

CARMINE NOVIELLO

MASTERING STM32

A step-by-step guide to the most complete
ARM Cortex-M platform, using a free
and powerful development environment
based on Eclipse and GCC

Mastering STM32

A step-by-step guide to the most complete ARM Cortex-M platform, using a free and powerful development environment based on Eclipse and GCC

Carmine Noviello

This book is for sale at <http://leanpub.com/mastering-stm32>

This version was published on 2022-02-13



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 - 2022 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](#)

Contents

Preface	i
Why Did I Write the Book?	i
Who Is This Book For?	ii
How to Integrate This Book?	iii
How Is the Book Organized?	iv
About the Author	vii
Errata and Suggestions	viii
Book Support	viii
How to Help the Author	ix
Copyright Disclaimer	ix
Credits	ix

Acknowledgments x

I Introduction 1

1.	Introduction to STM32 MCU Portfolio	2
1.1	Introduction to ARM Based Processors	2
1.1.1	Cortex and Cortex-M Based Processors	4
1.1.1.1	Core Registers	4
1.1.1.2	Memory Map	7
1.1.1.3	Bit-Banding	9
1.1.1.4	Thumb-2 and Memory Alignment	12
1.1.1.5	Pipeline	13
1.1.1.6	Interrupts and Exceptions Handling	15
1.1.1.7	SysTimer	17
1.1.1.8	Power Modes	17
1.1.1.9	CMSIS	19
1.1.1.10	Effective Implementation of Cortex-M Features in the STM32 Portfolio	20
1.2	Introduction to STM32 Microcontrollers	21
1.2.1	Advantages of the STM32 Portfolio....	22
1.2.2And Its Drawbacks	23

1.3	A Quick Look at the STM32 Subfamilies	24
1.3.1	F0	27
2.	Setting-Up the Tool-Chain	29
2.1	Why Choose Eclipse/GCC as Tool-Chain for STM32	30
2.1.1	Two Words About Eclipse...	31
2.1.2	... and GCC	31
2.2	Windows - Installing the Tool-Chain	32
2.2.1	Windows - Eclipse Installation	33
2.2.2	Windows - Eclipse Plug-Ins Installation	34
2.2.3	Windows - GCC ARM Embedded Installation	40
2.2.4	Windows – Build Tools Installation	41
2.2.5	Windows – OpenOCD Installation	41
2.2.6	Windows – ST Tools and Drivers Installation	42
2.2.6.1	Windows – ST-LINK Firmware Upgrade	42
3.	Hello, Nucleo!	44
3.1	Get in Touch With the Eclipse IDE	44
3.2	Create a Project	48
3.3	Connecting the Nucleo to the PC	55
3.4	Flashing the Nucleo using STM32CubeProgrammer	56
3.5	Understanding the Generated Code	57
4.	STM32CubeMX Tool	60
5.	Introduction to Debugging	61
5.1	Getting Started With OpenOCD	61
5.1.1	Launching OpenOCD	62
5.1.1.1	Launching OpenOCD on Windows	63
5.1.1.2	Launching OpenOCD on Linux and MacOS X.	64
5.1.2	Connecting to the OpenOCD Telnet Console	66
5.1.3	Configuring Eclipse	67
5.1.4	Debugging in Eclipse	73
5.2	ARM Semihosting	78
II	Appendix	79
B.	Troubleshooting guide	80
	GNU MCU Eclipse Installation Issues	80
	Eclipse related issue	80
	Eclipse cannot locate the compiler	81
C.	Nucleo pin-out	82
	Nucleo-F446RE	83

CONTENTS

Arduino compatible headers	83
Morpho headers	83
Nucleo-F411RE	84
Arduino compatible headers	84
Morpho headers	84
Nucleo-F410RB	85
Arduino compatible headers	85
Morpho headers	85
Nucleo-F401RE	86
Arduino compatible headers	86
Morpho headers	86
Nucleo-F334R8	87
Arduino compatible headers	87
Morpho headers	87
Nucleo-F303RE	88
Arduino compatible headers	88
Morpho headers	88
Nucleo-F302R8	89
Arduino compatible headers	89
Morpho headers	89
Nucleo-F103RB	90
Arduino compatible headers	90
Morpho headers	90
Nucleo-F091RC	91
Arduino compatible headers	91
Morpho headers	91
Nucleo-F072RB	92
Arduino compatible headers	92
Morpho headers	92
Nucleo-F070RB	93
Arduino compatible headers	93
Morpho headers	93
Nucleo-F030R8	94
Arduino compatible headers	94
Morpho headers	94
Nucleo-L476RG	95
Arduino compatible headers	95
Morpho headers	95
Nucleo-L152RE	96
Arduino compatible headers	96
Morpho headers	96
Nucleo-L073R8	97
Arduino compatible headers	97

CONTENTS

Morpho headers	97
Nucleo-L053R8	98
Arduino compatible headers	98
Morpho headers	98
E. History of this book	99
Release 0.1 - October 2015	99
Release 0.2 - October 28th, 2015	99
Release 0.2.1 - October 31th, 2015	99
Release 0.2.2 - November 1st, 2015	100
Release 0.3 - November 12th, 2015	100
Release 0.4 - December 4th, 2015	100
Release 0.5 - December 19th, 2015	100
Release 0.6 - January 18th, 2016	101
Release 0.6.1 - January 20th, 2016	101
Release 0.6.2 - January 30th, 2016	101
Release 0.7 - February 8th, 2016	101
Release 0.8 - February 18th, 2016	102
Release 0.8.1 - February 23th, 2016	102
Release 0.9 - March 27th, 2016	102
Release 0.9.1 - March 28th, 2016	102
Release 0.10 - April 26th, 2016	103
Release 0.11 - May 27th, 2016	103
Release 0.11.1 - June 3rd, 2016	104
Release 0.11.2 - June 24th, 2016	104
Release 0.12 - July 4th, 2016	104
Release 0.13 - July 18th, 2016	104
Release 0.14 - August 12th, 2016	104
Release 0.15 - September 13th, 2016	105
Release 0.16 - October 3th, 2016	105
Release 0.17 - October 24th, 2016	105
Release 0.18 - November 15th, 2016	106
Release 0.19 - November 29th, 2016	106
Release 0.20 - December 28th, 2016	106
Release 0.21 - January 29th, 2017	106
Release 0.22 - May 2nd, 2017	107
Release 0.23 - July 20th, 2017	107
Release 0.24 - December 11th, 2017	107
Release 0.25 - January 3rd, 2018	108
Release 0.26 - May 7th, 2018	108

Preface

As far as I know this book is the first attempt to write a systematic text about the STM32 platform and its official STM32Cube HAL. When I started dealing with this microcontroller architecture, I searched far and wide for a book able to introduce me to the subject, with no success.

The book is divided in three parts: an introductory part showing how to setup a complete development environment 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); a more advanced section covering aspects such as the use of a Real Time Operating Systems, the boot sequence and the memory layout of an STM32 application.

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 the STM32F4 MCU alone is almost one thousand pages, that is more than a book! Hence, this text will offer a hint to start diving inside 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.

Why Did I Write the Book?

I started to cover topics about STM32 programming on my personal blog in 2013. I first started writing posts only in Italian and then translating them into English. I covered several topics, ranging from how to setup a complete free tool-chain to specific aspects related to STM32 programming. Since then, I have received plenty of comments and requests about all kinds of topics. Thanks to the interaction with readers of my blog, I realized that it is not simple to cover complex topics in depth on a personal web site. A blog is an excellent place where to cover really specific and limited topics. If you need to explain broader topics involving software frameworks or hardware, a book is still the right answer. A book forces you to organize topics in a systematic way, and gives you all the necessary space to expand the subject as needed (I am one of those people who still believe reading long texts on a monitor is a bad idea).

For reasons that I do not know, there are no books¹ covering the topics presented here. To be honest, in the hardware industry is not so common to find books about microcontrollers, and this is really strange. Compared to software, hardware has much greater longevity. For example, all STM32 MCUs

¹This is not exactly true, since there is a good and free book from Geoffrey Brown of University of Indiana (<http://bit.ly/1Rc1tMl>). However, in my opinion, it goes too quickly to the point, leaving out important topics such as the use of a complete tool-chain. It also does not cover the STM32Cube HAL, which has replaced the old `std peripheral library`. Finally, it does not show the differences between each STM32 subfamily and it is focused only on the STM32F4 family.

have a guaranteed life of ten years starting from January 2017 (ST has been updating this “starting date” every year until now). This means that a book on this subject may potentially have the same life expectation, and this is really uncommon in computer science. Apart from some really important titles, most technical books have a shelf-life of two years or less.

I think that there are several reasons why this happens. First of all, in the electronics industry *know-how* is still a great value to protect. Compared to the software world, hardware requires years of field experience. Every mistake has a cost, and it is highly dependent on the product stage (if the device is already on the market, an issue may have dramatic costs). For this reason, electronics engineers and firmware developers tend to protect their know-how, and this may be one of the reasons discouraging really experienced users from writing books about these topics.

I believe another reason being that if you want to write a book about an MCU, you must be able to range from aspects of electronics to more high-level programming topics. This requires a lot of time and effort, and it is really hard especially when things change at a high pace (during the time of writing the first few chapters of this book, ST has released more than twenty versions of its HAL). In the electronics industry, hardware engineers and firmware developers are traditionally two different figures, and sometimes they do not know what the other is doing.

Finally, another important reason is that electronics design becomes sort of a niche when compared to the software world (there is great disparity between the number of software programmers and electronics designers), and the STM32 is itself a niche within the niche.

For these and other minor reasons, I decided to write this book using a self-publishing platform like *LeanPub*, which allows you to build a book progressively. I think that the idea behind *LeanPub* is perfect for books about niche subjects, and it gives authors the time and tools to write about as much complex topics as they want.

Who Is This Book For?

This book is addressed to novices of the STM32 platform, interested in learning in less time how to program these fantastic microcontrollers. 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 he/she does 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](https://www.arduino.cc/) is both a *brand* and an *ecosystem*. There are tens of Arduino development boards available on the market, 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. Atmel is one of the leading companies, together with Microchip^b, that rule the 8-bit MCU segment. 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](https://processing.org/)) and the Arduino libraries, which greatly simplify the development process on ATmega MCUs. This really large and continuously growing community has developed hundred of libraries to interface as many hardware devices, and thousand of examples and applications.

So, the question is: “Is Arduino good for professional applications or for those wanting to develop the last mainstream product on Kickstarter?”. The answer is: “YES, definitively.”. I myself have developed a couple of custom boards for a customer, and being these boards 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 an Open Source 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-bits 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/>

^bMicrochip has acquired Atmel in January 2016.

^c<https://processing.org/>

How to Integrate This Book?

This book does not aim to be a full-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 Yiu²](http://amzn.to/1P5sZwq) is the best source for every Cortex-M developer.

²<http://amzn.to/1P5sZwq>

How Is the Book Organized?

The book is divided in twenty-seven chapters, and they cover 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 testing board for the presented topics: the Nucleo.

Chapter 2 shows how to setup a complete and working tool-chain to start developing STM32 applications. The chapter is divided in 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 with no 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 an 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. Moreover, it explains how to dive into the generated code and customize it, as well as how to import a project generated with it into the Eclipse IDE.

Chapter 5 introduces the reader to debugging. A hand-on presentation of OpenOCD is given, showing how to integrate it in Eclipse. Moreover, a brief view of Eclipse's debugging capabilities is presented. Finally, the reader is introduced to a really important topic: ARM semihosting.

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. 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 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 it 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 introducing the essentials of the I²C protocol, and then it 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 introducing the essentials of the SPI specification, and then it 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 it shows the related CubeHAL module used to program it.

Chapter 17 is about IWDG and WWDG 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 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 really 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 the *vector table* relocation.

Chapter 21 provides an introduction to 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 has the ability to 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 layer developed by ST on top of the FreeRTOS API. Moreover, some advanced techniques, like the *tickless mode* in low-power design, are shown.

Chapter 24 introduces the reader to some advanced debugging techniques. The chapter starts 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, some Eclipse advanced debugging tools are presented, such as watchpoints and expressions, and how to use Keil Packs integrated in the GNU MCU Eclipse tool-chain. Finally, a brief introduction to SEGGER J-LINK professional debuggers is given, and to the way to use them in the Eclipse tool-chain.

Chapter 25 briefly introduces the reader to the FatFs middleware. This library allows to manipulate structured filesystems created with the widespread FAT12/16/32 filesystem. The chapter also shows the way ST engineers have integrated this library in 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 alike on an external SD card.

Chapter 27 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.

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 for 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 particular MCU. A black horizontal ruler (like the one below) closes the specific section. This means that the text returns to be 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 is 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 is a discussion box, and it is used to talk about the subject in a broader way.



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

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 has started fighting against bits.

I began my career in informatics when I was only a young boy with a 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 be 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 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 ten 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 it does not reached 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.

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 end, I am totally open to suggestions and improvements about 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 website: <http://www.carminenoviello.com/en/mastering-stm32/>³

Book Support

I have setup 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/>⁴.

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.

³<http://www.carminenoviello.com/en/mastering-stm32/>

⁴<http://www.carminenoviello.com/en/mastering-stm32/>

How to Help the Author

Almost twice a week I receive nice emails from readers of this book encouraging me to continue the work. Some of them would also donate additional money to help me during the book writing. Needless to say that these emails make me really happy for days on end :-)

However, if you really want to help me, you may consider to:

- give me feedback about unclear things or errors contained both in the text and examples;
- write a small review about what you think⁵ of this book in the [feedback section](#)⁶.
- use your favorite social network or blog *to spread the word*. The suggested hashtag for this book on Twitter is [#MasteringSTM32](#)⁷.

Copyright Disclaimer

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

ARTTM Accelerator, STM32, ST-LINK, STM32Cube 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, 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 forgot 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](#)⁸)

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

⁵Negative feedback is also welcome :-)

⁶<https://leanpub.com/mastering-stm32/feedback>

⁷<https://twitter.com/search?q=%23MasteringSTM32>

⁸<https://99designs.it/profiles/alemiglio>

Acknowledgments

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](http://www.erix.it)⁹) 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](http://ubidefeo.com)¹⁰) 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.

Finally, a special thanks to my business partners Antonio Chello and Francesco Vitobello who gave me a lot of help during last year with the management of our company: a book is probably the most time-consuming activity after a business development.

Regards,
Carminé I.D. Noviello

⁹<http://www.erix.it>

¹⁰<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 ten 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 500 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 due to the fact that there are many different ARM architecture revisions (ARMv6, ARMv6-M, ARMv7-M, ARMv7-A, 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 describes precisely 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 STM32F4 is a family of MCUs based on the Cortex-M4 core that implements this bit-banding feature. **Figure 1** clearly shows the relation between a Cortex-M3 based MCU and its Cortex-M3 core.

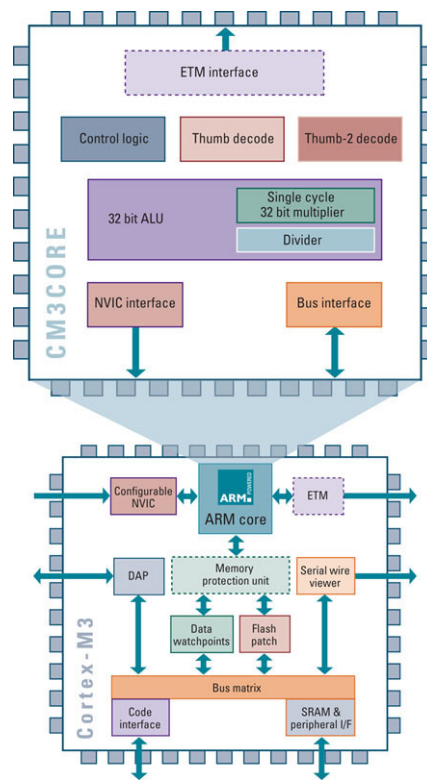


Figure 1: The relation between a Cortex-M3 core and a Cortex-M3 based MCU

ARM Holdings is a British¹ company that develops the instruction set and architecture for ARM-based products but does not manufacture devices. This is a really 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 the only manufacturer 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 really interesting 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 50 billion ARM processors have been produced as of 2014, 10 billion of which were produced in 2013. ARM based processors equip about 75 percent 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 architecture (ARMv8-A).

Being a sort of widespread standard, there are a lot of compilers and tools, as well as Operating

¹In July 2016 the Japanese *Softbank* announced a plan to acquire *ARM Holdings* for \$31 Billions. The deal has been closed on September 5th and on the following day the formerly British company has been de-listed from the London Stock Exchange.

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 or Freescale) to those who produce processors for other licensees. Among the most common cores in this segment, we can find Cortex-A7 and Cortex-A9 32-bit processors, as well as the latest ultra-performance 64-bit Cortex-A53 and Cortex-A57 cores.
- **Cortex-M**, which stands for **eM**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**ead-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 except² for two categories of instructions: load and store, used to transfer data between CPU registers and memory locations.

²This 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 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.

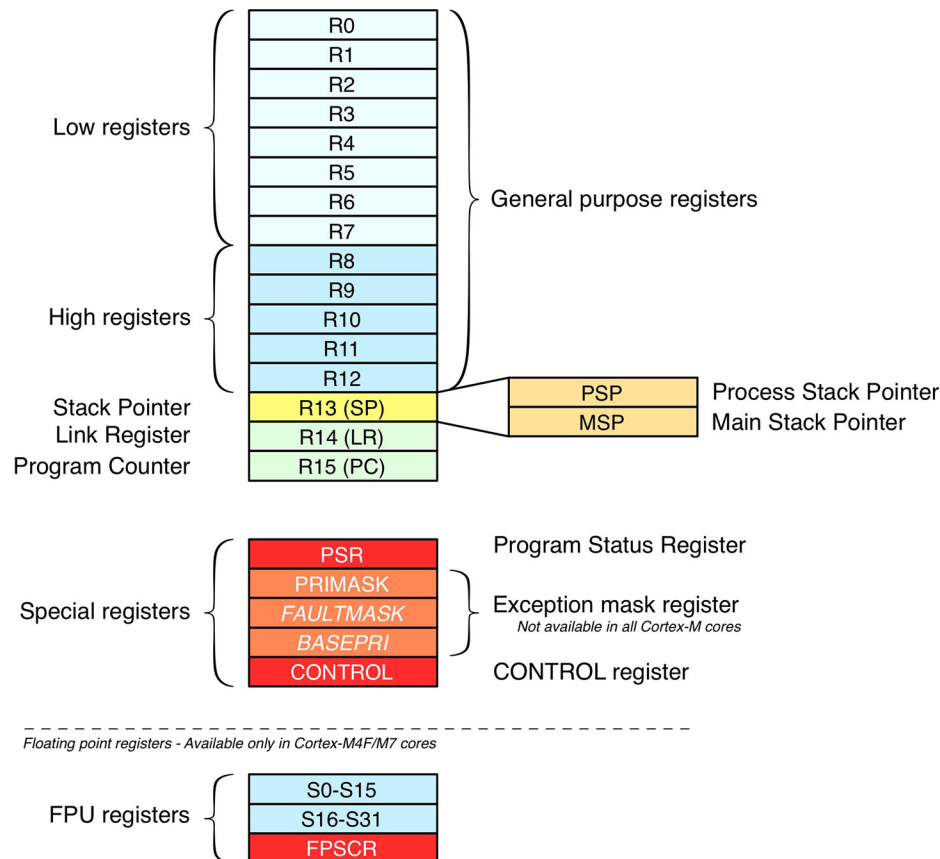


Figure 2: ARM Cortex-M core registers

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 code³:

```

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 (which is the *frame pointer*, as we will see in [Chapter 20](#)) 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.

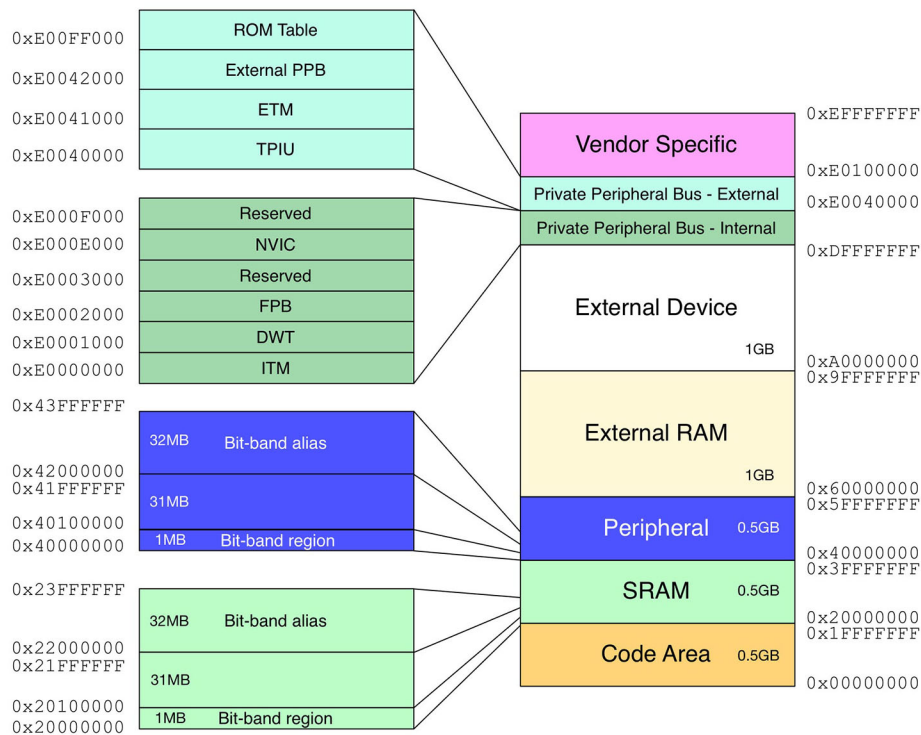


Figure 3: Cortex-M fixed memory address space

³That 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 manufacturer. The address space is 4GB wide, and it is organized in several sub-regions with different logical functionalities. **Figure 3** shows the memory layout of a Cortex-M processor ⁴.

The first 512MB are dedicated to code area. STM32 devices further divide this area in some sub-regions as shown in **Figure 4**. Let us briefly introduce them.

All Cortex-M processors map the code area starting at address `0x0000 0000`⁵. 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 16DC` can also be accessed from `0x0000 16DC`).

⁴Although 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.

⁵To 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.

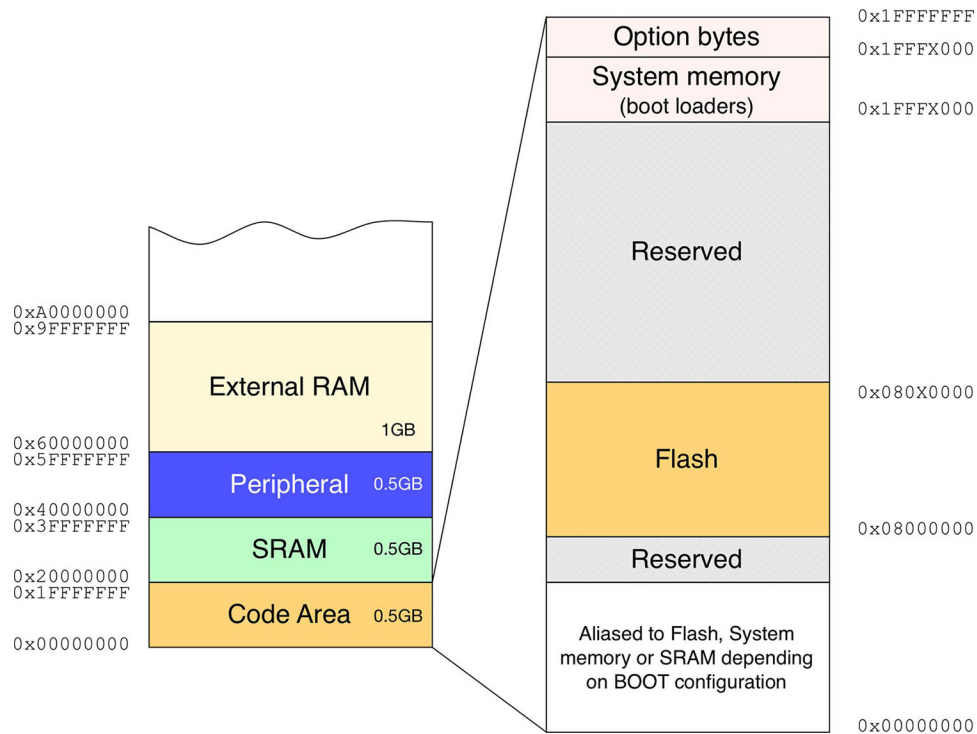


Figure 4: Memory layout of Code Area on STM32 MCUs

The last two sections are dedicated to *System memory* and *Option bytes*. The former 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`⁶. 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 an/or flashe storage. 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™

⁶The 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`.

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. Considering the previous C code, we can see that the compiler will generate the following ARM assembly code⁷:

```
#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 5** shows how the Cortex

⁷That 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

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.

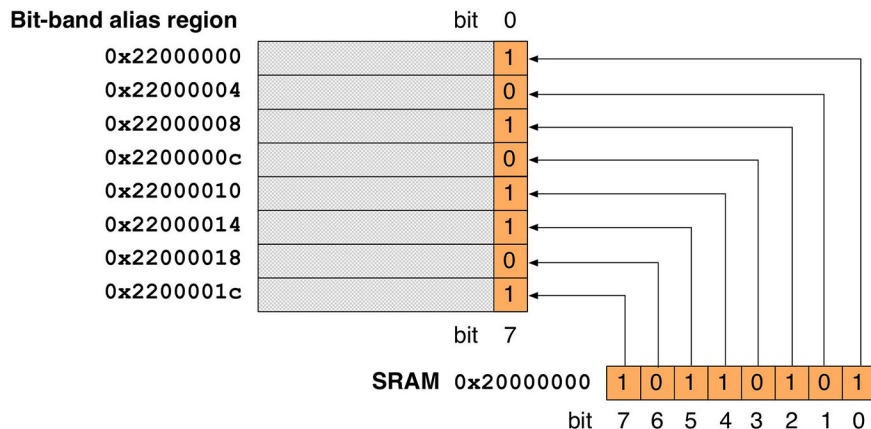


Figure 5: Memory mapping of SRAM address `0x2000 0000` in bit-banding region (first 8 of 32 bits shown)

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

$$\text{bit_band_address} = \text{alias_region_base} + (\text{region_base_offset} \times 32) + (\text{bit_number} \times 4)$$

For example, considering the memory address of **Figure 5**, to access bit 2 :

$$\text{alias_region_base} = 0x22000000$$

$$\text{region_base_offset} = 0x20000000 - 0x20000000 = 0$$

$$\text{bit_band_address} = 0x22000000 + 0 \times 32 + (0x2 \times 0x4) = 0x22000008$$

ARM defines two bit-band regions for Cortex-M based MCUs⁸, 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 6**.

⁸Unfortunately, Cortex-M7 based MCUs do not provide bit-banding capabilities.

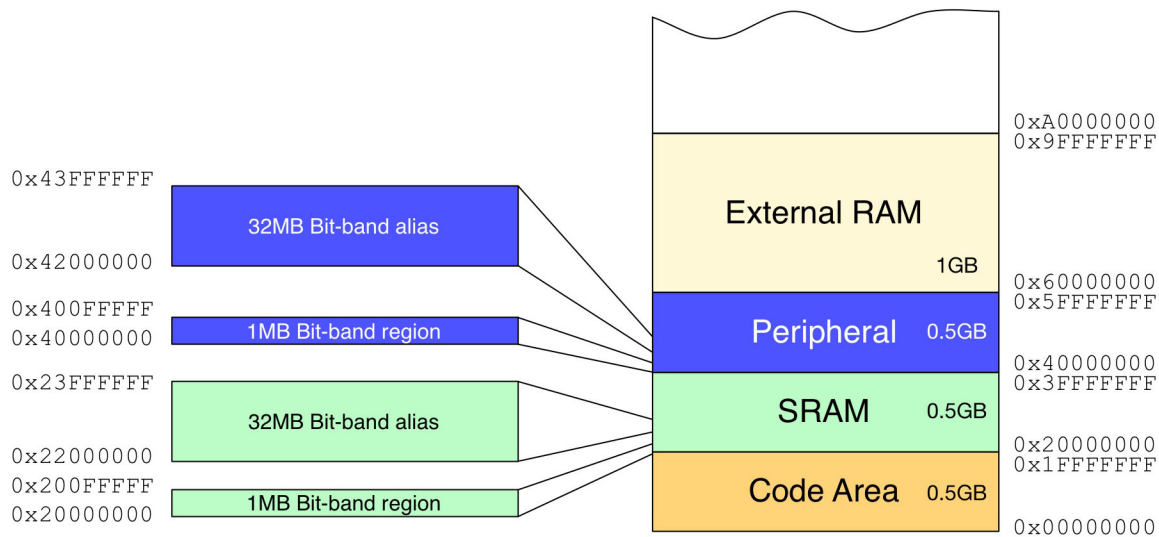


Figure 6: Memory map and bit-banding regions

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 port⁹, 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:

```
1 // Define base address of bit-band
2 #define BITBAND_SRAM_BASE 0x20000000
3 // Define base address of alias band
4 #define ALIAS_SRAM_BASE 0x22000000
5 // Convert SRAM address to alias region
6 #define BITBAND_SRAM(a,b) ((ALIAS_SRAM_BASE + ((uint32_t)&(a)-BITBAND_SRAM_BASE)*32 + (b*4)))
7
8 // Define base address of peripheral bit-band
9 #define BITBAND_PERI_BASE 0x40000000
10 // Define base address of peripheral alias band
11 #define ALIAS_PERI_BASE 0x42000000
12 // Convert PERI address to alias region
13 #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:

⁹Anyone who has already played with Nucleo boards, knows that user LED LD2 (the green one) is connected to that port pin.

```
1  #define GPIOA_PERH_ADDR 0x40020000
2  #define ODR_ADDR_OFF    0x14
3
4  uint32_t *GPIOA_ODR = GPIOA_PERH_ADDR + ODR_ADDR_OFF;
5  uint32_t *GPIOA_PIN5 = BITBAND_PERI(GPIOA_ODR, 5);
6
7  *GPIOA_PIN5 = 0x1; // Turns GPIO HIGH
```

1.1.1.4 Thumb-2 and Memory Alignment

Historically, ARM processors provide a 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.

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).

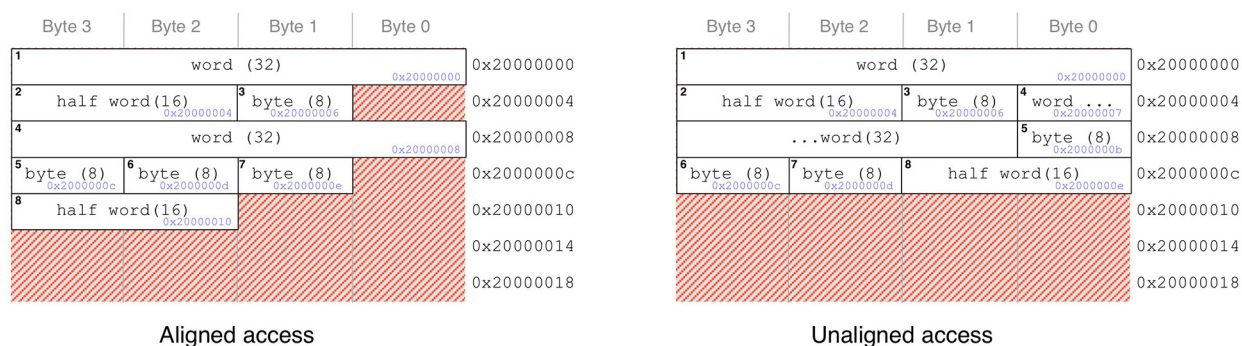


Figure 7: Difference between aligned and unaligned memory access

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 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 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 a number of 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.

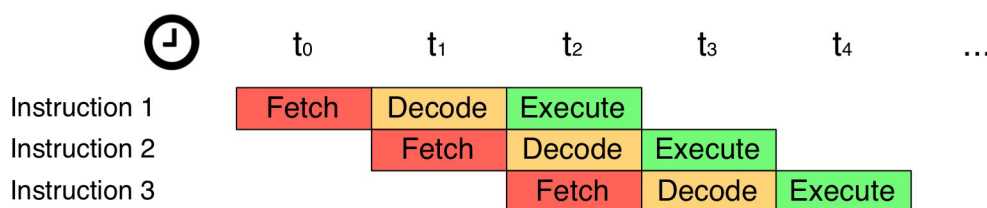


Figure 8: Three stage instruction pipeline

All Cortex-M based microcontrollers introduce a form of pipelining. The most common one is the *3-stage pipeline*, as shown in **Figure 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 has to 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 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*)

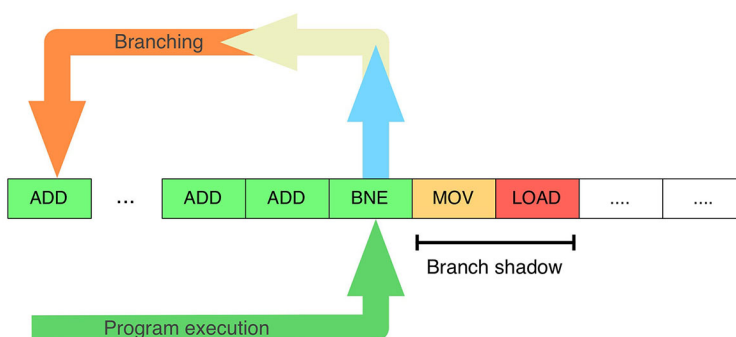


Figure 9: Branching in program execution related to pipelining

Even in this case there are several techniques to minimize the impact of branching. They are often referred as *branching prediction techniques*. The ideas 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 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](#)¹⁰ 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 execution¹¹.

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** 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.

¹⁰<http://bit.ly/1k7ggh6>

¹¹With 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.

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.
7-10	-	-	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-[47/240] ^d	IRQ	Configurable	IRQ Input

^aThe lower the priority number is, the higher the priority is.

^bIt's 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 ranges from 0 to 255.

^cThese exceptions are not available in Cortex-M0/0+.

^dCortex-M0/0+ allow 32 external configurable interrupts. Cortex-M3/4/7 allow 240 external configurable interrupts. However, in practice the number of interrupt inputs implemented in the real MCU is far less.

Table 1: Cortex-M exception types

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 NMI¹²).

¹²Also 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.

- **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 [explore next](#), 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 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](#).

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.

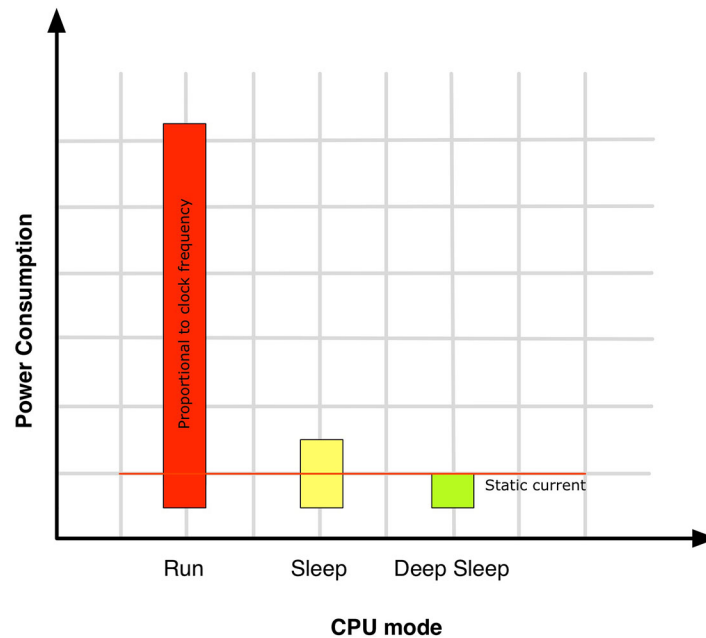


Figure 10: Cortex-M power consumption at different power modes

Traditionally, Cortex-M processors provide *user-defined power modes* via *System Control Register*(SCR). The first one is the *Run* mode (see **Figure 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 11** displaying the power consumption of an STM32F2 MCU running at 80MHz @30°C¹³. As we can see, the maximum power consumption is reached in *Run-mode* (that is, the *Active* 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.

¹³Source [ST AN3430](#)

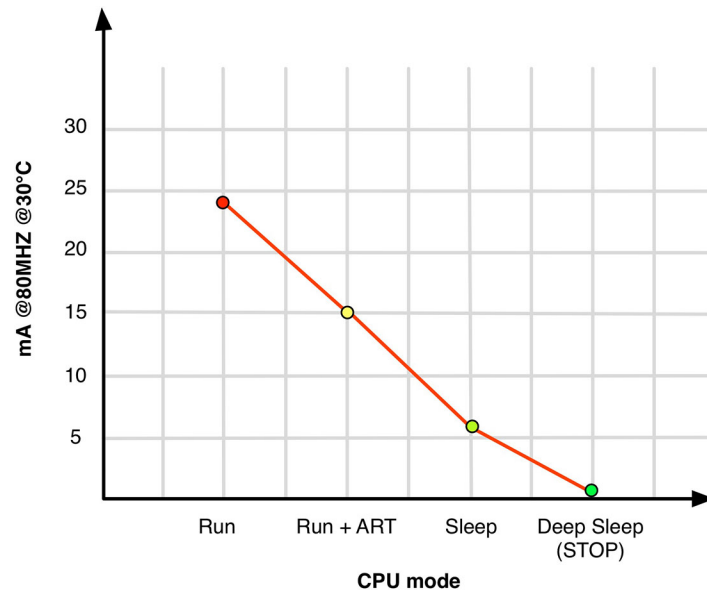


Figure 11: STM32F2 power consumption at different power modes

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 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 amongst 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-M0/3/4/7.
- **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 firmwares 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.

However, this initiative from ARM is still evolving, and the support to all components from ST is still very bare-bone. 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.

1.1.1.10 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 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	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	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	M4	Entire	Entire	32/64-bit result	Yes	Yes	Yes	Yes SP	ARMv7E-M
F7	M7	Entire	Entire	32/64-bit result	Yes	Yes	Yes	Yes SP & DP	ARMv7E-M

 Optional in ARM specification

Table 2: ARM Cortex-M instruction variations

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

 Optional in ARM specification

Table 3: ARM Cortex-M optional components

1.2 Introduction to STM32 Microcontrollers

STM32 is a broad range of microcontrollers divided in nine 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 under development. **Figure 12** shows the internal die of an STM32F103 MCU, one of the most widespread STM32 MCUs¹⁴. All STM32 microcontrollers have a Cortex-M core, plus some distinctive ST features (like the ARTTM 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.

¹⁴This picture is taken from Zeptobars.ru (<http://bit.ly/1FfqHsv>), a really fantastic blog. Its authors **decap** (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.

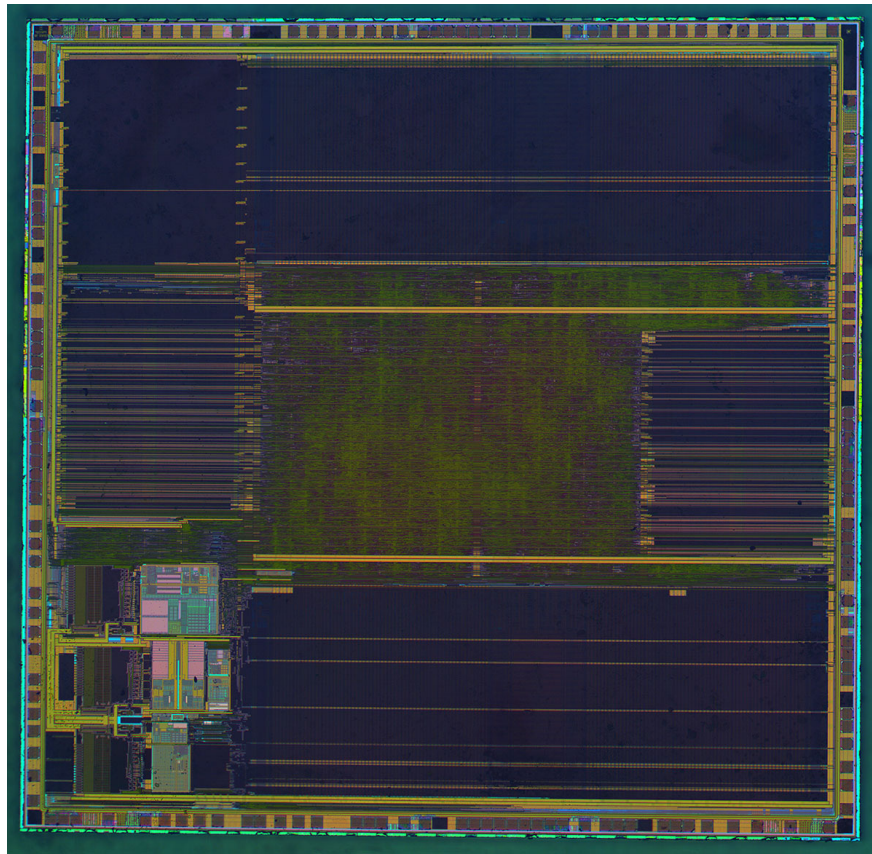


Figure 12: Internal die of an STM32F103 MCU³

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 50 billions of devices sold by 2014 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 boss) decide to switch to Cortex-M MCUs from other vendors (in theory).

- **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 really key to your application - a level shifter always introduce a parasitic capacitance that reduced the commutation frequency).
- **32 cents for 32 bit:** 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.

1.2.2And 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 really 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.
- **Lack of official tools:** this book will guide the reader through the process of setting up a full tool-chain for the STM32 platform. The fact that ST does not provide its official development environment (like, for example, Microchip does for its MCUs) pushes a lot of people away from this platform. This is a strategic mishap that people at ST should seriously take into account.
- **Fragmented and dispersive documentation:** ST is actively working on improving its official documentation for the STM32 platform. You can find a lot of really 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" files¹⁵, which are automatically generated from

¹⁵a 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.

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 HAL:** unfortunately, the official HAL from ST contains several bugs, and **some of them are really severe and lead to confusion in novices**. For example, during the development of this book I have found errors in several [linker scripts](#)¹⁶ (which are supposed to be the foundation blocks of the HAL) and in some critical routines that should work seamlessly. Every day at least a new post regarding HAL bugs appears in the official [ST forum](#)¹⁷, and this can be source of great frustration. 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. ST should seriously consider investing **less** on designing the next development kit and **more** on the development of a decent STM32 HAL, which is currently not adequate to the hardware development. I would respectfully suggest to release the whole HAL on a community for developers like *github*, and let the community help fixing the bugs. This would also greatly simplify the bug reporting process, which is currently demanded to scattered posts on the ST forum. A real pity.

1.3 A Quick Look at the STM32 Subfamilies

As you read, the STM32 is a rather complex product lineup, spanning over ten product sub-families. **Figure 13** and **Figure 14** summarize the current STM32 portfolio¹⁸. The diagrams aggregate the subfamilies in four macro groups: *High-performance*, *Mainstream*, *Wireless* and *Ultra Low-Power* 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 400MHz (H7). All MCUs in this group provide ART™ Accelerator, an ST technology that allows *0-wait* execution from flash memory.

¹⁶<http://bit.ly/1iRAKdf>

¹⁷<http://bit.ly/1LTf2MS>

¹⁸The diagram was taken from this ST Microelectronics brochure (<http://bit.ly/1G7HMFj>).









Common core peripherals and architecture:	High-performance										
	STM32H7 series – High performance with DSP, Double-precision FPU, JPEG Codec and Chrom-ART Accelerator™										
	400 MHz Cortex-M7 L1-Cache	Up to 2-Mbyte dual-bank Flash	Up to 1-Mbyte SRAM	2x USB 2.0 OTG FS/HS	2x 16-bit advanced MC timer HR timer	DFSDM HDMI-CEC Ethernet S/PDIF	Quad-SPI FMC MDIO Camera IF SDIO	Crypto-hash TRNG	4x SAI 3x I²S 2x FDCAN LCD-TFT	3x 16-bit ADC Op-amps comp.	
	STM32F7 series – High performance with DSP, FPU, ART Accelerator™ and Chrom-ART Accelerator™										
	216 MHz Cortex-M7 L1-Cache	Up to 2-Mbyte dual-bank Flash	Up to 512-Kbyte SRAM	2x USB 2.0 OTG FS/HS	2x 16-bit advanced MC timer	DFSDM HDMI-CEC Ethernet S/PDIF	Quad-SPI FMC MDIO Camera IF SDIO	Crypto-hash TRNG	2x SAI 2x I²S LCD-TFT Up to 3x CAN	MIPI-DSI	
	STM32F4 series – High performance with DSP, FPU, ART Accelerator™ and Chrom-ART Accelerator™										
	Up to 180 MHz Cortex-M4	Up to 2-Mbyte dual-bank Flash	Up to 384-Kbyte SRAM	2x USB 2.0 OTG FS/HS	2x 16-bit advanced MC timer	DFSDM HDMI-CEC Ethernet S/PDIF	Quad-SPI FMC MDIO Camera IF SDIO	Crypto-hash TRNG	2x SAI 5x I²S LCD-TFT Up to 2x CAN	MIPI-DSI	
	STM32F2 series – High performance with ART Accelerator™										
	120 MHz Cortex-M3 CPU	Up to 1-Mbyte Flash	Up to 128-Kbyte SRAM	2x USB 2.0 OTG FS/HS	2x 16-bit advanced MC timer	Ethernet	FSMC Camera IF SDIO	Crypto-hash TRNG	2x I²S Up to 2x CAN		
	Mainstream										
	STM32F3 series – Mixed-signal with DSP and FPU										
	72 MHz Cortex-M4	Up to 512-Kbyte Flash	Up to 80-Kbyte SRAM CCM-RAM	USB 2.0 FS	3x 16-bit advanced MC timer	3x DAC 7x comp. 4x PGA	FSMC CAN	HR-Timer	ADC 3x 16-bit $\Sigma\Delta$ 4x 12-bit (5 MSPS)		
	STM32F1 series – Mainstream										
	Up to 72 MHz Cortex-M3 CPU	Up to 1-Mbyte Flash	Up to 96-Kbyte SRAM	USB 2.0 OTG FS	2x 16-bit advanced MC timer	HDMI-CEC Ethernet	FSMC SDIO	2x I²S 2x CAN			
	STM32F0 series – Entry-level										
	48 MHz Cortex-M0 CPU	Up to 256-Kbyte Flash	Up to 32-Kbyte SRAM 20-byte backup data	USB 2.0 FS device Crystal less		Comp. HDMI-CEC	CAN DAC				
	Wireless										
	STM32WB series – Multiprotocol and ultra-low-power 2.4 GHz radio with DSP, FPU, ART Accelerator™ and IP Protection										
	64 MHz Cortex-M4 CPU	Up to 1-Mbyte Flash	Up to 256-Kbyte SRAM	USB 2.0 FS Crystal less BCD / LPM	1x 16-bit advanced MC timer	Cortex-M0+ BLE 5.0 802.15.4 Concurrent	LP ADC 12x-16bit 2x comp.	Quad-SPI 1x SAI (2ch)	PKA AES-256 TRNG CKS*	LCD 8x40 4x44	

Figure 13: STM32 portfolio

Common core peripherals and architecture:	Ultra-Low-Power									
	STM32L4+ series – Ultra-Low-Power and more Performance with DSP, FPU, ART Accelerator™ and Chrom-ART Accelerator™									
	120 MHz Cortex-M4 CPU	Up to 2-Mbyte dual-bank Flash	Up to 640-Kbyte SRAM	USB 2.0 OTG Crystal less	2x 16-bit advanced MC timer	DFSDM Op-amps comp.	2x Octo-SPI FSMC SDIO 2x SAI	SHA-256 AES-256 TRNG CAN	MIPI-DSI LCD-TFT Chrom-GRC™	STM32 L4+
	STM32L4 series – Ultra-Low-Power and Performance with DSP, FPU, ART Accelerator™ and Chrom-ART Accelerator™									
	80 MHz Cortex-M4 CPU	Up to 1-Mbyte dual-bank Flash	Up to 320-Kbyte SRAM	USB 2.0 OTG FS	2x 16-bit advanced MC timer	DFSDM Op-amps comp.	Quad-SPI FSMC SDIO 2x SAI	SHA-256 AES-256 TRNG 2x CAN	Up to LCD 8x40	STM32 L4
Communication peripherals: USART, SPI, I ² C	STM32L1 series – Ultra-Low-Power									
Multiple general-purpose timers	32 MHz Cortex-M3 CPU	Up to 512-Kbyte Flash	Up to 80-Kbyte SRAM	Up to 16-Kbyte EEPROM	USB 2.0 FS Device	Op-amps comp.	FSMC SDIO	AES-128	Up to LCD 8x40	STM32 L1
Integrated reset and brown-out warning	STM32L0 series – Ultra-Low-Power									
Multiple DMA	32 MHz Cortex-M0+ CPU	Up to 192-Kbyte SRAM	Up to 20-Kbyte SRAM	Up to 6-Kbyte EEPROM	USB 2.0 FS device Crystal less	DAC comp.	LP ADC 12-/16-bit	TRNG AES-128	LCD 8x48 / 4x52	STM32 L0
2x watchdogs Real-time clock										
Integrated regulator PLL and clock circuit										
Up to 3x 12-bit DAC										
Up to 4x 12-bit ADC (Up to 5 MSPS) Depending on series										
Main oscillator and 32 kHz oscillator										
Low- and high-speed internal RC oscillators										
-40 to +85 °C and up to 125 °C operating temperature range										
Low voltage 2.0 to 3.6 V or 1.65/1.7 to 3.6 V Depending on series										
Temperature sensor										

Figure 14: 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-M0/3/4 based MCUs, with maximum clock frequencies ranging from 48MHz (F0) to over 72MHz (F1/F3).

Wireless MCUs are the brand new lineup of dual-core STM32 microcontrollers with integrated 2.4GHz radio fronted suitable for wireless and Bluetooth applications. These MCUs feature a Cortex-M0+ core (named *Network Processor*) dedicated to the radio management (a companion BLE 5.0 stack is also provided by ST) and a user-programmable Cortex-M4 core (named *Application Processor*) for the main embedded application.

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.

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


Common to all STM32F0	<div><div>STM32 F0</div></div> <div><div>Core: Cortex-M0</div><div>Instruction set: <i>Thumb</i> subset, <i>Thumb-2</i> subset</div><div>Internal RC oscillators: HSI=8MHz , LSI=40KHz</div><div>External clocks: HSE=4 - 32MHz, LSE=32.768 - 1000 KHz</div><div>Maximum Core Frequency: 48MHz</div><div>Low power modes: Sleep, Stop and Standby</div><div>Year of commercialization: 2012</div><div>Available Packages: LQFP(32,48,64,100), TSSOP20, UFBGA(64,100), UFQFPN(28,32,48), WLCSP(25,36,49,64)</div></div>											
		Product Line	FLASH (KB)	RAM (KB)	Operating Voltage	Backup Memory	DAC	Touch Sense	Up to 2xSPI/I ² S, 2xI ² C	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)
MAINSTREAM												

Table 4: STM32F0 features

The STM32F0 series is the famous *32-cents for 32-bit* line of MCU from the STM32 portfolio. It is designed to have a street price able to compete with 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 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, TSSOP20¹⁹, UFBGA, UFQFPN, WLCSP (see **Table 4** for more about this).
- Operating voltage range is 2.0V to 3.6V with the possibility to go down to 1.8V $\pm 8\%$.

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

¹⁹F0/L0 are the only STM32 families that provides this convenient package.

2. Setting-Up the Tool-Chain

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 definitions/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).

There are several complete tool-chains for the STM32 Cortex-M family, both free and commercial. [IAR for Cortex-M¹](#) and [Keil²](#) are two of the most used commercial tool-chains for Cortex-M microcontrollers. They are a complete solution for developing applications for the STM32 platform, but being commercial products they have a street price that may be too high for small sized companies or students (they may cost more than \$5,000 according the features you need). However, this book does not cover commercial IDEs and, if you already have a license for one of these environments, you can skip this chapter, but you will need to arrange the instructions contained in this book according your tool-chain.

[CooCox³](#) and [System Workbench for STM32⁴](#) (shortened as SW4STM32) are two free development environments for the STM32 platform. These IDEs are essentially based on Eclipse and GCC. They do a good job trying to provide support for the STM32 family, and they work out of the box in most cases. However, there are several things to consider while evaluating these tools. First of all, CooCox IDE currently supports only Windows; instead, SWSTM32 provides support for Linux and MacOS too, but it lacks of some additional features found in the tool-chain described in this book. Moreover, they already come with all needed tools preinstalled and configured. While this could be

¹<http://bit.ly/1Qxtkql>

²<http://www.keil.com/arm/mdk.asp>

³<http://www.coocox.org/>

⁴<http://www.openstm32.org/>

an advantage if you are totally new to the development process for Cortex-M processors, it can be a strong limitation if you want to do serious work. It is really important to have the full control over the tools needed to develop your firmware, especially when dealing with Open Source software. So, the best choice is to set up a complete tool-chain from scratch. This allows you to become familiar with the programs and their configuration procedures, giving full control over your development environment. This could be annoying especially at the first time, but it is the only way to learn which piece of software is involved in a given development stage.

In this chapter I will show the required steps to setup a complete tool-chain for the STM32 platform on Windows, Mac OSX and Linux. The tool-chain is based on two main tools, Eclipse and GCC, plus a series of external tools and Eclipse plug-ins that allow you to build STM32 programs efficiently. Although the instructions are essentially equal for the three platforms, I will adapt them for each OS, showing dedicated screen captures and commands. This will simplify the installation procedure, and will allow you to setup a complete tool-chain in less time. This will also give us the opportunity to study in detail every component of our tool-chain. In the next chapter, I will show you how to setup a minimal application (a blinking LED - the *Hello World* application in electronics), which will allow us to test our tool-chain.

2.1 Why Choose Eclipse/GCC as Tool-Chain for STM32

Before we start setting up our tool-chain, there is a really common question to answer: which tool-chain is the best one to develop applications for the STM32 platform? The question is unfortunately not simple to answer. Probably the best answer is that it depends on the type of application. First of all, the audience should be divided between professionals and hobbyists. Companies often prefer to use commercial IDEs with annual fees that allow to receive technical support. You have to figure out that in business time means money and, sometimes, commercial IDE can reduce the learning curve (especially if you consider that ST gives explicit support to these environments). However, I think that even companies (especially small organizations) can take great advantages in using an open source tool-chain.

I think these are the most important reasons to use a Eclipse/GCC tool-chain for embedded development with STM32 MCUs:

- **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 commercial IDEs for STM32 (like TrueSTUDIO and others) 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.

- **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 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.

2.1.1 Two Words About Eclipse...

[Eclipse⁵](http://www.eclipse.org) 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 *Eclipse IDE for C/C++ Developers* is what fits our need.

Eclipse is designed to be expandable thanks to plug-ins. There are several plug-ins available in Eclipse Marketplace really 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 Collection⁶](https://gcc.gnu.org/) (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, ATMEL uses GCC as cross-compiler for its *AVR Studio* development environment.

⁵<http://www.eclipse.org>

⁶<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 a 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 collaborate to the development of ARM GCC. ST Microelectronics does not provide its development environment, but explicitly supports GCC based tool-chains. For this reason, it is relatively simple to setup a complete and working tool-chain to develop embedded applications with GCC.



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 Windows - Installing the Tool-Chain

The whole installation procedure assumes these requirements:

- A Windows based PC with sufficient hardware resources (I suggest to have at least 4Gb of RAM and 5Gb of free space on the Hard Disk); the screen captures in this section are based on Windows 7, but the instructions have been tested successfully on Windows XP, 7, 8.1 and the latest Windows 10.
- Java SE 8 Update 121 or later. If you do not have this version, you can download it for free from official [Java SE support page](#)⁷.



Please, take note that if you have a 64-bit Windows machine, you need to install the 64-bit *Java Virtual Machine* (JVM). Even if it is perfectly possible to use a 32-bit JVM on a 64-bit machine, Eclipse requires that you have a 64-bit Java if using a 64-bit machine.

⁷<http://bit.ly/2k5ppYR>



Choosing a Tool-Chain Folder

One interesting feature of Eclipse is that it is not required to be installed in a specific path on the hard disk. This allows the user to decide where to put the whole tool-chain and, if desired, to move it in another place or to copy it on another machine using a thumb drive (this is really useful if you have several machines to maintain).

In this book we will assume that the whole tool-chain is installed inside the C:\STM32Toolchain folder on the Hard Disk. You are free to place it elsewhere, but rearrange paths in the instructions accordingly.

2.2.1 Windows - Eclipse Installation

The first step is to install the Eclipse IDE. As said before, we are interested in the Eclipse version for C/C++ developers. The latest version at time of revising this chapter (August 2018) is Photon (Eclipse v4.8). However, it is strongly suggested to use the previous release, that is Oxygen.3a (Eclipse v4.7.3a), since the newest one is still not supported by the GNU MCU Eclipse plug-ins suite and by several other tools used in this book. It can be downloaded from the official [download page](#)⁸ as shown in Figure 1⁹.

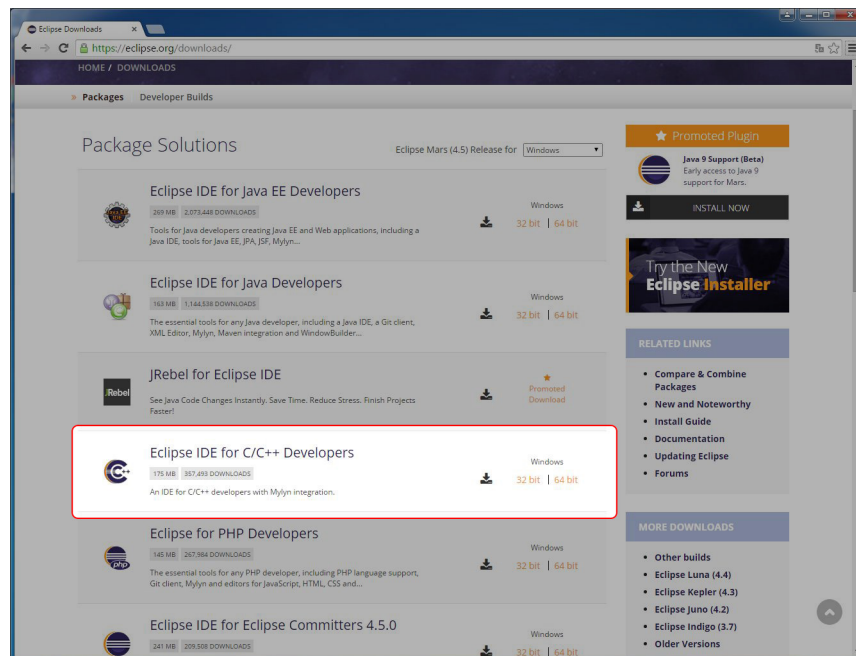


Figure 1: Eclipse download page

Choose the release (32bit or 64bit) for your PC.

⁸<https://www.eclipse.org/downloads/packages/release/oxygen/3a/>

⁹Some screen captures may appear different from the ones reported in this book. This happens because the Eclipse IDE is updated frequently. Don't worry about that: the installation instructions should work in any case.

The Eclipse IDE is distributed as a ZIP archive. Extract the contents of the archive inside the folder `C:\STM32Toolchain`. At the end of the process you will find the folder `C:\STM32Toolchain\eclipse` containing the whole IDE.

Now we can execute for the first time the Eclipse IDE. Go inside the `C:\STM32Toolchain\eclipse` folder and run the `eclipse.exe` file. After a while, Eclipse will ask you for the preferred folder where all Eclipse projects are stored (this is called *workspace*), as shown in **Figure 2**.

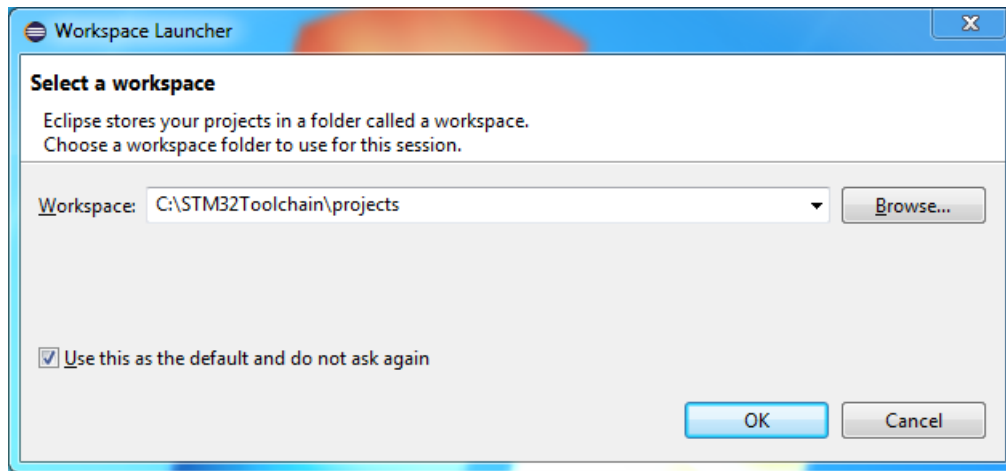


Figure 2: Eclipse workspace setting

You are free to choose the folder you prefer, or leave the suggested one. In this book we will assume that the Eclipse workspace is located inside the `C:\STM32Toolchain\projects` folder. Arrange the instructions accordingly if you choose another location.

2.2.2 Windows - Eclipse Plug-Ins Installation

Once Eclipse is started, we can proceed to install some relevant plug-ins.



What Is a Plug-In?

A plug-in is an external software module that extends Eclipse functionalities. A plug-in must adhere to a standard API defined by Eclipse developers. In this way, it is possible for third party developers to add features to the IDE without changing the main source code. We will install several plug-ins in this book to adapt Eclipse to our needs.

The first plug-in we need to install is the *C/C++ Development Tools SDK*, also known as Eclipse CDT, or simply CDT. CDT provides a fully functional C and C++ *Integrated Development Environment* (IDE) based on the Eclipse platform. Features include: support for project creation and managed build for various tool-chains, standard make build, source navigation, various source knowledge tools, such as type hierarchy, call graph, includes browser, macro definition browser, code editor with syntax highlighting, folding and hyperlink navigation, source code refactoring and code generation, visual debugging tools, including memory, registers, and disassembly viewers.

To install CDT we have to follow this procedure. Go to *Help->Install new software...* as shown in **Figure 3**.

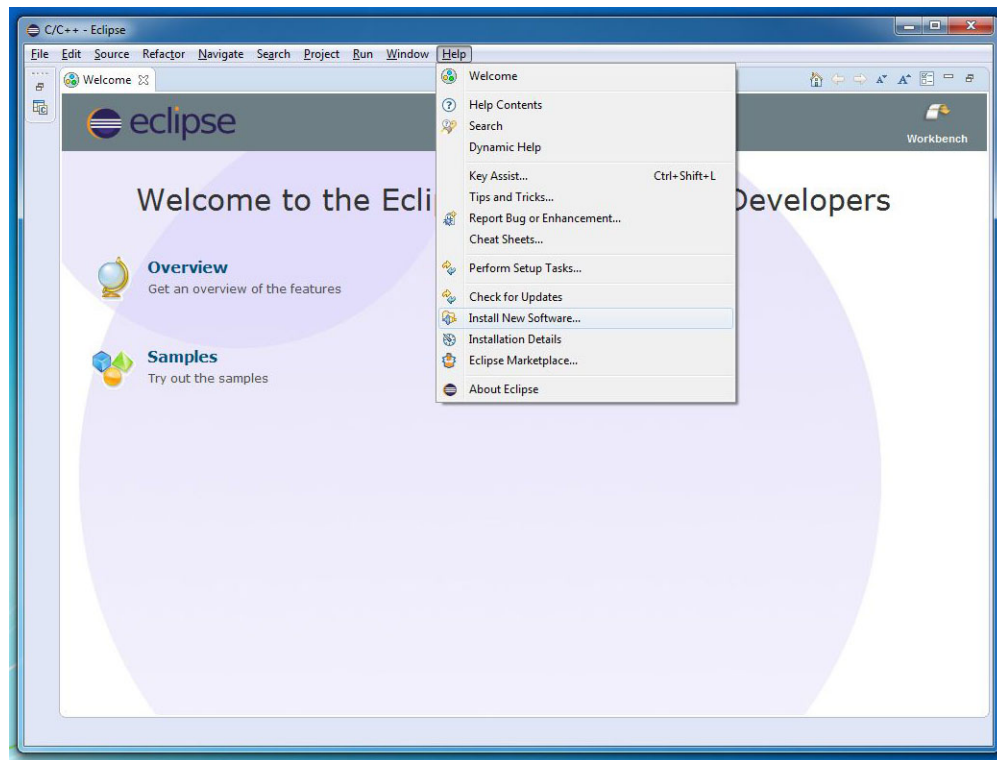


Figure 3: Eclipse plug-in install menu

In the plug-ins install window, we need to enable other plug-in repositories by clicking on *Manage...* button. In the Preferences window, select the “*Install/Update->Available Software Sites*” entry on the left and then check “*CDT*” entry as shown in **Figure 4**. Click on the OK button.

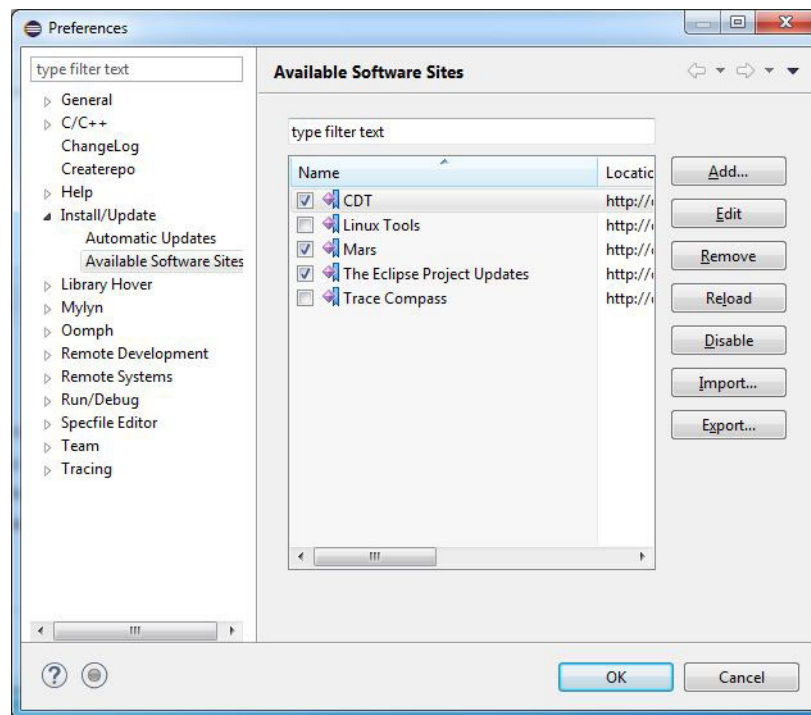


Figure 4: Eclipse plug-in repository selection

Now, from “work with” drop-down menu choose “CDT” repository, as shown in **Figure 5**, and then select “CDT Main Features->C/C++ Development Tools” and “CDT Optional Features->C/C++ GDB Hardware Debugging” entries, as shown in **Figure 6**. Click on “Next” button and follow the instructions to install the plug-in. At the end of installation process (the installation takes a while depending your Internet connection speed), restart Eclipse when requested.

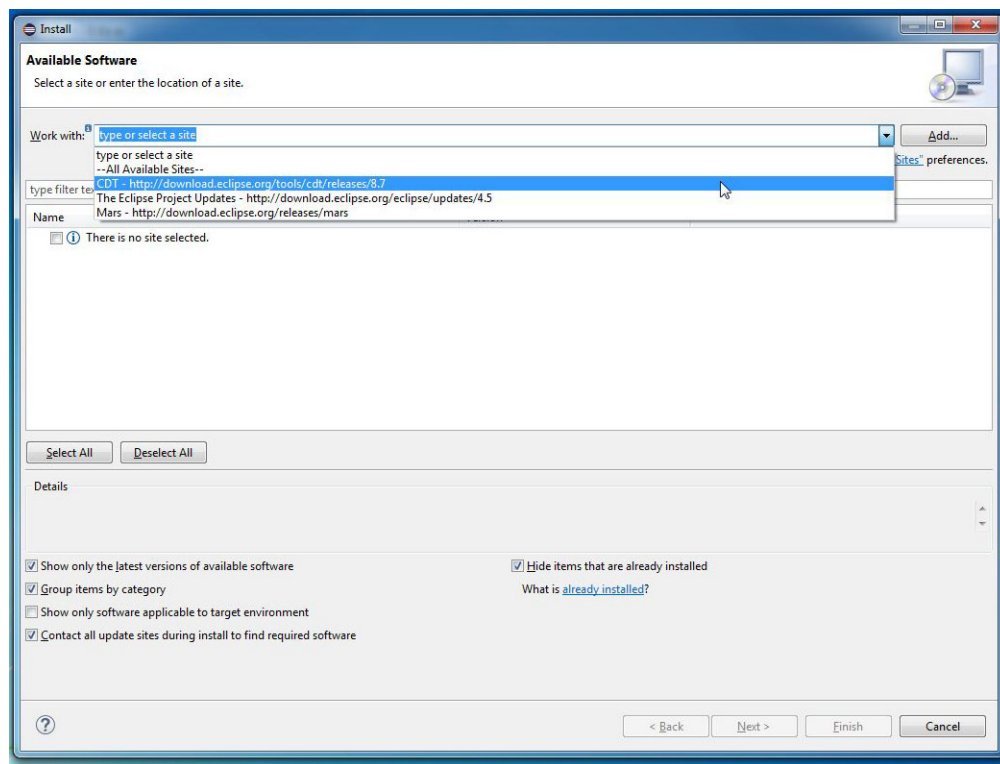


Figure 5: CDT repository selection

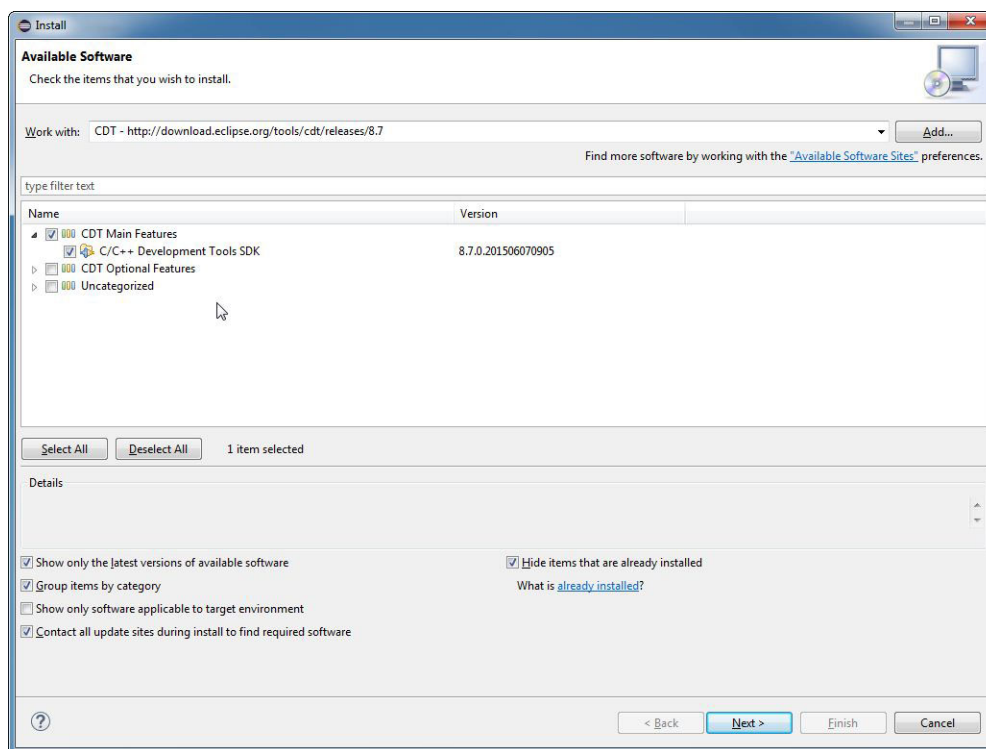


Figure 6: CDT plug-in selection

Now we have to install the [GNU MCU plug-ins for Eclipse](https://gnu-mcu-eclipse.github.io/)¹⁰. These plug-ins add a rich set of features to Eclipse CDT to interface the GCC ARM tool-chain. Moreover, they provide specific functionalities for the STM32 platform. Plug-ins are developed and maintained by Liviu Ionescu, who did a really excellent work in providing support for the GCC ARM tool-chain. Without these plug-ins it is almost impossible to develop and run code with Eclipse for the STM32 platform.

To install GCC ARM plug-ins go to *Help->Install new software....* In the Install window, click the **Add...** button and fill the fields in the following way (see Figure 7):

Name: GNU MCU Eclipse Plug-ins

Location: <http://gnu-mcu-eclipse.netlify.com/v4-neon-updates>

Click on the **OK** button. Now, from “work with” drop-down menu choose “GNU MCU Eclipse Plug-ins” repository. A list of installable packages appears. Check the packages to install according to Figure 8.

¹⁰<https://gnu-mcu-eclipse.github.io/>

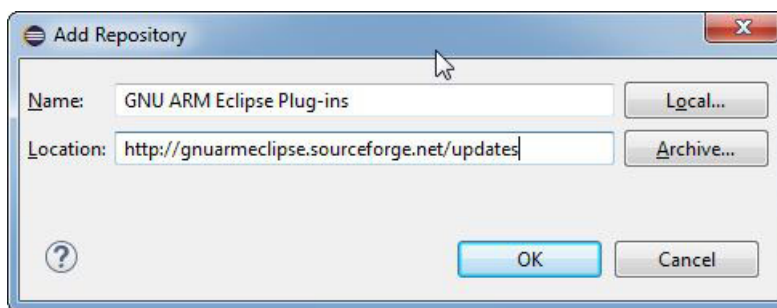


Figure 7: GNU MCU plug-ins installation

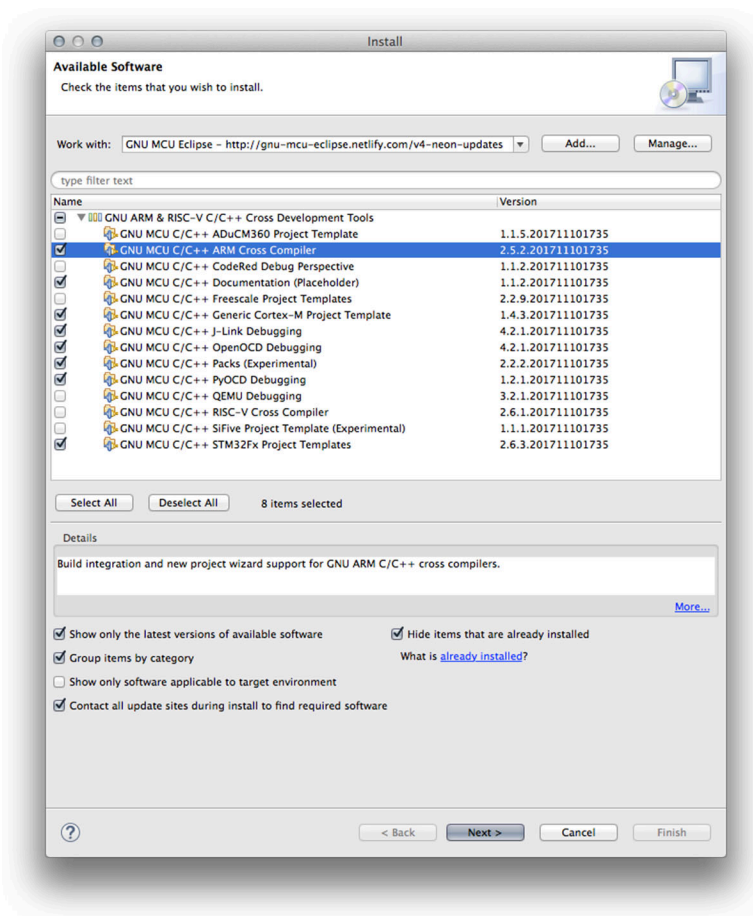


Figure 8: GNU MCU plug-ins selection

Click on “Next >” button and follow the instructions to install the plug-ins. At the end of installation process, restart Eclipse when requested. he end of installation process, restart Eclipse when requested.



Read Carefully

If you run in troubles during the plug-ins installation (handshake error, provisioning error or something like that), please refer to the [troubleshooting section](#).

Eclipse is now essentially configured to start developing STM32 applications. Now we need the cross-compiler suite to generate the firmware for the STM32 family.

2.2.3 Windows - GCC ARM Embedded Installation

The next step in tool-chain configuration is installing the GCC suite for ARM Cortex-M and Cortex-R microcontrollers. This is a set of tools (macro preprocessor, compiler, assembler, linker and debugger) designed to cross-compile the code we will create for the STM32 platform.

The latest release of ARM GCC can be downloaded from [ARM Developer](https://developer.arm.com/open-source/gnu-toolchain/gnu-rm)¹¹. At the time of writing this chapter, the latest available version is the 6.0. The Windows Installer can be downloaded from the [download section](#)¹².

Once download is complete, run the installer. When the installer asks for the destination folder, choose C:\STM32Toolchain\gcc-arm and then click on “*Install*” button, as shown in **Figure 9**.

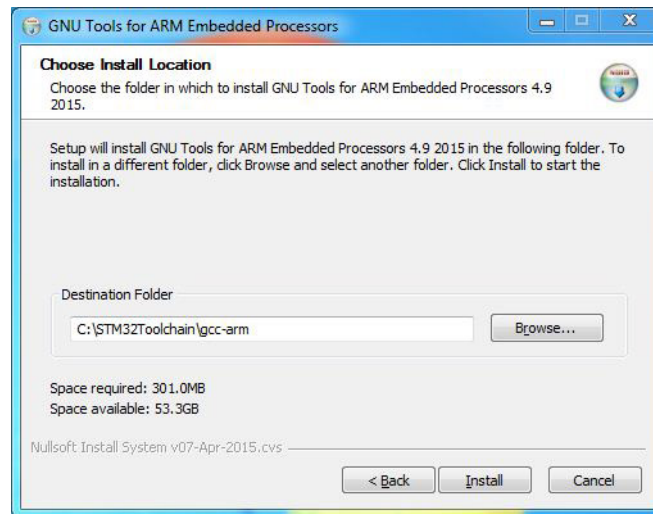


Figure 9: Selection of GCC destination folder



The installer, by default, suggests a destination folder that is related to the GCC version we are going to install (6.0 2017q2). This is not convenient, because when GCC is updated to a newer version we need to change settings for each Eclipse project we have made.

Once the installation is complete, the installer will show us a form with four different checkboxes. If only one GCC is installed on your system, or you do not know, check the entry **Add path to environment variable** and **Add registry information** (two checked boxes), as shown in **Figure 10**.

¹¹<https://developer.arm.com/open-source/gnu-toolchain/gnu-rm>

¹²<https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads>

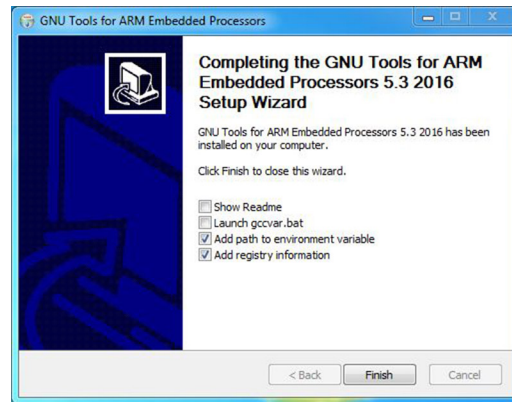


Figure 10: Final GCC install options



If you have multiple copies of GCC installed in your system, then I suggest to leave that two options unchecked, and to handle the PATH environment variable using Eclipse. Refer to the Troubleshooting Appendix (paragraph named “[Eclipse cannot locate the compiler](#)”) where it is explained how to configure GCC paths in Eclipse.

2.2.4 Windows – Build Tools Installation

Windows historically lacks some tools that are a must in the UNIX world. One of these is *make*, the tool that controls the compilation process of programs written in C/C++. If you have already installed a product like MinGW or similar (and it is configured in your PATH environment correctly), you can skip this process. If not, you can install the *Build Tools* package made by the same author of GCC ARM plug-ins for Eclipse. You can download setup program from [here](#)¹³. Choose the version that fits your OS release (32 or 64 bit). At the time of writing this chapter, the last available version is 2.8.

When asked, install the tools in this folder: `C:\STM32Toolchain\Build Tools`. Restart Eclipse if it is already running.

2.2.5 Windows – OpenOCD Installation

[OpenOCD](#)¹⁴ is a tool that allows to upload the firmware on the Nucleo board and to do the step-by-step debugging. Originally started by Dominic Rath, OpenOCD is now actively maintained by the community and several companies, including STM. We will discuss it in depth in Chapter 5, which is dedicated to the debugging. But we will install it in this chapter, because the procedure changes between the three different platforms (Windows, Linux and Mac OS). The latest official release at the time of writing this book is the 0.10.

Compiling a tool like OpenOCD, expressly designed to be compiled on UNIX like systems, is not a trivial task. It requires a complete UNIX C tool-chain like MinGW or Cygwin. Luckily, Liviu

¹³<http://bit.ly/2g2bu5R>

¹⁴<http://openocd.org/>

Ionescu has already done the dirty job for us. You can download the latest development version of OpenOCD (0.10.0-5-20171110-* at the time of writing this chapter) from the [GNU MCU Eclipse official repository](#)¹⁵. Choose the .exe package for your Windows platform (32- or 64-bits). When asked, install the files inside the C:\STM32Toolchain\openocd folder (pay attention to write openocd as-is).



Once again, this ensures us that we should not change Eclipse settings when a new release of OpenOCD will be released, but we only need to replace the content inside C:\STM32Toolchain\openocd folder with the new software release.

2.2.6 Windows – ST Tools and Drivers Installation

ST provides several tools that are useful for developing STM32 based applications. We will install them in this chapter, and we will discuss their use later in this book.

STM32CubeMX is a graphical tool used to generate setup files in C programming language for an STM32 MCU, according the hardware configuration of our board. For example, if we have the Nucleo-F401RE, which is based on the STM32F401RE MCU, and we want to use its user LED (marked as LD2 on the board), then STM32CubeMX will automatically generate all source files containing the C code required to configure the MCU (clock, peripheral ports, and so on) and the GPIO connected to the LED (port GPIO 5 on port A on almost all Nucleo boards). You can download STM32CubeMX from the official [ST website](#)^{16,17} (the download link is at the bottom of the page), and follow the installation instructions.

Another important tool is the [STM32CubeProgrammer](#)¹⁸. 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 use it in the next chapter. The STM32CubeProgrammer installation package also provides necessary drivers to interface ST development boards with Windows. You can download STM32CubeProgrammer from the official [ST page](#)¹⁹ (the download link is at the bottom of the page in the GET SOFTWARE section), and follow the installation instructions.

2.2.6.1 Windows – ST-LINK Firmware Upgrade



Warning

Read this paragraph carefully. Do not skip this step!

I bought several Nucleo boards and I saw that all boards come with an old ST-LINK firmware. In order to use the Nucleo with OpenOCD, the firmware must be updated at least to the 2.29.18 version.

¹⁵<http://bit.ly/2khxhXL>

¹⁶<http://bit.ly/1RLCa4G>

¹⁷To download the software, you need to register to the ST website providing a valid email.

¹⁸<http://bit.ly/2CK4aFa>

¹⁹<http://bit.ly/2CK4aFa>

Once the ST-LINK drivers are installed, we can download the latest ST-LINK firmware update from [ST website](#)²⁰. The firmware is distributed as ZIP file. Extract it in a convenient place. Connect your Nucleo board using a USB cable and go inside the Windows sub-folder and execute the file ST-LINKUpgrade. Click on *Device Connect* button.

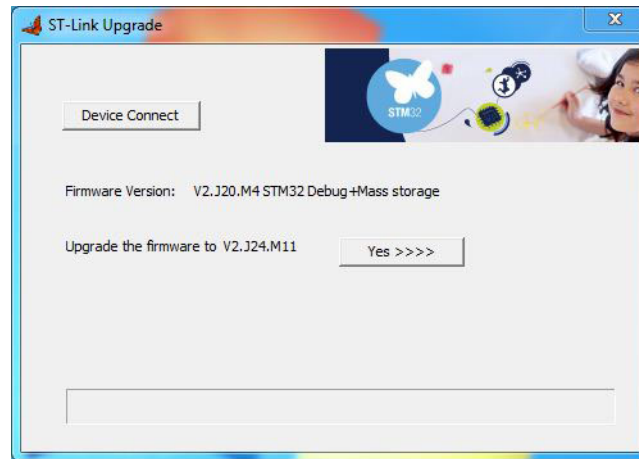


Figure 12: The ST-LINK Upgrade program

After a while, ST-LINK Upgrade will show if your Nucleo firmware needs to be updated (pointing out a different version, as shown in **Figure 12**). If so, click on *Yes >>>>* button and follow the instructions.

Congratulation. The tool-chain is now complete, and you can jump to the [next chapter](#).

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

²⁰<http://bit.ly/1RLDp3H>

3. Hello, Nucleo!

There is no programming book that does not begin with the classic “Hello world!” program. And this book will follow the tradition. In the previous chapter we have configured the development environment needed to program STM32 based boards. So, we are now ready to start coding.

In this chapter we will create a really basic program: a blinking LED. We will use the GNU MCU Eclipse plug-in to create a complete application in a few steps without dealing, in this phase, with aspects related to the ST *Hardware Abstraction Layer* (HAL). I am aware that not all details presented in this chapter will be clear from the beginning, especially if you are totally new to embedded programming.

However, this first example will allow us to become familiar with the development environment. Following chapters, especially the [next one](#), will clarify a lot of obscure things. So I suggest you to be patient and try to take the best from the following paragraphs.



A Note on GNU MCU Eclipse Plug-ins

Experienced programmers might observe that these plug-ins are not strictly necessary to generate code for the STM32 platform. It is perfectly possible to start importing the HAL in an empty C/C++ project and to configure the tool-chain accordingly. Moreover, as we will see in the [next chapter](#), it is better to directly use the code from the latest HAL release and the one automatically generated by STM32CubeMX tool. However, the GNU MCU plugin brings several features that simplify the project management. Moreover, I think that for newbies it is recommended to start with an automatic-generated project to avoid a lot of confusion. When writing code for the STM32 platform, we need to deal with a lot of tools and libraries. Some of them are mandatory, while others may lead to confusion. So it is best to start gradually and dive inside the whole stack. Once you get familiar with the development environment, it will be really easy to adapt it to your needs.

If you are totally new to Eclipse IDE, the next paragraph will briefly explain its main functionalities.

3.1 Get in Touch With the Eclipse IDE

When you start Eclipse, you might be a bit puzzled by its interface. **Figure 1¹** shows how Eclipse appears when started for the first time.

¹Starting from this chapter, 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.

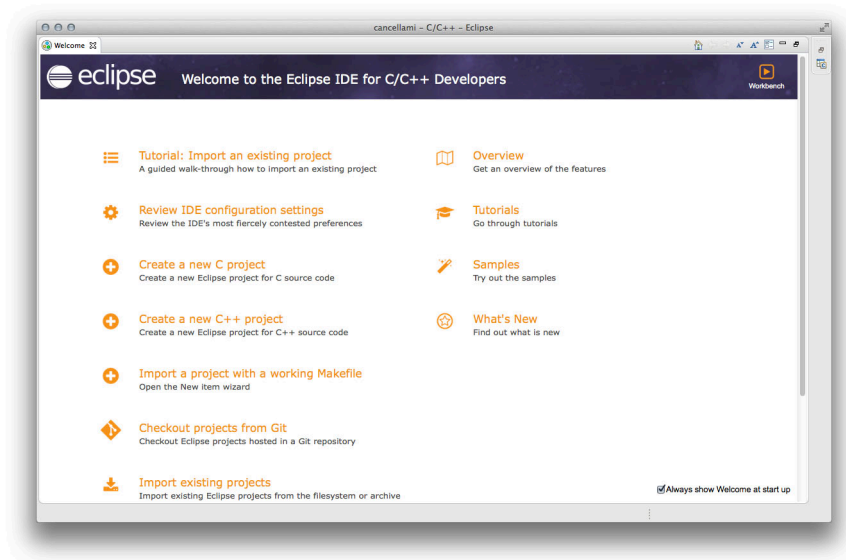


Figure 1: The Eclipse interface once started for the first time

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*.

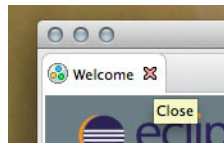


Figure 2: How to close the *Welcome view* by clicking on the *X*.

To close the *Welcome view*, click on the cross icon, as shown in **Figure 2**. Once the *Welcome view* goes away, the *C/C++ perspective* appears, as shown in **Figure 3**.

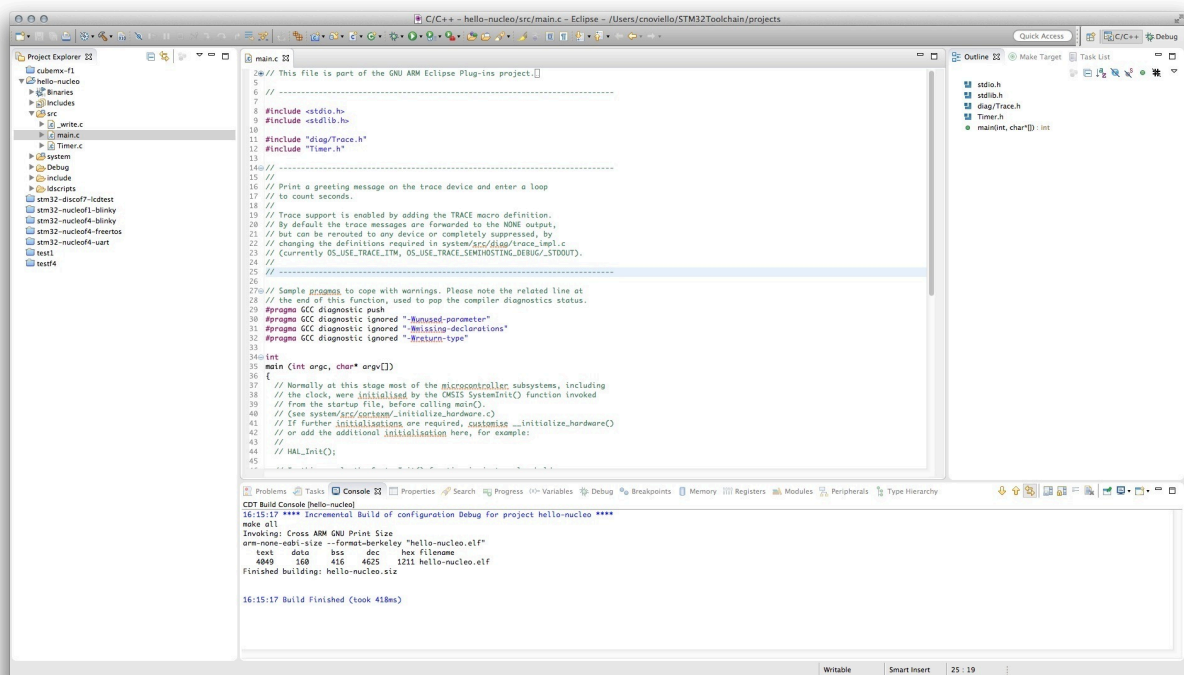


Figure 3: The C/C++ perspective view in eclipse (with a main.c file loaded later)

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.



If you recall from the previous chapter, the first time we started Eclipse we had to choose the workspace directory. The *workspace* is the place where a group of projects are stored. Please note that we say *a group of projects* and not *all the projects*. This means that we can have several workspaces (that is, directories) where different groups of projects are stored. However, a workspace also contains IDE configurations, and we can have different configurations for every 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 view in the bottom of Eclipse window is dedicated to several activities related to coding and compiling, and it is 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 content of 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.



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 4** 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.

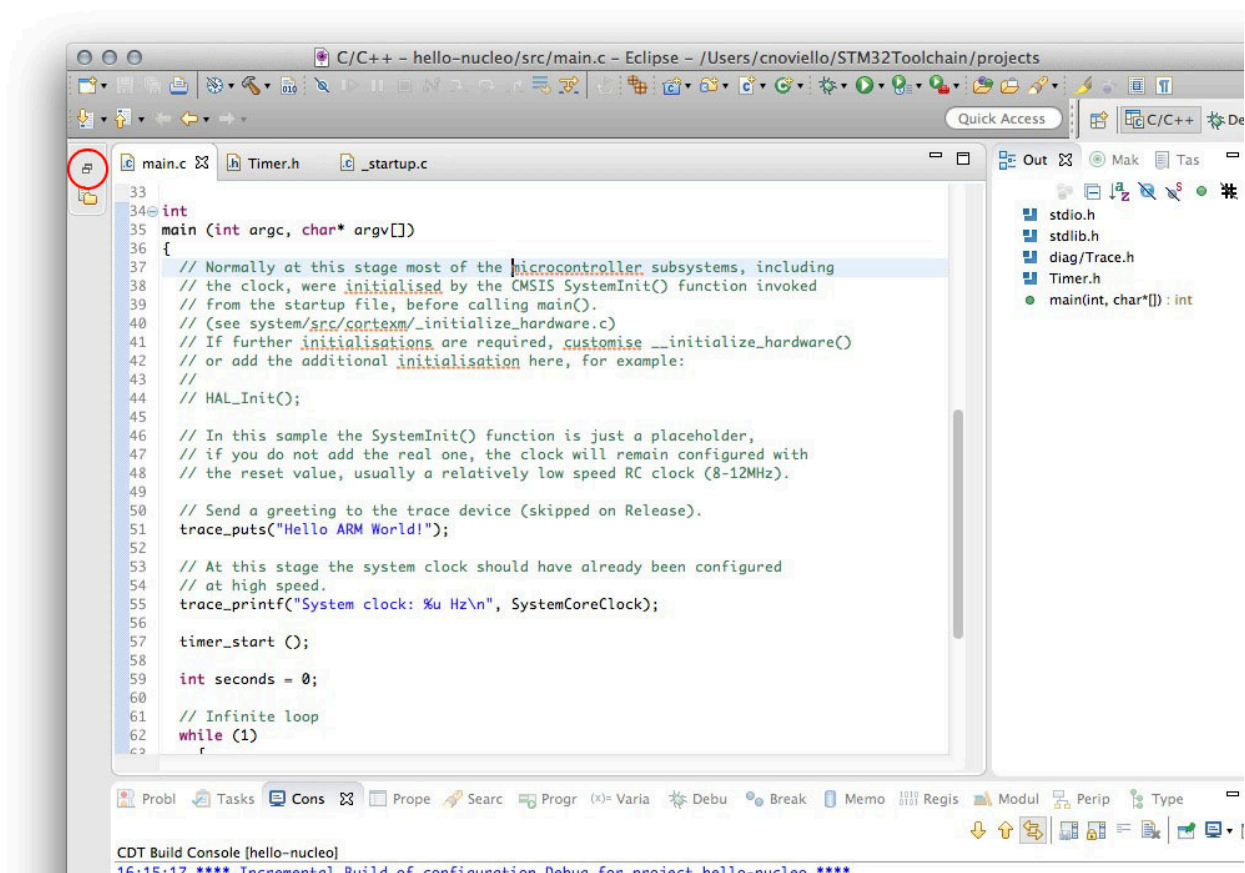


Figure 4: *Project Explorer* view minimized

To switch between different perspectives you can use the specific toolbar available in the top-right side of Eclipse (see **Figure 5**)

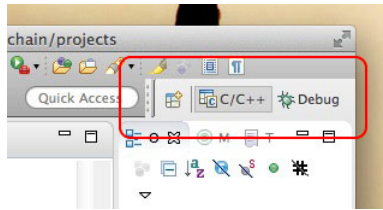


Figure 5: Perspective switcher toolbar

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.

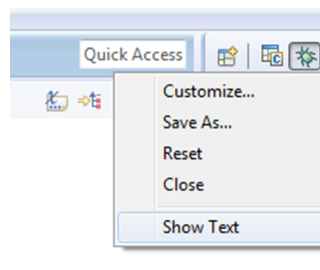


Figure 6: How to enable the name of a perspective in the *perspective switcher toolbar*

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

3.2 Create a Project

Let us create our first project. We will create a simple application that makes the Nucleo LD2 LED (the green one) blink.

Go to **File->New->C Project**. Eclipse shows a wizard that allows us to create our test project (see Figure 7).

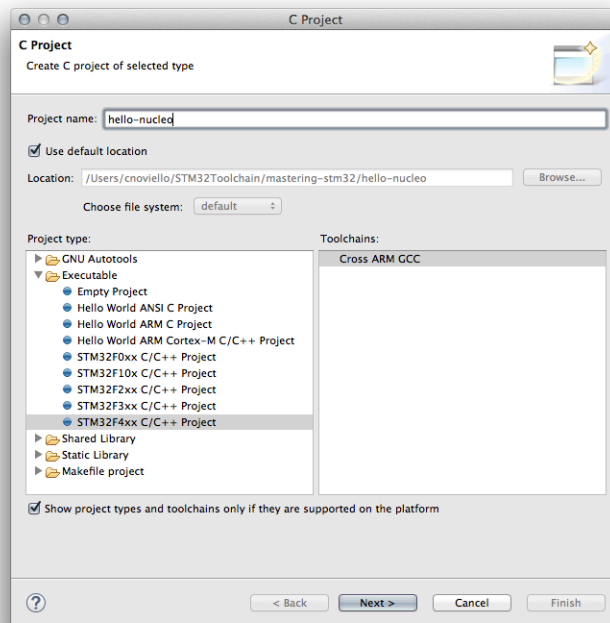


Figure 7: Project wizard - STEP 1

In the **Project name** field write *hello-nucleo* (you are totally free to choose the project name you like). The important part, indeed, is the **Project type** section. Here we have to choose the STM32 family of our Nucleo board. For example, if we have a *NUCLEO-F401RE* we have to choose *STM32F4xx C/C++ Project*.



Unfortunately, Liviu Ionescu still has not implemented project templates for the STM32L0/1/4 families. Moreover, project templates for some Nucleo boards are missed. If your Nucleo is based on one of these series, you have to jump to the next chapter, where we will see a more general way to generate projects for the STM32 platform. However, it could be that by the time you read this chapter, the plug-in has been updated with new templates.

Now click on the **Next** button. In this step of the wizard it is **really important** to select the right size of RAM and flash memory (if those fields do not match the quantity of RAM and flash of the MCU equipping your Nucleo, it will be impossible to start the example application)². Use **Table 1** to choose the correct values for your Nucleo board³.

²Owners of STM32F4 and STM32F7 development boards will not find the entry to specify the RAM size. Do not complain about this, since the project wizard is designed to properly configure the right amount of RAM if you choose the right **Chip family** type.

³In case you are using a different development board (e.g. a Discovery kit), check on the ST web site for right values of RAM and flash.

Nucleo P/N	STM32 MCU to select in wizard	Cortex-M Core	RAM (KB)	CCM RAM (KB)	FLASH (KB)
NUCLEO-F446RE	STM32F446xx	M4	128	-	512
NUCLEO-F411RE	STM32F411xE	M4	128	-	512
NUCLEO-F410RB	STM32F410Rx	M4	32	-	128
NUCLEO-F401RE	STM32F401xE	M4	96	-	512
NUCLEO-F334R8	N/A	M4	12	4	64
NUCLEO-F303RE	STM32F30x/31x	M4	64	16	512
NUCLEO-F302R8	N/A	M4	16	-	64
NUCLEO-F103RB	STM32F10x Medium Density	M3	20	-	128
NUCLEO-F091RC	N/A	M0	32	-	128
NUCLEO-F072RB	STM32F072	M0	16	-	128
NUCLEO-F070RB	N/A	M0	16	-	128
NUCLEO-F030R8	STM32F030	M0	8	-	64
NUCLEO-L476RG	N/A	M4	96	-	1024
NUCLEO-L152RE	N/A	M3	80	-	512
NUCLEO-L073RZ	N/A	M0+	20	-	192
NUCLEO-L053R8	N/A	M0+	8	-	64

Table 1: RAM and flash size to select according the given Nucleo

So, fill the fields of second step in the following way⁴ (see **Figure 8** for reference):

Chip Family: Select the exact MCU equipping your Nucleo (see **Table 1**).

Flash size: pick the right value from **Table 1**.

RAM size: pick the right value from **Table 1**.

External clock(Hz): it is ok to leave this field as is.

Content: Blinky (blink a LED).

Use system calls: Freestanding (no POSIX system calls).

Trace output: None (no trace output).

Check some warnings: *Checked*.

Check most warnings: Unchecked.

Enable -Werror: Unchecked.

Use -Og on debug: *Checked*.

Use newlib nano: *Checked*.

Exclude unused: *Checked*.

Use link optimizations: Unchecked.

F334

F303

Those of you having a STM32F3 Nucleo, will find an additional field in the wizard step. It is named **CCM RAM Size (KB)**, and it is related to the *Core Coupled Memory* (CCM), a special internal and fast memory that we will study in a [following chapter](#). If you have a Nucleo-F334 or a Nucleo-F303 board, fill the field with the value from **Table 1**. For other STM32F3 based boards place a zero in that field.

⁴Please, take note that, depending the actual STM32 family of your development board, some of those fields may be absent in the second step. Don't care about this, because it means that the project generator knows how to fill them.

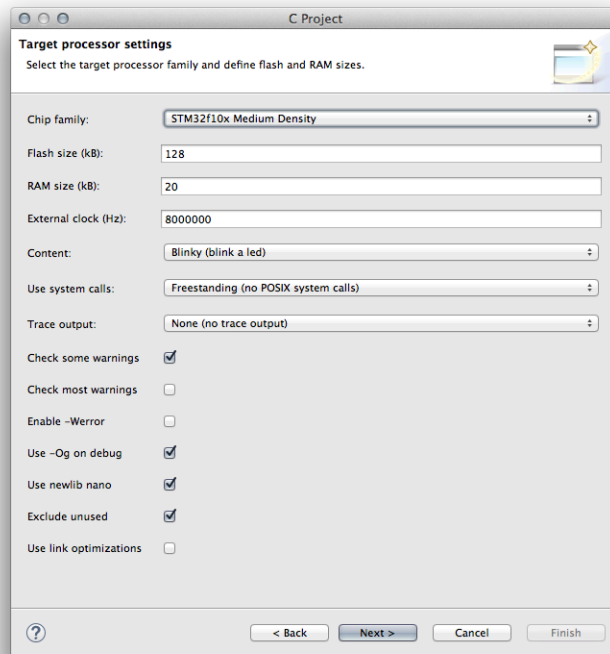


Figure 8: Project wizard - STEP 2

Now click on the **Next** button. In the next two wizard steps, leave all parameters as default. Finally, in the last step you have to select the GCC tool-chain path. In the previous chapter, we have installed GCC inside the `~/STM32Toolchain/gcc-arm` folder (in Windows the folder was `C:\STM32Toolchain\gcc-arm`). So, select that folder as shown in **Figure 9** (either typing the pathname or using the Browse button), and ensure that the **Toolchain name** field contains *GNU Tools for ARM Embedded Processors (arm-none-eabi-gcc)*, otherwise select it from the drop-down menu. Click on the **Finish** button.

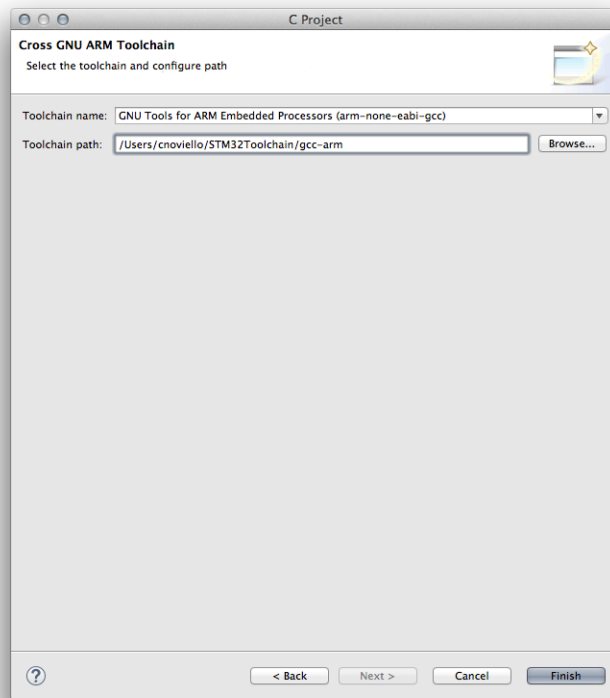


Figure 9: Project wizard - STEP 5

Our test project is almost complete. We only need to modify one thing to make it work on the Nucleo. However, before we complete the example, it is better to take a look at what has been generated by the GNU MCU plug-in.

Figure 10 shows what appears in the Eclipse IDE after the project has been generated. The *Project Explorer* view shows the project structure. This is the content of the first-level folders (going from top to bottom):

Includes: this folder shows all folders that are part of the *GCC Include Folders*⁵.

src: this Eclipse folder contains the `.c` files⁶ that make up our application. One of these files is `main.c`, which contains the `int main(int argc, char* argv[])` routine.

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

include: this folder contains the header files of our main application.

ldscripts: this folder contains some relevant files that make our application work on the MCU. These are LD (the GNU Link eDitor) script files, and we will study them in depth in a [following chapter](#).

⁵Every C/C++ compiler needs to be aware of where to look for include files (files ending with `.h`). These folders are called *include folders* and their path must be specified to GCC using the `-I` parameter. However, as we will see later, Eclipse is able to do this for us automatically.

⁶The exact type and amount of files in this folder depends on the STM32 family. Do not worry if you see additional files than the ones shown in **Figure 10**, and focus your attention exclusively on the `main.c` file.

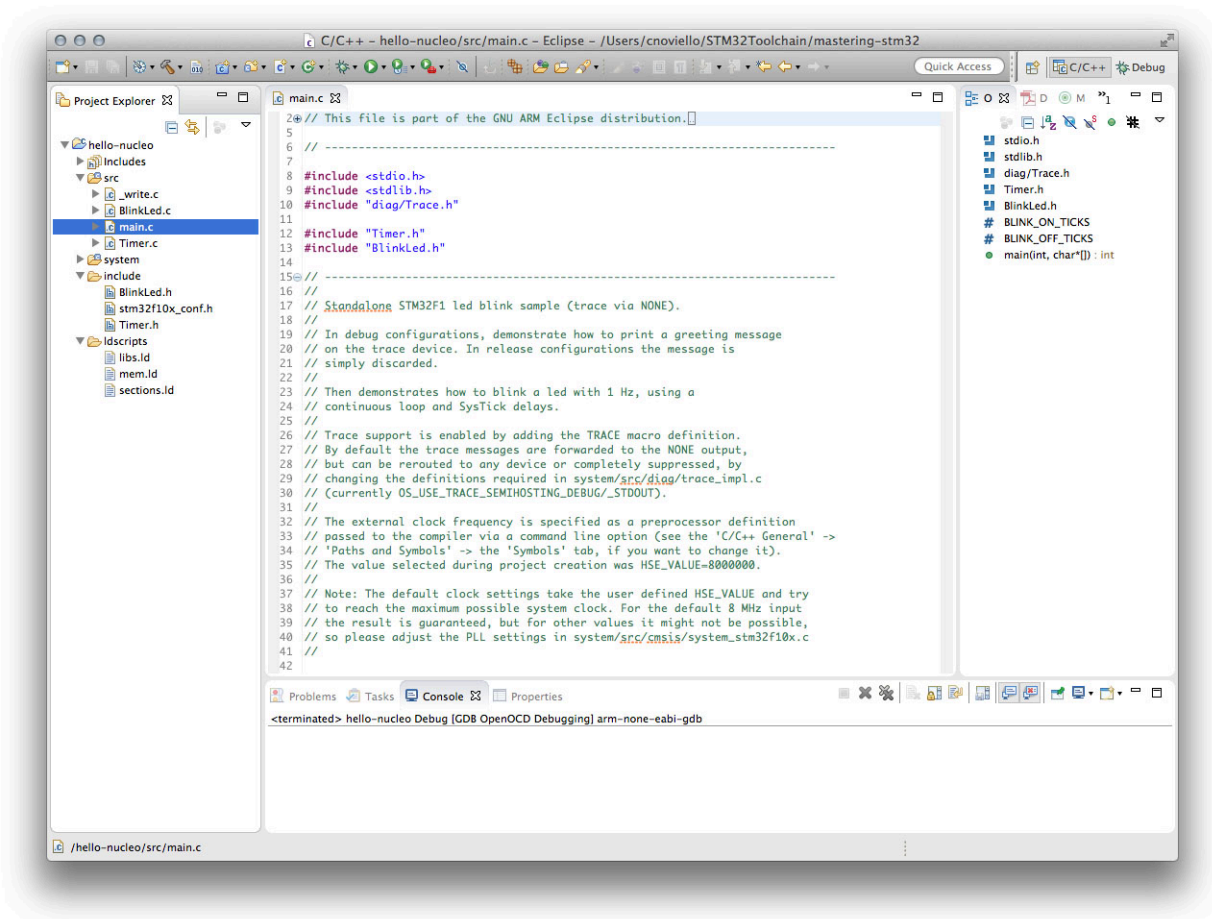


Figure 10: The project content after its generation

As said before, we need to modify one more thing to make the example project work on our Nucleo board. The GNU MCU plugin generates an example project that fits the Discovery hardware layout. This means that the LED is routed to a different MCU I/O pin. We need to modify this.

How can we know to which pin the LED is connected? ST [provides schematics](http://bit.ly/1FAVXSw)⁷ of the Nucleo board. Schematics are made using the *Altium Designer* CAD, a really expensive piece of software used in the professional world. However, luckily for us, ST provides a convenient PDF with schematics. Looking at page 4, we can see that the LED is connected to the PA5 pin⁸, as shown in **Figure 11**.

⁷<http://bit.ly/1FAVXSw>

⁸Except for the Nucleo-F302RB, where LD2 is connected to PB13 port. More about this next.

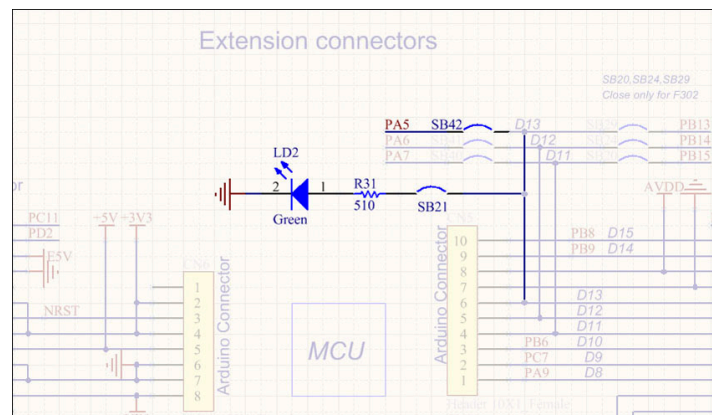


Figure 11: LD2 connection to PA5

PA5 is shorthand for PIN5 of GPIOA port, which is the standard way to indicate a GPIO in the STM32 world.

We can now proceed to modify the source code. Open the `Include/BlinkLed.h` and go to line 19. Here we find the macro definition for the GPIO associated to the LED. We need to change the code in the following way:

Filename: `include/BlinkLed.h`

```
30 #define BLINK_PORT_NUMBER      (0)
31 #define BLINK_PIN_NUMBER      (5)
```

BLINK_PORT_NUMBER defines the GPIO port (in our case GPIOA=0), and BLINK_PIN_NUMBER the pin number.

Nucleo-F302

Nucleo-F302R8 is the only Nucleo board that has a different hardware configuration regarding the pin used for LED LD2, because it is connected to pin PB13, as you can see in schematics. This means that the right pin configuration is:

```
30 #define BLINK_PORT_NUMBER (1)
31 #define BLINK_PIN_NUMBER (13)
```

We can now compile the project. Go to menu **Project->Build Project**. After a while, we should see something similar to this in the output console[^ch3-flash-image-size].

Invoking: Cross ARM GNU Create Flash Image

```
arm-none-eabi-objcopy -O ihex "hello-nucleo.elf" "hello-nucleo.hex"
```

Finished building: hello-nucleo.hex

Invoking: Cross ARM GNU Print Size

```
arm-none-eabi-size --format=berkeley "hello-nucleo.elf"
```

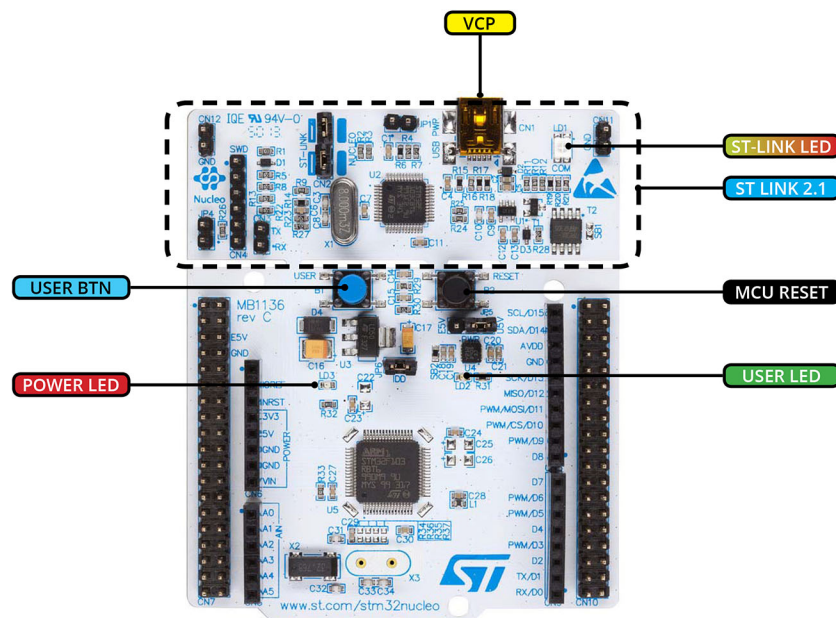
text	data	bss	dec	hex	filename
5697	176	416	6289	1891	hello-nucleo.elf

Finished building: hello-nucleo.siz

09:52:01 Build Finished (took 6s.704ms)

3.3 Connecting the Nucleo to the PC

Once we have compiled our test project, you can connect the Nucleo board to your computer using an USB cable connected to micro-USB port (called **VCP** in Figure 12). After few seconds, you should see at least two LED turning ON.



The first one is the LD1 LED, which in Figure 12 is called **ST-LINK LED**. It is a red/green LED and it is used to signal the ST-LINK activity: once the board is connected to the computer, that LED is green; during a debug session or while uploading the firmware on the MCU it blinks green and red alternatively.

Another LED that turns ON when the board is connected to the computer is the LED LD3, which is called **POWER LED** in Figure 12. It is a red LED that turns ON when the USB port ends *enumeration*, that is the ST-LINK interface is properly recognized by the computer OS as a USB peripheral. The

target MCU on the board is powered only when that LED is ON (this means that the ST-LINK interface also manages the powering of the target MCU).

Finally, if you have not still flashed your board with a custom firmware, you will see that the LD2 LED, a green LED named **USER LED** in **Figure 12**, also blinks: this happens because ST preloads the board with a firmware that makes the LD2 LED blinking. To change the blinking frequency you can press the **USER BUTTON** (the blue one).

Now we are going to replace the on-board firmware with the one made by us before.

3.4 Flashing the Nucleo using STM32CubeProgrammer

ST has recently introduced a new and really practical tool to flash firmware on the target board: STM32CubeProgrammer. Its aim is to replace the historical ST-LINK Utility tool and the good news is that it is finally multi-platform. The tool is not still perfectly stable, but I am sure that next releases will fix its early bugs. **Figure 13** shows the STM32CubeProgrammer's main interface.

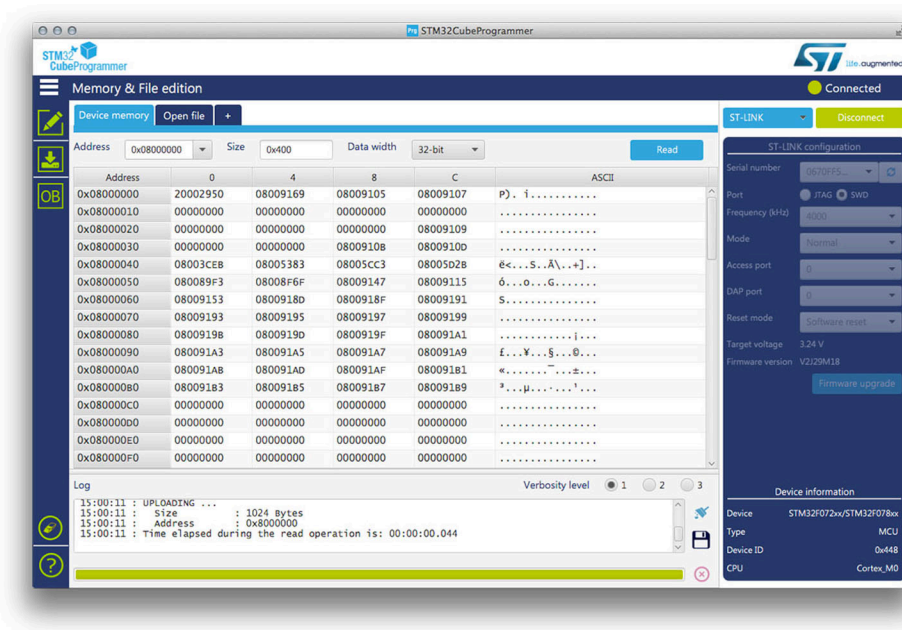


Figure 13: The STM32CubeProgrammer interface once connected to the board

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

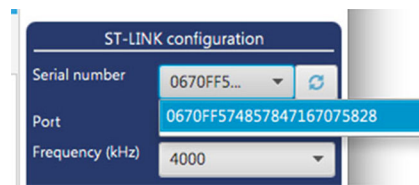


Figure 14: The ST-LINK interface serial number as shown by STM32CubeProgrammer tool



Read Carefully

If the label “*Old ST-LINK Firmware*” appears instead of the ST-LINK interface serial number, then you need to update the ST-LINK firmware to the latest version. Click on the **Firmware upgrade** button at the bottom of the **ST-LINK Configuration** pane and following the instruction. Alternatively, follow the upgrade instructions reported in Chapter 2.

Once the ST-LINK board has been identified, click the **Connect** button. After a while you will see the content of flash memory, as shown in **Figure 13** (ensure that all connection parameters are the same of the ones reported in **Figure 13**).

Ok, let us upload the example firmware to the board. Click on the **Erase & programming** icon (the second green icon on the left). Then, click on the **Browse** button in the **File programming** section and select the file `C:\STM32Toolchain\projects\hello-nucleo\Debug\hello-nucleo.hex` in Windows or `~/STM32Toolchain/projects/hello-nucleo/Debug/hello-nucleo.hex` in Linux and Mac OS. Check the **Verify programming** and **Run after programming** flags and click on **Start Programming** button to start flashing. At the end of flashing procedure your Nucleo green LED will start blinking. Congratulations: welcome to the STM32 world ;-)

3.5 Understanding the Generated Code

Now that we brought a cold piece of hardware to life, we can give a first look at the code generated by the GNU MCU plugin. Opening `main.c` file we can see the content of `main()` function, the entry point⁹ of our application.

⁹Experienced STM32 programmers know that it is improper to say that the `main()` function is the entry point of an STM32 application. The execution of the firmware begins much earlier, with the calling of some important setup routines that create the execution environment for the firmware. However, from the *application point of view*, its start is inside the `main()` function. A [following chapter](#) will show in detail the bootstrap process of an STM32 microcontroller.

Filename: src/main.c

```

45 // Keep the LED on for 2/3 of a second.
46 #define BLINK_ON_TICKS (TIMER_FREQUENCY_HZ * 3 / 4)
47 #define BLINK_OFF_TICKS (TIMER_FREQUENCY_HZ - BLINK_ON_TICKS)
48
49 int main(int argc, char* argv[])
50 {
51     trace_puts("Hello ARM World!");
52     trace_printf("System clock: %u Hz\n", SystemCoreClock);
53
54     timer_start();
55
56     blink_led_init();
57
58     uint32_t seconds = 0;
59
60     // Infinite loop
61     while (1)
62     {
63         blink_led_on();
64         timer_sleep(seconds == 0 ? TIMER_FREQUENCY_HZ : BLINK_ON_TICKS);
65
66         blink_led_off();
67         timer_sleep(BLINK_OFF_TICKS);
68
69         ++seconds;
70
71         trace_printf("Second %u\n", seconds);
72     }
73 }

```

Instructions at line 51, 52 and 71 are related to debugging¹⁰ and we will see them in depth in Chapter 5. Function `timer_start()`; initializes the **SysTick timer** so that it fires an interrupt every 1ms. This is used to compute delays, and we will study how it works in Chapter 7. The function `blink_led_init()`; initializes the GPIO pin PA5 to be an output GPIO. Finally, the infinite loop turns ON and OFF the LED LD2, keeping it ON for 2/3 of second and OFF for 1/3 of second.

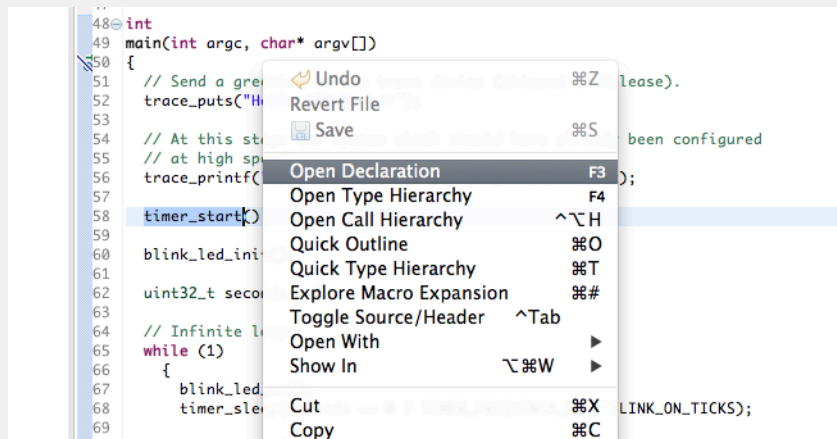


The only way to learn something in this field is to get your hands dirty writing code and making a lot of mistakes. So, if you are new to the STM32 platform, it is a good idea to start looking inside the code generated by the GNU MCU plugin, and trying to modify it. For example, a good exercise is to modify the code so that the LED starts blinking when the user button (the blue one) is pressed. A hint? The user button is connected to PC13 pin.

¹⁰For the sake of completeness, they are tracing functions that use *ARM semihosting*, a feature allowing to execute code in the host PC invoking it from the microcontroller - a sort of remote procedure call.

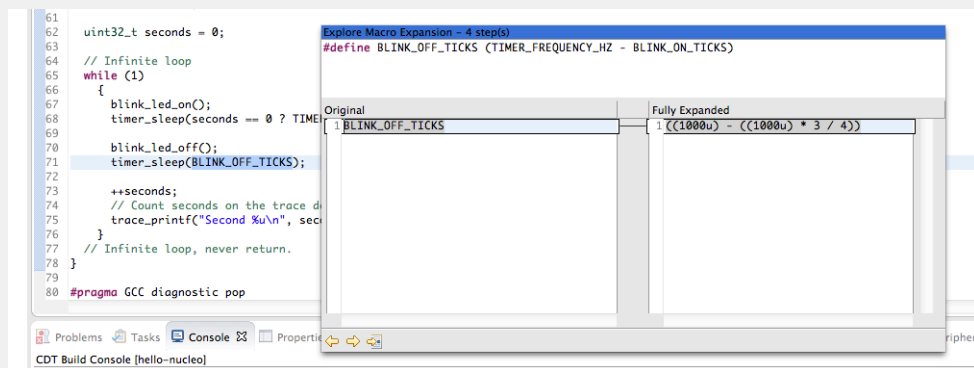
Eclipse intermezzo

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



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

Another interesting Eclipse feature is the ability to expand complex macros. For example, click with right mouse button on the `BLINK_OFF_TICKS` macro at line 71, and choose the entry **Explore macro expansion**. The following contextual window will appear.



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

4. STM32CubeMX Tool

STM32CubeMX¹ is the Swiss army knife of every STM32 developer, and it is a fundamental tool especially if you are new to the STM32 platform. It is a quite complex piece of software distributed freely by ST, and it is part of the [STCube initiative](#)², which aims to provide to developers with a complete set of tools and libraries to speed up the development process.

Although there is a well-established group of people that still develops embedded software in pure assembly code³, time is the most expensive thing during project development nowadays, and it is really important to receive as much help as possible for a quite complex hardware platform like the STM32.

In this chapter we will see how this tool from ST works, and how to build Eclipse projects from scratch using the code generated by it. This will make GNU MCU plugin a less critical component for project generation, allowing us to create better code and ready to be integrated with the STM32Cube HAL. However, this chapter is not a substitute for the [official ST documentation for CubeMX tool](#)⁴, a document made of more than 170 pages that explains in depth all its functionalities.

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

¹STM32CubeMX name will be simplified in *CubeMX* in the rest of the book.

²<http://bit.ly/1YKvl85>

³Probably, one day someone will explain them that, except for really rare and specific cases, a modern compiler can generate better assembly code from C than could be written directly in assembly by hand. However, we have to say that these habits are limited to ultra low-cost 8-bit MCUs like PIC12 and similar.

⁴<http://bit.ly/1O50wrp>

5. Introduction to Debugging

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

In this chapter we will start analyzing an important debugging tool: OpenOCD. It has become a sort of standard in the embedded development world, and thanks to the fact that many companies (including ST) are officially supporting its development, OpenOCD is facing a rapid growth. Every new release includes the support for tens of microcontrollers and development boards. Moreover, being portable among the three major Operating Systems (Windows, Linux and Mac OS), it allows us to use one unique and consistent tool to debug examples in this book.

This chapter also covers another important debugging mechanism: *ARM semi-hosting*. It is a way to communicate input/output requests from application code to a host computer running a debugger and it is extremely useful to execute functions that would be too complicated (or impossible due to the lack of some hardware features) to execute on the target microcontroller.

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

5.1 Getting Started With OpenOCD

The [Open On-Chip Debugger](#)¹ (OpenOCD) started as thesis work by Dominic Rath and now is actively developed and maintained by a large and growing community, with the official support from several silicon vendors.

OpenOCD aims to provide debugging, in-system programming and boundary-scan testing for embedded target devices. It does so with the assistance of a hardware debug adapter, which provides the right kind of electrical signaling to the target being debugged. In our case, this adapter is the integrated ST-LINK debugger provided by the Nucleo board². Every debug adapter uses a *transport protocol* that mediates between the hardware under debugging and the host software, that is OpenOCD.

¹<http://openocd.org>

²The Nucleo ST-LINK debugger is designed so that it can be used as standalone adapter to debug an external device (e.g., a board designed by you equipping an STM32 MCU).

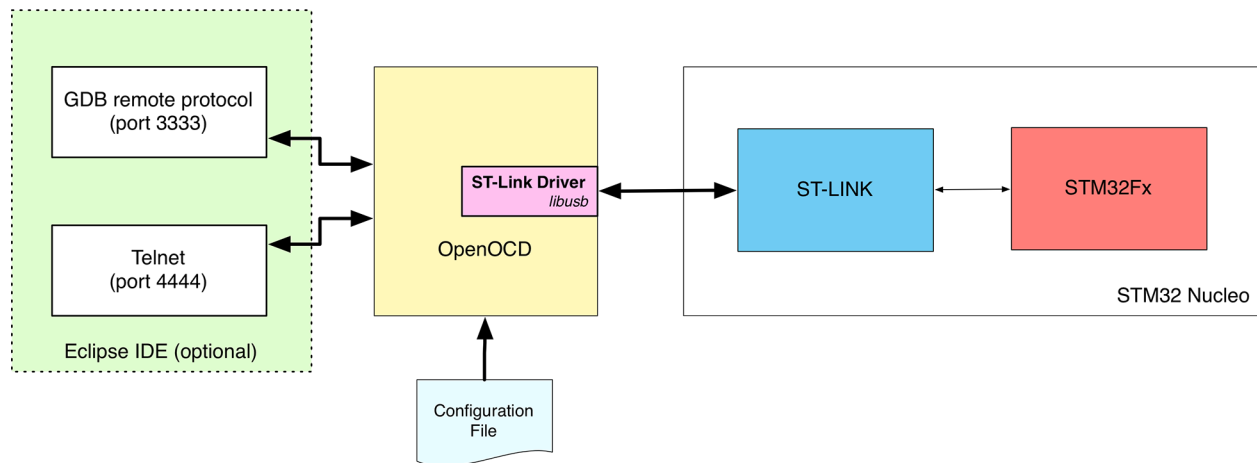


Figure 1: How OpenOCD interacts with a Nucleo board

OpenOCD is designed to be a generic tool able to work with tens of hardware debuggers, using several transport protocols. This requires a way to configure how to interface the specific debugger, and this is done through the use of script files. OpenOCD uses an extended definition of Jim-TCL, which in turn is a subset of the TCL programming language.

Figure 1 shows a typical debugging environment for the Nucleo board. Here we have the hardware part, composed by a Nucleo with its integrated ST-LINK interface, and OpenOCD interacting with the ST-LINK debugger using *libusb*, or any API-compatible library able to allow user-space applications to interface USB devices. OpenOCD also provides needed drivers to interact with the internal STM32 flash memory³ and the ST-LINK protocol. So it is instructed about the specific hardware under debugging (and the used debugger) through configuration files.

Once OpenOCD has established the connection with the board to debug, it provides two ways to communicate with the developer. The first one is through a local `telnet` connection on the port 4444. OpenOCD provides a convenient shell that is used to send commands to it and to receive information about the board under debugging. The second option is offered by using it as remote server for GDB. OpenOCD also implements the GDB remote protocol and it is used as “mediator” component between GDB and the hardware. This allows us to debug the firmware using GDB and, more important, using Eclipse as graphical debugging environment.

5.1.1 Launching OpenOCD

Before we configure Eclipse to use OpenOCD in our project, it is better to take a look at how OpenOCD works at a lower level. This will allow us to familiarize with it and, in case something does not work properly, it will allow to better investigate for issues related to the OpenOCD configuration.

The instructions to start OpenOCD are different between Windows and UNIX like systems. So, jump to the paragraph that fits your OS.

³One common misunderstanding about the STM32 platform is that all STM32 devices have a common and standardized way to access to their internal flash. This is not true, since every STM32 family has specific capabilities regarding their peripherals, including the internal flash. This requires OpenOCD to provide drivers to handle all STM32 devices.

5.1.1.1 Launching OpenOCD on Windows

Open the Windows Command Line tool⁴ and go inside the C:\STM32Toolchain\openocd\scripts folder and execute the following command:

```
$ cd C:\STM32Toolchain\openocd\scripts
$ ..\bin\openocd.exe -f board\<nucleo_conf_file.cfg>
```

where <nucleo_conf_file.cfg> must be substituted with the config file that fits your Nucleo board, according to Table 1⁵. For example, if your Nucleo is the Nucleo-F401RE, then the proper config file to pass to OpenOCD is st_nucleo_f4.cfg.

Table 1: Corresponding OpenOCD board file for a given Nucleo

Nucleo P/N	OpenOCD 0.10.0 board script file
NUCLEO-F446RE	st_nucleo_f4.cfg
NUCLEO-F411RE	st_nucleo_f4.cfg
NUCLEO-F410RB	st_nucleo_f4.cfg
NUCLEO-F401RE	st_nucleo_f4.cfg
NUCLEO-F334R8	stm32f334discovery.cfg
NUCLEO-F303RE	st_nucleo_f3.cfg
NUCLEO-F302R8	st_nucleo_f3.cfg
NUCLEO-F103RB	st_nucleo_f103rb.cfg
NUCLEO-F091RC	st_nucleo_f0.cfg
NUCLEO-F072RB	st_nucleo_f0.cfg
NUCLEO-F070RB	st_nucleo_f0.cfg
NUCLEO-F030R8	st_nucleo_f0.cfg
NUCLEO-L476RG	st_nucleo_l476rg.cfg
NUCLEO-L152RE	st_nucleo_l1.cfg
NUCLEO-L073RZ	st_nucleo_l073rz.cfg
NUCLEO-L053R8	stm32l0discovery.cfg

If everything went the right way, you should see messages similar to those appearing in Figure 2.

⁴It is strongly suggested to use a decent terminal emulator like ConEmu(<https://conemu.github.io/>) or similar.

⁵OpenOCD 0.10.0 still does not provide full support to all types of Nucleo boards, but the community is working hard on this and in the next main release the support will be completed. However, you can use alternative configuration files to work with your Nucleo at the time of writing this chapter.

```

cmd - .\bin\openocd.exe -f board\st_nucleo_f4.cfg
<1> cmd - .\bin\op...
Microsoft Windows [Version 6.1.7601]

cnoviello@CARMINENOVI6178 C:\Users\cnoviello
> cd C:\STM32Toolchain\openocd\scripts

cnoviello@CARMINENOVI6178 C:\STM32Toolchain\openocd\scripts
> ..\bin\openocd.exe -f board\st_nucleo_f4.cfg
Open On-Chip Debugger 0.9.0 (2015-05-19-12:06)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v20 API v2 SWIM v4 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.255790
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints

openocd.exe:1208

```

Figure 2: What appears on the command line prompt when OpenOCD starts correctly

At the same time, the LED LD1 on the Nucleo board should start blinking GREEN and RED alternatively. Now we can jump to the next paragraph.

5.1.1.2 Launching OpenOCD on Linux and MacOS X.

Linux and MacOS X users share the same instructions. Go inside the `~/STM32Toolchain/openocd/scripts` folder and execute the following command:

```

$ cd ~/STM32Toolchain/openocd/scripts
$ ../bin/openocd -f board/<nucleo_conf_file.cfg>

```

where `<nucleo_conf_file.cfg>` must be substituted with the config file that fits your Nucleo board, according to Table 1. For example, if your Nucleo is the Nucleo-F401RE, then the proper config file to pass to OpenOCD is `st_nucleo_f4.cfg`.

If everything went the right way, you should see messages similar to those appearing in Figure 2. At the same time, the LED LD1 on the Nucleo board should start blinking GREEN and RED alternatively. Now we can jump to the next paragraph.

Common OpenOCD Issues on the Windows Platform

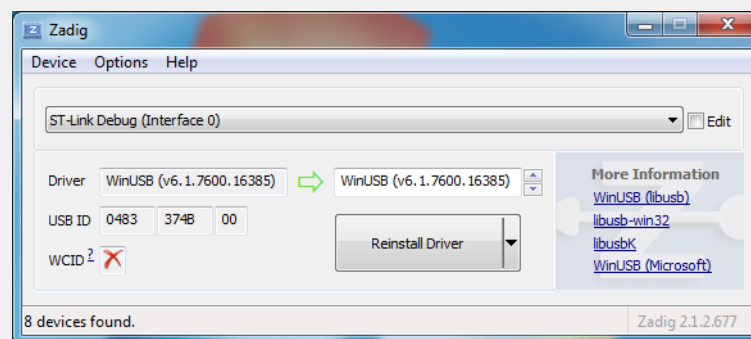
If you experienced issues trying to use OpenOCD on Windows, probably this paragraph could help you solving them.

It happens really often that Windows users cannot use OpenOCD the first time they install it. When OpenOCD is executed, an error message regarding libusb is thrown, as shown at lines 12-14 below.

```

1  Open On-Chip Debugger 0.10.0 (2015-05-19-12:09)
2  Licensed under GNU GPL v2
3  For bug reports, read http://openocd.org/doc/doxygen/bugs.html
4  Info : The selected transport took over low-level target control. The results might differ com\
5  pared to plain JTAG/SWD
6  adapter speed: 2000 kHz
7  adapter_nsrst_delay: 100
8  none separate
9  srst_only separate srst_nogate srst_open_drain connect_deassert_srst
10 Info : Unable to match requested speed 2000 kHz, using 1800 kHz
11 Info : Unable to match requested speed 2000 kHz, using 1800 kHz
12 Info : clock speed 1800 kHz
13 Error: libusb_open() failed with LIBUSB_ERROR_NOT_SUPPORTED
14 Error: libusb_open() failed with LIBUSB_ERROR_NOT_SUPPORTED
15 Error: libusb_open() failed with LIBUSB_ERROR_ACCESS
16 Error: open failed
17 in procedure 'init'
18 in procedure 'ocd_bouncer'
```

This happens because a wrong version of libusb is used to interface the ST-LINK Debug Interface. To solve this, download the [Zadig utility](http://zadig.akeo.ie/)^a for your Windows version. Launch the Zadig tool ensuring that your Nucleo board is plugged to the USB port, and go to the **Option->List All Devices** menu. After a while the ST-LINK Debug (Interface 0) entry should appear inside the device list combo box. If the installed driver is not the WinUSB one, then select it and click on **Reinstall Driver** button, as shown below.



^a<http://zadig.akeo.ie/>

5.1.2 Connecting to the OpenOCD Telnet Console

Once OpenOCD starts, it acts as a daemon program⁶ waiting for external connections. OpenOCD offers two ways to interact with it. One of these is mode is through GDB (the GNU Debugger), as we will see later. The other one is through a telnet⁷ connection to the localhost port 4444⁸. Let us start a connection.

```
$ telnet localhost 4444
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
>
```

To access to the list of supported commands, we can type `help`. The list is quite huge, and its content is outside of the scope of this book (the official OpenOCD document is a good place to start understanding what those commands are used for). Here, we will simply see how to flash the firmware.

Before we can upload a firmware to the target MCU of our Nucleo, we have to halt the MCU. This is done issuing a `reset init` command:

```
Open On-Chip Debugger
> reset init
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080002a8 msp: 0x20018000, semihosting
```

OpenOCD says to us that the micro is now halted and we can proceed to upload the firmware using the `flash write_image` command:

⁶*Daemon* is the way in UNIX to name those programs that works like a service. For example, a HTTP server or an FTP server is called a *daemon* in UNIX. In the Windows world these kind of programs are called *services*.

⁷Starting from Windows 7, telnet is an optional component to install. However, it is strongly suggested to use a more evolute telnet client like putty (<http://bit.ly/1jsQjnt>).

⁸The default port can be changed issuing a `telnet_port` command inside the board configuration file. This can be useful if we are debugging two different boards using two OpenOCD sessions, as we will see next.

```
> flash write_image erase <path to the .elf file>
auto erase enabled
Padding image section 0 with 3 bytes
target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x20000042 msp: 0xffffffff, semihosting
wrote 16384 bytes from file <path to the .elf file> in 0.775872s (20.622 KiB/s) >
```

where <path to the .elf file> is the full path to the binary file (it is usually stored inside the Debug subdirectory in the Eclipse project folder).

To start running our firmware we can simply type the `reset` command to the OpenOCD command line.

There are other few OpenOCD commands that may be useful during firmware debugging, especially when dealing with hardware faults. The `reg` commands shows the current status of all Cortex-M core registries when the target MCU is halted:

```
> reset halt
...
> reg
===== arm v7m registers
(0) r0 (/32): 0x00000000
(1) r1 (/32): 0x00000000
...
```

Another group of useful commands are `md[whb]` to read a word, half-word and byte respectively. For example, the command:

```
> mdw 0x80000000
0x08000000: 12345678
```

reads 32 bit (a word) from the address `0x8000 000`. The commands `mw[whb]` are the equivalent commands to store data in a given memory location.

Now you can close the OpenOCD daemon sending the `shutdown` command to the telnet console. This will also close the telnet session.

5.1.3 Configuring Eclipse

Now that we are familiar with the way OpenOCD works, we can configure Eclipse to debug our application from the IDE. This will dramatically simplify the debugging process, allowing us to easily set breakpoints in our code, to inspect the content of variables and to do step-by-step execution.

Eclipse is a generic and high configurable IDE, and it allows to create configurations that easily integrate external tools like OpenOCD in its development life-cycle. The process we are going to

accomplish here is essentially to create a *debug configuration*. There are at least three ways to integrate OpenOCD in Eclipse, but only one is probably the more convenient way when we deal with the ST-LINK debugger.

We will configure OpenOCD as *external debugging tool* that we execute only once and leave as daemon process, like we have done in the previous paragraph executing it from command line prompt. The next step is to create a GDB debug configuration that instructs GDB to connect to OpenOCD port 3333 and use it as GDB server.

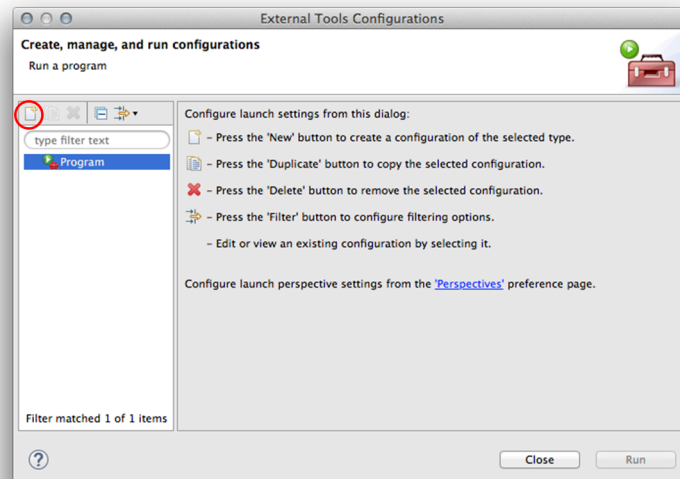


Figure 3: The *External Tools Configurations* dialog

First, ensure that you have a project opened in Eclipse. Then, go to **Run->External Tools->External Tools Configurations...** menu. The *External Tools Configurations* dialog appears. Highlight the **Program** entry in the list view on the left and click on the **New** icon (the one circled in red in Figure 3). Now, fill the following fields in this way:

- **Name:** write the name you like for this configuration; it is suggested to use *OpenOCD FX*, where FX is the STM32 family of your Nucleo board (F0, F1, and so on).
- **Location:** choose the location of the OpenOCD executable (C:\STM32Toolchain\openocd\bin\openocd.exe for Windows users, ~/STM32Toolchain/openocd/bin/openocd for Linux and Mac OS users).
- **Working directory:** choose the location of the OpenOCD scripts directory (C:\STM32Toolchain\openocd\scripts for Windows users, ~/STM32Toolchain/openocd/scripts for Linux and Mac OS users).
- **Arguments:** write the command line arguments for OpenOCD, that is “-f board\<nucleo_conf_file.cfg>” for Windows users and “-f board/<nucleo_conf_file.cfg>” for Linux and Mac OS users. <nucleo_conf_file.cfg> must be substituted with the config file that fits your Nucleo board, according to **Table 1**.

When completed, click on the **Apply** button and then on the **Close** one. To avoid mistakes that could

cause confusion, **Figure 4** shows how to fill the fields on Windows and **Figure 5** on a UNIX-like system (arrange the home directory accordingly).

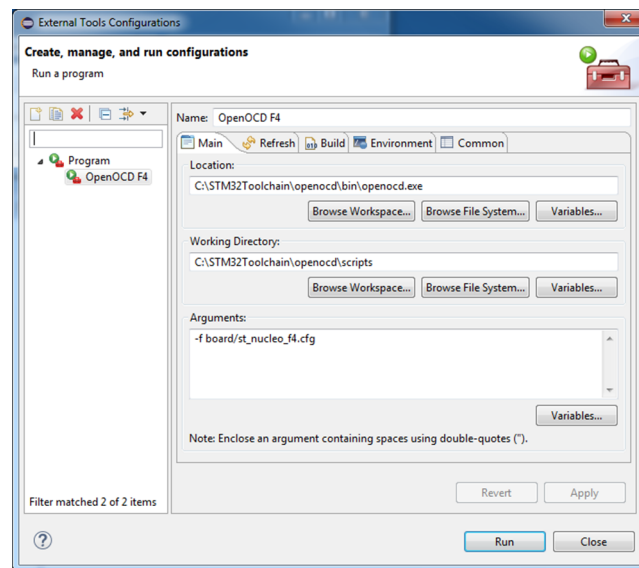


Figure 4: How to fill the *External Tools Configurations* fields on Windows

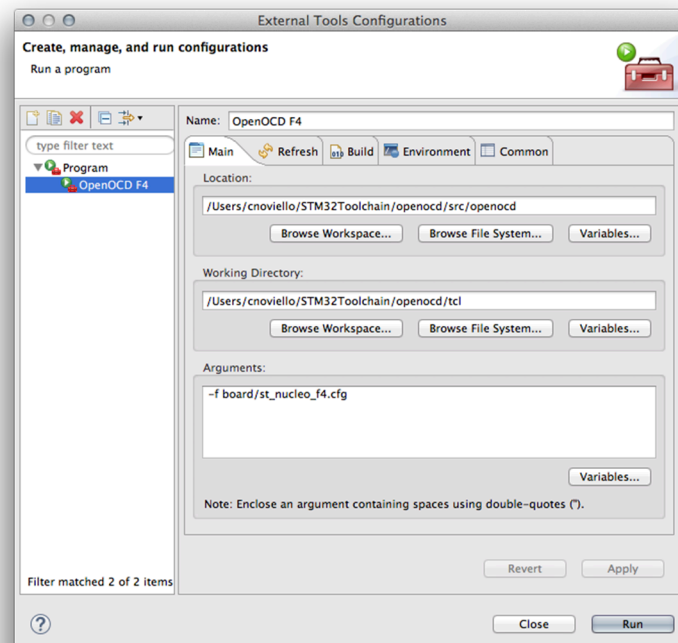


Figure 5: How to fill the *External Tools Configurations* fields on UNIX systems

To launch OpenOCD now you can simply go to **Run->External Tools** menu and choose the configuration you have created. If everything went the right way, you should see the classical

OpenOCD messages inside the Eclipse Console, as shown in **Figure 6**. At the same time, the LED LD1 on the Nucleo board should start blinking GREEN and RED alternatively.

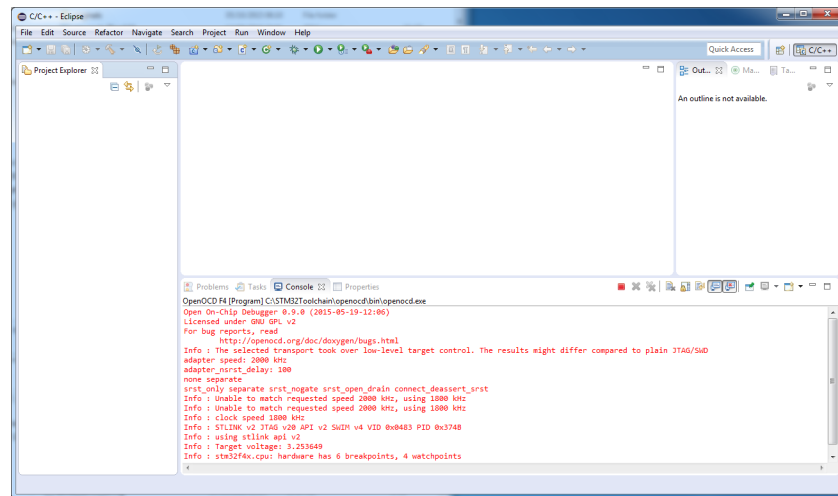


Figure 6: The OpenOCD output in the Eclipse console

Now we are ready to create a *Debug Configuration* to use GDB in conjunction with OpenOCD. This operation must be repeated every time we create a new project.

Go to **Run->Debug Configurations...** menu. Highlight the **GDB OpenOCD Debugging** entry in the list view on the left and click on the **New** icon (the one circled in red in **Figure 7**).

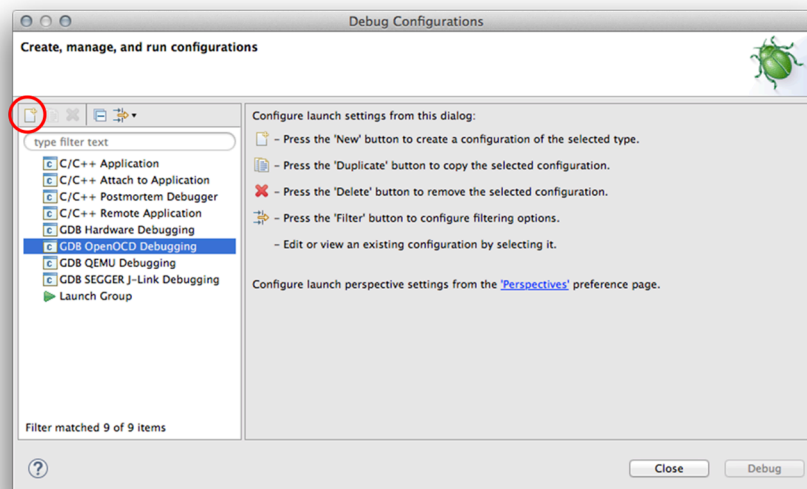


Figure 7: The *Debug Configuration* dialog

Eclipse fills automatically all the needed fields in the **Main** tab. However, if you are using a project with several *build configurations*, you need to click on the **Search Project** button and choose the ELF file for the active build configuration.



Unfortunately, sometimes Eclipse is not able to automatically locate the binary file. This is probably a bug, or at least a weird behaviour. It may happens really often especially when there are more than one project opened. To address this issue, click on the **Browse** button and find the binary file in the project folder (usually you find it inside the <project-dir>/Debug sub-directory).

Alternatively, another solution consists in closing the **Debug Configuration** dialog, then refreshing the whole project tree (by clicking with the right mouse button on the project root and selecting the **Refresh** entry). You will notice that Eclipse updates the content of **Binaries** subfolder. Now you can re-open again the **Debug Configuration** dialog and complete the configuration by clicking on the **Search Project** button.

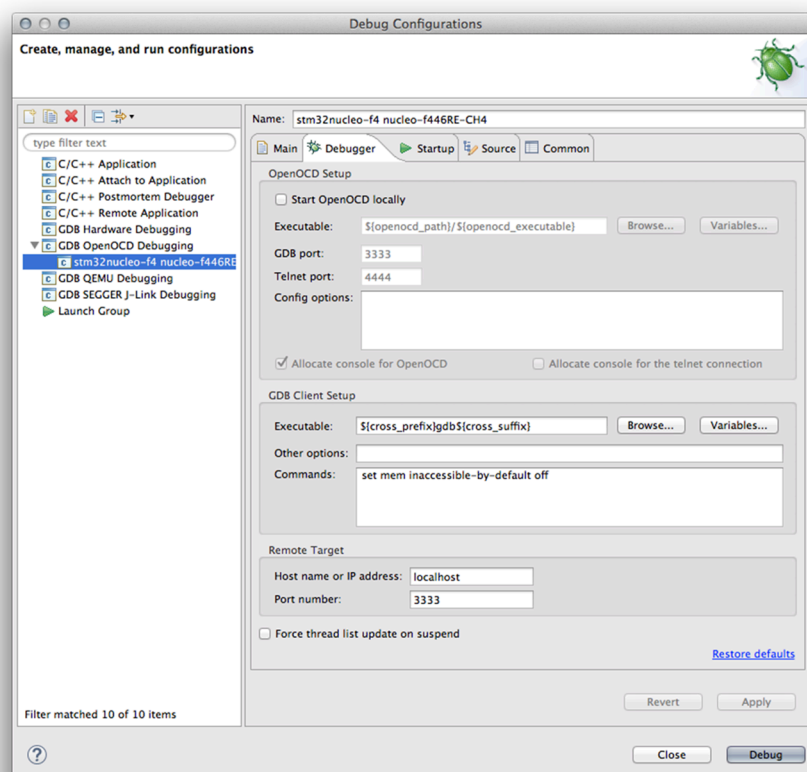


Figure 8: The *Debug Configuration* dialog - *Debugger* section

Next, go in the **Debugger** tab and uncheck the entry **Start OpenOCD locally**, since we have created the specific OpenOCD external tool configuration. Ensure that all other fields are equal to the ones shown in **Figure 8**.

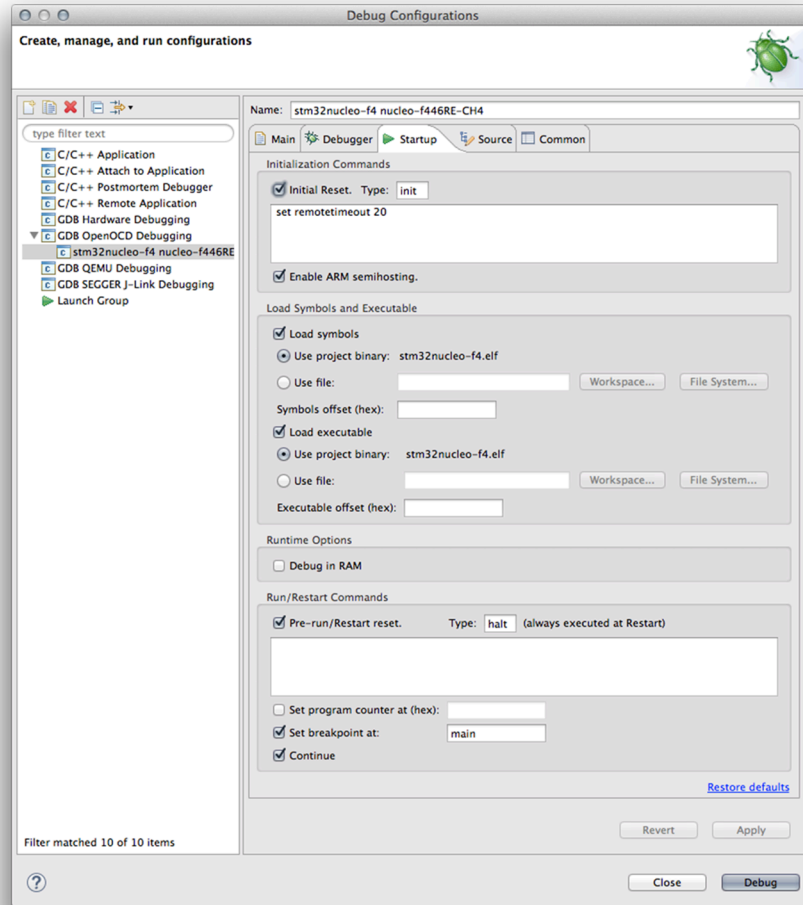


Figure 9: The *Debug Configuration* dialog - *Startup* section

Now, go in the **Startup** section and leave all options as default but do not forget to add the OpenOCD command `set remotetimeout 20` as shown in **Figure 9**.



What Do All Those Fields Mean?

If you take a pause and look at the fields in this section, you should recognize most of commands we have typed when using the OpenOCD telnet session to load the firmware on our Nucleo board.

The **Initial reset** checkbox is the equivalent of the `reset init` to reset the MCU. The **Load symbols** and the **Load executables** are the equivalent flash `write_image` command⁹. Finally, the `set remotetimeout 20` increases the keep alive time between GDB and OpenOCD, which ensures that the OpenOCD backend is still alive. 20(ms) is a proven value to use.

⁹Experienced STM32 users would dispute this sentence. They would be right: here we are issuing different GDB `load` commands, and not the OpenOCD flash `write_image` command. However, for the sake of simplicity, consider that sentence true. A later chapter will explain this better.

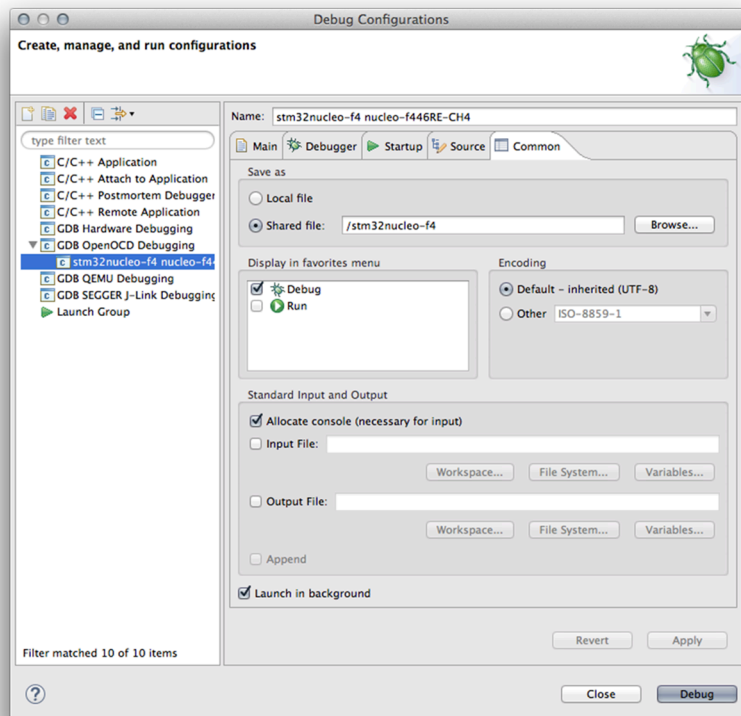


Figure 10: The *Debug Configuration* dialog - *Common* section

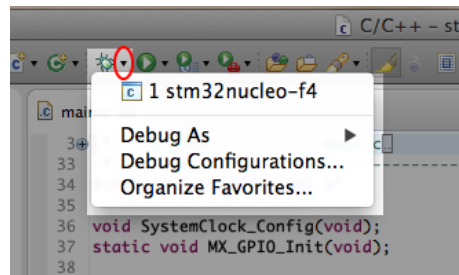
Finally, go in the **Common** section and check the option **Shared file**¹⁰ in **Save as** frame box and check the entry **Debug** in **Display in favorites menu** frame box, as shown in **Figure 10**.

Click on the **Apply** button and then on the **Close** one. Now we are ready to start debugging.

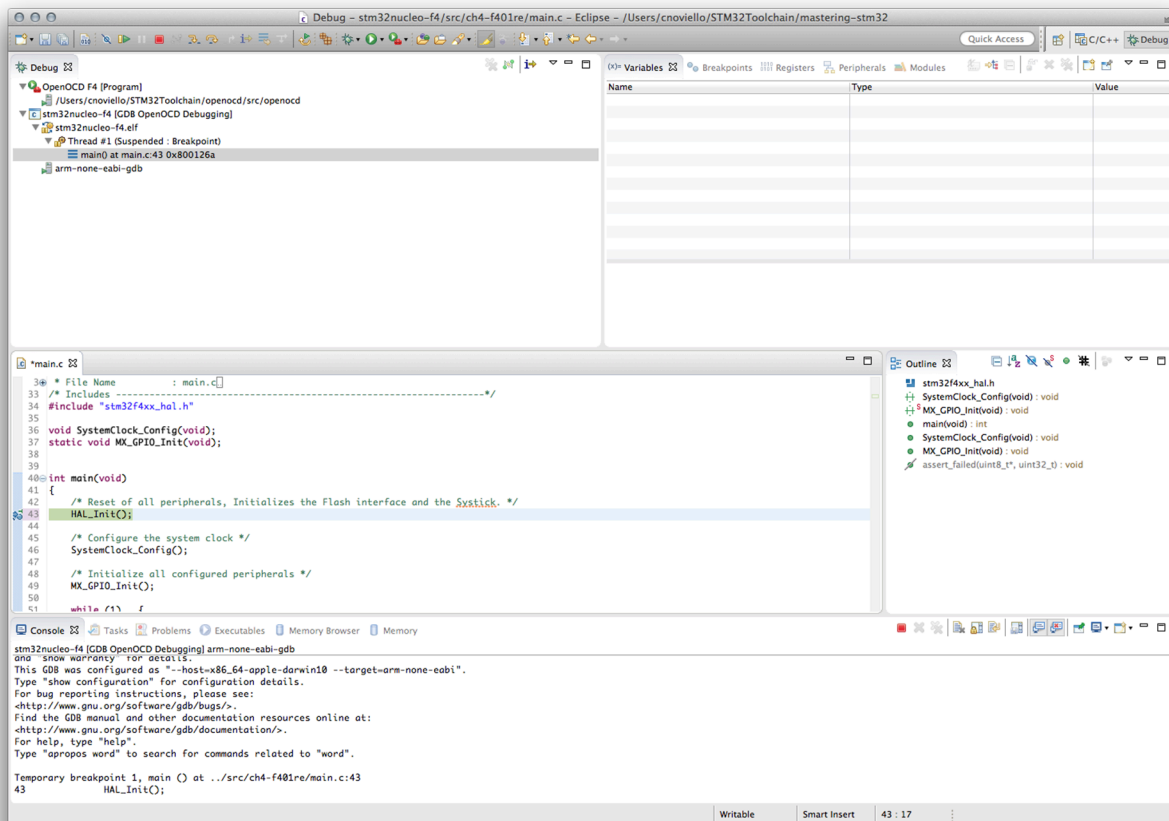
5.1.4 Debugging in Eclipse

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

¹⁰This setting saves the debug configuration at project level, and not as global Eclipse setting. This will allow us to share the configuration with other people if we work in team.

Figure 11: The *Debug* icon to start debugging in Eclipse

To start a new debug session using the debug configuration made earlier, you can click on the arrow near the **Debug** icon on the Eclipse toolbar and choose the debug configuration, as shown in **Figure 11**. Eclipse will ask you if you want to switch to the *Debug Perspective*. Click on the **Yes** button (it is strongly suggested to flag the **Remember my decision** checkbox). Eclipse switches to the *Debug Perspective*, as shown in **Figure 12**.

Figure 12: The *Debug Perspective*

Let us see what each view is used for. The top-left view is called **Debug** and it shows all the running debug activities. This is a tree-view, and the first entry represents the OpenOCD process launched using the external debug configuration. We can eventually stop the execution of OpenOCD

highlighting the executable program and clicking on the **Terminate** icon on the Eclipse toolbar, as shown in Figure 13.

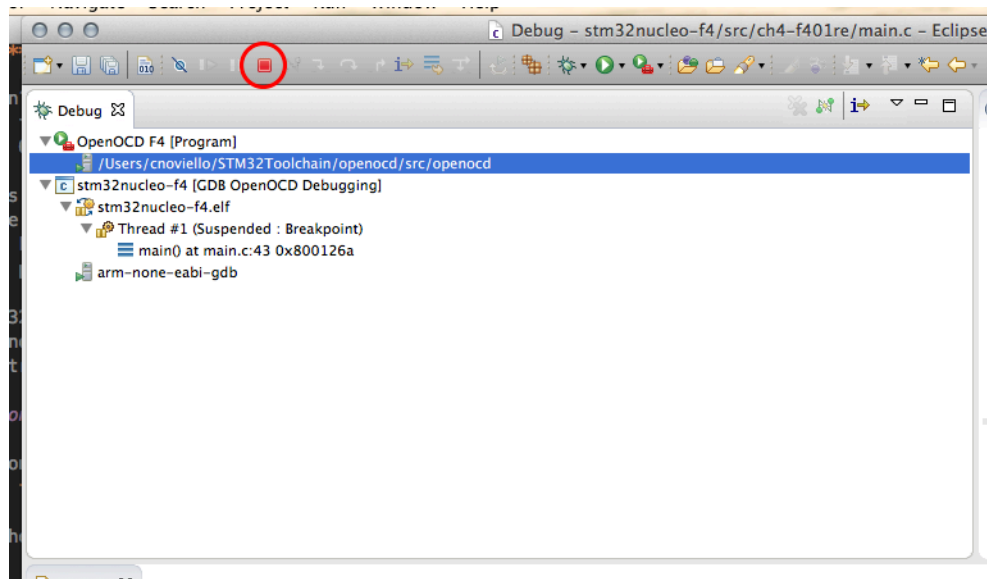


Figure 13: How to terminate the execution of a debug activity

The second activity showed in the **Debug** view represents the GDB process. This activity is really useful, because when the program is halted the complete call stack is shown here and it offers a quick way to navigate inside the call stack.

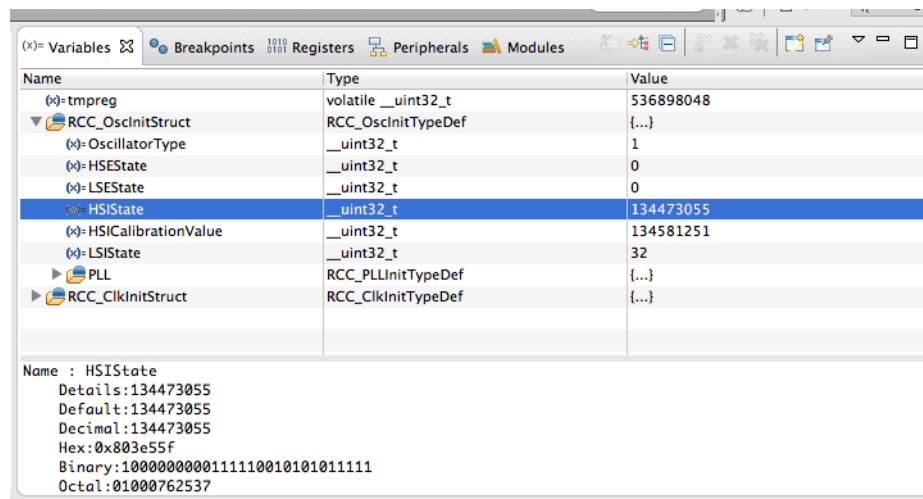


Figure 14: The variables inspection pane in the *debug perspective*

The top-right view contains several sub-panes. The **Variables** one offers the ability to inspect the content of variables defined in the current stack frame (that is, the selected procedure in the call stack). Clicking on an inspected variable with the right button of mouse, we can further customize the way the variable is shown. For example, we can change its numeric representation, from decimal (the default one) to hexadecimal or binary form. We can also cast it to a different datatype (this is

really useful when we are dealing with raw amount of data that we know to be of a given type - for example, a bunch of bytes coming from a stream file). We can also go to the memory address where the variable is stored clicking on the **View Memory...** entry in the contextual menu.

The **Breakpoint** pane lists all the used breakpoints in the application. A *breakpoint* is a hardware primitive that allows to stop the execution of the firmware when the *Program Counter*(PC) reaches a given instruction. When this happens, the debugger is warned and Eclipse will show the context of the halted instruction. Every Cortex-M base MCU has a limited number of hardware breakpoints. **Table 2** summarizes the maximum breakpoints and watchpoints¹¹ for a given Cortex-M family.

Table 2: Available breakpoints/watchpoints in Cortex-M cores

Cortex-M	Breakpoints	Watchpoints
M0/0+	4	2
M3/4/7	6	4

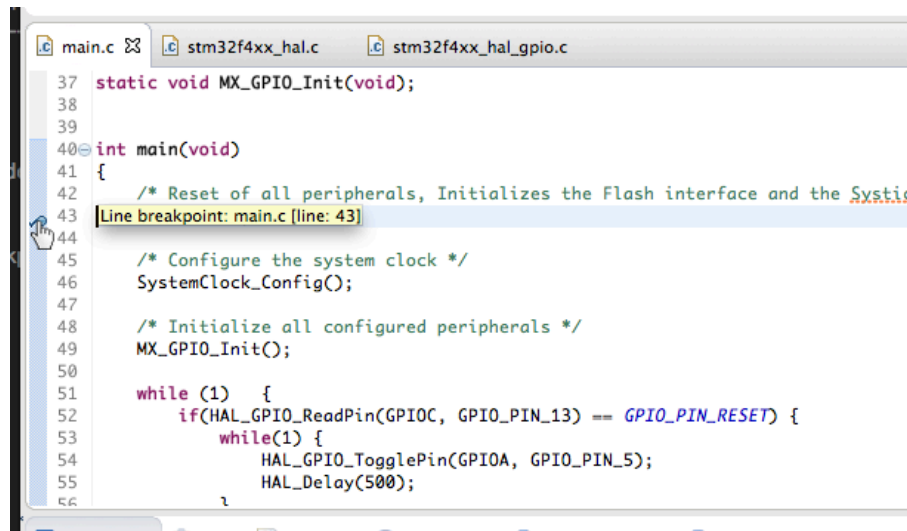


Figure 15: How to add a breakpoint at a given line number

Eclipse allows to easily setup breakpoints inside the code from the editor view in the center of **Debug perspective**. To place a breakpoint, simply double-click on the blue stripe on the left of the editor, near to the instruction where we want to halt the MCU execution. A blue bullet will appear, as shown in **Figure 15**.

When the program counter reaches the first assembly instruction constituting to that line of code, the execution is halted and Eclipse shows the corresponding line of code as shown in **Figure 12**. Once we have inspected the code, we have several options to resume the execution.

¹¹A watchpoint, indeed, is a more advanced debugging primitive that allows to define conditional breakpoints over data and peripheral registers, that is the MCU halts its execution only if a variable satisfies an expression (e.g. `var == 10`). We will analyze watchpoints in [Chapter 24](#).

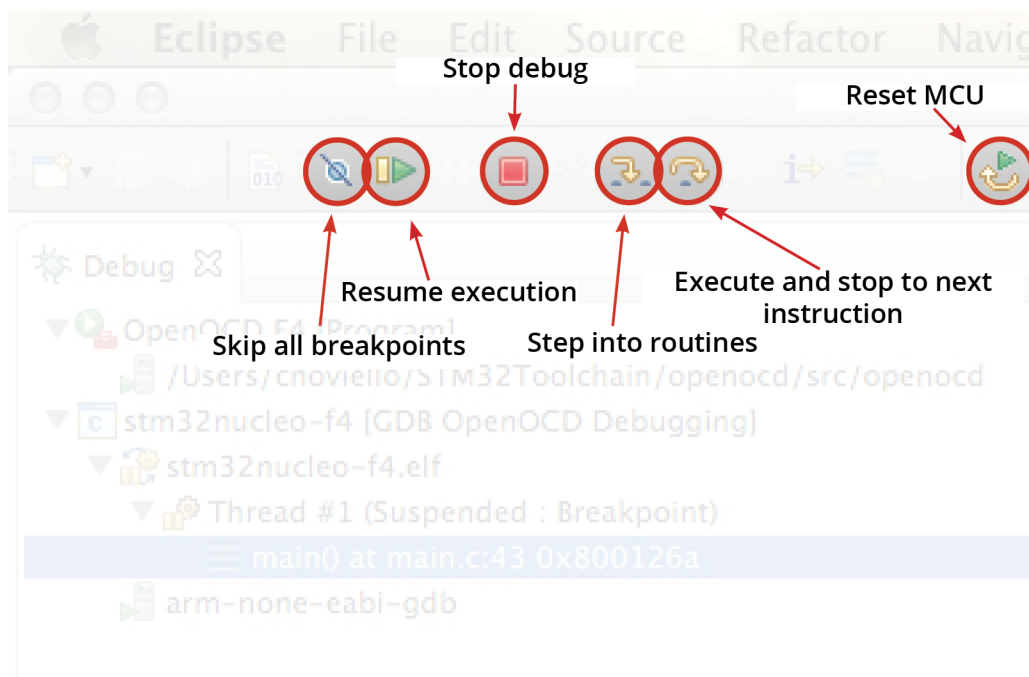


Figure 16: The Eclipse debug toolbar

Figure 16 shows the Eclipse debug toolbar. The highlighted icons allow to control the debug process. Let us see each of them in depth.

- **Skip all breakpoints:** this toggle icon allows to temporarily ignore all the breakpoint used. This allows to run the firmware without interruption. We can resume breakpoints by deactivating the icon.
- **Resume execution:** this icon restarts the execution of the firmware from the current PC. The adjacent icon, the pause, will stop the execution on request.
- **Stop debug:** this icon causes the end of the debug session. GDB is terminated and the target board is halted.
- **Step into routine:** 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.
- **Step over:** the next icon of the debug toolbar has 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.
- **Reset MCU:** this icon is used to do a soft reset of MCU, without stopping the debug and relaunch it again.

Finally, another interesting pane of that view is the **Registers** one. It displays the content of all Cortex-M registers and it is the equivalent of the `reg` OpenOCD command we have seen before.

It can be really useful to understand the current state of the Cortex-M core. In [Chapter 24](#) about debugging we will see how to deal with Cortex-M exceptions and we will learn how to interpret the content of some important Cortex-M registers.

5.2 ARM Semihosting

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

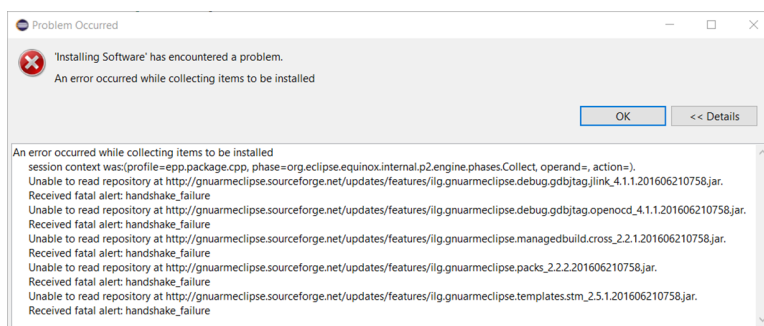
II Appendix

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): <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`

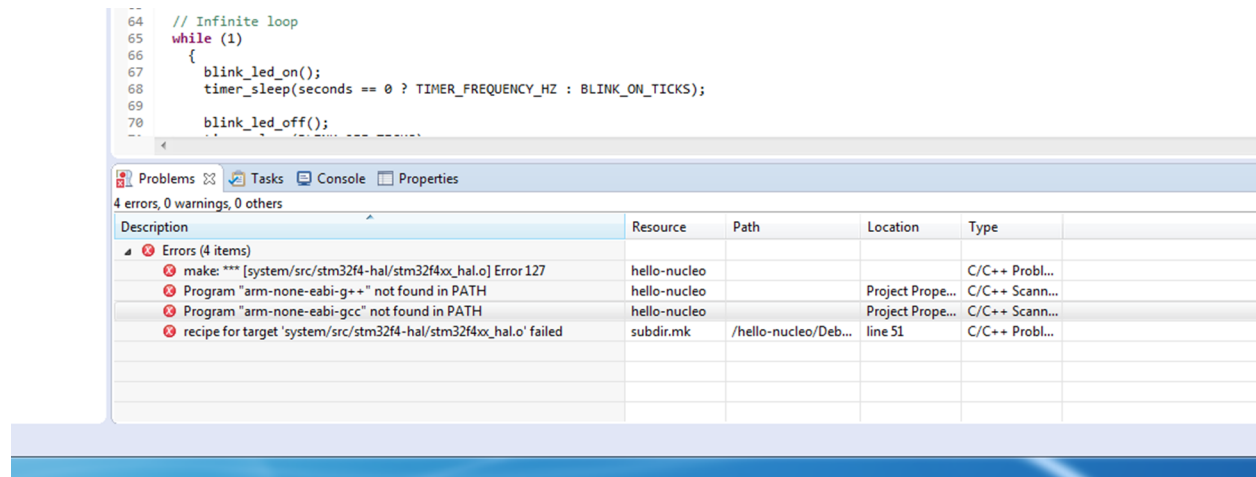
Restart Eclipse. You should be able to install GNU MCU Eclipse plug-ins now.

Eclipse related issue

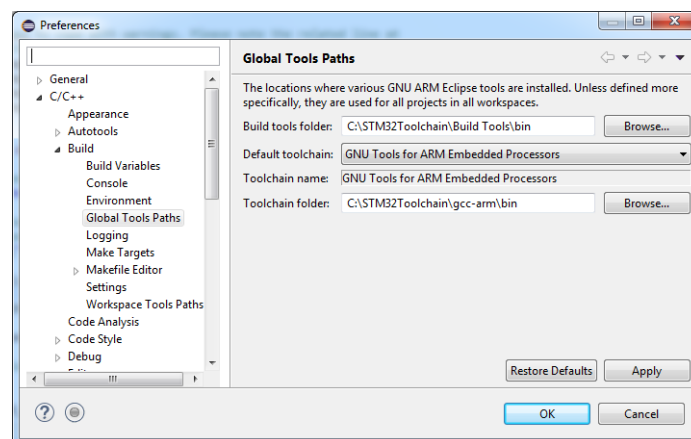
This section contains a list of frequently issues related with the Eclipse IDE.

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](https://developer.mbed.org/platforms/?tvend=10)¹².

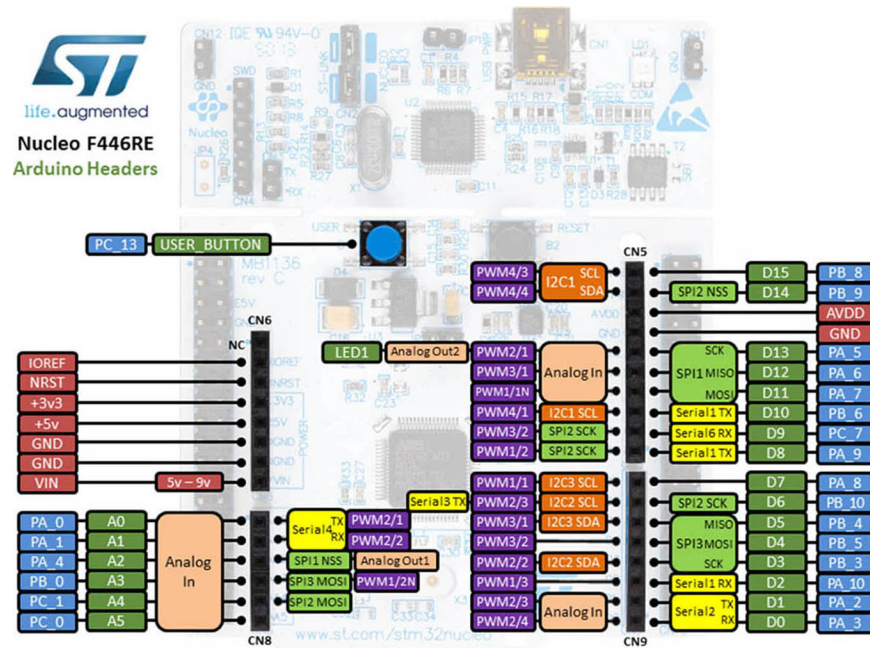
Nucleo Release

[Nucleo-F446RE](#)
[Nucleo-F411RE](#)
[Nucleo-F410RB](#)
[Nucleo-F401RE](#)
[Nucleo-F334R8](#)
[Nucleo-F303RE](#)
[Nucleo-F302R8](#)
[Nucleo-F103RB](#)
[Nucleo-F091RC](#)
[Nucleo-F072RB](#)
[Nucleo-F070RB](#)
[Nucleo-F030R8](#)
[Nucleo-L476RG](#)
[Nucleo-L152RE](#)
[Nucleo-L073RZ](#)
[Nucleo-L053R8](#)

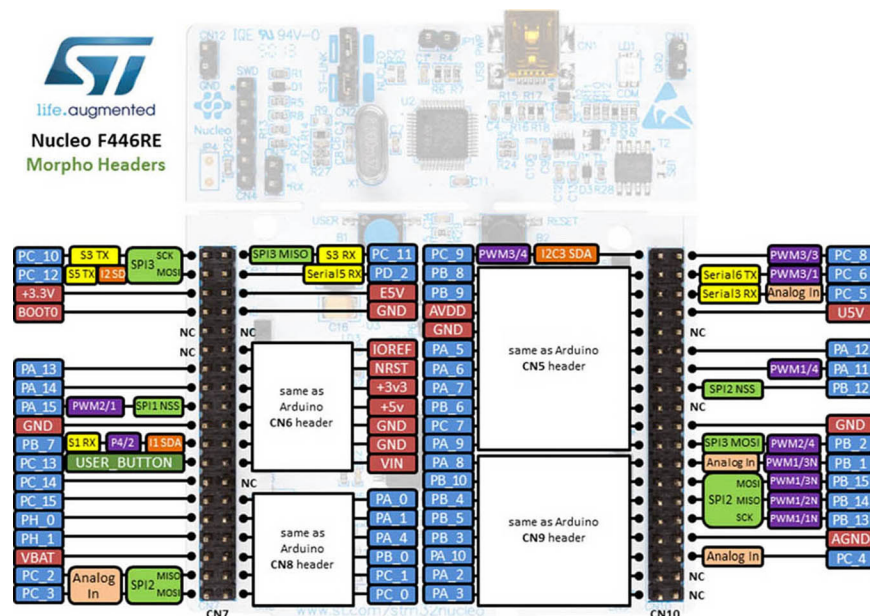
¹²<https://developer.mbed.org/platforms/?tvend=10>

Nucleo-F446RE

Arduino compatible headers

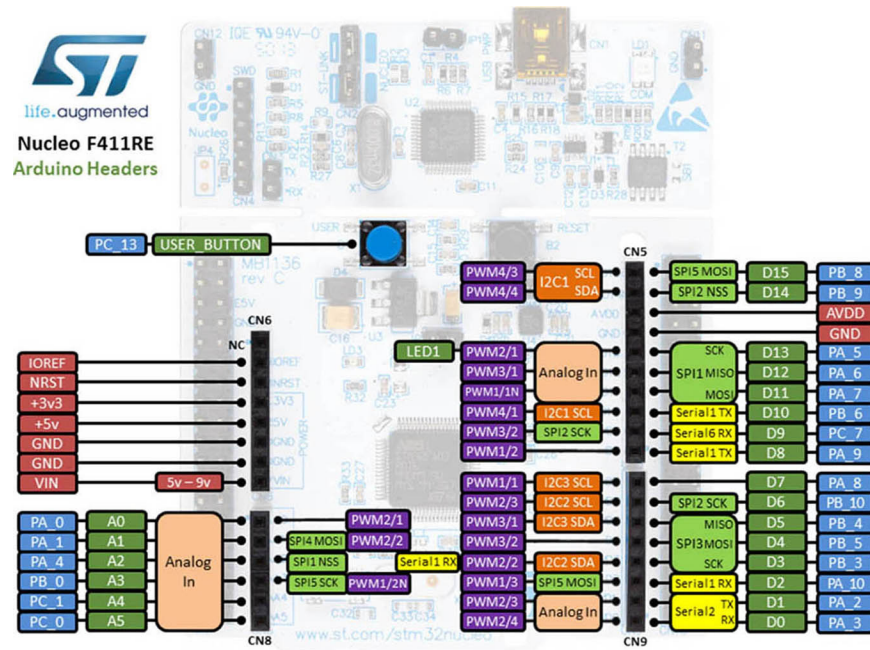


Morpho headers

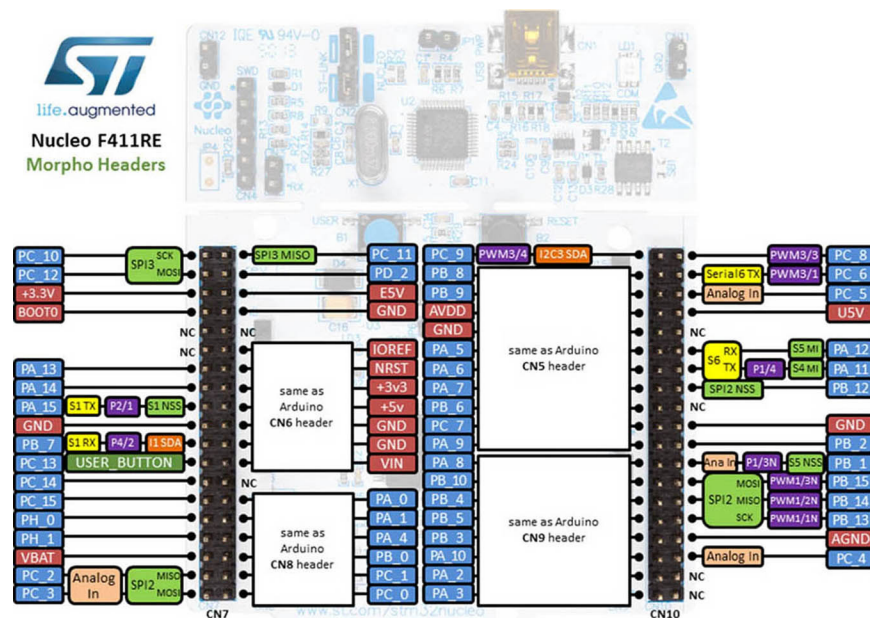


Nucleo-F411RE

Arduino compatible headers

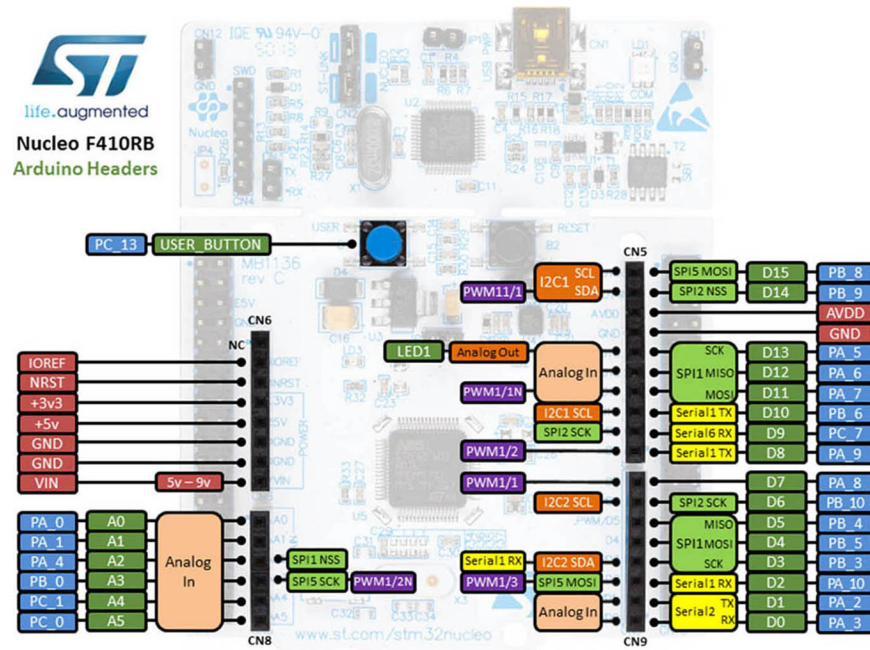


Morpho headers

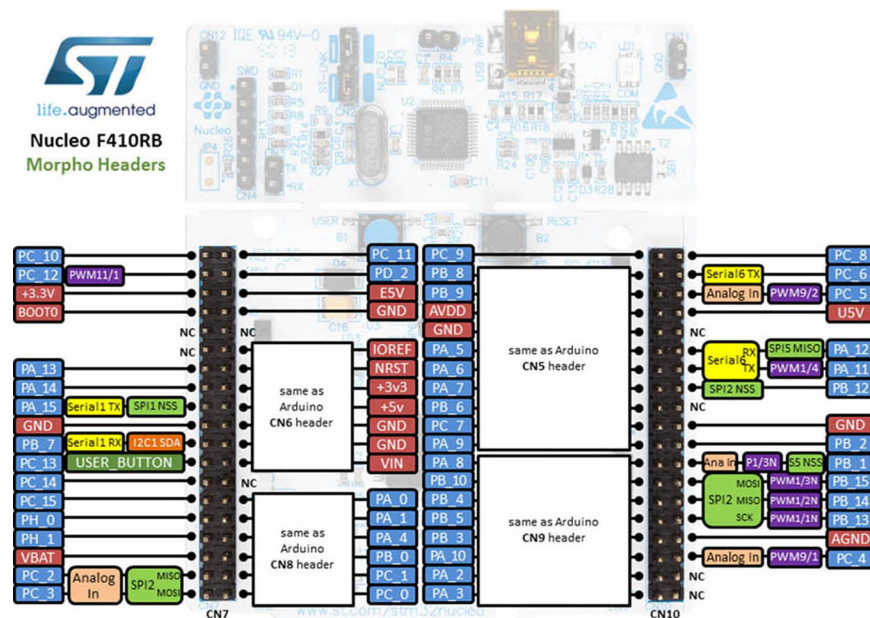


Nucleo-F410RB

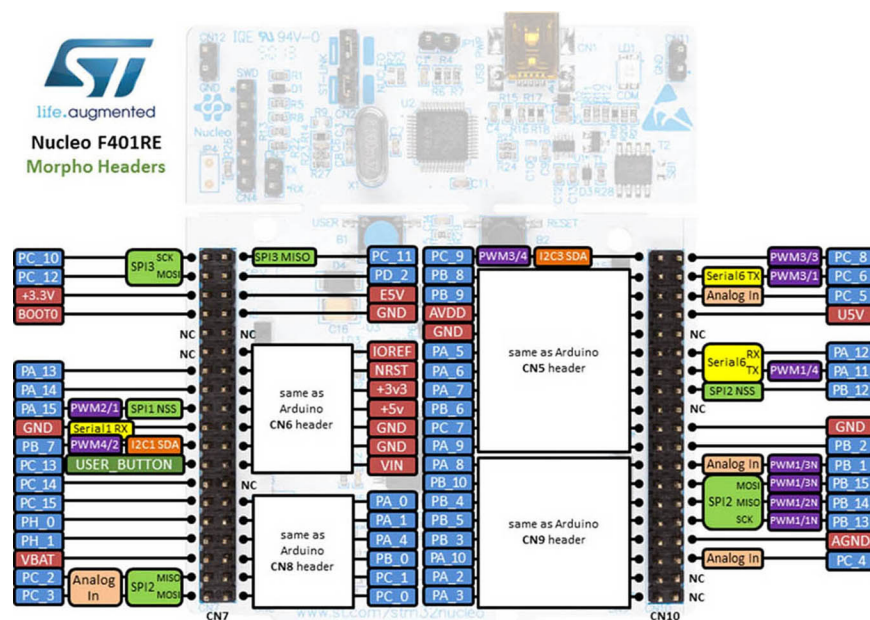
Arduino compatible headers



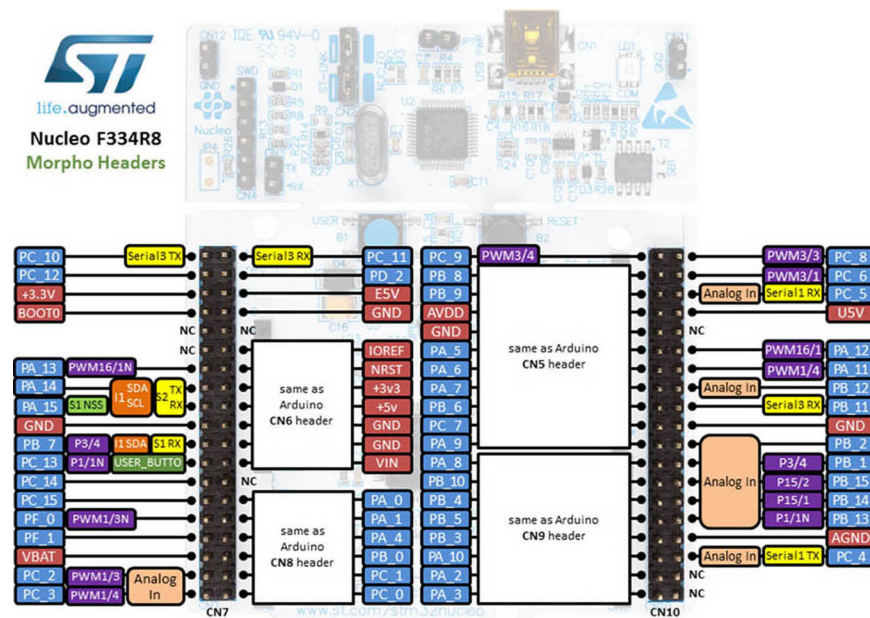
Morpho headers



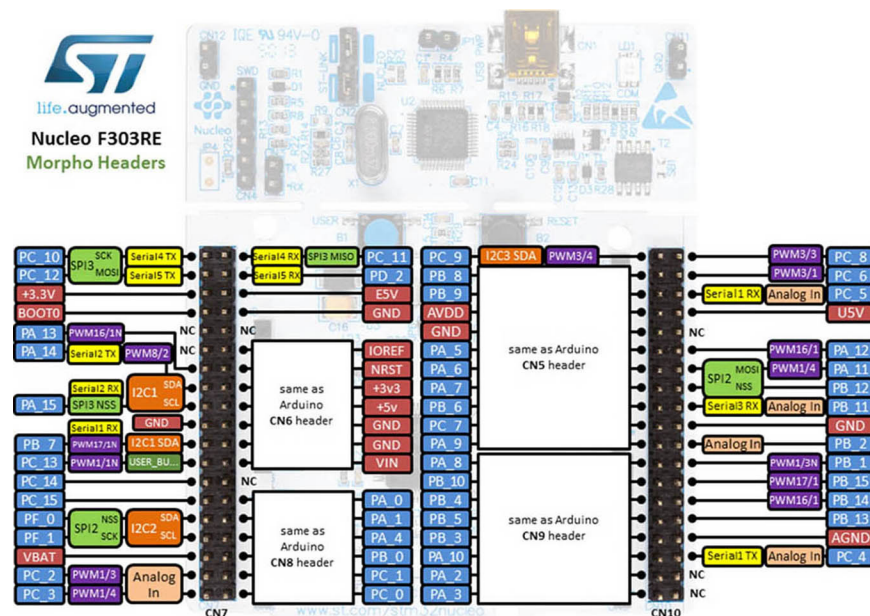
Arduino compatible headers



Arduino compatible headers

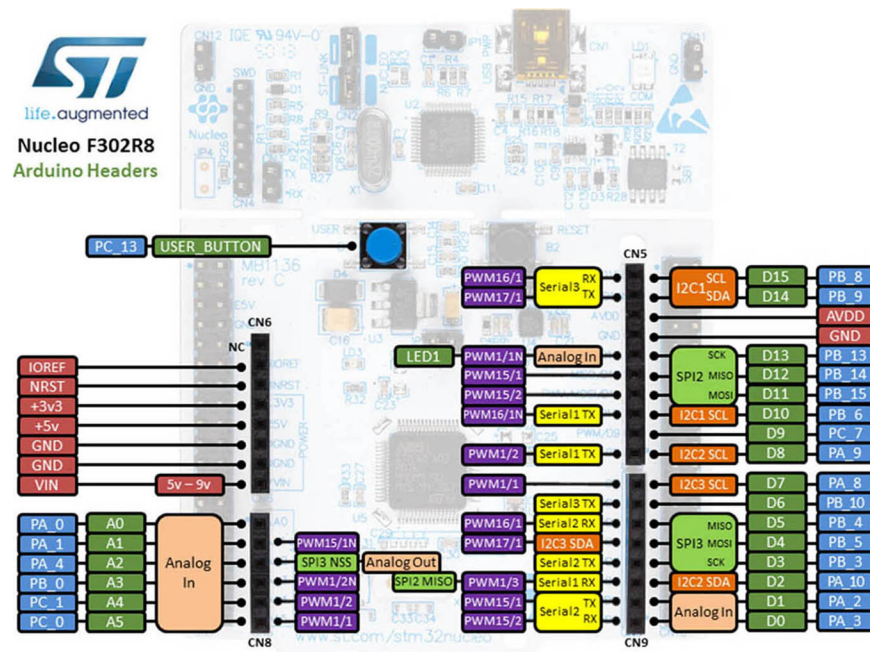


Arduino compatible headers

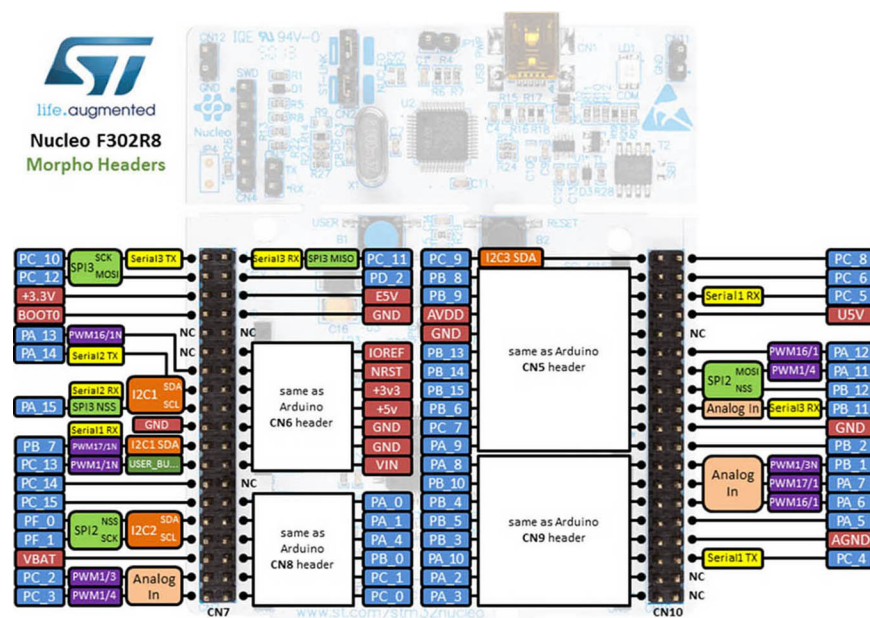


Nucleo-F302R8

Arduino compatible headers

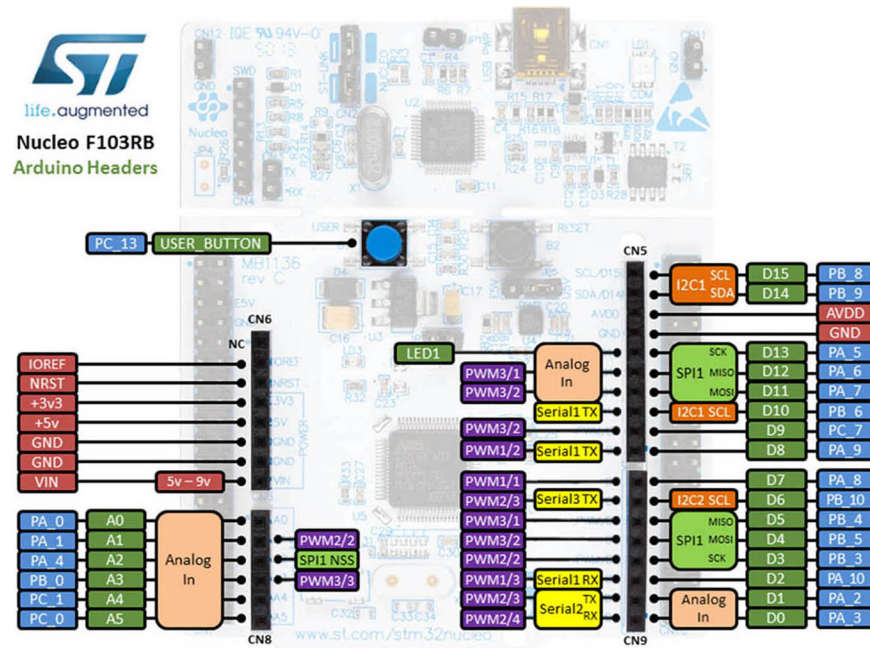


Morpho headers

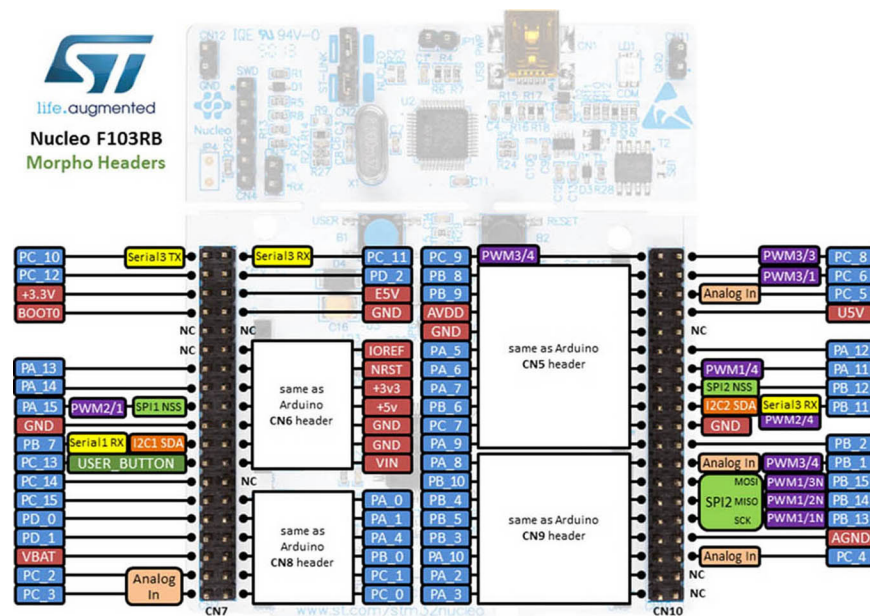


Nucleo-F103RB

Arduino compatible headers

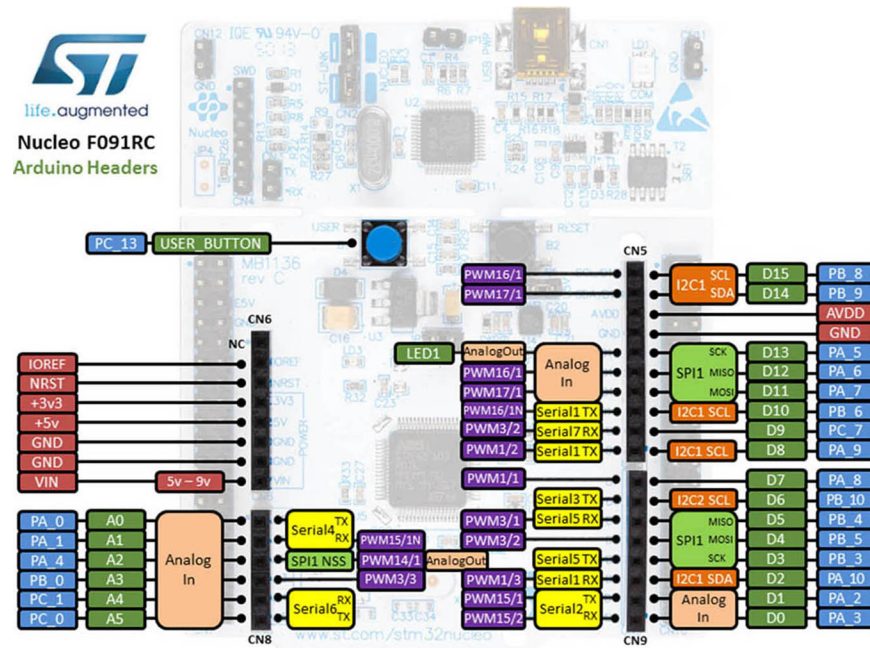


Morpho headers

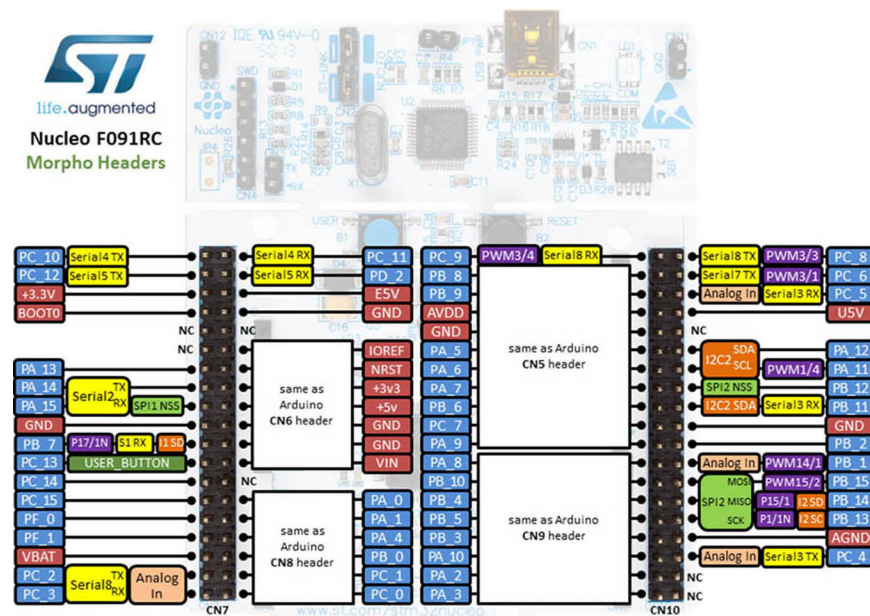


Nucleo-F091RC

Arduino compatible headers

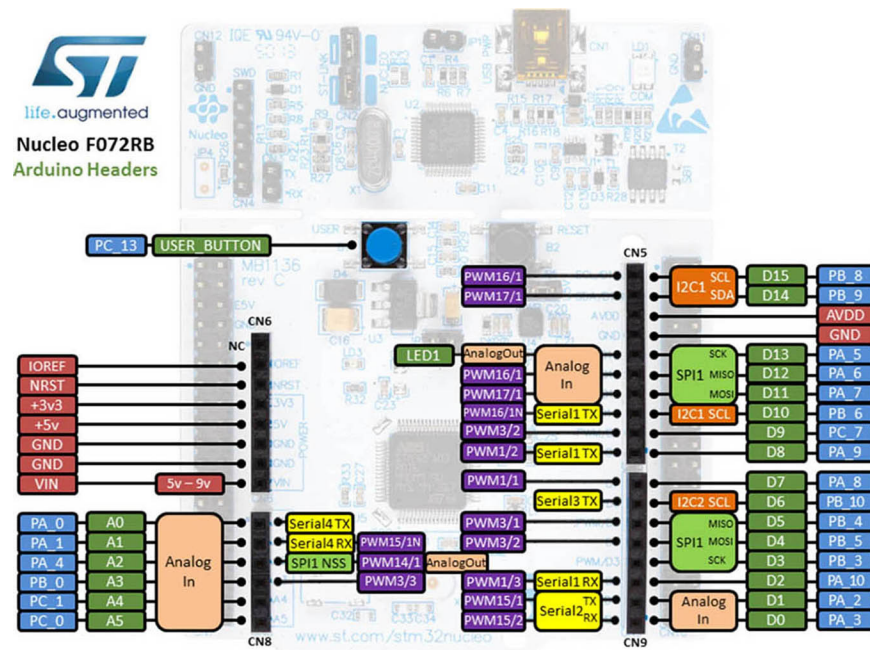


Morpho headers

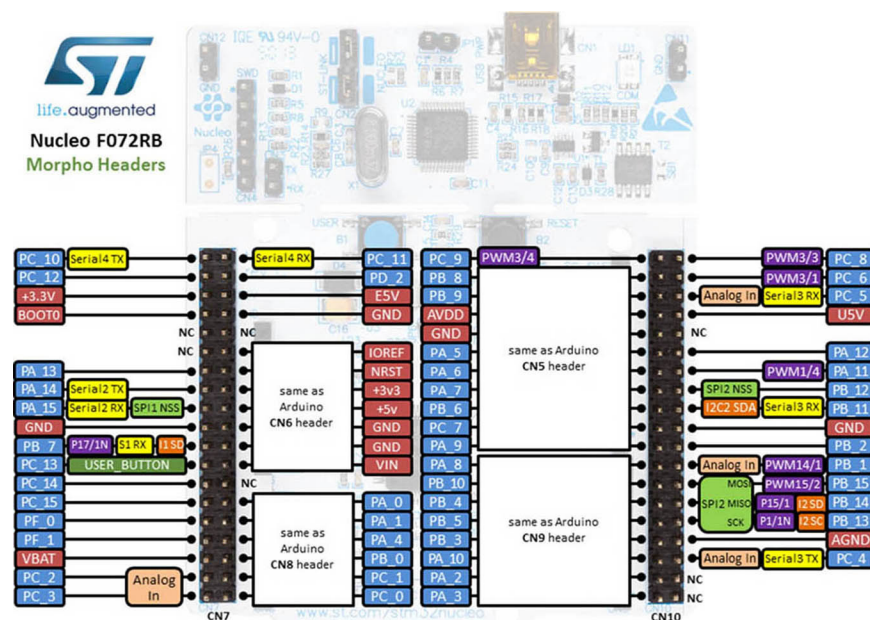


Nucleo-F072RB

Arduino compatible headers

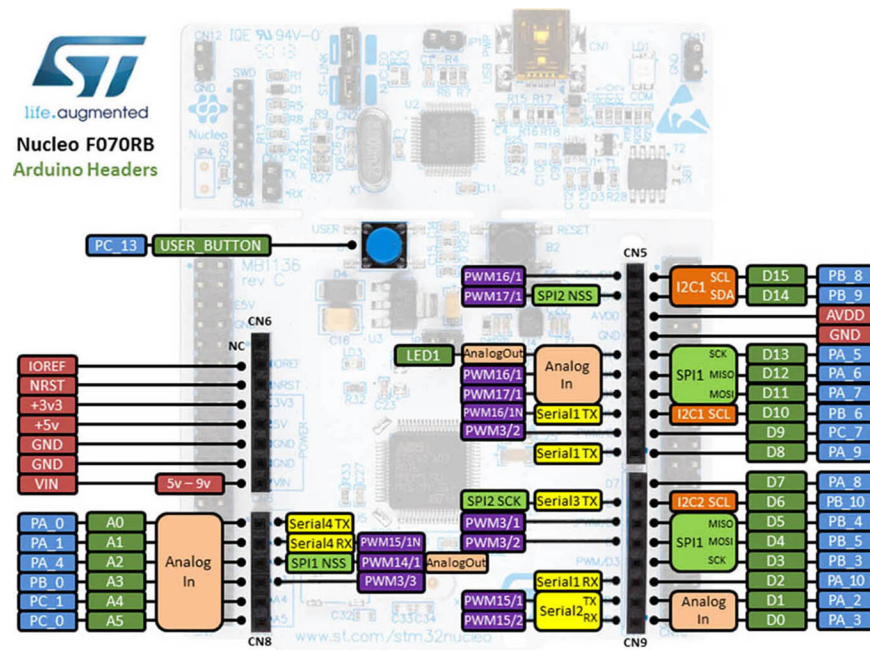


Morpho headers

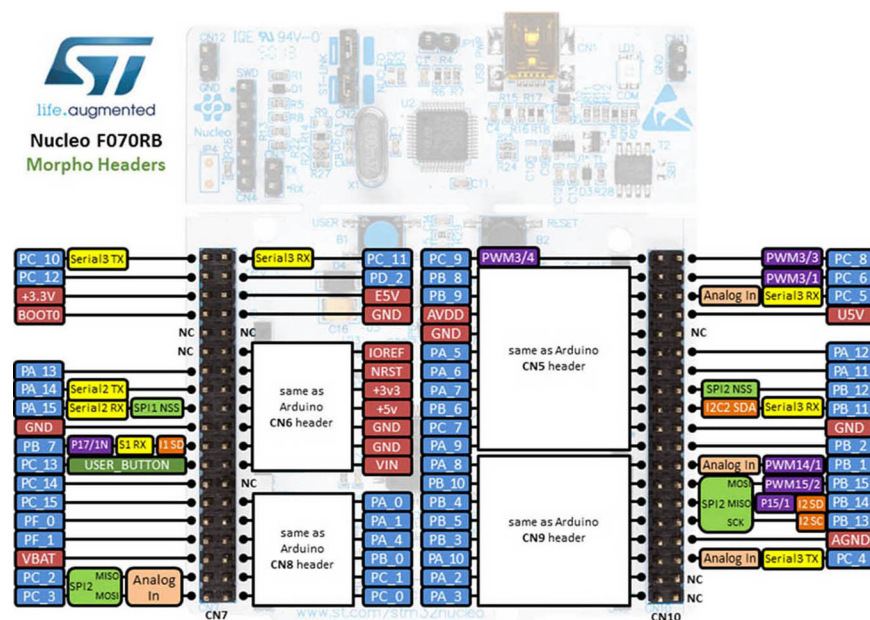


Nucleo-F070RB

Arduino compatible headers

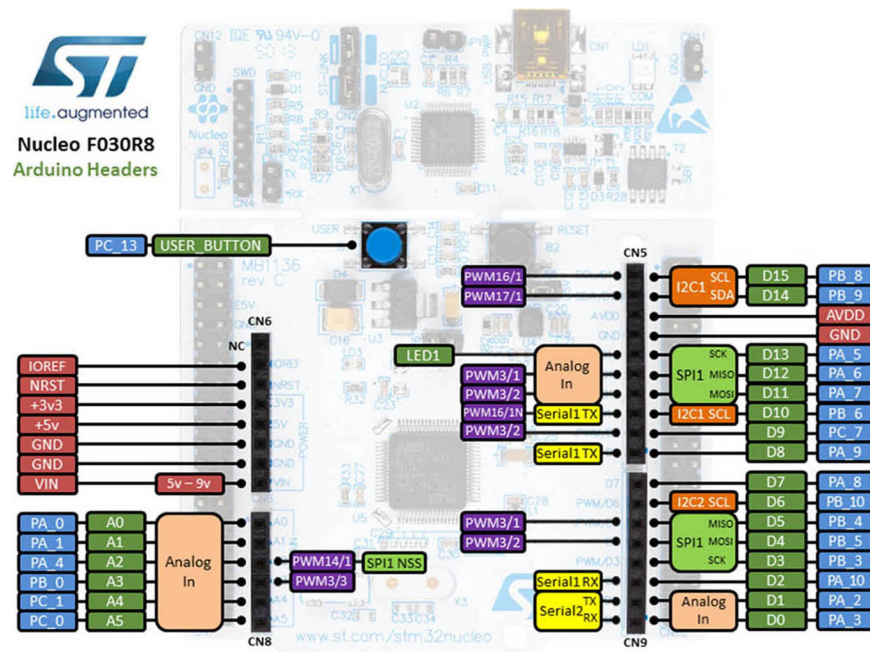


Morpho headers

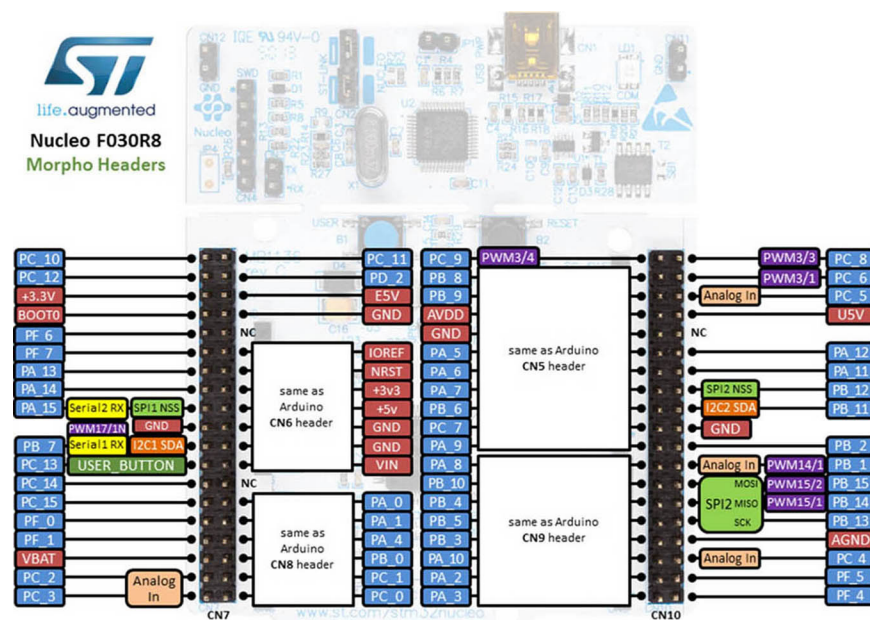


Nucleo-F030R8

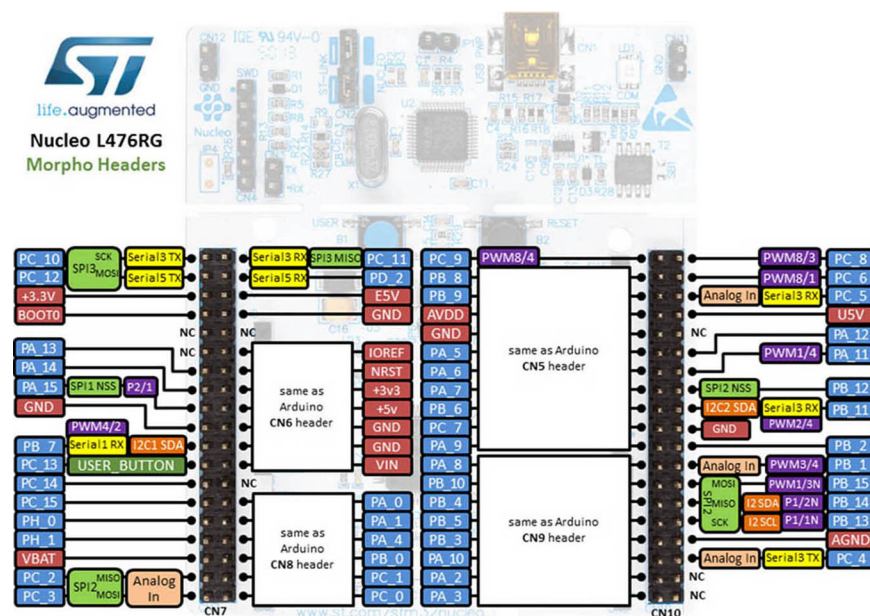
Arduino compatible headers



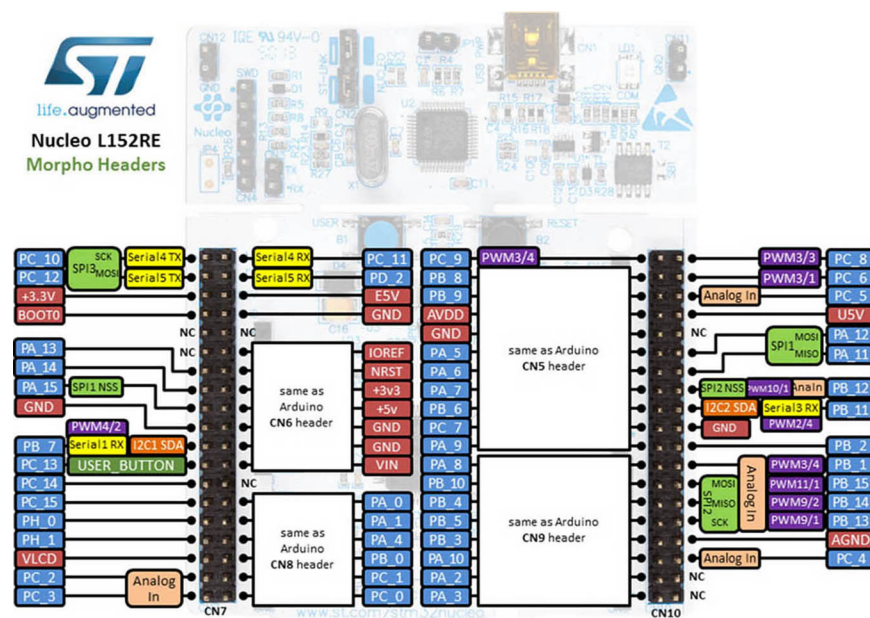
Morpho headers



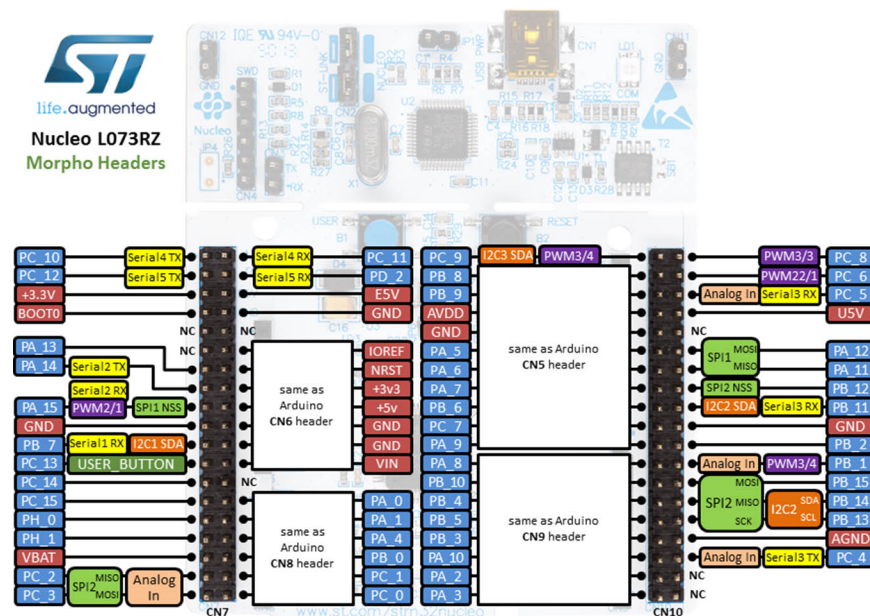
Arduino compatible headers



Arduino compatible headers

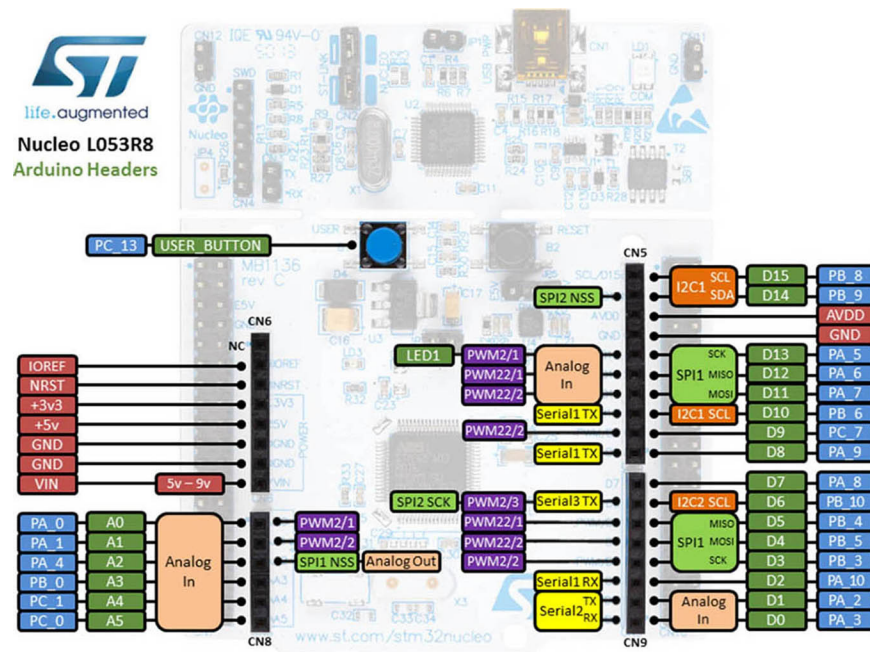


Arduino compatible headers

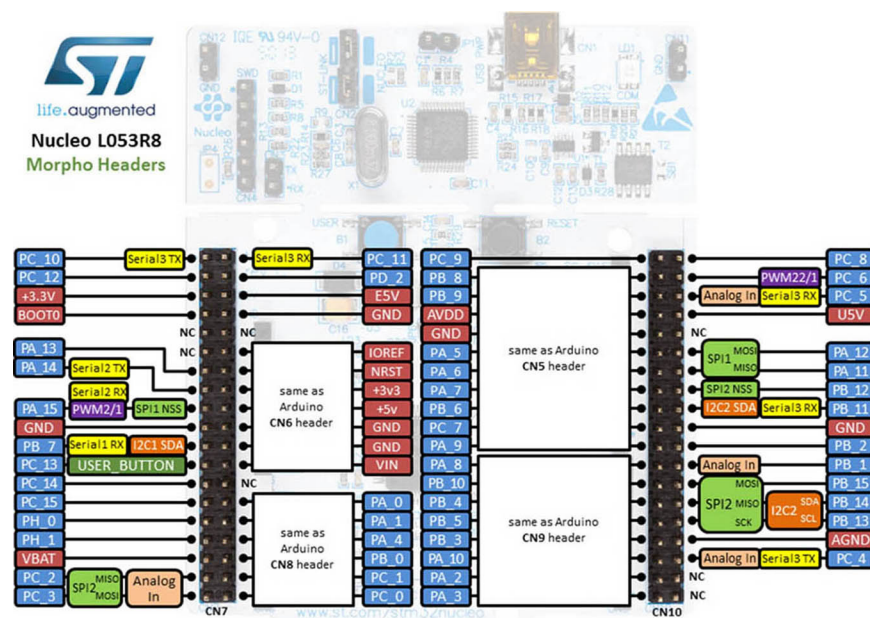


Nucleo-L053R8

Arduino compatible headers



Morpho headers



E. History of this book

Being this an in-progress book, it is interesting to publish a complete history of modifications.

Release 0.1 - October 2015

First public version of the book, made of 5 chapters.

Release 0.2 - October 28th, 2015

This release contains the following fixes:

- Changed the [Table 1 in Chapter 1](#): it wrongly stated that Cortex-M0/0+ allows 16 external configurable interrupts. Instead, it is 32.
- Paragraph 1.1.1.6 wrongly stated that the number of cycles required to service an interrupt is 12 for all Cortex-M processors. Instead it is equal to 12 cycles for all Cortex-M3/4/7 cores, 15 cycles for Cortex-M0, 16 cycles for Cortex-M0+.
- Fixed a lot of errors in the text. Really thanks to Enrico Colombini (aka Erix - <http://www.erix.it>) who is doing this dirty job.

This release adds the following chapters:

- Chapters 6 about GPIOs management.
- Added a troubleshooting section in the appendix.
- Added a section in the appendix about miscellaneous HAL functions.

Release 0.2.1 - October 31th, 2015

This release contains the following fixes:

- Changed again the [Table 1 in Chapter 1](#): it did not indicate which Cortex exceptions are not available in Cortex-M0/0+ based processors.
- Added several remarks to Chapter 4 (thanks again to Enrico Colombini) that better clarify some steps during the import of CubeMX generated output in the Eclipse project. Moreover, it is better explained why the [startup file](#) differs between Cortex-M0/0+ and Cortex-M3/4/7 processors.

Release 0.2.2 - November 1st, 2015

This release contains the following fixes:

- Changed in Chapter 4 (~pg. 140) the description of project generated by CubeMX, since ST has updated the template files after this author submitted a bug report. Now the code generated is generic and works with all Nucleo boards (even the F302 one).

Release 0.3 - November 12th, 2015

This release contains the following changes:

- Tool-chain installation instructions have been successfully tested on Windows XP, 7, 8.1 and the latest Windows 10.
- Added in chapter 4 the description of the CubeMXImporter, a tool made by this author to automatically import a CubeMX project into an Eclipse project made with the GNU MCU plug-in.

This release adds the following chapter:

- Chapter 7 about NVIC controller.

Release 0.4 - December 4th, 2015

This release contains the following changes:

- Added in Chapter 5 the definition of *freestanding environment*.
- Figures 11 and 12 in Chapter 5 have been updated to better clarify the signal levels.
- Added a paragraph about 96-bit Unique-ID in the Appendix A.

This release adds the following chapter:

- Chapter 8 about UART peripheral.

Release 0.5 - December 19th, 2015

This release adds the following chapter:

- Chapter 9 about how to start a new custom design with STM32 MCUs.

Release 0.6 - January 18th, 2016

This release adds the following chapter:

- Chapter 9 about DMA controller and HAL_DMA module.

Release 0.6.1 - January 20th, 2016

This release contains the following changes:

- Better clarified in paragraphs 7.1 and 7.2 the relation between NVIC and EXTI controller.
- In chapter 9 clarified that the BusMatrix also allows to automatically interconnect several peripherals between them. This topic will be explored in a subsequent chapter.
- Clarified at page 266 that we have to enable the DMA controller, using the macro `__DMA1_CLK_ENABLE()`, before we can use it.

Release 0.6.2 - January 30th, 2016

This release contains the following changes:

- The **Figure 4** in Chapter 1, and the text describing it, was completely wrong. It wrongly placed the boot loaders at the beginning of code area (`0x0000 0000`), while they are contained inside the *System memory*. Moreover, the role of the aliasing of flash addresses is better clarified, both there and in Chapter 7.
- Better clarified the role of *I-Bus*, *D-Bus* and *S-Bus* in Chapter 9.
- Fixed several errors in the text. Really thanks to Omar Shaker who is helping me.

Release 0.7 - February 8th, 2016

This release adds the following chapter:

- Chapter 10 about memory layout and linker scripts.
- Appendix C with correct pin-out for all Nucleo boards.

This release also better introduces the whole Nucleo lineup in Chapter 1. Moreover, BB-8 droid by Sphero is now among us. We welcome BB-8 (can you find it? :-)).

Release 0.8 - February 18th, 2016

This release adds the following chapter:

- Chapter 10 about clock tree configuration.

This release contains the following changes:

- In paragraph 4.1.1.2 the meaning of each IP Tree pane symbol has been better clarified.
- Fixed several errors in the text. Again, really thanks to Omar Shaker who is helping me.

Release 0.8.1 - February 23th, 2016

This release contains the following changes:

- The GCC tool-chain has been updated to the latest 5.2 release. There is nothing special to report.

Release 0.9 - March 27th, 2016

This release adds the following chapter:

- Chapter 11 about timers.

This release contains the following changes:

- The paragraph 9.2.6 has been updated: after several tests, I reach to the conclusion that the *peripheral-to-peripheral* transfer is possible only if the bus matrix is expressly designed to trigger transfers between the two peripherals.
- The paragraph 9.2.7 has been completely rewritten to better specify how to use the HAL_UART module in DMA mode.
- Added the paragraph 9.4 that explains the correct way to declare buffers for DMA transfers.
- Added the paragraph 10.1.1.1 about the MSI RC clock source in STM32L MCUs.
- Added the paragraph 10.1.3 about clock source options in Nucleo boards.
- Added in Appendix C the Nucleo-L073 and Nucleo-F410 pinout diagrams.

Release 0.9.1 - March 28th, 2016

This release contains the following changes:

- Installation instructions have been updated to the latest CubeMX 4.14, which now officially supports MacOS and Linux.

Release 0.10 - April 26th, 2016

This release adds the following chapter:

- Chapter 12 about low-power modes.

This release contains the following changes:

- Explained in paragraph 6.2.2 why the field `GPIO_InitTypeDef.Alternate` is missed in `CubeF1 HAL`.
- Fixed example 3 in Chapter 9. The example contained two errors, one related to the `EXTI2_3_IRQHandler()` and one to the priority of IRQs. The code in the book examples repository was instead correct.
- Added few words about I/O debouncing at page 207.
- The paragraph 7.6 has been completely rewritten to cover also the `BASEPRI` register.
- Added the paragraph 11.3.3 about how to generate timer-related events by software.
- ST engineers have changed the way a peripheral clock is enabled/disabled: now all the `__<PPP>_CLK_ENABLE()` macros have been renamed to `__HAL_RCC_<PPP>_CLK_ENABLE()`. The whole book has been updated. However, they are still leaving the old macro available for compatibility.

Release 0.11 - May 27th, 2016

This release adds the following chapter:

- Chapter 14 about FreeRTOS.

This release contains the following changes:

- Changed **Figure 16** in Chapter 7: the temporal sequences of ISR B and C were wrong.
- Changed **Figure 17** in Chapter 7: the sub-priority of ISRs B and C were wrong, because according to that execution sequence, the right sub-priority is 0x0 for C and 0x1 for B.
- Added another figure in Chapter 7 (the actual Figure 20), which better explains what happens when the *priority grouping* is lowered from 4 to 1 in that example. Thanks to Omar Shaker that helped me in refining this part.
- Paragraph 11.3.10.4 has been completely rewritten to better describe the update process of `TIMx->ARR` register.
- Clarified in Chapter 9 that, when using the UART in DMA mode, it is also important to enable the corresponding UART interrupt and to add a call to the `HAL_UART_IRQHandler()` from the ISR.
- Added an *Eclipse intermezzo* at the end of Chapter 6: it shows how to customize Eclipse appearance with themes.
- Added paragraph 12.3.3 regarding an important issue encountered with STM32F103 MCUs.
- Now the book has a brand new and professionally designed cover ;-)

Release 0.11.1 - June 3rd, 2016

This release contains the following changes:

- Better explained the *vector table* relocation process in 13.3.1 (in the previous releases of the book, the physical copy of the `.ccm` section from the flash memory to the CCM one was missed). The example 6 has been changed accordingly.

Release 0.11.2 - June 24th, 2016

This release contains the following changes:

- Tool-chain installation instruction have been updated to Eclipse 4.6 (Neon) and GCC 5.3.

Release 0.12 - July 4th, 2016

This release adds the following chapter:

- Chapter 12 about ADC.

This release contains the following changes:

- Better clarified in paragraph 7.2 the difference between enabling an interrupt at NVIC level and at the peripheral level.

Release 0.13 - July 18th, 2016

This release adds the following chapter:

- Chapter 13 about DAC.

Release 0.14 - August 12th, 2016

This release adds the following chapter:

- Chapter 17 about flash memory management.

This release contains the following changes:

- Clarified in paragraph 12.2.8 that the `hadc.Init.ContinuousConvMode` field must be set to `DISABLE`, otherwise the ADC performs conversions by itself without waiting the timer trigger.
- Added the paragraph 12.2.6.1 about how to convert multiple times the same channel in DMA mode (paragraph 12.2.6.1 is now 12.2.6.2).

Release 0.15 - September 13th, 2016

This release adds the following chapter:

- Chapter 17 about booting process in STM32 microcontrollers.

This release contains the following changes:

- Equation [4] in Chapter 9 was wrong because, to properly measure the period between two consecutive captures, the right formula is the following one (thanks to Davide Ruggiero to point me this out):

$$Period = Capture \cdot \left(\frac{TIMx_CLK}{(Prescaler + 1)(CH_{Prescaler})(PolarityIndex)} \right)^{-1} \quad [4]$$

- Described in Chapter 19 how to configure Eclipse to generate binary images of the firmware in *Release* mode.
- Added a new *Eclipse Intermezzo* at the end of the Chapter 7. It explains how to use code templates to increase coding productivity.

Release 0.16 - October 3th, 2016

This release adds the following chapter:

- Chapter 14 about I²C peripheral.

This release contains the following changes:

- Added the paragraph 16.4 about MPU unit.

Release 0.17 - October 24th, 2016

This release adds the following chapter:

- Chapter 15 about SPI peripheral.

This release contains the following changes:

- Better clarified in paragraph 12.2.8 that the timer's TRGO line must be properly configured to trigger the ADC conversion by using the `HAL_TIMEx_MasterConfigSynchronization()` routine, even if the timer is not configured in master mode.

Release 0.18 - November 15th, 2016

This release adds the following chapter:

- Chapter 21 about advanced debugging techniques.

This release contains the following changes:

- Added the paragraph 12.2.6.2 that explains how to perform multiple and not continuous conversions in DMA mode.
- Added the paragraph 1.3.7 that briefly mentions the new STM32H7-series.
- OpenOCD installation instructions for Windows, Linux and MacOS have been completely revised. Since the next OpenOCD release (0.10) is still under development, I have decided to use the precompiled packages made by Liviu Ionescu. This because they support the latest STM development boards. Several of you are, in fact, experiencing issues with OpenOCD 0.9. The latest development packages by Liviu should address these issues definitively. Please, Mac users take note that MacOS releases prior to 10.11 (aka El Capitan) are no longer supported.

Release 0.19 - November 29th, 2016

This release adds the following chapter:

- Chapter 16 about CRC peripheral.

Release 0.20 - December 28th, 2016

This release adds the following chapter:

- Chapter 17 about IWDT and WWDT timers.

Release 0.21 - January 29th, 2017

This release adds the following chapter:

- Chapter 24 about FatFs middleware library.

This release contains the following changes:

- Installation instructions have been updated to the latest official OpenOCD 0.10, Eclipse Neon.2 and GCC 5.4. **Please, take not that the latest ARM GCC 6.x appears to be incompatible with the current GNU MCU Eclipse plug-ins. So keep using the 5.4 branch until Liviu fixes incompatibilities.** Take also note that latest version of Eclipse needs Java SE 8 update 121.

Release 0.22 - May 2nd, 2017

This release adds the following chapter:

- Chapter 25 about W5500 ethernet processor.

This release contains the following changes:

- Chapter 22 has been updated to the latest FreeRTOS 9.x. *Please take note that ST still has not completed the rollout of latest FreeRTOS release to all STM32 families.*
- Equation [1] in Chapter 17 was wrong. Thank you to Michael Kaiser to let me know that.
- Instructions in paragraph 23.6 have been updated to better clarify how to retrieve the right ST-LINK serial number in Windows.
- Instructions in paragraph 8.3.1 have been updated to better clarify how to install RXTX library in Windows.
- ST refactored the HAL_IWDG and HAL_WWDG modules. The chapter 17 has been updated to cover the new APIs.
- This book is almost finished! Now it is the right time to add an acknowledgments section to thank all those people that helped me to make this work possible.

Release 0.23 - July 20th, 2017

This release adds the following chapter:

- Chapter 18 about RTC.

This release contains the following changes:

- Chapter 4 has been updated to cover CubeMX 4.22 features.
- Updated paragraph 23.3.4 to cover the new behaviour in FreeRTOS 9.x: now if one thread deletes another thread, then the memory allocated by FreeRTOS to the deleted thread is freed immediately.

Release 0.24 - December 11th, 2017

This release contains the following changes:

- Chapter 1 has been updated to cover the new STM32L4+ family. Moreover, the STM32L4 series has been updated to cover the latest MCUs.
- Installation instructions in Chapter 2 have been updated to cover Eclipse Oxygen and the latest GNU MCU Eclipse plug-ins.

Release 0.25 - January 3rd, 2018

This release contains the following changes:

- ST has released a new flashing utility named STM32CubeProgrammer. The big news is that STM32CubeProgrammer is now multi-platform, and it runs on Windows, Mac and Linux. The tool is not yet perfectly stable, but it is a good start. That allowed me to review installation instructions: now there is no longer need to install QSTLink2 and texane.

Release 0.26 - May 7th, 2018

This release contains the following changes:

- Chapter 1 has been updated to cover the new STM32WB family.
- Minor fixes to the text.