ability to define custom labels for individual MCU signals. By right-clicking on an enabled pin, you can select **Enter User Label**. A contextual pop-up will appear, as shown in **Figure 4.9**. The label can be in the format `LABEL [Comment]`. The `LABEL` part is used to generate a corresponding macro inside the `main.h` file, while the `[Comment]` part is simply a note for the developer, displayed in the *Pinout View*.

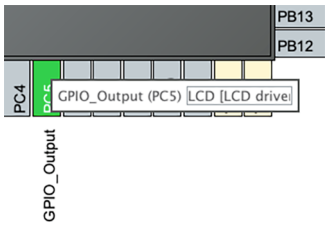


Figure 4.9: How to add custom label to MCU I/Os

As an alternative to the *Pinout view*, the *System view* provides an overview of all components that can be configured in software: GPIOs, peripherals, DMA, NVIC, Middleware, and additional software components. Clickable buttons allow you to open the configuration options for a given component

through the *Mode* and *Configuration panels*. The color of the button icon reflects the status of the component's configuration.

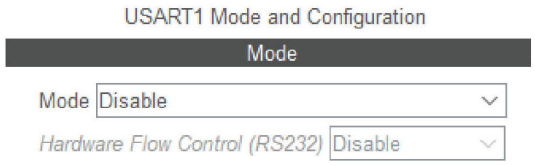
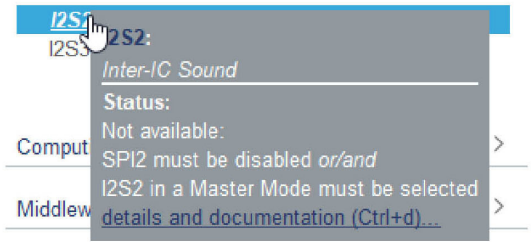
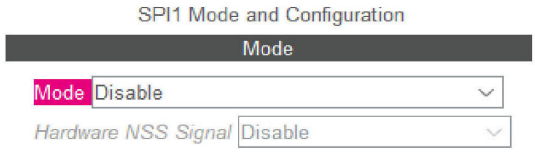
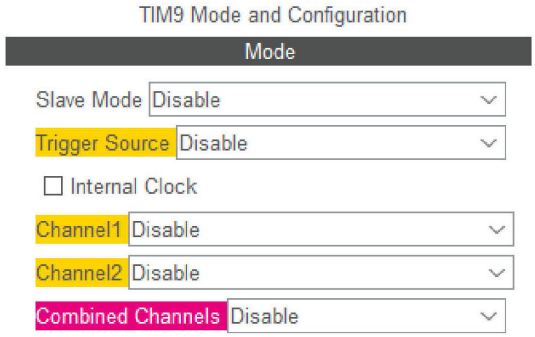
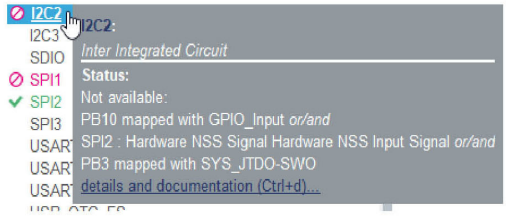
Case	Display	Component status	Corresponding Mode view / Tooltips
1	USART1 (Plain Black Text)	The peripheral is not configured (no mode is set) and all modes are available.	
2	*I2S2* (Gray Italic Text)	Peripheral is not available because some constraints are not solved. See tooltip.	
3	✔ CAN1 ⊗ CAN2	The 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

On the left side of the *Pinout & Configuration* view, we have the *Categories list* (also referred to as the *Components list* in the official ST documentation). This list can be displayed either in alphabetical order or by categories. By default, it contains the list of peripherals and Middleware components that

the target MCU supports, providing a convenient way to enable/disable and configure the desired peripherals and Middleware. Selecting an entry from this list opens two additional panels (*Mode* and *Configuration*) that allow users to set the functional mode and configure the initialization parameters, which 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 are selectable. For example, for a USART interface, all possible modes (*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 peripherals use the same GPIOs, making it impossible to use them simultaneously. Hovering over it will display the conflicting peripheral. For instance, with an STM32F401RE MCU, you cannot 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 still available. A green check mark signifies that all parameters are properly configured, while a fuchsia cross indicates incomplete configurations.
- **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. Hovering over the peripheral name will display a tooltip describing the conflict.

4.1.2.2 Clock Configuration View

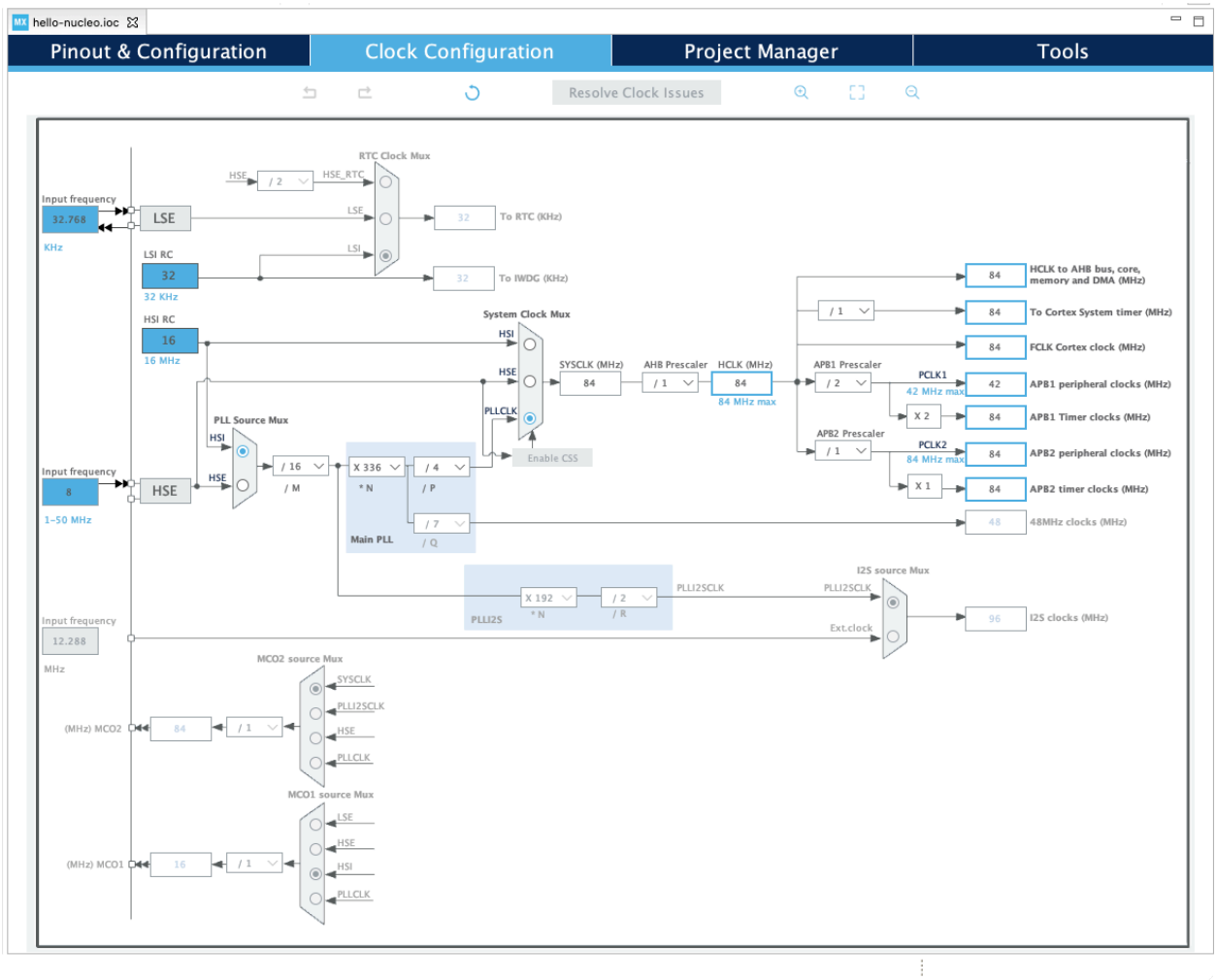


Figure 4.10: The CubeMX clock view

The *Clock Configuration* view is the pane where all configurations related to clock management are handled. Here, you can configure both the core clock and the peripheral clocks. All clock sources and PLL (Phase-Locked Loop) configurations are presented in a graphical format (see **Figure 4.10**). At first glance, the number of configuration options might seem overwhelming to new users. However, with a bit of practice, this becomes the easiest way to manage the STM32 clock configuration, which is considerably more complex compared to 8-bit MCUs.

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

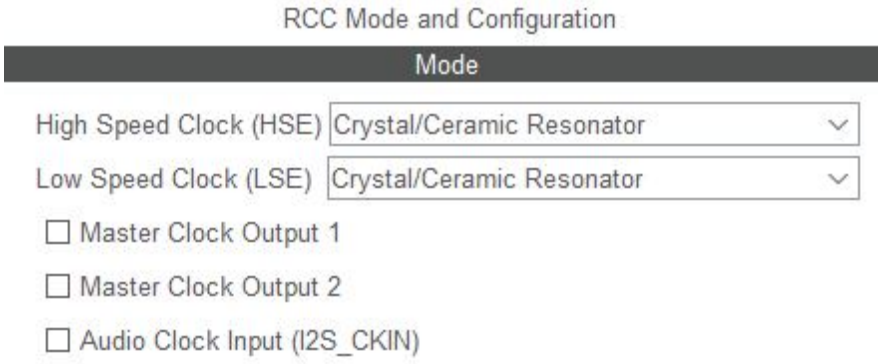


Figure 4.11: HSE and LSE enabling in CubeMX

Once this is accomplished, you will be able to modify clock sources in the *Clock Configuration* view.

The clock tree configuration will be explored in detail in Chapter 10. To avoid confusion at this stage, it is recommended to leave all parameters as they are automatically configured by CubeMX.

Overclocking

A common hacking practice is to overclock the MCU core by adjusting the PLL configuration to run at a higher frequency. The author strongly discourages this practice, as it can not only cause permanent damage to the microcontroller but also lead to abnormal behavior that is difficult to debug.

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

4.1.3 Project Manager

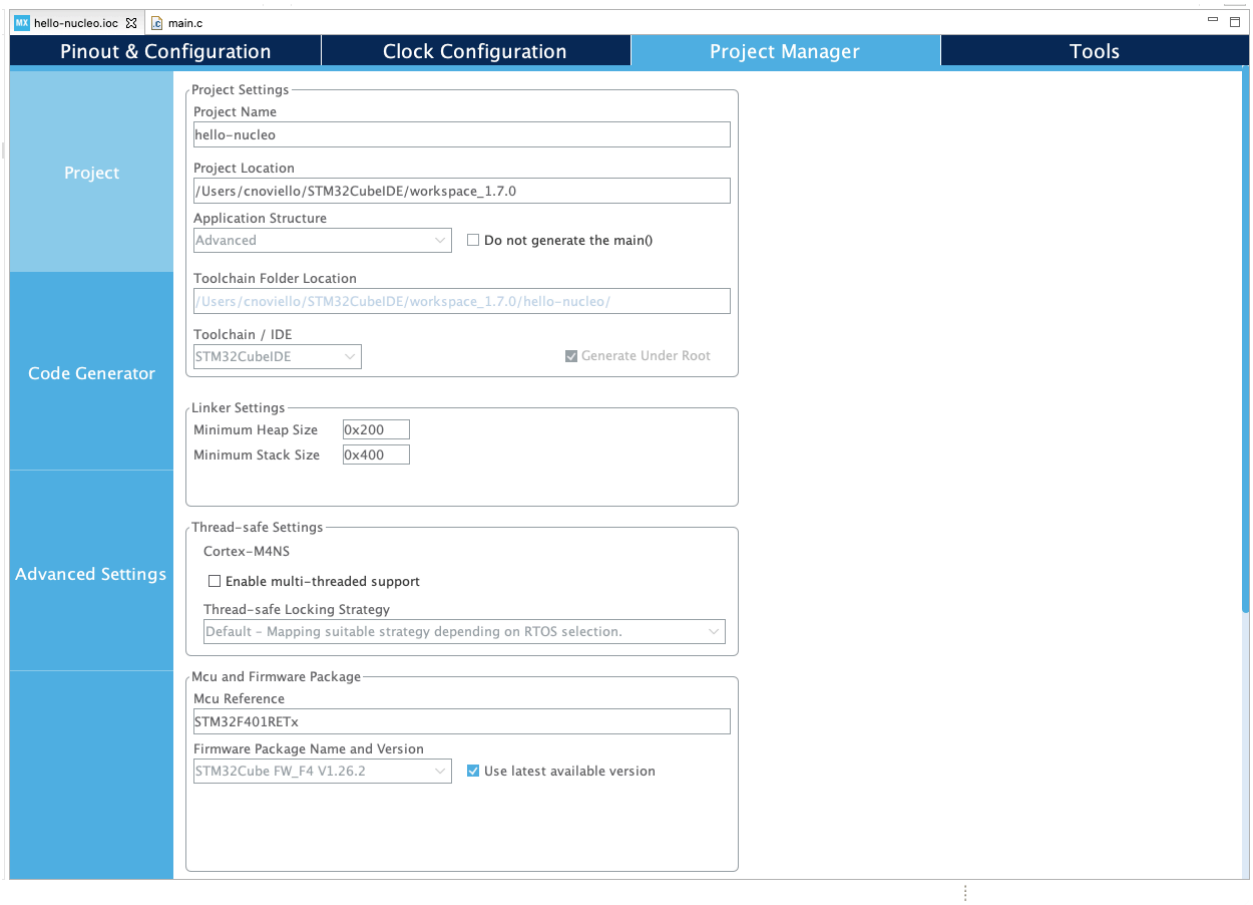


Figure 4.12: The Project Manager view

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

- **Project**: This section includes general project settings such as the project name, its location on the filesystem, the tool-chain, and the CubeHAL libraries version.
- **Code Generator**: This section offers additional options related to CubeMX code generation, such as how HAL `*.c/h` files are included in the project, how the template file structure is maintained when changes are applied to the project settings, and more.
- **Advanced Settings**: This section covers more advanced project options, mainly related to the type of CubeHAL used to generate initialization code for a given peripheral. You can choose between CubeHAL and the more optimized Cube-LL library. Additionally, you can opt not to generate code for specific peripherals or middleware components, allowing the programmer to add custom code if desired.

What is the Cube Low-Layer API?

With the advent of the STCube initiative, ST completely redesigned the SDK for the STM32 series by introducing the *Hardware Abstraction Layer* (HAL) library and phasing out the old *Standard Peripheral Library* (SPL), which had been quite popular in the ST community. Despite its popularity, the SPL lacked many features needed for the more modern and powerful STM32 MCUs. However, over the years, the HAL library has received a lot of criticism—not just because it had numerous bugs in its early stages, but more importantly, because it is not considered an example of optimized code for embedded application development.

The CubeHAL is not a high-performance library, and for a simple reason: it is designed to be abstract and to simplify porting user code between MCUs within the same series or across different STM32 series. This results in a library filled with `if` and `then` statements, as well as unnecessary code when working with a specific microcontroller. But this is the trade-off when prioritizing streamlined development and, more importantly, the adoption of complex microcontroller architectures like those in the STM32 portfolio. The HAL APIs are divided into two categories: generic APIs, which provide common functionality across all STM32 series, and extension APIs, which offer specific, customized functions for a particular line or part number. HAL drivers provide a complete set of ready-to-use APIs to simplify user application development. These drivers are feature-oriented rather than IP-oriented. For instance, the timer APIs are split into several categories based on IP functions, such as basic timers, capture, and pulse width modulation (PWM). The HAL driver layer also implements run-time failure detection by checking the input values of all functions, enhancing the firmware's robustness.

In recent years, ST responded to criticism of the HAL library's performance by introducing the *Cube Low-Layer* (abbreviated as **LL**) set of drivers. As the name suggests, the LL library was created to be highly optimized, giving programmers responsibility for managing the specific characteristics of a given STM32 series and part number. LL drivers offer hardware services based on the exact capabilities of the STM32 peripherals. These services mirror the hardware's functionality and provide atomic operations, which must follow the programming model outlined in the product line's reference manual. Consequently, LL services are not based on standalone processes and do not require extra memory resources for state management, counters, or data pointers. All operations are performed by directly manipulating the associated peripheral registers. Unlike the HAL, LL APIs are not provided for peripherals where optimized access is not a key feature, or for those requiring significant software configuration and/or a complex upper-level stack (such as USB). LL-based code consists primarily of a sequence of C macros that expand into a series of statements, minimizing the use of branches and unpredictable instructions that could hinder performance.

This book does not cover topics related to the LL library, as doing so would require a completely different approach and a narrow focus on a few STM32 part numbers. Instead, this book aims to be general, providing an overview of the most relevant features for designing powerful and complex electronic boards. If you need precise control over every aspect of a peripheral to achieve the most optimized code, the LL library is for you. However, I suggest starting with CubeHAL for firmware design, especially if you are not an experienced firmware developer, and transitioning to LL only when necessary.

4.1.4 Tools View

The *Tools* view contains additional configuration panes, some of which are specific to more advanced STM32 MCUs, such as the STM32MP1 series. However, for all STM32 microcontrollers, the *Power Consumption Calculator* (PCC) is available. This feature of CubeMX, when provided with a microcontroller, a battery model, and a user-defined power sequence, estimates the following parameters:

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

Users can also add custom batteries through a dedicated interface.
For each step in the power sequence, the user can select VBUS as a potential power source instead of the battery, which will affect the battery life estimation. If power consumption measurements are available at different voltage levels, CubeMX will also offer a selection of voltage values.

The PCC view will be explored in a following chapter.

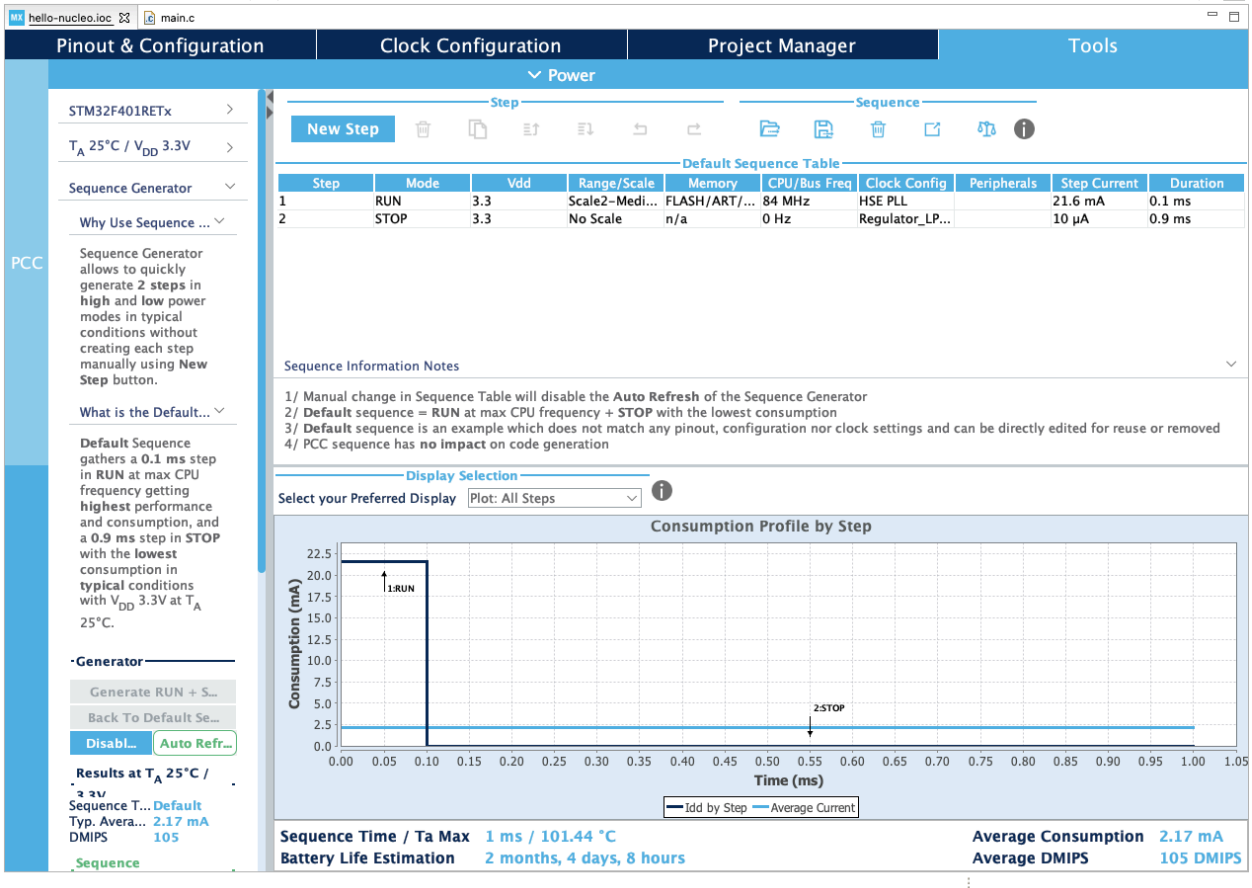


Figure 4.13: The CubeMX Tools view

4.2 Understanding Project Structure

Once the configuration of the MCU, its peripherals, and Middleware components is completed, CubeMX can be used to generate the C project skeleton for you. Code generation can be initiated in two ways:

- By saving the CubeMX project (the `.ioc` file), which automatically triggers the code generation process.
- By going to the **Project->Generate Code** menu (with the `.ioc` file selected in the main perspective) or by clicking the corresponding icon in the Eclipse toolbar (see **Table 2.1**).

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

At first, the generated project structure might seem overwhelming, especially if you are new to embedded programming or dealing with such complex architectures[9]. But do not worry! In the following chapters, we will dive into the details of each auto-generated source file. However, to start programming without feeling lost, it is helpful to take a quick look at the main project folders and their contents. **Figure 4.14** will serve as a good reference for this walkthrough.

`Binaries`
: This folder contains the final binary file generated by the compiler at the end of the build process. The file is in the *Executable and Linkable Format* (ELF), a common object file format in Linux-based operating systems. This folder is part of Eclipse CDT's project organization, and its content is somewhat redundant, as the same binary file can also be found inside the `Debug` folder.

`Includes`
: This folder serves two purposes. It is a graphical representation of all `Include` paths for the compiler (i.e., the directories where GCC will look for C header files (`.h`)). Additionally, it can be used to add references to other include files without dealing directly with compiler-specific arguments and include paths. For more information on this topic, refer to the Eclipse CDT documentation.

`Core`
: This folder contains all application-specific files. The files within this folder, along with its subfolders, are specific to the project settings and MCU configuration in CubeMX. The `Core` folder should house all files necessary for application development, though Eclipse allows you to rearrange these files into different folders if needed. However, there is a significant drawback: modifying the structure of the `Core` folder will prevent you from updating the source code when making changes in CubeMX. For this reason, it is recommended to leave the `Core` folder structure unchanged, at least during the early stages of the project.
  Several files within the `Core` folder serve a special role in the project. Let us take a closer look at them.

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

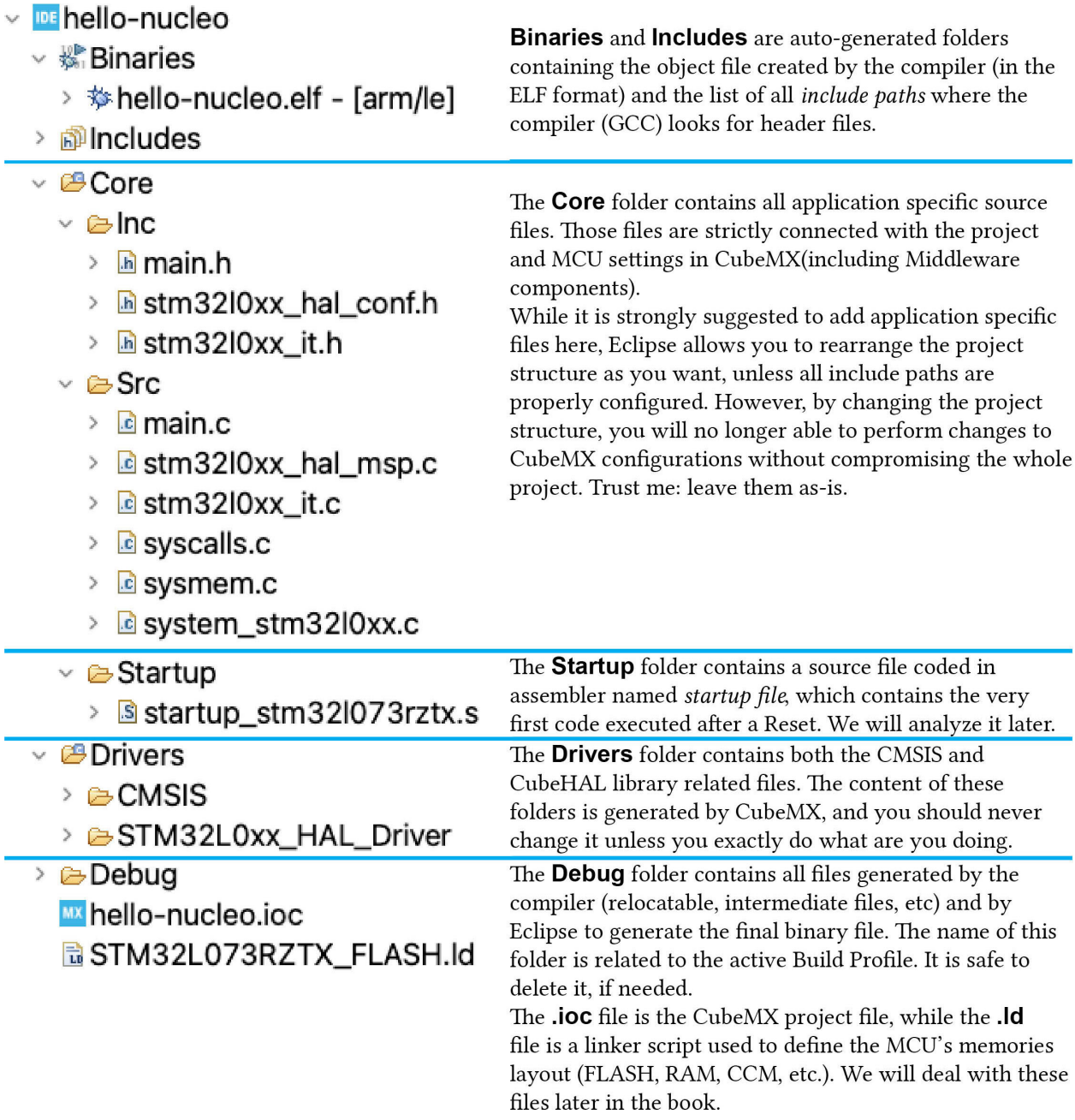


Figure 4.14: The typical structure of CubeMX project

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

`Core/Inc/stm32XXxx_hal_conf.h`
: This file translates the HAL configurations into C code through various macro definitions. These macros are used to instruct the HAL about the enabled MCU functionalities. You will find many 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, as long as the necessary `.c/.h` files are already included in the project. We will explore 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 essential components of the project. They store all the *Interrupt Service Routines* (ISRs) generated by CubeMX. Depending on the CubeMX configuration, these files contain definitions for several functions. In the case of the *hello-nucleo* project from Chapter 3, the main relevant function is `void SysTick_Handler(void)`. This function is the ISR for the *SysTick* timer, which is invoked whenever the SysTick timer reaches `0`. But where is this ISR actually 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 }
```

The answer to this question gives us an opportunity to explore one of the most fascinating features of Cortex-M processors: the *Nested Vectored Interrupt Controller* (NVIC). Table 1.1 in Chapter 1 lists the Cortex-M exception types. As you may recall, we mentioned that interrupts in a Cortex-M CPU are a special type of exception. Cortex-M defines the `SysTick_Handler` as 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, commonly referred to as the *startup file*. By opening this file, we can observe 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 note that *startup files* may vary slightly between different ST HALs. The line numbers mentioned here could differ from those in the startup file for your specific MCU. Additionally, the `MemManage Fault`, `Bus Fault`, `Usage Fault`, and `Debug Monitor` exceptions are not available (and the corresponding vector entries are RESERVED—see 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+ and Cortex-M3/4/7 based MCUs.

`Core/Src/stm32XXxx_hal_msp.c`

This is another crucial file to analyze. First, let us clarify the meaning of "MSP." It stands for *MCU Support Package*, and it defines all the initialization functions required to configure the on-chip peripherals according to the user configuration (such as pin allocation, clock enablement, DMA usage, and interrupts). To explain this further, consider that a peripheral is essentially composed of two things: the peripheral itself (e.g., the SPI2 interface) and the hardware pins associated with this peripheral.

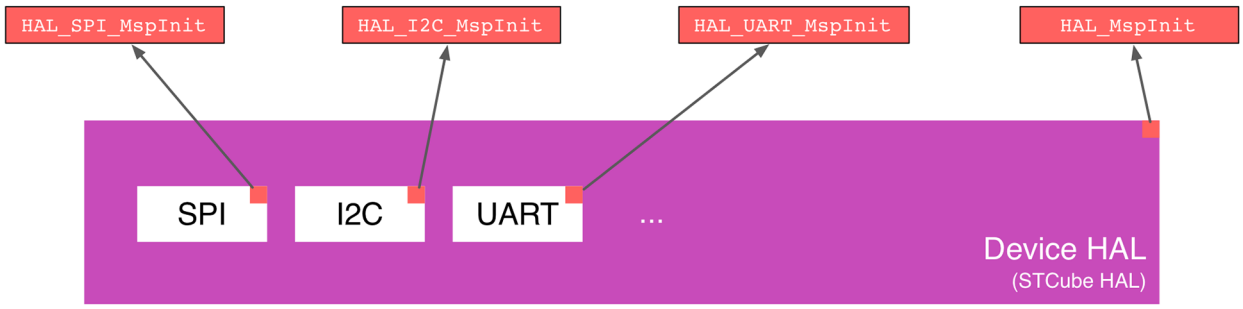


Figure 4.15: The relation between MSP files and the HAL

The ST HAL is designed so that the SPI module (and other modules) of the HAL is generic and abstracted from specific I/O settings, which may vary depending on the MCU package and the user-defined hardware configuration. Therefore, ST developers have left it to the user to "fill" this part of the HAL with the necessary code to configure the peripheral, using a form of *callback* routines. 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 for the actual configuration of the pin pairs associated with the USART peripheral (specifically, `PA2` and `PA3`). **Figure 4.16** illustrates the call hierarchy of the `HAL_UART_MspInit()` function: it is invoked by the generic HAL function `HAL_UART_Init()`, which, in turn, is called in `main.c` by the function `MX_USART2_UART_Init()`.

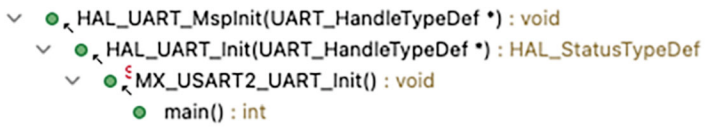


Figure 4.16: The Call Hierarchy of the function `HAL_UART_MspInit()`

The last file we need to analyze is `Core/Src/main.c`. This file essentially contains four routines: `SystemClock_Config(void)`, `MX_GPIO_Init(void)`, `MX_USART2_UART_Init(void)`, and `int main(void)`. The first function, `SystemClock_Config(void)`, initializes the core and peripheral clocks. Although its explanation is beyond the scope of this chapter, the code is not too difficult to understand if you are familiar with clock configurations.

The function `MX_GPIO_Init(void)` configures the GPIOs connected to the `LD2` pin and the `B1` pin (the pin connected to the blue switch on the Nucleo board). Chapter 6 will explore GPIO configuration in detail.

Lastly, 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-explanatory. First, the HAL is initialized by calling the `HAL_Init()` function. Then, the clocks, GPIOs, and USART2 are initialized. Finally, the application enters an infinite loop—this is where your application code should be placed.

Peripheral Initialization and Deinitialization

If you examine the file `Core/Src/stm32XXxx_hal_msp.c`, you will find the definition of the `HAL_UART_MspDeInit()` function. Similar to `HAL_UART_MspInit()`, the `HAL_UART_MspDeInit()` function is invoked by the HAL function `HAL_UART_DeInit()`. As a good programming practice, it is recommended to always de-initialize a peripheral when it is no longer in use. For most peripherals, the deinitialization procedure involves setting the associated GPIOs to a high-impedance state (to avoid power leakage) and stopping any ongoing peripheral activity by shutting down its clock source. The CubeHAL is designed to properly handle peripheral deinitialization.

To keep things simple, proper deinitialization procedures are omitted in most of the examples in this text. However, remember that in embedded programming, the more control you have, the lower the risk of encountering unwanted behaviors.

`Core/Src/syscalls.c`

ARM GCC, and by extension the entire STM32 development environment, relies on a reduced version of the standard C run-time library for embedded systems called *newlib nano*. Every

time we compile our firmware for an STM32 microcontroller, pieces of the `newlib nano` library are linked with CubeHAL and our source code. `newlib nano` allows the use of some traditional C library functions in embedded applications. For instance, it is perfectly possible to use standard I/O functions like `printf()` and `scanf()`, even if the board does not have a screen or keyboard. However, some of the `newlib nano` functions rely on lower-level routines (called *system calls* or *syscalls*) that handle hardware-specific features. The `Core/Src/syscalls.c` file contains a dummy implementation of these routines; without them, the linking process would fail.

In upcoming chapters, we will learn how to customize certain *syscalls* to implement advanced debugging capabilities. We will explore several ways to establish communication between the board and a host PC, using methods such as the *Instrumentation Trace Macrocell* (ITM), *ARM Semihosting*, or even a simple UART communication.

`Core/Src/sysmem.c`

Similar to `syscalls.c`, this file contains a functional implementation of the `_sbrk()` routine. In UNIX-based environments, this syscall controls the amount of memory allocated to a process's data segment (the *heap*). In Chapter 20, we will explore the memory layout of a typical STM32 application in detail, which will help us understand the logic behind `_sbrk()` and how to customize it for specific needs.

`Drivers`

This folder contains both the CMSIS-CORE package and the CubeHAL library. By default, CubeMX places only the necessary files for using the peripherals enabled via the *Device Configuration Tool* in this folder and its subfolders. The CMSIS-CORE package implements the basic run-time system for a Cortex-M device and provides access to the processor core and peripherals through convenient C macros. Specifically, it defines:

- **HAL** for Cortex-M processor registers, offering standardized definitions for *SysTick*, NVIC, System Control Block registers, MPU registers, FPU registers, and core access functions.
- **System exception names** to interface with system exceptions in a standardized way, avoiding compatibility issues.
- **Methods for organizing header files** to make it easier to learn new Cortex-M microcontroller products and improve software portability. This includes consistent naming conventions for device-specific interrupts.
- **Methods for system initialization** provided by MCU vendors. For instance, the standardized `SystemInit()` function is crucial for configuring the clock system when the device starts.
- **Intrinsic functions** used to generate CPU instructions that are not supported by standard C functions.
- A **global variable** called `SystemCoreClock`, which helps 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 to access the core registers (*SysTick*, NVIC, ITM, DWT, etc.). These files are generic and can be used with 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 produce the final binary file in ELF or another binary format. The name `Debug` comes from the name of the active *Build Configuration*. *Build configurations* is a feature supported by all modern IDEs, allowing multiple project configurations within the same project. Every Eclipse project has at least two build configurations: *Debug* and *Release*. The former is used to generate a binary suitable for debugging, while the latter generates optimized firmware for production.
It is safe to delete this folder entirely if needed.

`STM32XXxx_FLASH.ld` and `STM32XXxx_RAM.ld`

These files (note that the `_RAM.ld` file may not be present in projects for some STM32 MCUs) are linker scripts that describe the memory layout of the application. They define the amount of FLASH and RAM memory and, more importantly, how these memories are organized at runtime. In Chapter 20, we will explore the memory layout of a typical STM32 application in detail. This will help us understand the content of these files and how to customize them as needed.

4.3 Downloading Book Source Code Examples

All examples presented in this book are available for download from its GitHub repository: http://github.com/cnoviello/mastering-stm32-2nd[10].

The examples are organized by Nucleo model, as shown in **Figure 4.17**. You can clone the entire repository using the `git` command:

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

Alternatively, you can download only the repository content as a .zip package following this link[11]. The repository is divided into nine subfolders, each corresponding to one of the Nucleo boards used to build the examples in this book.
Now, you need to import the relevant Eclipse projects for your Nucleo board into your Eclipse workspace.

Open Eclipse and switch to a new workspace. Go to **File->Import**.... The Import dialog will appear. Select **General->Existing Project into Workspace** and click the **Next** button. Then, browse to the folder containing the example projects for your Nucleo board by clicking the **Browse** button. Once the main folder is selected, a list of the contained projects will appear. Check all the projects you are interested in and make sure to select **Search for nested projects** and **Copy projects into workspace**, as shown in **Figure 4.18**. Finally, click the **Finish** button.

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

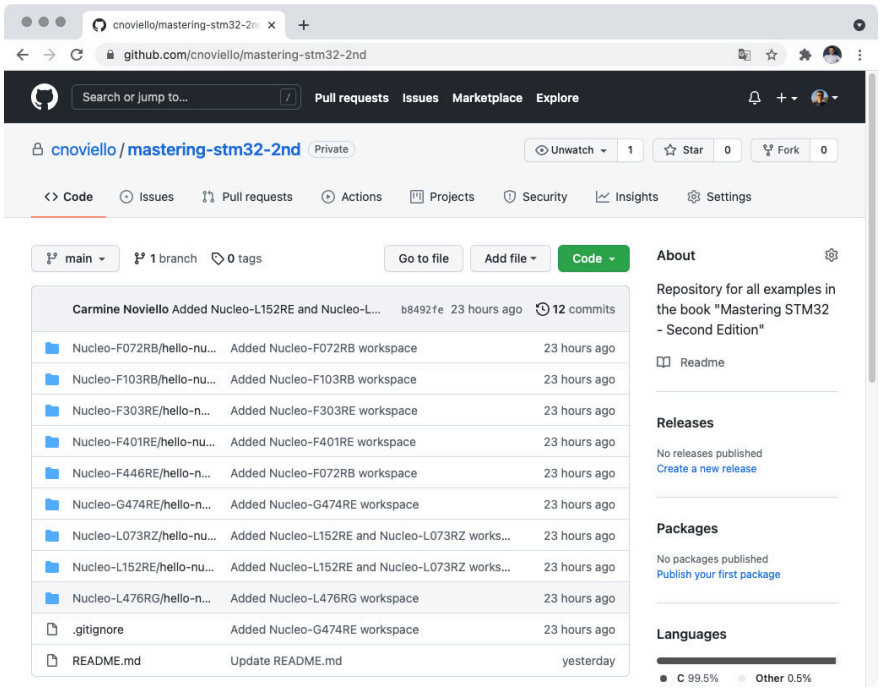


Figure 4.17: The content of the GitHub repository containing all the book examples

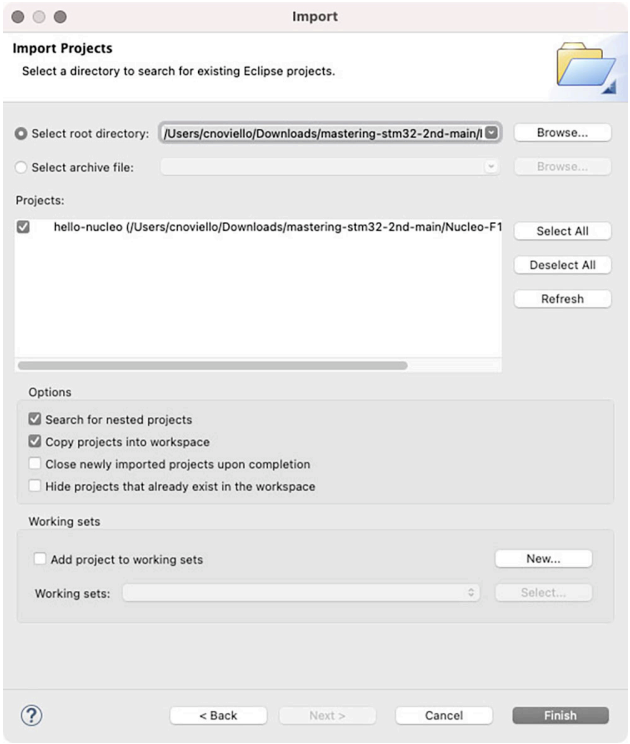


Figure 4.18: Eclipse project import wizard

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

Each project corresponds to a given chapter, and all the examples discussed in that chapter are available within the same project. To switch between different examples, simply select the corresponding *Build Configuration*, as shown in **Figure 4.19**.

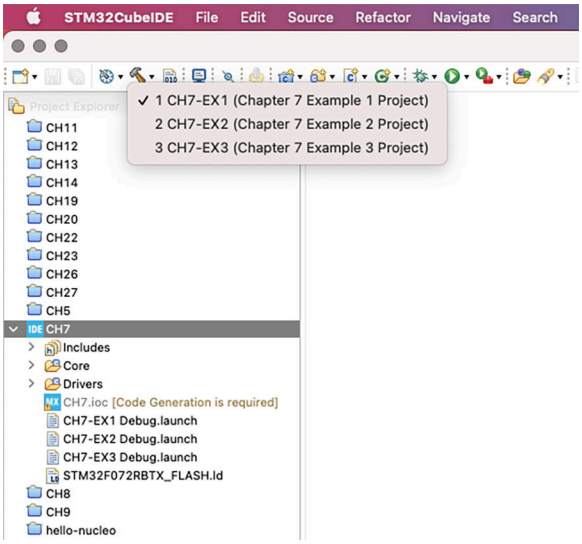


Figure 4.19: How to switch to a different project configuration to select other chapter's examples

The rest of the chapter is not available in the book sample

5. Introduction to Debugging

"Coding is all about debugging", a friend once told me. And this is profoundly true. No matter how well we write our code, sooner or later, we will encounter software bugs (and let us not even start on the nightmare of hardware bugs!). Mastering the art of debugging embedded software is essential to becoming a happy and effective embedded developer.

In this chapter, we will begin by exploring the fundamental debugging features provided by STM32CubeIDE. As we will see, STM32CubeIDE includes a robust set of debugging tools that make identifying and fixing bugs or unexpected behavior a much more manageable task. STMicroelectronics has done an excellent job integrating these tools directly into Eclipse, making them straightforward and intuitive to use. Unlike in the past, when developers had to rely on external programs or specialized hardware, today all you need is a simple, affordable ST-LINK debug probe and STM32CubeIDE.

This chapter offers an introductory look at the debugging process, but it is important to note that debugging—especially for complex systems—could easily warrant its own book, even for relatively simple architectures like the STM32. In Chapter 24, we will delve deeper into advanced debugging techniques, focusing on the Cortex-M exception mechanism, a key feature of this platform.

5.1 What is Behind a Debug Session

Before we dive into how to start a debug session and perform common debugging tasks—such as adding breakpoints, step-by-step execution, and step-into operations, it is helpful to first take a quick look at the software and hardware tools involved. **Figure 5.1** provides an overview of the debugging setup happening behind the scenes.

In a GCC-based development environment, the primary tool for performing debugging operations is the *GNU Debugger* (GDB). GDB is a command-line tool with an integrated shell and a wide array of commands and options. It shares the same design philosophy as GCC: it is abstracted from the specifics of the target architecture (whether it be x86, MIPS, ARM, etc.), the programming language (C, C++, etc.), or the host operating system (Windows, Linux, macOS, etc.).

To maintain this level of portability across various architectures, GDB is built with a clear separation between its frontend (the core of GDB responsible for binary manipulation, interpreting debug information in object files, etc.) and its backend, which handles the details of the target hardware and software architecture. As a result, GDB operates in a client-server model, where the client (frontend) communicates with the server (backend) using a defined protocol over a network connection. This connection can be established either between two separate machines or within the same machine.

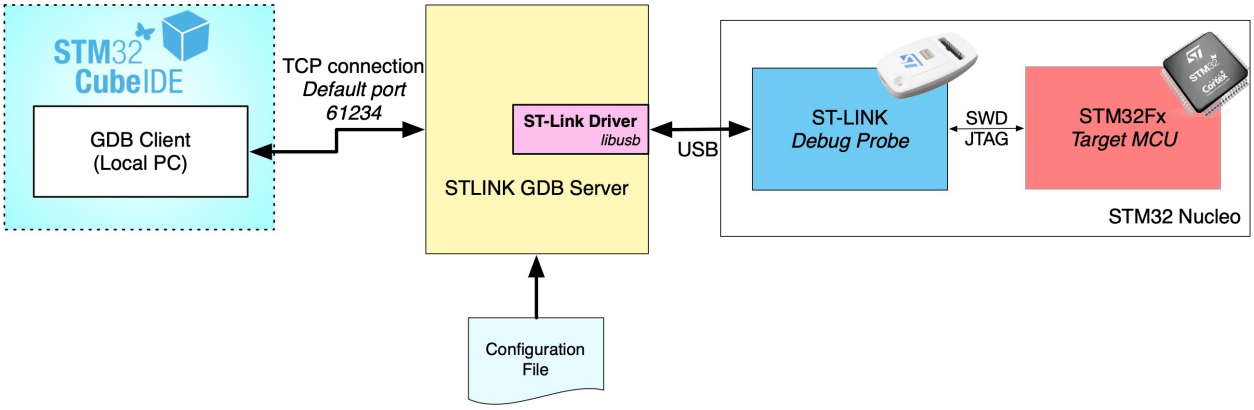


Figure 5.1: How OpenOCD interacts with a Nucleo board

For embedded architectures like the Cortex-M, it is common that the server part is not provided within the ARM-GCC distribution. This is because a debug session always involves a dedicated debug adapter—a piece of hardware that translates, both physically and logically, "high-level" commands into JTAG or SWD signals and instructions. For all Nucleo boards, this adapter is the integrated ST-LINK interface[1].

To accommodate this, ST provides a dedicated backend server for GDB, called *ST-LINK GDB Server*, which communicates with the ST-LINK adapter via a USB connection using *libusb* or any API-compatible library that allows user-space applications to interface with USB devices. Thanks to a set of configuration files included in STM32CubeIDE distributions, *ST-LINK GDB Server* understands how to interface with the specific target MCU (e.g., STM32F030, STM32F401), its specific *Debug Access Port* (DAP), its unique FLASH memory[2], bus architecture, and more.

When a debug session starts, the following main steps occur[3]:

1. STM32CubeIDE runs the *ST-LINK GDB Server* in the background, passing several command-line arguments that specify parameters like the path to *STM32CubeProgrammer*, the TCP/IP port to accept connections from the GDB client, the type of debug mode (SWD, JTAG), debug port speed, and more[4]. If the *ST-LINK GDB Server* successfully communicates with the ST-LINK probe and the target board, it waits for commands on the designated TCP/IP port (usually port 61234).
2. STM32CubeIDE then executes the *GDB client*, which connects to the remote GDB server (the *ST-LINK GDB Server*) using the provided TCP/IP port.
3. STM32CubeIDE loads the binary file into the target MCU's FLASH memory and starts the firmware execution.

[1]The Nucleo ST-LINK debugger is designed to be used as a standalone adapter to debug an external device (e.g., a board you have designed that features an STM32 MCU). Refer to your Nucleo board's documentation for configuration details.

[2]A common misconception about the STM32 platform is that all STM32 devices have a standardized method for accessing internal FLASH memory. This is incorrect, as each STM32 family has distinct peripheral capabilities, including internal flash. As a result, *ST-LINK GDB Server* must provide drivers to handle all STM32 devices.

[3]These steps provide a simplified overview of the operations carried out during a debug session. Many details are omitted, as a full description would require in-depth knowledge of Cortex-M internals, STM32-specific details, and a solid understanding of the GDB framework.

[4]The complete list of command-line arguments is documented in the user manual available here: https://bit.ly/3EfV2aH.

Finally, Eclipse-CDT provides all the necessary logic to control GDB in the background, allowing the user to perform typical debugging tasks through a graphical interface without needing to know any GDB shell commands.

5.2 Debugging With STM32CubeIDE

Eclipse provides a dedicated perspective for debugging, designed to offer the essential tools required during the debugging process. This perspective can also be customized with additional plug-ins as needed (more about this later).

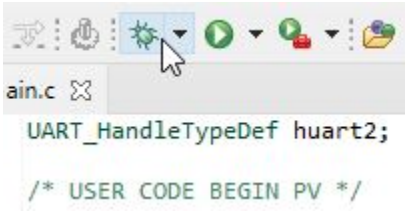


Figure 5.2: The *Debug* icon to start debugging in Eclipse

To start a new debug session, simply click on the **Debug** icon in the Eclipse toolbar, as shown in **Figure 5.2**. Eclipse will prompt you to switch to the *Debug Perspective*. Click **Yes** (it is recommended to check the **Remember my decision** checkbox). Eclipse will then switch to the *Debug Perspective*, as shown in **Figure 5.3**.

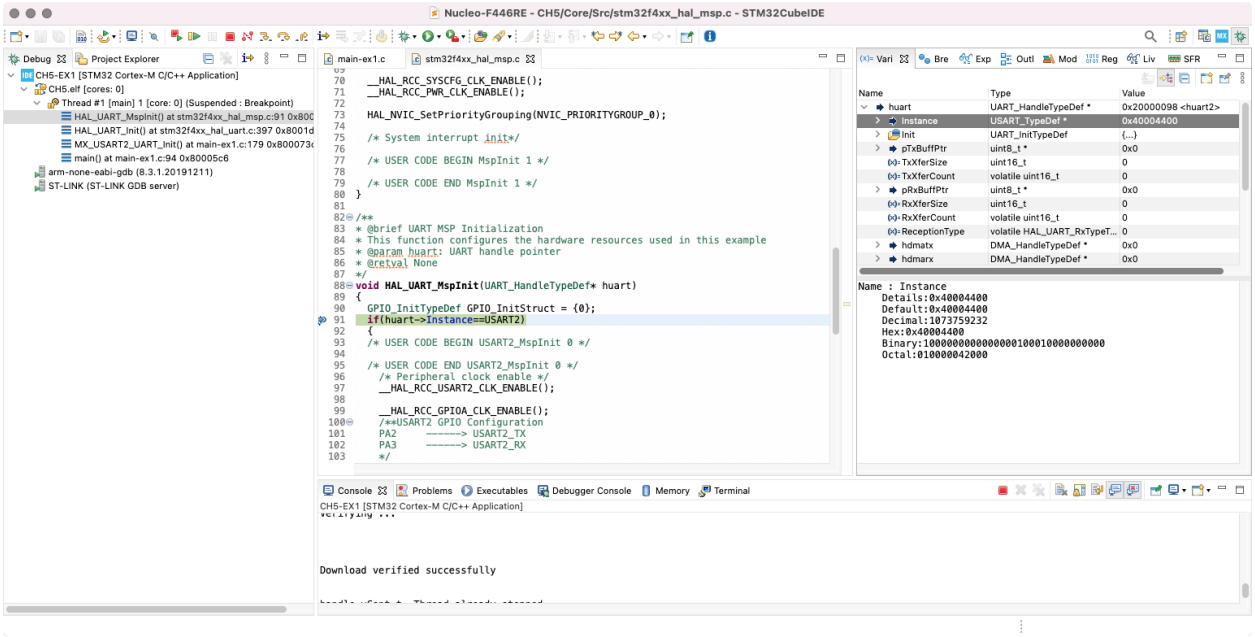


Figure 5.3: The *Debug Perspective*

5.2.1 Views in the Debug Perspective

Let us explore the purpose of each view in the *Debug Perspective*. The top-left view is called **Debug**, and it displays all active debugging sessions. This is a tree-view, and when the firmware execution is paused, it shows the complete call stack, providing a quick way to navigate through it.

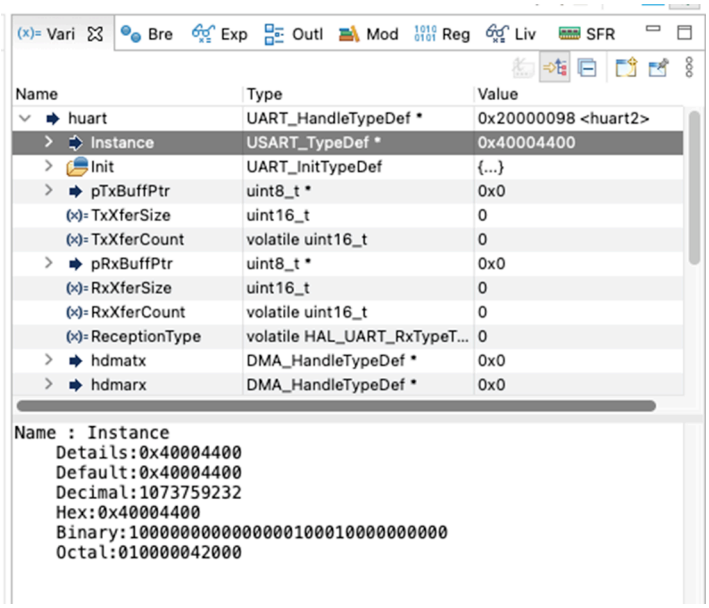


Figure 5.4: The variables inspection pane in the *Debug Perspective*

The top-right view contains several sub-panes. The **Variables** pane allows you to inspect the contents of variables within the current stack frame (i.e., the selected procedure in the call stack). Right-clicking on a variable lets you further customize how the variable is displayed. For instance, you can change its numeric representation from decimal (the default) to hexadecimal or binary. You can also cast the variable to a different data type, which is particularly useful when working with raw data, such as a stream of bytes that represent a specific type. Additionally, you can navigate to the memory address where the variable is stored by selecting **View Memory**... from the contextual menu.

The **Breakpoint** pane lists all breakpoints used in the application. A *breakpoint* is a hardware feature that stops the firmware execution when the *Program Counter* (PC) reaches a specified instruction. When this happens, the debugger halts, and Eclipse displays the context of the stopped instruction. Each Cortex-M-based MCU has a limited number of hardware breakpoints. **Table 5.1** summarizes the maximum breakpoints and watchpoints[5] available for each Cortex-M family.

[5] A watchpoint is a more advanced debugging primitive that allows you to define conditional breakpoints over data and peripheral registers. This means the MCU halts its execution only if a variable satisfies a specified condition (e.g., `var == 10`). We will explore watchpoints in greater detail in Chapter 24.

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

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

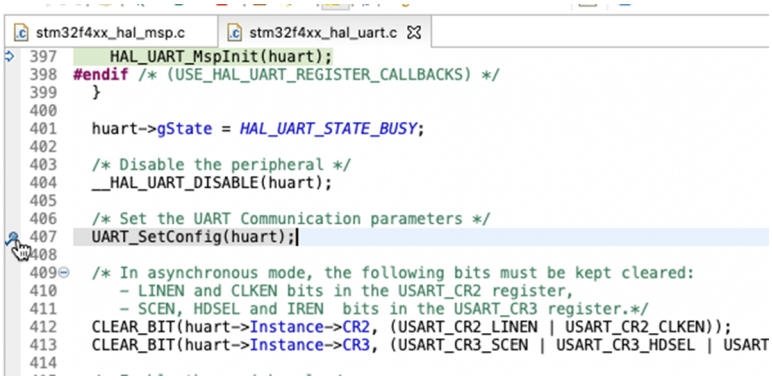


Figure 5.5: How to add a breakpoint at a given line number

Eclipse makes it easy to set up breakpoints directly from the editor view, located at the center of the **Debug Perspective**. To place a breakpoint, simply double-click on the grey stripe to the left of the editor, next to the instruction where you want to halt the MCU's execution. A blue bullet will appear, as shown in **Figure 5.5**.

When the program counter reaches the first assembly instruction corresponding to that line of code, the execution halts, and Eclipse highlights the corresponding line, as shown in **Figure 5.3**. After inspecting the code, you have several options to resume execution. **Table 5.2** explains the function of the most important 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.

Table 5.2: Most relevant icons on the Eclipse debug toolbar

Icon	Description
	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.
	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 will find two more useful views: **SFR** and **Registers**. These display the contents of both the hardware registers specific to the STM32 MCU and the Cortex-M core registers. These views can be extremely helpful for understanding the current state of a peripheral or the Cortex-M core itself. In Chapter 24, where we discuss debugging, we will explore how to handle Cortex-M exceptions and learn how to interpret the contents of several important Cortex-M registers.

5.2.2 Debug Configurations

Eclipse is a highly configurable and generic IDE, allowing the creation of multiple debug configurations that can easily be adapted to different development scenarios.
So far, we have started debug sessions by simply clicking on the corresponding icon in the toolbar (see **Figure 5.2**). However, when we do this for the first time, STM32CubeIDE automatically configures the debug operations for us.

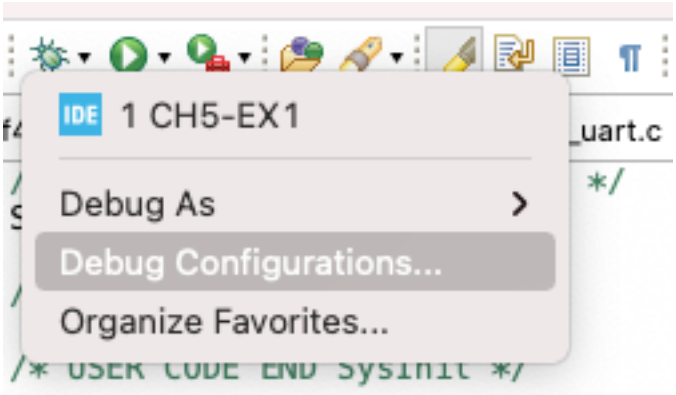


Figure 5.6: Debug Contextual Menu

By clicking on the down arrow next to the debug icon, you can access the debug contextual menu (see **Figure 5.6**). From there, selecting **Debug Configurations**... allows you to manage all debug configurations, as shown in **Figure 5.7**.

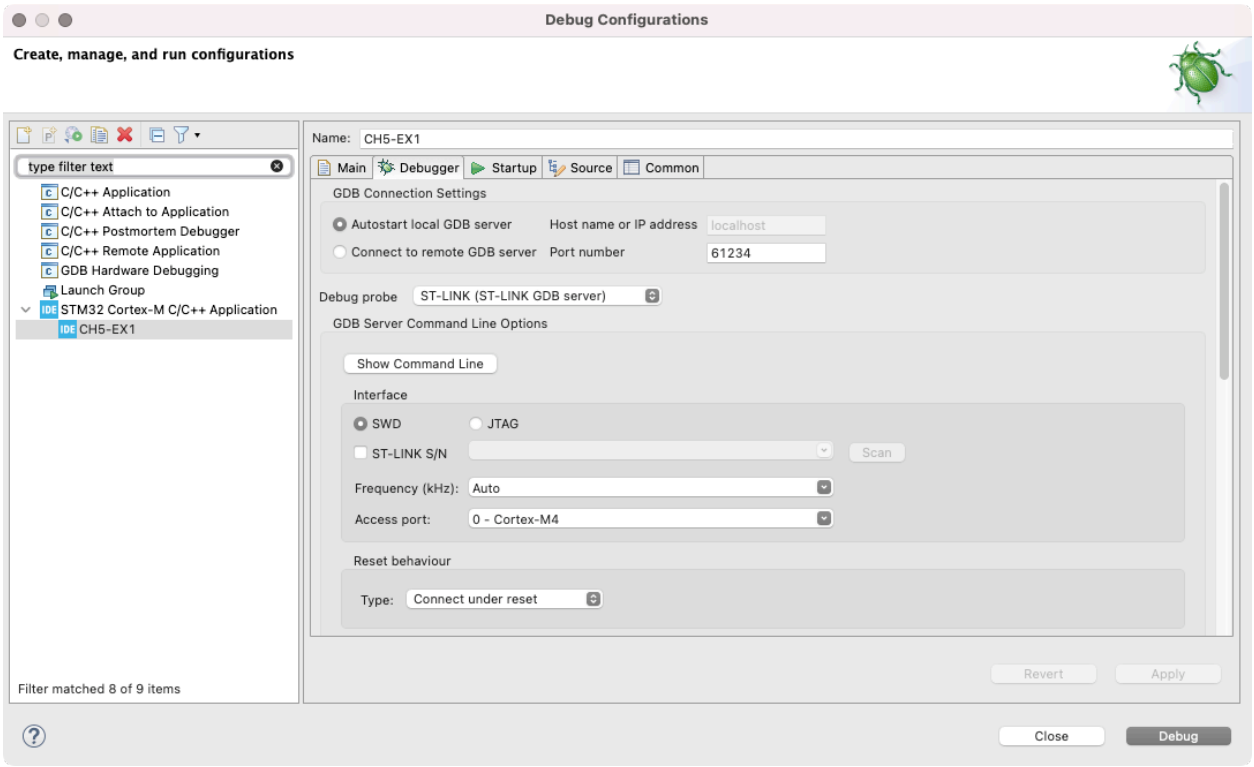


Figure 5.7: Debug Configurations Dialog

The view is divided into two main panes. On the left, there is a tree pane containing several configuration types. We are particularly interested in the **STM32 Cortex-M C/C++ Application**. By expanding this entry, you will see the debug configuration that was created automatically (the configuration name corresponds to the project name).
On the right, there is a tabbed pane with several tabs. The most notable ones are **Main**, **Debugger**,

and **Startup**.

The **Main** tab primarily contains the project name and specifies which binary file to load onto the target MCU to begin a debug session.

The **Debugger** tab includes several important options for configuring the debug session. Some of these options are advanced topics that we will cover in later chapters. Here, we will focus on the most critical ones.

- **GDB Connection Settings**: This group of settings configures the GDB Server. You can select whether to connect to a local or remote server, specify its IP address, and set the port number. It is strongly recommended to leave all options at their default values.
- **Debug Probe**: STM32CubeIDE supports three different debug probes: the standard ST-LINK, SEGGER J-Link, and OpenOCD. In this text, we assume the use of the ST-LINK debug probe integrated into the Nucleo board. However, we will discuss the other two in a later chapter.
- **Interface**: These settings allow you to choose which MCU debug port to use. Most STM32 MCUs support both JTAG and SWD interfaces. In this book, we assume the use of the SWD interface.

The **Reset Behavior** section requires a deeper explanation. Occasionally, you may encounter issues where the MCU cannot be flashed or debugged using ST-LINK. One noticeable symptom is that the ST-LINK LD1 LED (which usually blinks red and green while debugging) stops blinking and remains frozen, with both LEDs ON.

When this occurs, it indicates that the ST-LINK debugger is unable to access the target MCU's debug port (via the SWD interface), or that the flash is locked, preventing debugger access.

There are typically three causes for this condition:

- The SWD pins have been reconfigured as general-purpose GPIOs (this often happens after resetting pin configurations in CubeMX).
- The MCU is in a deep *low-power* mode that has disabled the debug port.
- An issue with the option bytes configuration—perhaps the flash has been write-protected, or read protection level 1 has been enabled.

To resolve this issue, the ST-LINK debugger must be forced to connect to the target MCU while keeping its `nRST` pin low. This procedure is known as *connection under reset* and can be performed by selecting one of the **Reset Behavior** options, which are described next.

- **Connect under reset** (default): The ST-LINK reset line is activated, and the ST-LINK connects in SWD or JTAG mode while the reset is active. Once connected, the reset line is deactivated.
- **Software system reset**: A system reset is triggered by writing to the `RCC` register in software. This resets the core and peripherals, and can also reset the entire system as the target's reset pin asserts itself.

- **Hardware reset**: The ST-LINK reset line is activated and then deactivated (a pulse on the reset line), after which the ST-LINK connects in SWD or JTAG mode.
- **Core reset**: A core reset is triggered by writing to a Cortex-M register in software (not possible on Cortex®-M0/0+/33 cores). This resets only the core, without affecting the peripherals or the reset pin.
- **None**: This option is used for attaching to a running target where the program has already been loaded into the device. There should be no file programming command in the **Startup** tab.

The **Startup** tab configures how the debug session begins. The **Initialization Commands** field can be customized with GDB or GDB server `monitor` commands, which are sent to the GDB server before loading the program. For example, when using the ST-LINK GDB server, the command `monitor flash mass_erase` can be added if a full FLASH memory erase is required before loading.

The **Load Image and Symbols** list box should include the file(s) to be debugged. The **Runtime Options** section contains checkboxes for setting the start address, setting a breakpoint, and enabling exception handling and resuming. The **Set breakpoint at** checkbox is enabled by default, with the field displaying `main`. This means that a breakpoint is automatically set at the `main()` function when debugging begins. As a result, execution halts at `main()` at the start of every debug session.

Three exception checkboxes make it easier to identify problems during debugging:

- **Exception on divide by zero**: Enabled by default to catch divide-by-zero errors during debugging.
- **Exception on unaligned access**: Can be enabled to trigger exceptions for unaligned memory access.
- **Halt on exception**: Enabled by default to halt program execution whenever an exception occurs during debugging.

The rest of the chapter is not available in the book sample

II Diving into the HAL

6. GPIO Management

All STM32 microcontrollers include a variable number of *General Purpose Input/Output* (GPIO) pins, with the exact count depending on several factors:

- The package type chosen (e.g., LQFP48, BGA176).
- The specific microcontroller family (e.g., F0, F1, etc.).
- The use of external crystals for high-speed and low-speed external oscillators (HSE and LSE).

GPIOs are the primary means through which an MCU interfaces with external devices. On every electronic board, varying numbers of I/Os are used to control peripherals (e.g., LEDs) or to exchange data through different communication protocols (UART, USB, SPI, etc.).

This chapter begins our exploration of CubeHAL with one of its simplest modules: `HAL_GPIO`. Although we have already used functions from this module in earlier examples, now is the time to delve into the full range of capabilities offered by this widely-used and straightforward peripheral. Before discussing specific HAL features, however, it is useful to understand how STM32 peripherals are mapped to logical addresses and represented within the HAL library.

6.1 STM32 Peripherals Mapping and HAL *Handlers*

Every STM32 peripheral is interconnected to the MCU core by several orders of buses, as shown in **Figure 6.1**[1].

[1]Here, to simplify this topic, we are considering the bus organization of one of the simplest STM32 microcontrollers, the STM32F072. STM32F4 and STM32F7, for example, have a more advanced bus interconnection system, which is outside the scope of this book. Please, always refer to the reference manual of your MCU.

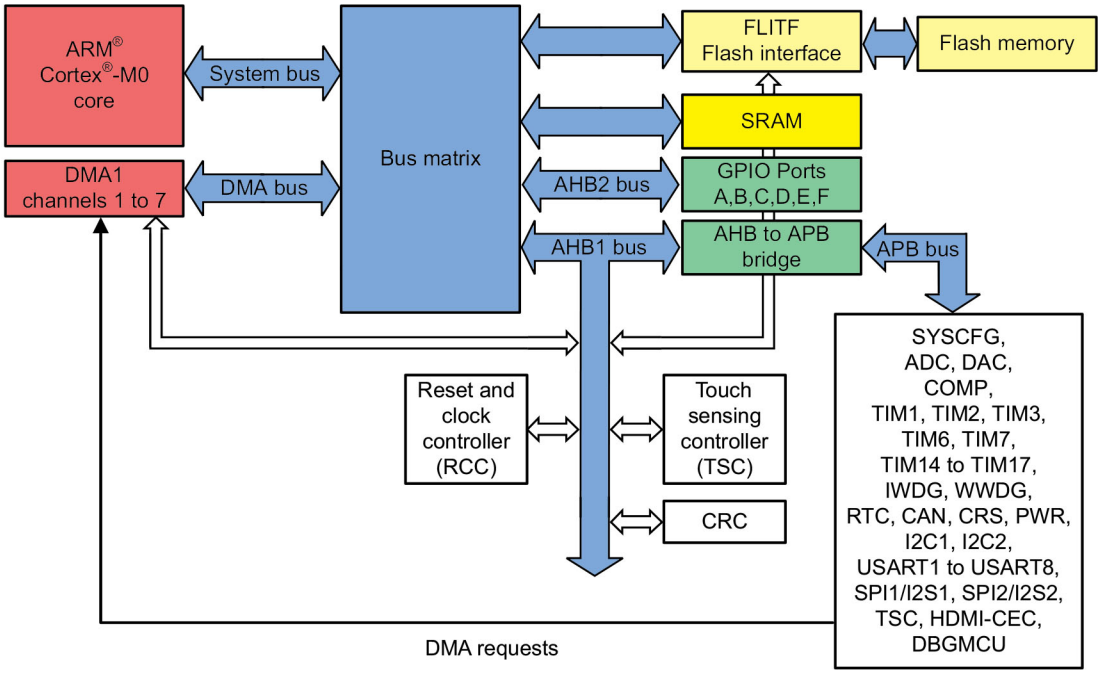


Figure 6.1: Bus architecture of an STM32F072 microcontroller

- The *System bus* connects the system bus of the Cortex-M core to a *Bus Matrix*, which manages arbitration between the core and the DMA. Both the core and the DMA act as masters.
- The *DMA bus* connects the *Advanced High-performance Bus* (AHB) master interface of the DMA to the Bus Matrix, which controls CPU and DMA access to SRAM, flash memory, and peripherals.
- The *Bus Matrix* manages access arbitration between the core system bus and the DMA master bus, using a Round Robin algorithm. It consists of two masters (CPU, DMA) and four slaves (FLASH memory interface, SRAM, AHB1 with an AHB-to-*Advanced Peripheral Bus* (APB) bridge, and AHB2). AHB peripherals are connected to the system bus through this Bus Matrix, enabling DMA access.
- The *AHB to APB bridge* provides fully synchronous connections between the AHB and the APB bus, where most peripherals are connected.

As we will see in a later chapter, each of these buses is connected to different clock sources, which determine the maximum speed for the peripherals connected to each bus[2].

In Chapter 1, we learned that peripherals are mapped to a specific region of the 4GB address space, starting from `0x4000 0000` and extending up to `0x5FFF FFFF`. This region is further divided into several sub-regions, each mapped to a specific peripheral, as shown in **Figure 6.2**.

[2]If the above description seems complex, do not worry; these concepts will become clearer as you progress through the chapter. Additional detail will also be covered in the chapter dedicated to the DMA.

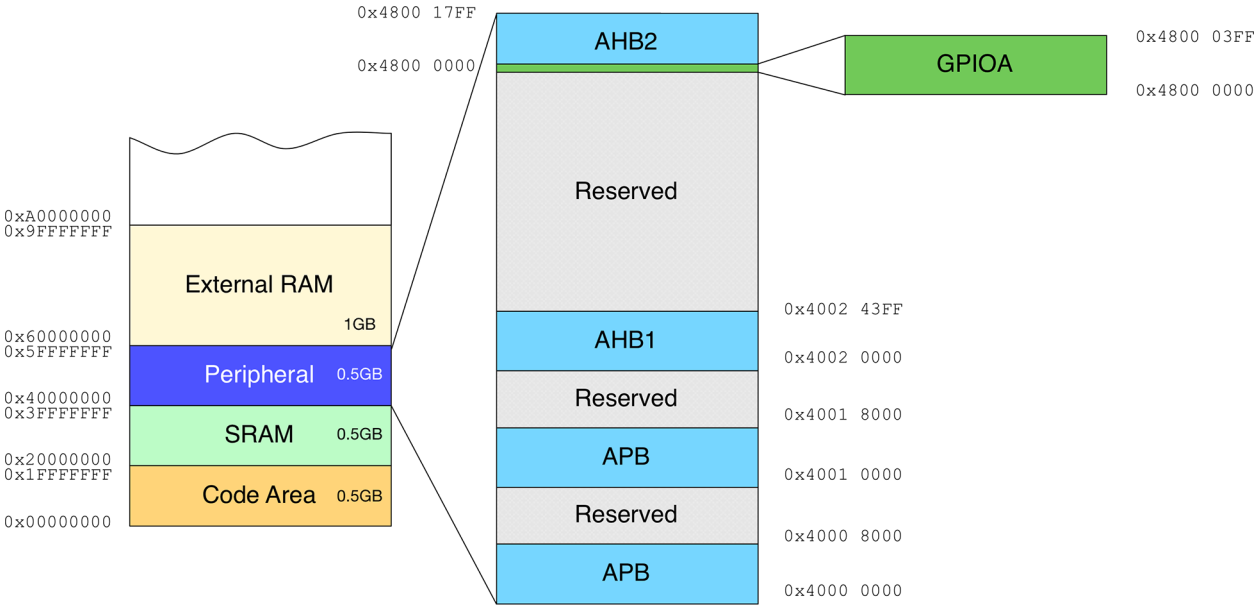


Figure 6.2: Memory map of peripheral regions for an STM32F072 microcontroller

The organization of this memory space, and thus the mapping of peripherals, is specific to each STM32 microcontroller. For instance, in an STM32F072 microcontroller, the AHB2 bus is mapped to the address range from `0x4800 0000` to `0x4800 17FF`. This range spans 6144 bytes. Within this range, the space is further divided into sub-regions, each corresponding to a specific peripheral. Continuing with this example, the `GPIOA` peripheral (which manages all pins connected to PORT-A) is mapped from `0x4800 0000` to `0x4800 03FF`, meaning it occupies 1KB of the aliased peripheral memory.

The way this memory-mapped space is organized depends on the specific peripheral. **Table 6.1**[3] provides the memory layout of a GPIO peripheral.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Figure 6.3: GPIO MODER register memory layout

[3]Both Table 6.1 and Figure 6.1 are sourced from the ST STM32F072 Reference Manual (https://bit.ly/2XzzJ3s).

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**GPIOA_MODER**	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x00	**GPIOx_MODER** (where x = **B..F**)	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x04	**GPIOx_OTYPER** (where x = **A..F**)	Res.		Res.		Res.		Res.		Res.		Res.		Res.		Res.		OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	**GPIOx_OSPEEDR** (where x = **B..F**)	OSPEEDR15[1:0]		OSPEEDR14[1:0]		OSPEEDR13[1:0]		OSPEEDR12[1:0]		OSPEEDR11[1:0]		OSPEEDR10[1:0]		OSPEEDR9[1:0]		OSPEEDR8[1:0]		OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1[1:0]		OSPEEDR0[1:0]	
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	**GPIOA_PUPDR**	PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]		PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
	Reset value	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	**GPIOx_IDR** (where x = **A..F**)	Res.		Res.		Res.		Res.		Res.		Res.		Res.		Res.		IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
	Reset value																	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0x14	**GPIOx_ODR** (where x = **A..F**)	Res.		Res.		Res.		Res.		Res.		Res.		Res.		Res.		ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x18	**GPIOx_BSRR** (where x = **A..F**)	BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0	BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x1C	**GPIOx_LCKR** (where x = **A..B**)	Res.		Res.		Res.		Res.		Res.		Res.		Res.		Res.		LCKK															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x20	**GPIOx_AFRL** (where x = **A.., B**)	AFRLAFR7[3:0]				AFRLAFR6[3:0]				AFRLAFR5[3:0]				AFRLAFR4[3:0]				AFRLAFR3[3:0]				AFRLAFR2[3:0]				AFRLAFR1[3:0]				AFRLAFR0[3:0]			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x24	**GPIOx_AFRH** (where x = **A..B**)	AFRHAFR15[3:0]				AFRHAFR14[3:0]				AFRHAFR13[3:0]				AFRHAFR12[3:0]				AFRHAFR11[3:0]				AFRHAFR10[3:0]				AFRHAFR9[3:0]				AFRHAFR8[3:0]			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x28	**GPIOx_BRR** (where x = **A..F**)	Res.		Res.		Res.		Res.		Res.		Res.		Res.		Res.		BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.1: GPIO peripheral memory map for an STM32F072 microcontroller

A peripheral is controlled by modifying and reading each register within these mapped regions. For example, with the `GPIOA` peripheral, to enable the PA5 pin as an output, we configure the `MODER`

register so that bits [11:10] are set to `01` (which corresponds to *General purpose output mode*), as shown in **Figure 6.3**. Next, to pull the pin high, we set the corresponding bit [5] inside the *Output Data Register* (ODR), which, according to **Table 6.1**, is mapped to the `GPIOA + 0x14` memory location—equivalent to `0x4800 0000 + 0x14`.

The following minimal example demonstrates how to use pointers to access the GPIOA peripheral mapped memory in an STM32F072 MCU.

```
int main(void) {
  volatile uint32_t *GPIOA_MODER = 0x0, *GPIOA_ODR = 0x0;

  GPIOA_MODER = (uint32_t*)0x48000000;        // Address of the GPIOA->MODER register
  GPIOA_ODR = (uint32_t*)(0x48000000 + 0x14); // Address of the GPIOA->ODR register

  //This ensures that the peripheral is enabled and connected to the AHB1 bus
  __HAL_RCC_GPIOA_CLK_ENABLE();

  *GPIOA_MODER = *GPIOA_MODER | 0x400; // Sets MODER[11:10] = 0x1
  *GPIOA_ODR = *GPIOA_ODR | 0x20;      // Sets ODR[5] = 0x1, that is pulls PA5 high
  while(1);
}
```

It is important to clarify once again that each STM32 family (e.g., F0, F1, etc.) and each member within a given family (e.g., STM32F072, STM32F103, etc.) provides its own subset of peripherals, mapped to specific addresses. Furthermore, the implementation of these peripherals varies between STM32 series.

One of the roles of the HAL is to abstract from specific peripheral mappings. This is achieved by defining various *handlers* for each peripheral. A handler is simply a C `struct`, used as a reference to point to the actual peripheral address. Let us examine one of these handlers.

In previous chapters, we configured the PA5 pin with the following code:

```
/*Configure GPIO pin : PA5 */
GPIO_InitStruct.Pin = GPIO_PIN_5;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

Here, the `GPIOA` variable is a pointer of type `GPIO_TypeDef` defined in this way:

```
typedef struct {
        volatile uint32_t MODER;
        volatile uint32_t OTYPER;
        volatile uint32_t OSPEEDR;
        volatile uint32_t PUPDR;
        volatile uint32_t IDR;
        volatile uint32_t ODR;
        volatile uint32_t BSRR;
        volatile uint32_t LCKR;
        volatile uint32_t AFR[2];
        volatile uint32_t BRR;
} GPIO_TypeDef;
```

The `GPIOA` pointer is defined so that it points[4] to the address `0x4800 0000`:

```
GPIO_TypeDef *GPIOA = 0x48000000;

GPIOA->MODER |= 0x400;
GPIOA->ODR   |= 0x20;
```

The rest of the chapter is not available in the book sample

[4] This is not entirely accurate, as the HAL, to save RAM space, defines `GPIOA` as a macro (`#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)`).

7. Interrupts Management

Hardware management involves handling asynchronous events, most of which originate from hardware peripherals. For example, a timer reaching a configured period value or a UART indicating the arrival of data. Other events are triggered by external interactions, such as a user pressing a switch that, unpredictably, causes the board to hang, leading to a day spent troubleshooting.

All microcontrollers offer a feature called *interrupts*. An interrupt is an asynchronous event that temporarily halts the current code execution based on priority (higher-priority interrupts can suspend lower-priority ones). The code that responds to the interrupt is called the *Interrupt Service Routine* (ISR).

Interrupts enable multiprogramming: the hardware handles them by saving the current execution context (such as the stack frame, the *Program Counter* (PC), and a few other essentials) before switching to the ISR. Interrupts are essential in *Real-Time Operating Systems* (RTOS) for introducing *tasks*. Without hardware support, a true preemptive system — one that switches between multiple execution contexts without losing the current execution flow — would be impossible.

Interrupts can be triggered by both hardware and software. The ARM architecture differentiates between these: *interrupts* originate from hardware, while *exceptions* originate from software (e.g., an access to an invalid memory location). In ARM terminology, an interrupt is a specific type of exception.

Cortex-M processors include a dedicated unit for exception management, called the *Nested Vectored Interrupt Controller* (NVIC). This chapter focuses on programming this essential hardware component to manage interrupts. Exception handling, however, will be covered in Chapter 24, which discusses advanced debugging techniques.

7.1 NVIC Controller

The NVIC is a dedicated hardware unit within Cortex-M-based microcontrollers responsible for handling exceptions. **Figure 7.1** illustrates the relationship between the NVIC unit, the processor core, and peripherals. Here, we distinguish between two types of peripherals: those that are external to the Cortex-M core but internal to the STM32 MCU (e.g., timers, UARTs), and peripherals that are entirely external to the MCU. Interrupts from the latter type of peripherals are triggered by MCU I/O, which can be configured either as general-purpose I/O (e.g., a tactile switch connected to a pin configured as an input) or to interface with an external advanced peripheral (e.g., I/Os configured to exchange data with an *Ethernet PHY* via the RMII interface). A dedicated programmable controller, called the *External Interrupt/Event Controller* (EXTI), manages the connection between external I/O signals and the NVIC controller, as we will see shortly.

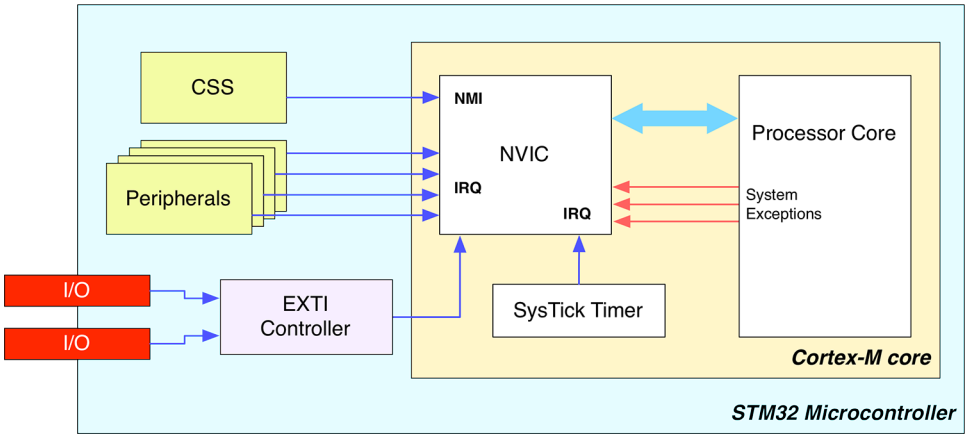


Figure 7.1: The relation between the NVIC controller, the Cortex-M core, and STM32 peripherals

As mentioned, ARM differentiates between system exceptions, which originate within the CPU core, and hardware exceptions from external peripherals, also known as *Interrupt Requests* (IRQ). Programmers manage exceptions by writing specific ISRs, typically coded at a higher level (most often in C). The processor knows where to locate these routines using an indirect table containing the memory addresses of the Interrupt Service Routines. This table is commonly referred to as the *vector table*, and each STM32 microcontroller defines its own. Let us examine this in depth.

7.1.1 Vector Table in STM32

All Cortex-M processors reserve a fixed set of fifteen exceptions common to all Cortex-M families. However, not all of these exceptions are currently defined (they are marked as **RESERVED** in the official ARM documentation), and only a subset is available in Cortex-M0/0+ cores. We first encountered these exceptions in Chapter 1. For convenience, **Table 7.1** below provides the same information. Let us review these exceptions (we will cover fault exceptions in more detail in Chapter 24 on advanced debugging).

- **Reset**: This exception is raised immediately after the CPU resets. Its handler serves as the entry point for the running firmware. In an STM32 application, everything starts from this exception. The handler includes assembly-coded functions that initialize the execution environment, such as the main stack, the `.bss` area, and more. Chapter 22 on the booting process covers this in depth.
- **NMI**: This is a special exception with the highest priority after the *Reset* exception. Like *Reset*, it cannot be masked and is typically associated with critical, non-deferrable activities. In STM32 microcontrollers, it is linked to the *Clock Security System* (CSS). CSS is a diagnostic feature that detects failure of the external clock, called HSE. If this occurs, HSE is disabled (automatically enabling the internal HSI), and an NMI interrupt is raised to alert the software of an HSE failure. More details are provided in Chapter 10.
- **Hard Fault**: This is the general fault exception, related to software interrupts. When other fault exceptions are disabled, it acts as a catch-all for all types of exceptions (e.g., if a memory access

to an invalid location raises a Bus Fault and that exception is disabled, the Hard Fault exception will handle it).

- **Memory Management Fault**[1]: This occurs when code attempts to access an illegal location or violates a rule of the Memory Protection Unit (MPU). This is covered further in Chapter 20.
- **Bus Fault**[2]: This occurs when the AHB interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch, or *data abort* if it is a data access). It can also be caused by illegal accesses (e.g., accessing a non-existent SRAM memory location).
- **Usage Fault**[3]: This occurs due to program errors such as illegal instructions, alignment issues, or attempts to access a non-existent co-processor.
- **SVCCall**: This is not a fault condition but is raised when the *Supervisor Call* (SVC) instruction is executed. RTOS uses this exception to perform privileged operations (a task requiring privileged operations executes the SVC instruction, and the OS performs the requested operation, similar to a system call in other OSes).
- **Debug Monitor**[4]: This exception is raised for software debug events when the core is in Monitor Debug-Mode. It also handles debug events like breakpoints and watchpoints for software-based debugging.
- **PendSV**: This exception is used in RTOS. Unlike the SVCall exception, which is executed immediately after an SVC instruction, PendSV can be delayed, allowing the RTOS to complete higher-priority tasks.
- **SysTick**: This exception is also commonly used in RTOS tasks. Every RTOS requires a timer to periodically interrupt the current code execution and switch to another task. All STM32 microcontrollers provide a *SysTick* timer within the Cortex-M core. Although other timers can be used for scheduling, the dedicated SysTick timer ensures portability across STM32 families (as not all timers are externally available in all models due to MCU die optimization). Additionally, even if an RTOS is not used, it is important to note that ST CubeHAL relies on the *SysTick* timer for time-related activities (**and assumes that the SysTick timer is configured to generate an interrupt every 1 ms**).

The remaining exceptions that can be defined for a given MCU are related to IRQ handling. Cortex-M0/0+ cores support up to 32 external interrupts, Cortex-M3/4/7 cores allow silicon manufacturers to define up to 240 interrupts, and Cortex-M33 cores support up to 480 IRQ lines.

Where can we find the list of usable interrupts for a specific STM32 microcontroller? The primary source of information is the MCU's datasheet, which details the available interrupts. However, we can also refer to the *vector table* provided by ST in its HAL. This table is defined in the startup file for our MCU, located in the `Core/Startup` folder of our project as an assembly file with a `.s` extension (e.g., for an STM32F446RET MCU, the file is `startup_stm32f446retx.s`). Opening this file, we can find the complete vector table for that MCU, starting around line 128 (see the example in Chapter 4).

[1]This exception is not available in Cortex-M0/0+ based microcontrollers.
[2]This exception is not available in Cortex-M0/0+ based microcontrollers.
[3]This exception is not available in Cortex-M0/0+ based microcontrollers.
[4]This exception is not available in Cortex-M0/0+ based microcontrollers.

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 7.1: Cortex-M exception types

Even though the *vector table* contains the addresses of the handler routines (it is, in fact, an indirect table), the Cortex-M core requires a method to locate the *vector table* in memory. By convention, the *vector table* begins at the hardware address `0x0000 0000` in all Cortex-M processors.

If our firmware places the *vector table* in internal flash memory (a common scenario), it will start at address `0x0800 0000` in all STM32 MCUs. However, as noted in Chapter 1, the `0x0800 0000` address is automatically aliased to `0x0000 0000` when the CPU boots up[5].

[5]Except for the Cortex-M0, other Cortex-M cores allow the *vector table* position in memory to be *relocated*. It is also possible to configure the MCU to boot from different memory regions besides internal flash. These advanced topics will be discussed in Chapter 20 on memory layout and in Chapter 22 on the booting process. To simplify, we will assume here that the *vector table* position is fixed at `0x0000 0000`.

Figure 7.2 illustrates the organization of the *vector table* in memory. The first entry in this array is the address of the *Main Stack Pointer* (MSP) within the SRAM. Typically, this address corresponds to the end of SRAM, calculated as its base address plus its size (more on STM32 memory layout in Chapter 20). From the second entry onward, the table lists the addresses for exception and interrupt handlers. Thus, the *vector table* length is 48 entries for Cortex-M0/0+ based microcontrollers and 256 entries for Cortex-M3/4/7.

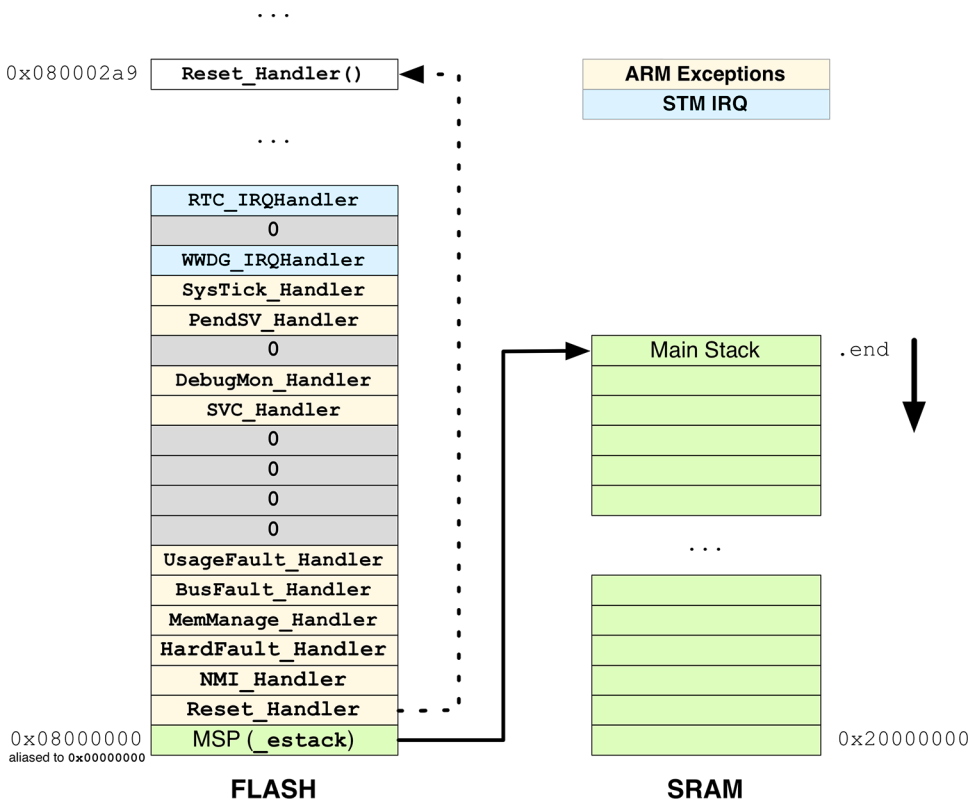


Figure 7.2: The minimal layout of the *vector table* in an STM32 MCU based on a Cortex-M3/4/7 core

It is important to clarify some details about the *vector table*.

1. The names of the exception handlers are simply conventions, and you are free to rename them if you prefer different names. They are merely *symbols* (similar to variables and functions in a program). However, keep in mind that CubeMX generates ISRs with specific names following ST conventions, so if you rename a handler, you must also update the ISR name in the CubeMX-generated code.
2. As mentioned, the *vector table* must be located at the beginning of flash memory, where the processor expects it. This placement is handled by the GCC linker, which positions the *vector table* at the start of the flash section when generating the *absolute file* — the binary file we upload to flash memory. In Chapter 20, we will explore the content of the `STM32XXxx_FLASH.ld` file, which includes the directives that instruct GNU LD on this configuration.

The rest of the chapter is not available in the book sample

8. Universal Asynchronous Serial Communications

The electronics industry today offers a wide range of serial communication protocols and hardware interfaces. Many of these focus on achieving high data transmission rates, as seen in recent standards such as USB 2.0 and 3.x, as well as FireWire (IEEE 1394). While some standards have been around for years, they remain prevalent, particularly for communication between modules on the same circuit board. One such enduring interface is the *Universal Synchronous/Asynchronous Receiver/Transmitter*, commonly abbreviated as USART.

Almost every modern microcontroller includes at least one UART peripheral. In the STM32 microcontroller family, nearly all models provide at least two UART/USART interfaces, with many offering more — some supporting up to eight interfaces, depending on the I/O capacity of the specific MCU package.

In this chapter, we will explore how to program this versatile peripheral using the CubeHAL library. We will examine application development with UART in both *polling* and *interrupt* modes, with the third operational mode, *DMA*, covered in greater detail in the next chapter.

8.1 Introduction to UARTs and USARTs

Before diving into the analysis of the functions provided by the HAL to manage universal serial devices, it is helpful to first examine the UART/USART interface and its communication protocol.

When we want to exchange data between two (or more) devices, we have two primary options: transmit data in parallel—using a set of communication lines equal to the data word size (e.g., eight lines for an 8-bit word)—or transmit each bit of the word sequentially over a single line. A UART/USART device translates a parallel sequence of bits, typically grouped in a byte, into a continuous stream of signals transmitted over a single wire.

When information flows between two devices over a common channel, both devices (referred to here as *the sender* and *the receiver*) must agree on the *timing*, which determines how long it takes to transmit each individual bit. In **synchronous transmission**, the sender and receiver share a common clock signal, generated by one of the devices—typically the one acting as *the master* in this communication system.

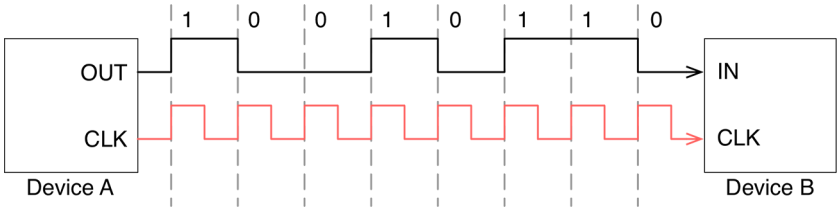


Figure 8.1: A serial communication between two devices using a shared clock source

In **Figure 8.1**, we see a typical timing diagram[1] illustrating *Device A* sending one byte (`0b01101001`) serially to *Device B* using a common reference clock. This shared clock synchronizes the start of *sampling* for the bit sequence: when the master device begins *clocking* the designated line, it signals the start of a bit sequence transmission.

In **synchronous transmission**, the transmission speed and duration are dictated by the clock frequency, which determines how quickly a single byte can be transmitted over the communication channel[2]. However, if both devices agree on the timing of each bit is transmission and the start and stop points for sampling transmitted bits, they can avoid using a dedicated clock line. This setup is referred to as **asynchronous transmission**.

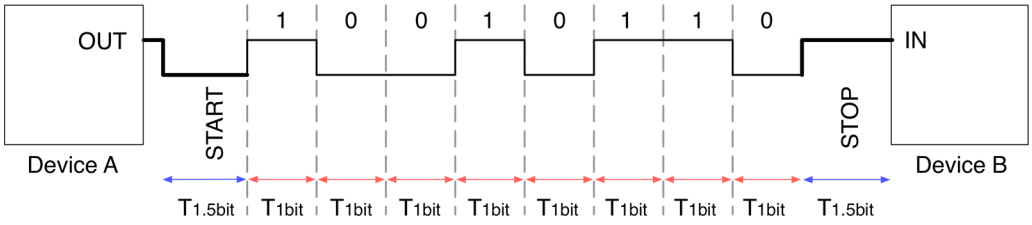


Figure 8.2: The timing diagram of a serial communication without a dedicated clock line

Figure 8.2 illustrates the timing diagram of an asynchronous transmission. The idle state (when no transmission is occurring) is represented by a high signal level. Transmission begins with a **START** bit, marked by a low-level signal. The receiver detects the negative edge, and 1.5 bit periods after this transition (as shown in Figure 8.7.1s $T_{1.5bit}$), it starts sampling the incoming bits.

Eight data bits are sampled sequentially, with the least significant bit (LSB) typically transmitted first. Following the data bits, an optional parity bit may be included for error checking. This parity bit is often omitted if the transmission channel is assumed to be noise-free or if error checking is handled in higher protocol layers. The transmission concludes with a STOP bit, which lasts for 1.5 bit periods.

[1] A Timing Diagram is a representation of a set of signals over time.

[2] However, keep in mind that the maximum transmission speed is influenced by several factors, such as the electrical characteristics of the channel, the ability of each device involved in transmission to sample high-speed signals, and other related parameters.

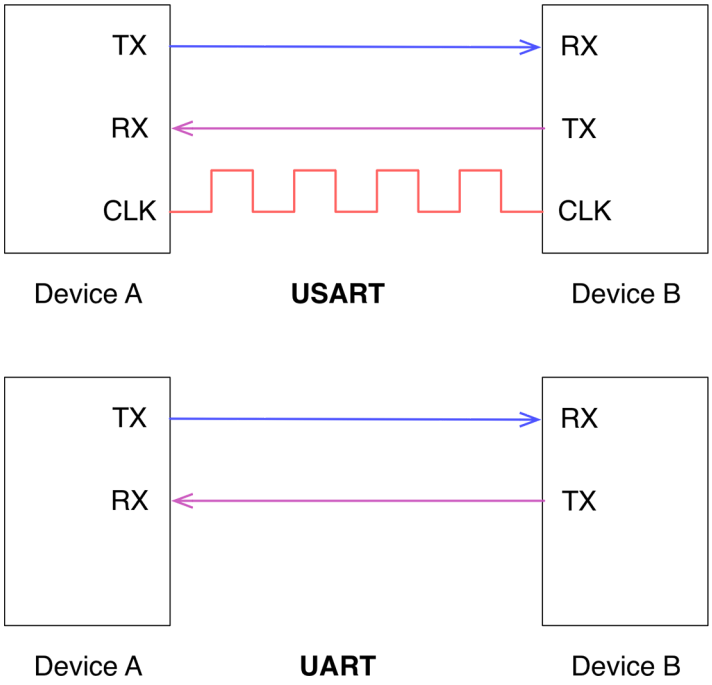


Figure 8.3: The signaling difference between a USART and a UART

A *Universal Synchronous Receiver/Transmitter* (USART) interface is a device capable of transmitting data words serially using two I/Os — one acting as the transmitter (TX) and one as the receiver (RX) — plus an additional I/O as a clock line. In contrast, a *Universal Asynchronous Receiver/Transmitter* (UART) requires only two I/Os, TX and RX (see **Figure 8.3**). Conventionally, we refer to the first interface as **USART** and to the second as **UART**.

While a UART/USART defines the signaling method, it does not specify voltage levels. This means that an STM32 UART/USART will use the voltage levels of the MCU's I/Os, typically close to VDD (commonly known as *TTL voltage levels*). Translating these voltage levels for serial communication beyond the board requires other communication standards. For example, EIA-RS232 and EIA-RS485 are two popular standards that define not only signaling voltages but also timing, signal meaning, and the physical specifications of connectors. Additionally, UART/USART interfaces can facilitate data exchange across other physical and logical serial interfaces. For instance, the FT232RL is a widely-used IC that maps a UART to a USB interface, as illustrated in **Figure 8.4**.

The presence of a dedicated clock line or an agreed transmission frequency does not ensure that the receiver can process data at the same rate as the sender. For this reason, some communication standards, such as RS232 and RS485, allow the use of a dedicated *Hardware Flow Control* line. For example, two devices communicating over an RS232 interface may share two additional lines, *Request To Send* (RTS) and *Clear To Send* (CTS). The sender sets its RTS line, signaling the receiver to monitor its data input line. When ready to receive data, the receiver raises its complementary CTS line, signaling the sender to start transmission and to monitor the receiver's data output line.

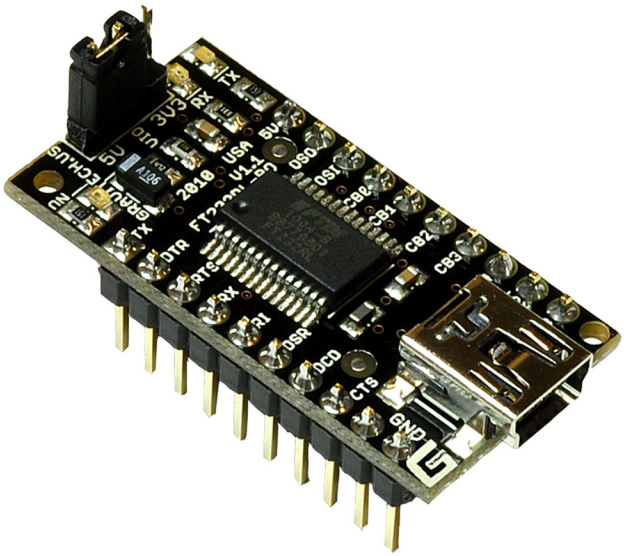


Figure 8.4: A typical circuit based on FT232RL used to convert a 3.3V TTL UART interface to USB

STM32 microcontrollers offer a variable number of USART interfaces, which can be configured to operate in both *synchronous* and *asynchronous* modes. Some STM32 MCUs also provide interfaces limited to UART functionality. **Table 8.1** lists the UART/USART interfaces available on STM32 MCUs featured in all Nucleo boards used in this text. Most USARTs also support automatic *Hardware Flow Control*, compatible with both the RS232 and RS485 standards.

All Nucleo-64 boards are designed so that the USART2 of the target MCU is connected to the ST-LINK interface[3]. When the ST-LINK drivers are installed, an additional driver for the *Virtual COM Port* (VCP) is also installed, enabling access to the target MCU's USART2 over USB without the need for a dedicated TTL/USB converter. By using a terminal emulation program, we can communicate with our Nucleo board to exchange messages and data.

The CubeHAL library distinguishes between APIs for managing UART and USART interfaces. Functions and data types for USARTs are prefixed with `HAL_USART` and located in `stm32xxx_hal_usart.{c,h}`, while those for UARTs use the `HAL_UART` prefix and reside in `stm32xxx_hal_uart.{c,h}`. Since both modules are conceptually similar, and UART is the most common form of serial interconnection between modules, this book will focus on the `HAL_UART` module.

[3]Note that this may not apply if you are using a Nucleo-32 or Nucleo-144 board. Consult the ST documentation for details.

Nucleo P/N	USARTs + UARTs	USART#	HW Flow Control RS232	HW Flow Control RS485
NUCLEO-F446RE	4 + 2	USART1/2/3	Y	-
		USART6	-	-
		UART4/5	Y	-
NUCLEO-G474RE	3 + 2	USART1/2/3	Y	Y
		UART4/5	Y	Y
NUCLEO-F401RE	3 + 0	USART1/2	Y	-
		USART6	-	-
NUCLEO-F303RE	3 + 2	USART1/2/3	Y	Y
		UART4/5	-	-
NUCLEO-F103RB	3 + 0	USART1/2/3	Y	-
NUCLEO-F072RB	4 + 0	USART1/2/3/4	Y	Y
NUCLEO-L476RG	3 + 2	USART1/2/3	Y	Y
		UART4/5	Y	Y
NUCLEO-L152RE	3 + 2	USART1/2/3	Y	-
		UART4/5	-	-
NUCLEO-L073RZ	4 + 2	USART1/2/4	Y	Y
		UART5	*RTS Only*	Y

Table 8.1: The list of available USARTs and UARTs on all Nucleo boards

The rest of the chapter is not available in the book sample

9. Memory layout

Every time we compile our firmware using the GCC ARM tool-chain, a series of non-trivial things takes place. The compiler translates the C source code in the ARM assembly and organizes it to be flashed on a given STM32 MCU. Every microprocessor architecture defines an execution model that needs to be "matched" with the execution model of the C programming language. This means that several operations are performed during bootstrap, whose task is to prepare the execution environment for our application: the stack and heap creation, the initialization of data memory, the *vector table* initialization are just some of the activities performed during startup. Moreover, some STM32 microcontrollers provide additional memories, or allow to interface external ones using the FSMC controller, that can be assigned to specific tasks during the firmware lifecycle.

This chapter aims to throw light to those questions that are common to a lot of STM32 developers. What does it happen when the MCU resets? Why providing the `main()` function is mandatory? And how long does it take to execute since the MCU resets? How to store variables in flash instead of SRAM? How to use the STM32 CCM memory?

9.1 The STM32 Memory Layout Model

In Chapter 1 we have analyzed the typical memory layout of an STM32 microcontroller. Figure 1.4 shows that the first 1GB address space is divided between the FLASH and the SRAM memories. These memory areas are in turn subdivided in some several sub-regions. Let us analyze the way they are organized in a typical STM32 application by taking as reference the **Figure 20.1**.

9.1.1 Flash Memory Typical Organization

In an STM32 microcontroller, the internal flash memory is mapped starting from the address `0x0800 0000`[1]. In Chapter 7 we learned that the very initial bytes of flash memory are dedicated to the *Main Stack Pointer* (MSP). The MSP contains the address in SRAM where the stack begins. The Cortex-M architecture gives maximum freedom of placing the stack in the SRAM memory as well as in other internal memories (for example, the CCM RAM available in some STM32 MCUs) or external ones (connected to the FSMC controller). This explains the need for the MSP.
The Cortex-M architecture defines that the memory locations right after the MSP are dedicated to the *vector table*, a sequence of 32-bit addresses pointing to the ISR routines. The length of this table depends on the Cortex-M architecture, as seen in **Table 7.1**.

Apart from these architectural constraints, that can be "relaxed" in such a way as we will see later in this chapter, the compiler is free to arrange the rest of flash memory according to the programming

[1]Remember that, as we will see next, the Cortex-M architecture defines the `0x0000 0000` address as the memory location where starting to place MSP and *vector table*. This means that the flash starting address (`0x0800 0000`) is aliased to `0x0000 0000`.

language execution model. In a typical ARM-GCC C application, usually the rest of flash memory is used to store program code, read-only data (also known as *const data*, since variables declared as `const` are automatically placed in this memory) and initialization data, that is the initialization values of variables in SRAM.

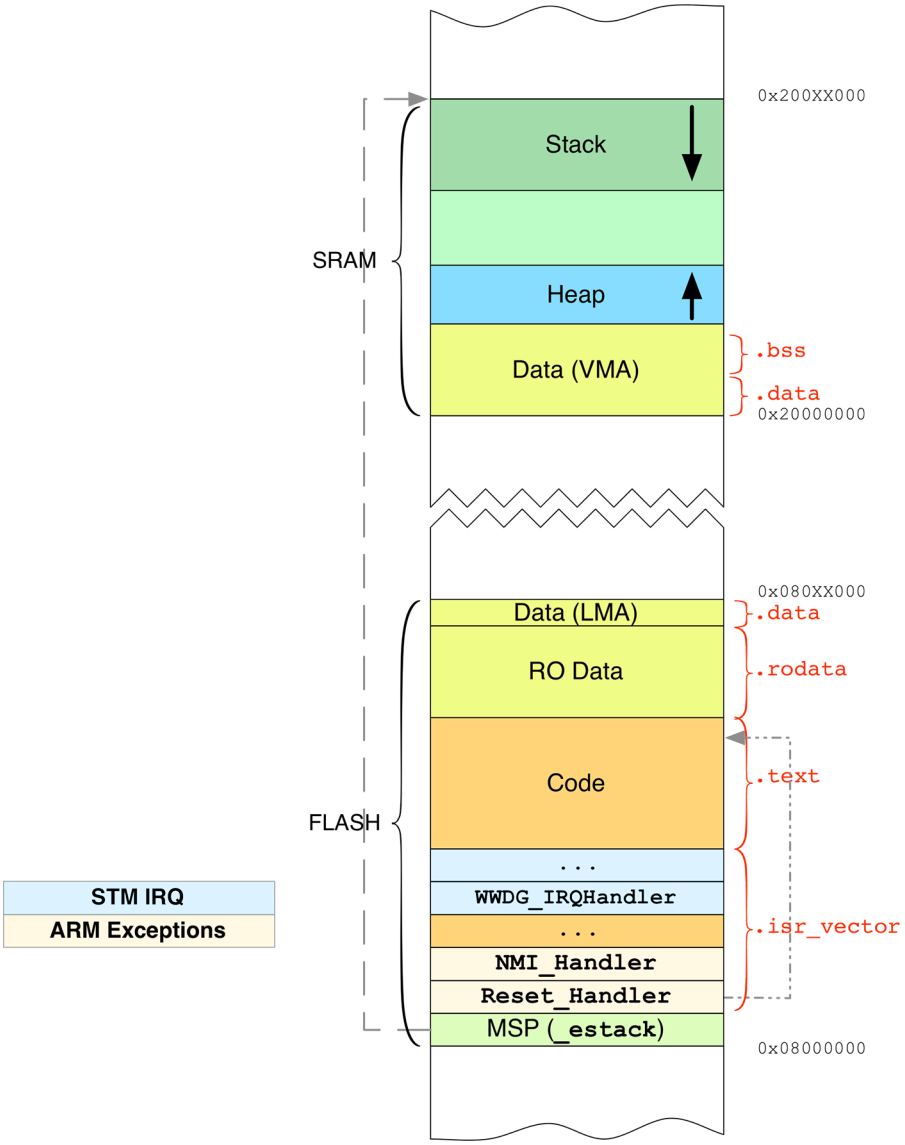


Figure 20.1: The typical layout of flash and SRAM memories

From the compiler point of view, these sections are traditionally named in a different way inside the application binary. For example, the section containing assembly code is named `.text`, `.rodata` is the one containing `const` variables and strings, while the section for initialized data is named `.data`. These names are also common to other computer architectures, like x86 and MIPS. Others are specific of "microcontrollers world". For example, the `.isr_vector` section is the one designated to store the *vector table* in Cortex-M based MCUs. The number and the naming of these sections is, however, well defined and they adhere to a more general specification called ARM *Embedded*

Application Binary Interface (EABI). This specification states how many and what kind of sections an ELF[2] binary file must provide, so that all the firmware application can be properly loaded and executed on a given Cortex-M architecture.

9.1.2 SRAM Memory Typical Organization

The internal SRAM memory is mapped starting from the `0x2000 0000` address and it is also organized in several sub-regions. A variable-sized region starting from the **end** of SRAM and growing downwards (that is, its base address has the highest SRAM address) is dedicated to the *stack*. This happens because Cortex-M cores use a stack memory model called *full-descending stack*. The base stack pointer, that is the MSP, is computed at compile time, and it is stored at `0x0800 0000` flash memory location, as seen before. Once we call a function, a new *stack frame* is pushed on the stack. This means that the pointer to the current stack frame (SP) is automatically decremented at every function call (this means that the ARM assembly `push` instruction automatically decrements it).

The SRAM is also used to store variable data, and this region usually starts at beginning of SRAM (`0x2000 0000`). This region is in turn divided between *initialized* and *un-initialized* data. To understand the difference, let us consider this code fragment:

```
...
uint8_t var1 = 0xEF;
uint8_t var2;
...
```

`var1` and `var2` are two global variables. `var1` is an initialized variable (we fix its starting value at compile time), while the value `var2` is un-initialized: during the very first instructions after an MCU reset, a set of dedicated routines initialize them by setting `var2` to zero and `var1` to the value stored in `.data` section inside the flash memory. We will study these operations later in this chapter.

Finally, the SRAM memory could contain another growing region: the *heap*. It stores variables that are allocated dynamically during the execution of the firmware (by using the C `malloc()` routine or similar). This area can be in turn organized in several sub-regions, according to the *allocator* used (in the next chapter we will see how FreeRTOS provides several allocators to handle dynamic memory allocation). The heap grows upwards (that is, the base address is the lowest in its region) and it has a fixed maximum size.

Since every STM32 MCU has its own quantity of SRAM and flash, and since every program has a variable number of instructions and variables, the dimension and location in memory of these sections differ among several MCUs. Before we can see how to instruct the compiler to generate the binary file for the specific MCU, we have to understand all the steps and tools involved during the generation of *object files*.

[2]ELF is acronym for *Executable and Linkable Format* and it is a common standard file format for executable files, object code, shared libraries, and core dumps. It is the typical file format of UNIX like systems (Linux and MacOS use this format too) and ARM based environments.

The rest of the chapter is not available in the book sample

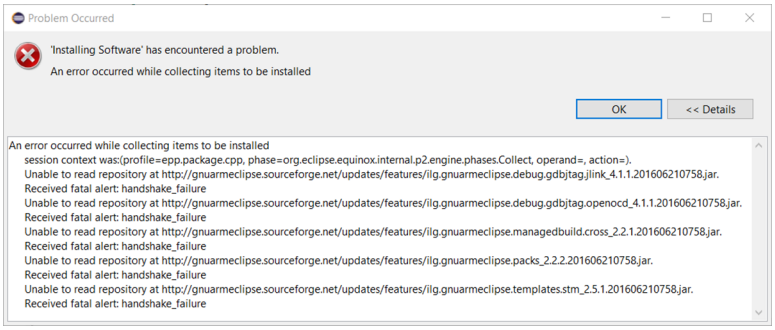
III Appendix

B. Troubleshooting guide

Here you can find common issues already reported from other readers. Before posting from any kind of problem you can encounter, it is a good think to have a look here.

GNU MCU Eclipse Installation Issues

Several readers are reporting me issues in installing GNU MCU Eclipse plug-ins. During the installation, Eclipse cannot access to the packages repository, and the following error appears:



This error is caused by Java, which does not support natively strong encryption due to limitations to cryptographic algorithms in some countries. The workaround is described in this stackoverflow answer: http://stackoverflow.com/a/38264878. Essentially, you need to download an additional package (http://bit.ly/2jiC7GE) from the Java website; extract the ".zip" file and copy the content of the `UnlimitedJCEPolicyJDK8` directory inside the following dir:

- In Windows: `C:\Program Files\Java\jre1.8.0_121\lib\security`
- In Linux:`/usr/lib/jvm/java-8-oracle/lib/security`
- In MacOS: `/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/jre/lib/security`

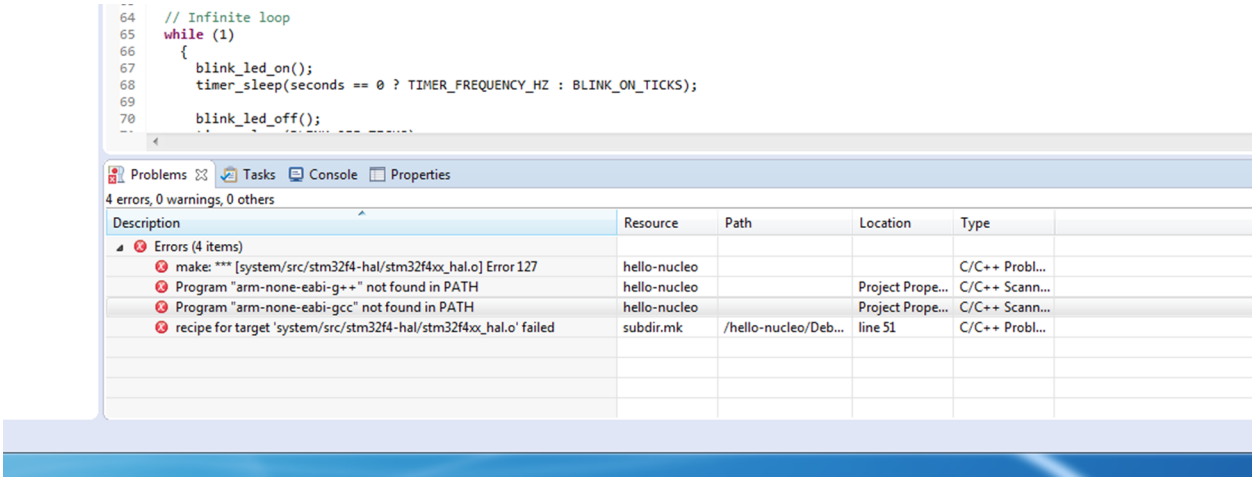
Restart Eclipse. You should be able to install GNU MCU Eclipse plug-ins now.

Eclipse related issue

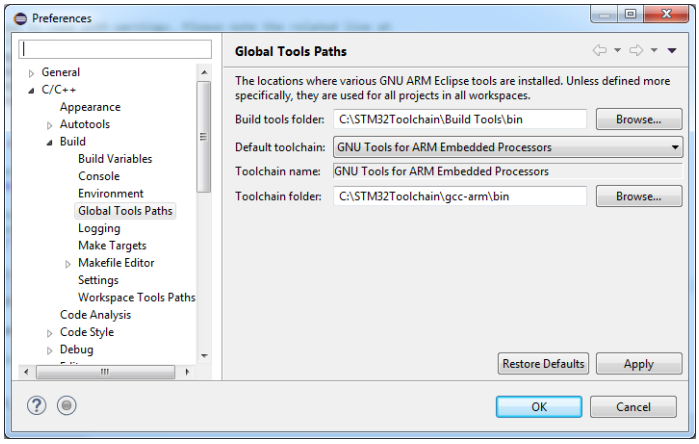
This section contains a list of frequently issues related with the Eclipse IDE.

Eclipse cannot locate the compiler

This is a problem that happens frequently on Windows. Eclipse cannot find the compiler installation folder, and it generates compiling errors like the ones shown below.



This happens because the GNU MCU plug-in cannot locate the GNU cross-compiler folder. To address this issue, open the Eclipse preferences clicking on the **Window->Preferences** menu, then go to **C/C++->Build->Global Tools Paths** section. Ensure that the **Build tools folder** path points to the directory containing the Build Tools (`C:\STM32Toolchain\Build Tools\bin` if you followed the instructions in Chapter 3, or arrange the path accordingly), and the **Toolchain folder** paths point to the GCC ARM installation folder (`C:\STM32Toolchain\gcc-arm\bin`). The following image shows the right configuration:



The rest of the chapter is not available in the book sample

C. Nucleo pin-out

In the next paragraphs, you can find the correct pin-out for all Nucleo boards. The pictures are taken from the mbed.org website[3].

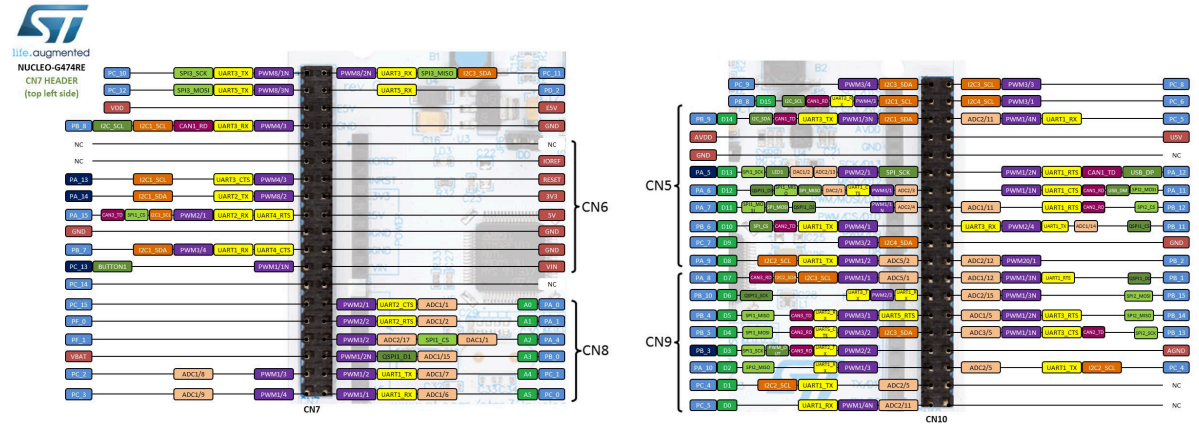
Nucleo Release
Nucleo-G474RE
Nucleo-F446RE
Nucleo-F401RE
Nucleo-F303RE
Nucleo-F103RB
Nucleo-F072RB
Nucleo-L476RG
Nucleo-L152RE
Nucleo-L073RZ

[3] https://developer.mbed.org/platforms/?tvend=10

Nucleo-G474RE

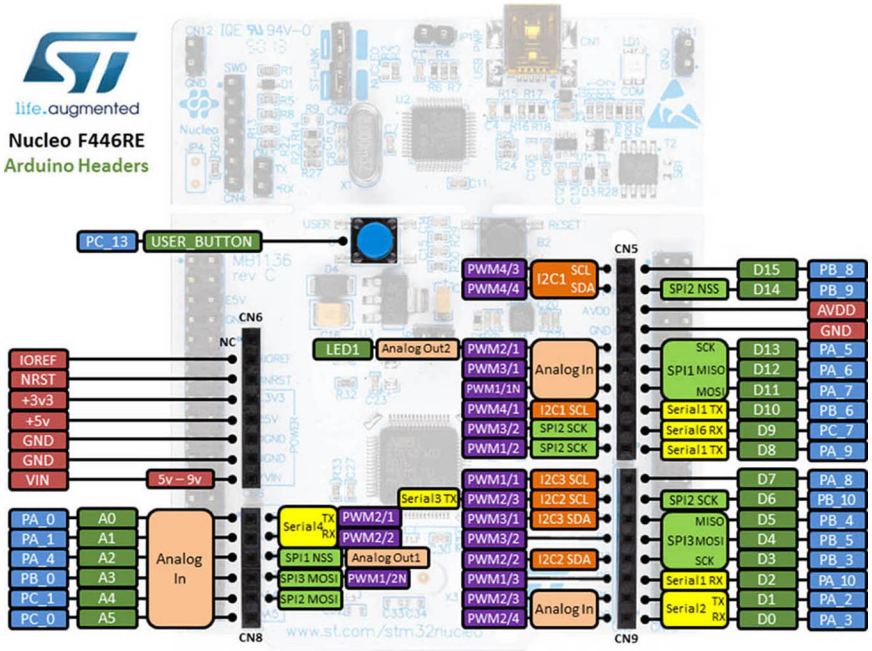
Arduino compatible headers

Morpho headers

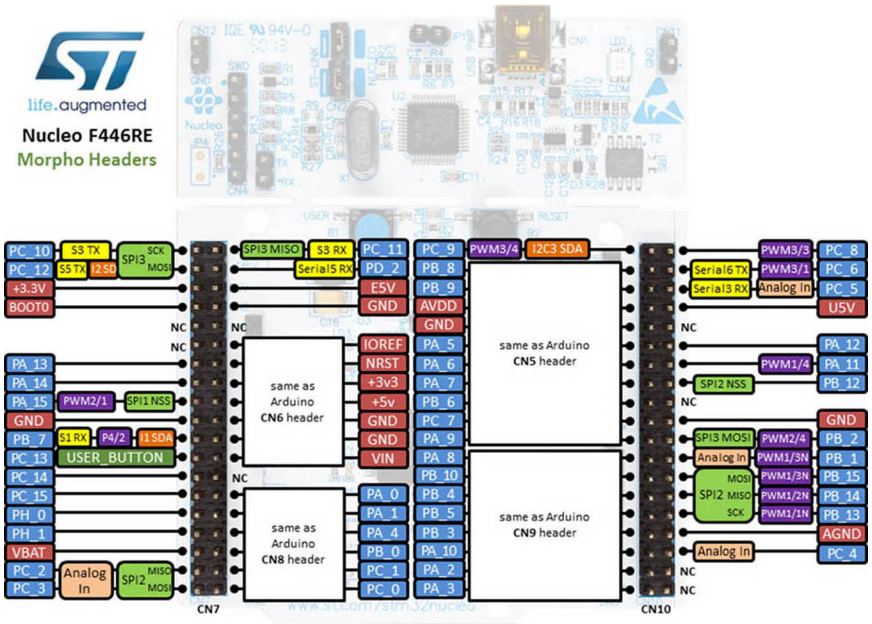


Nucleo-F446RE

Arduino compatible headers

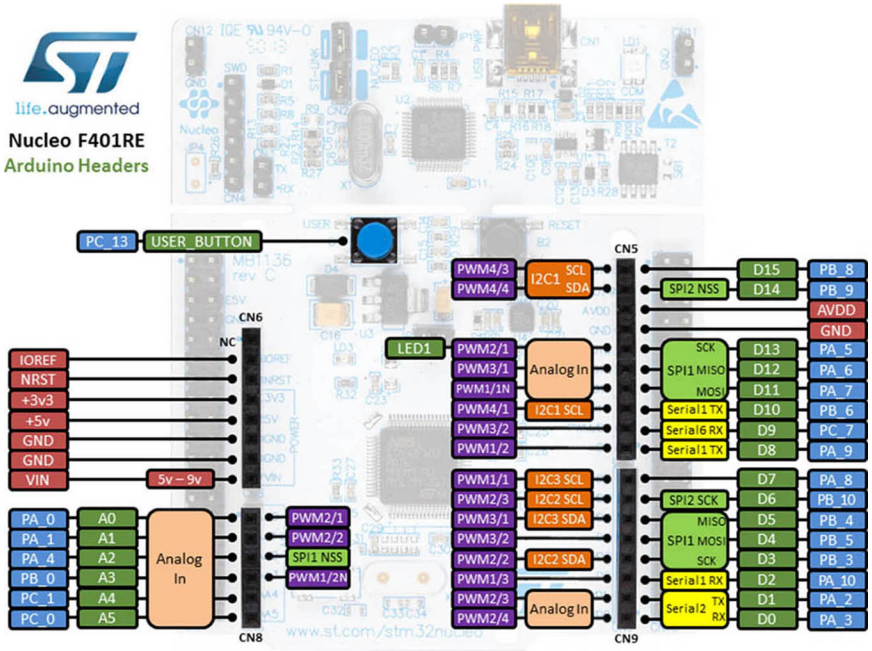


Morpho headers

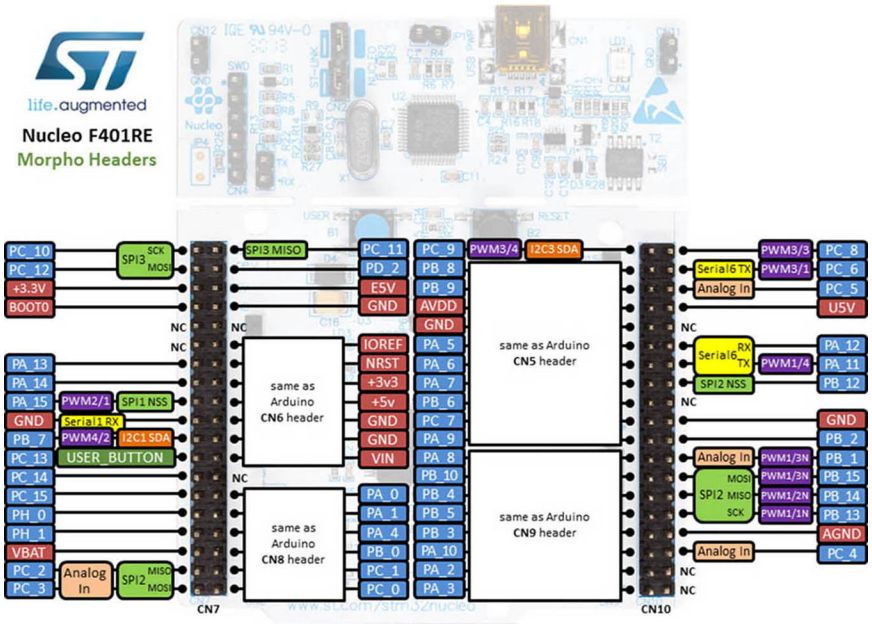


Nucleo-F401RE

Arduino compatible headers

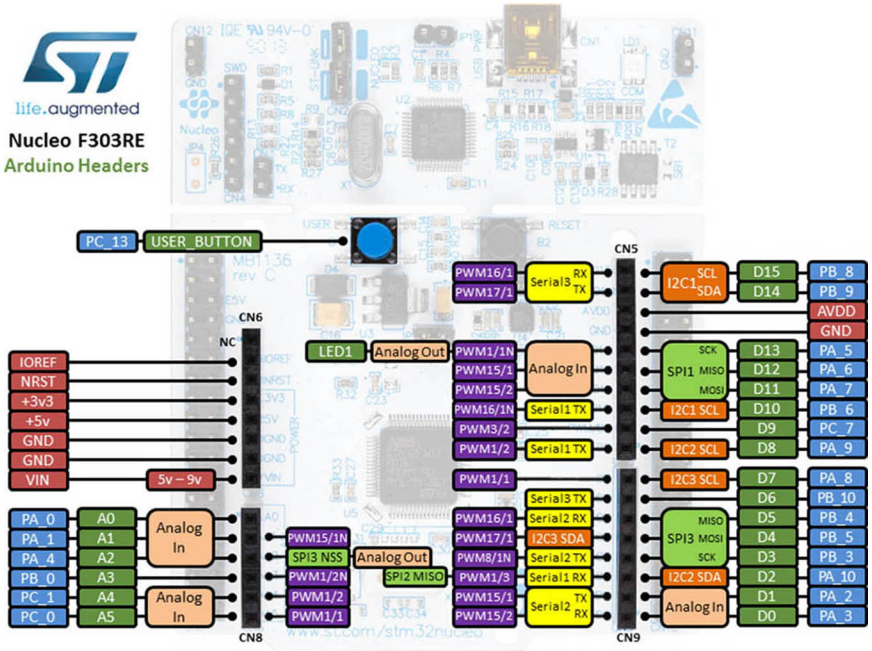


Morpho headers

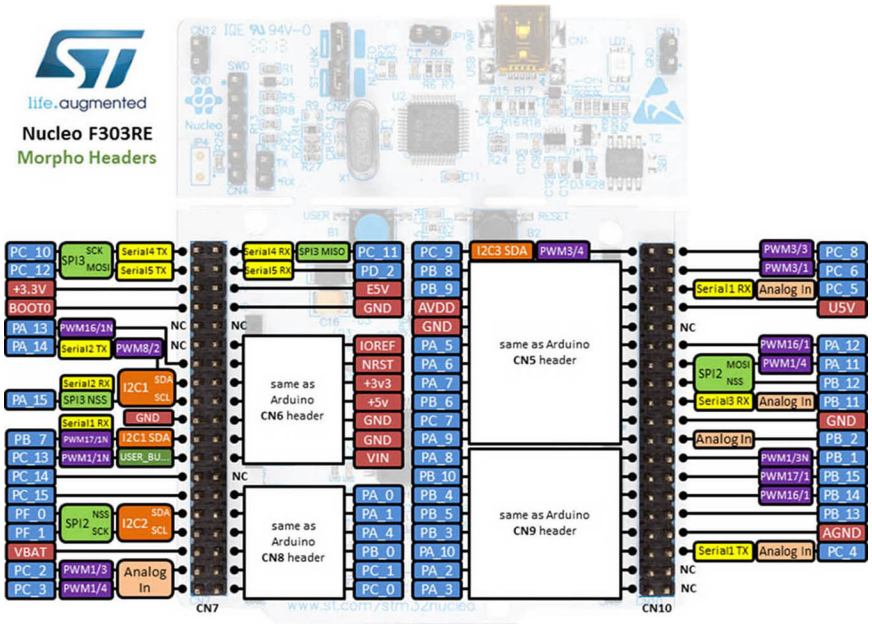


Nucleo-F303RE

Arduino compatible headers

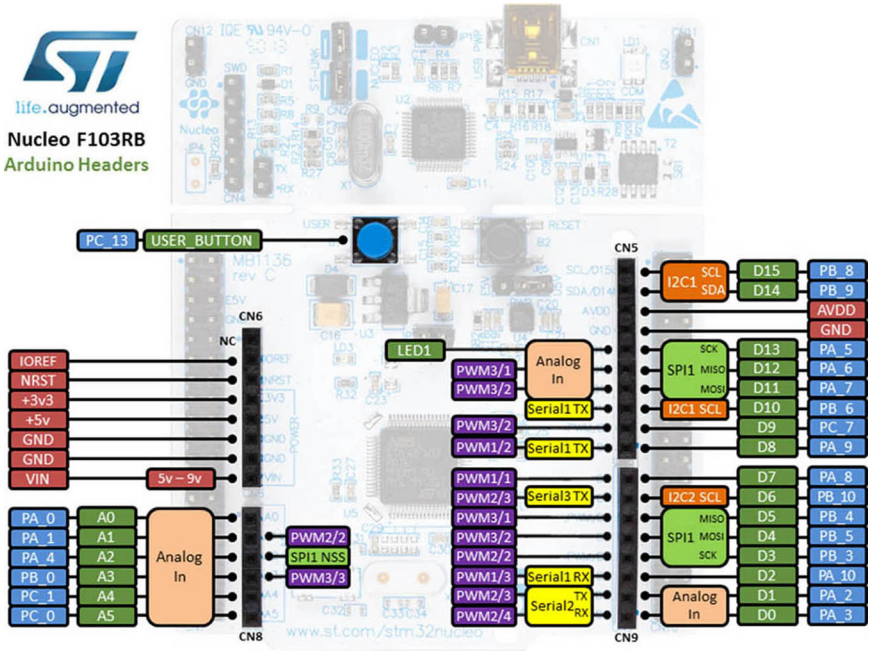


Morpho headers

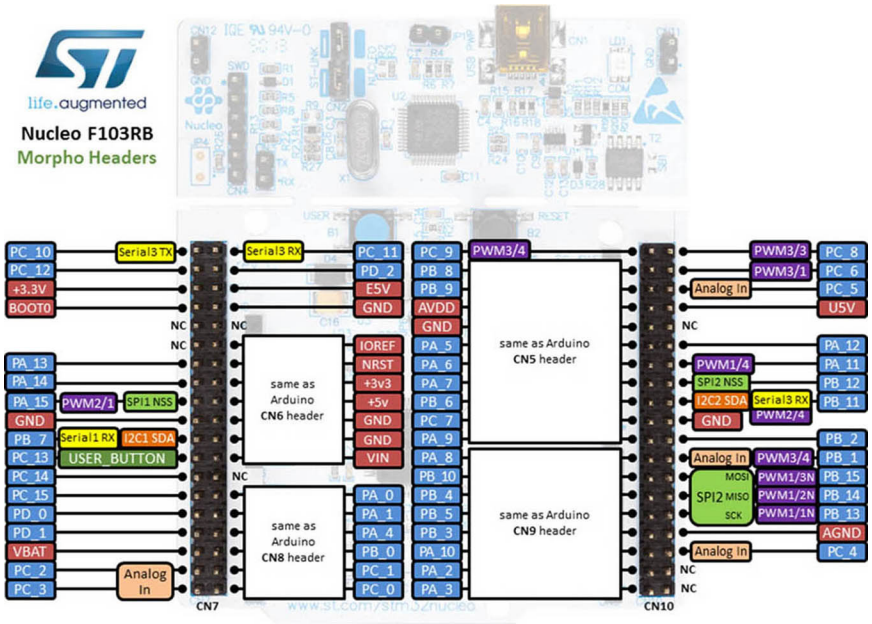


Nucleo-F103RB

Arduino compatible headers

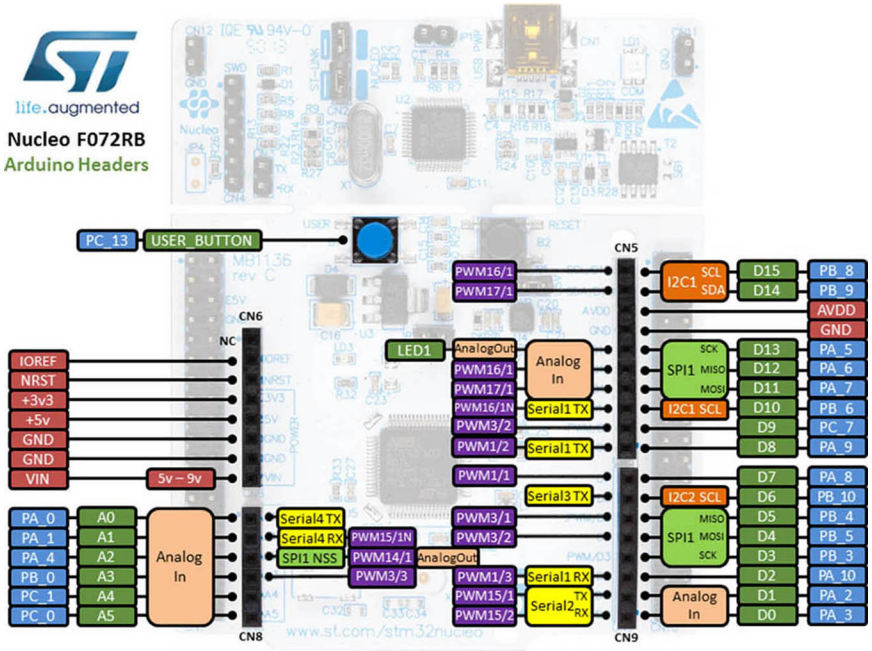


Morpho headers

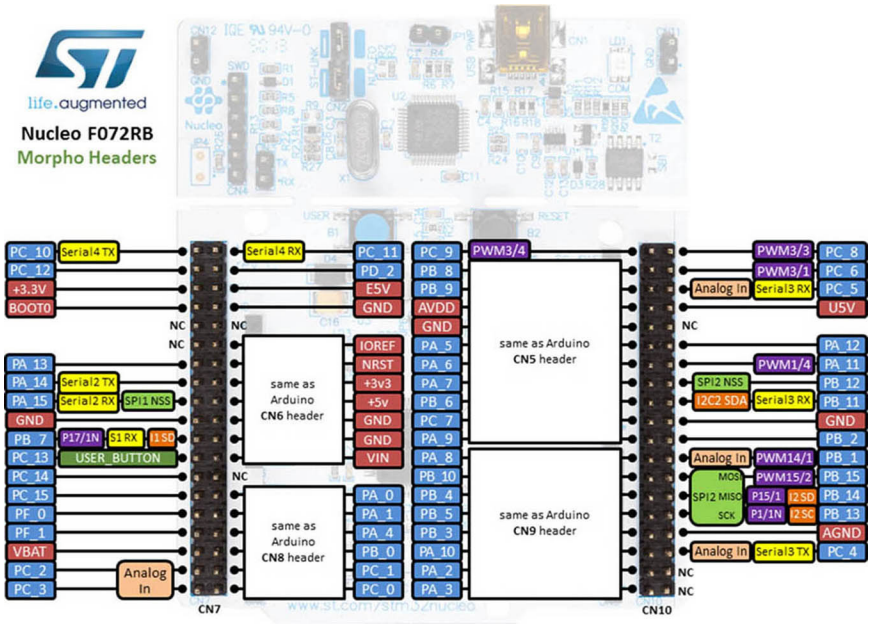


Nucleo-F072RB

Arduino compatible headers

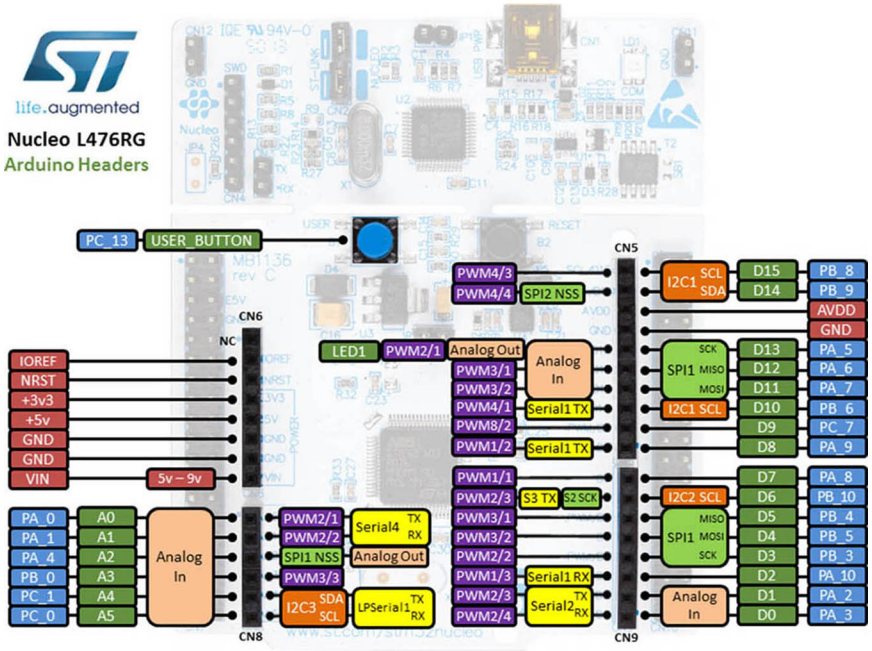


Morpho headers

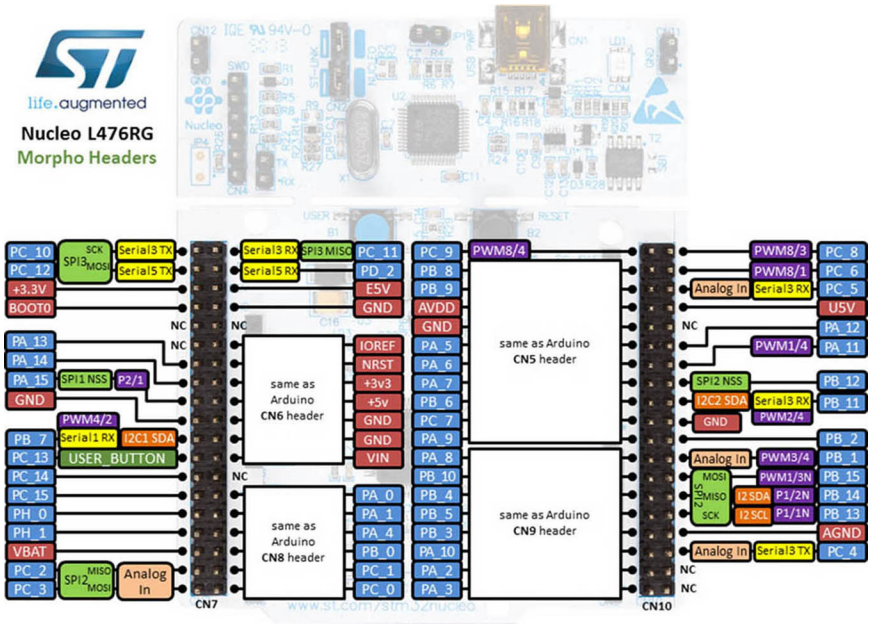


Nucleo-L476RG

Arduino compatible headers

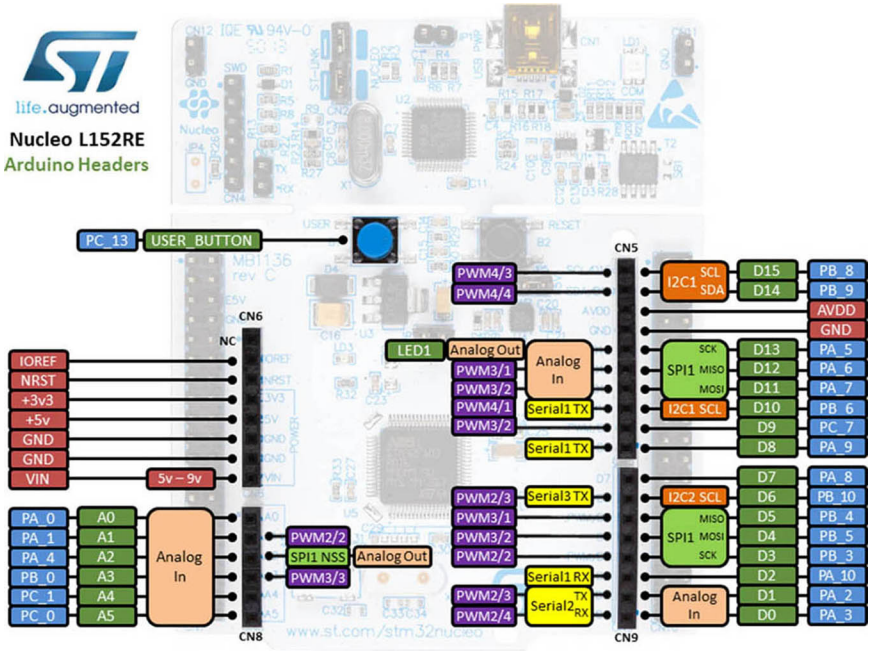


Morpho headers

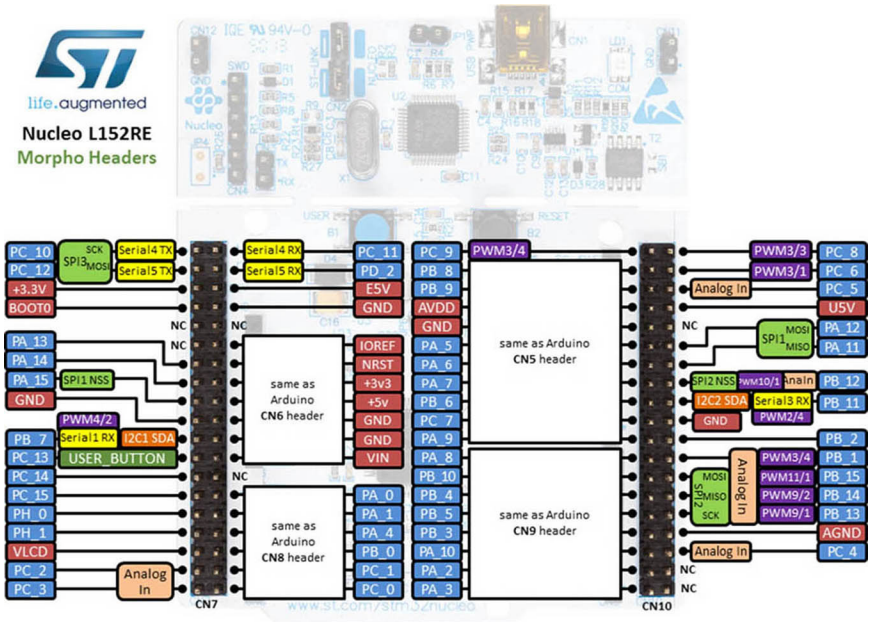


Nucleo-L152RE

Arduino compatible headers

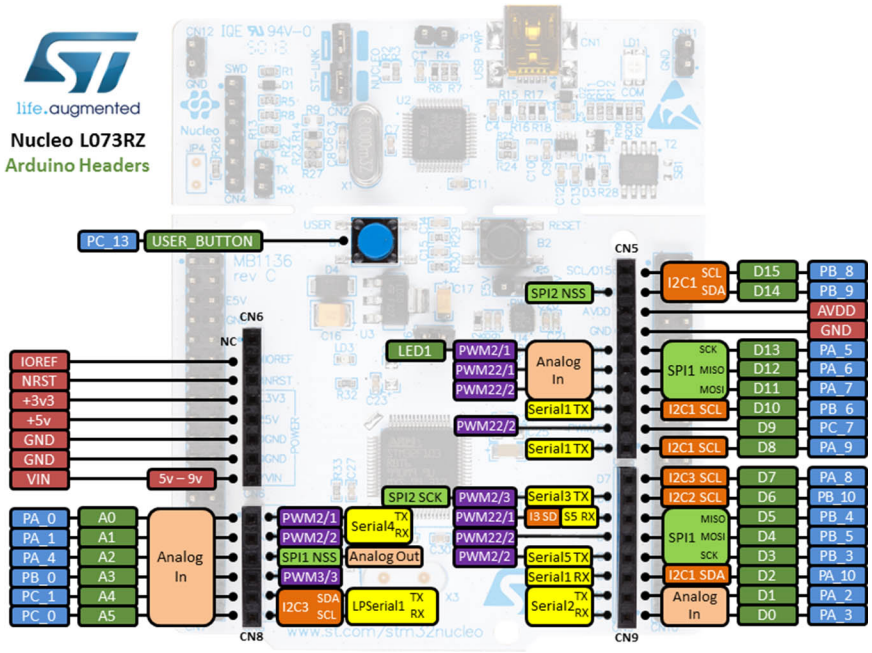


Morpho headers

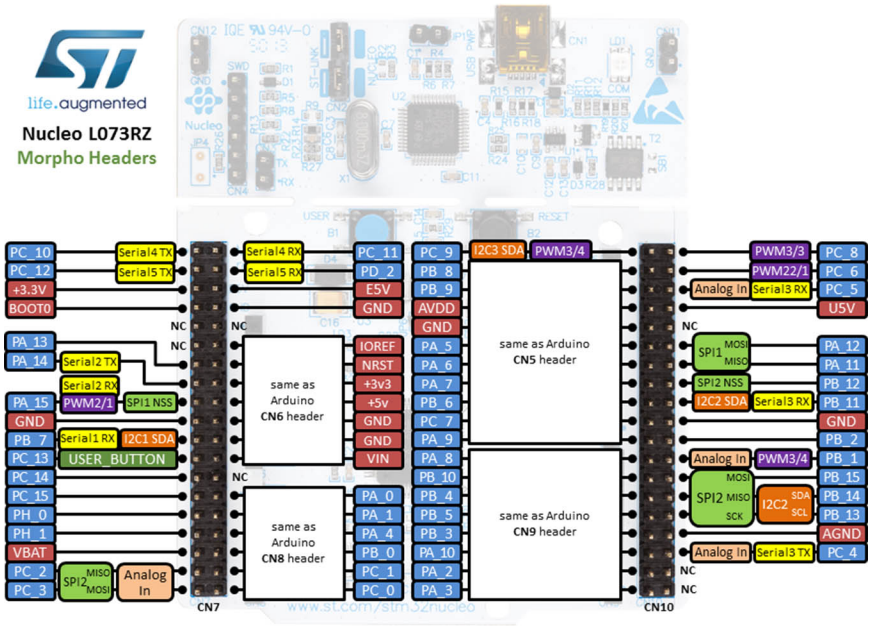


Nucleo-L073R8

Arduino compatible headers



Morpho headers



D. Differences with the 1st edition

The next paragraphs report all differences between the 1st and 2nd edition of the book.

Chapter 1

- New paragraphs about:
  - ARM TrustZone
  - STM32G0, STM32G4, STM32L5, STM32U5 families
  - Minor modifications to the text removing no longer updated information and adapting some parts to more recent evolutions of the STM32 portfolio
- Updated paragraphs about:
  - STM32F7 and STM32H7 series

Chapter 2

Chapter 2 was completely rewritten to cover STM32CubeIDE installation on Windows, MacOS and Linux.

Chapter 3 and 4

Chapter 3 and 4 were completely rewritten according to the project generation procedure of the STM32CubeIDE tool-chain and the new STM32CubeMX 6.x

Chapter 5

Chapter 5 was completely rewritten to cover STM32CubeIDE debug capabilities. The first edition of the book was based on the usage of OpenOCD acting as GDB server. While the STM32CubeIDE still offers the possibility to use OpenOCD as alternative to the *ST-LINK GDB Server*, I cannot see any notably reason to not use the official ST tooling. Finally, the I/O retargeting (that is, the usage of standard C `printf()`/`scanf()` routines) is now shown in this chapter instead of the Chapter 8.

Chapter 6

Chapter 6 was adapted so that all examples, figures and tables are related to the STM32F072RB MCU, since the STM32F030 is no longer used as platform for book examples.
Fixed some errors in the text.

Chapter 7

Chapter 7 was updated to cover some feature of the Cortex-M33 cores, used on STM32U5 and STM32L5 families.
Fixed some errors in the text.

Chapter 8

Chapter 8 was updated to cover more recent CubeMX features. Example 3 was improved with a better implementation of the circular buffer. Fixed some errors in the text.

Chapter 9

Chapter 9 was updated to cover recent STM32 MCUs with the more advanced DMAMUX module. All examples were updated accordingly.
The Chapter was updated to cover more recent CubeMX features.
Fixed some errors in the text.

Chapter 10

Chapter 10 was updated with all recent STM32 MCUs. Moreover, the new Nucleo-64 boards with integrated ST-LINK v3 debugger are documented in a separated section.
The Chapter was updated to cover more recent CubeMX features.
Fixed some errors in the text.

Chapter 11

Chapter 11 was updated to cover more recent CubeMX features. Some examples were improved with a better implementation. Fixed some errors in the text.

Chapter 12-22

Chapters 12-22 were updated to cover more recent CubeMX features and more recent STM32 MCUs. Fixed several errors in the text.

Chapter 23

Chapter 23 was updated to cover FreeRTOS 10.x features and the new CMSIS-RTOS v2 layer. Moreover, some new advanced topics have been added. For example, it is now deeply explained how to configure the project to enable re-entrancy of the `newlib` C run-time library.

Chapter 24

Chapter 24 was completely rewritten to cover the advanced debugging features offered by the STM32CubeIDE tool-chain.

Chapter 25-26

Chapters 25-26 were updated to cover more recent CubeMX features and more recent STM32 MCUs. Fixed several errors in the text.

Chapter 27

Chapters 27 is totally new.

Chapter 28

Chapters 28 were updated to cover more recent CubeMX features. Fixed several errors in the text.