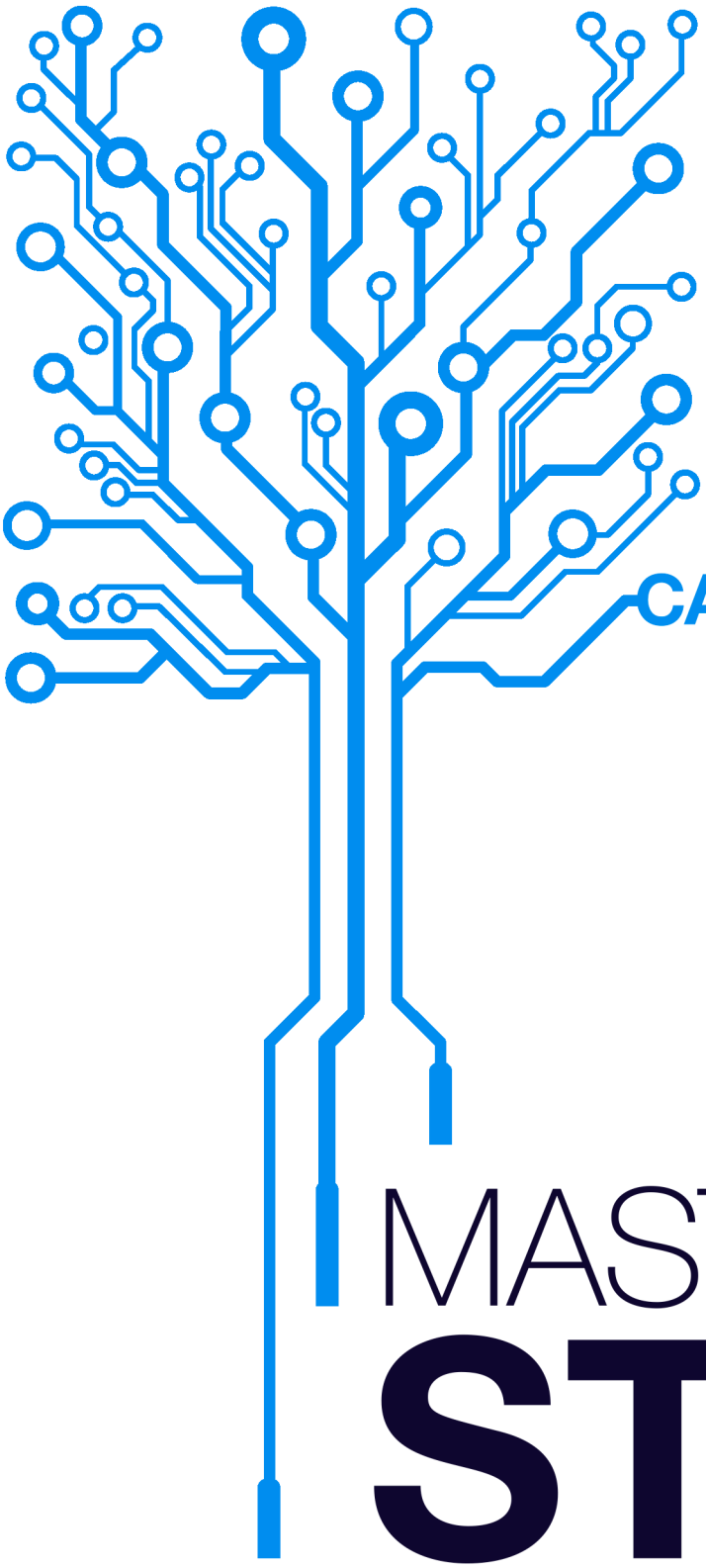
RELEASE **2.00**



CARMINE NOVIELLO

MASTERING STM32

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

SECOND EDITION

Mastering STM32 - Second Edition

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

Carmine Noviello

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

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

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

To my daughter Giulia, who completely upset my projects

Contents

Preface . . . . . **i**
Who Is This Book For? ii
How to Integrate This Book? iii
How Is the Book Organized? iv
Differences With the First Edition vii
About the Author viii
Errata and Suggestions ix
Book Support ix
How to Help the Author ix
Copyright Disclaimer ix
Credits x

Acknowledgments to the First Edition . . . . . **xi**

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 8
1.1.1.4 Thumb-2 and Memory Alignment 11
1.1.1.5 Pipeline 13
1.1.1.6 Interrupts and Exceptions Handling 14
1.1.1.7 SysTimer 16
1.1.1.8 Power Modes 17
1.1.1.9 TrustZone™ 18
1.1.1.10 CMSIS 19
1.1.1.11 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

1.2.2 ....And Its Drawbacks . . . 23
1.3 A Quick Look at the STM32 Subfamilies . . . 24
1.3.1 F0 . . . 26
1.3.2 F1 . . . 27
1.3.3 F2 . . . 28
1.3.4 F3 . . . 29
1.3.5 F4 . . . 31
1.3.6 F7 . . . 33
1.3.7 H7 . . . 35
1.3.8 L0 . . . 37
1.3.9 L1 . . . 38
1.3.10 L4 . . . 39
1.3.11 L4+ . . . 41
1.3.12 L5 . . . 42
1.3.13 U5 . . . 43
1.3.14 G0 . . . 45
1.3.15 G4 . . . 46
1.3.16 STM32WB . . . 48
1.3.17 STM32WL . . . 50
1.3.18 How to Select the Right MCU for You? . . . 51
1.4 The Nucleo Development Board . . . 54

2. Get In Touch With SM32CubeIDE . . . 60
2.1 Why Choose STM32CubeIDE as Tool-Chain for STM32 . . . 60
2.1.1 Two Words About Eclipse... . . . 62
2.1.2 ... and GCC . . . 62
2.2 Downloading and Installing the STM32CubeIDE . . . 63
2.2.1 Windows - Installing the Tool-Chain . . . 64
2.2.2 Linux - Installing the Tool-Chain . . . 67
2.2.3 Mac - Installing the Tool-Chain . . . 68
2.3 STM32CubeIDE overview . . . 70

3. Hello, Nucleo! . . . 76
3.1 Create a Project . . . 76
3.2 Adding Something Useful to the Generated Code . . . 79
3.3 Connecting the Nucleo to the PC . . . 84
3.3.1 ST-LINK Firmware Upgrade . . . 85
3.4 Flashing the Nucleo using STM32CubeProgrammer . . . 86

4. STM32CubeMX Tool . . . 89
4.1 Introduction to CubeMX Tool . . . 89
4.1.1 Target Selection Wizard . . . 90
4.1.1.1 MCU/MPU Selector . . . 91
4.1.1.2 Board Selector . . . 92

4.1.1.3 Example Selector . . . 92
4.1.1.4 Cross Selector . . . 94
4.1.2 MCU and Middleware Configuration . . . 94
4.1.2.1 Pinout View & Configuration . . . 95
4.1.2.2 Clock Configuration View . . . 100
4.1.3 Project Manager . . . 102
4.1.4 Tools View . . . 104
4.2 Understanding Project Structure . . . 105
4.3 Downloading Book Source Code Examples . . . 113
4.4 Management of STM32Cube Packages . . . 115

5. Introduction to Debugging . . . 117
5.1 What is Behind a Debug Session . . . 117
5.2 Debugging With STM32CubeIDE . . . 119
5.2.1 Debug Configurations . . . 122
5.3 I/O Retargeting . . . 125

II Diving into the HAL . . . 129

6. GPIO Management . . . 130
6.1 STM32 Peripherals Mapping and HAL *Handlers* . . . 130
6.2 GPIOs Configuration . . . 135
6.2.1 GPIO Mode . . . 136
6.2.2 GPIO Alternate Function . . . 139
6.3 Driving a GPIO . . . 140
6.4 De-initialize a GPIO . . . 140

7. Interrupts Management . . . 143
7.1 NVIC Controller . . . 143
7.1.1 Vector Table in STM32 . . . 144
7.2 Enabling Interrupts . . . 148
7.2.1 External Lines and NVIC . . . 149
7.2.2 Enabling Interrupts with CubeMX . . . 153
7.3 Interrupt Lifecycle . . . 155
7.4 Interrupt Priority Levels . . . 159
7.4.1 Cortex-M0/0+ . . . 159
7.4.2 Cortex-M3/4/7/33 . . . 164
7.4.3 Setting Interrupt Priority in CubeMX . . . 171
7.5 Interrupt Re-Entrancy . . . 171
7.6 Mask All Interrupts at Once or an a Priority Basis . . . 173

8. Universal Asynchronous Serial Communications . . . 176
8.1 Introduction to UARTs and USARTs . . . 176

8.2 UART Initialization . . . 180
8.2.1 UART Configuration Using CubeMX . . . 187
8.3 UART Communication in *Polling Mode* . . . 188
8.3.1 Installing a Terminal Emulator in Eclipse . . . 192
8.4 UART Communication in *Interrupt Mode* . . . 193
8.4.1 UART Related Interrupts . . . 194
8.5 Error Management . . . 201
8.6 List of Available Callbacks in the HAL_UART Module . . . 202

9. DMA Management . . . 205
9.1 Introduction to DMA . . . 205
9.1.1 The Need of a DMA and the Role of the Internal Buses . . . 206
9.1.2 The DMA Controller . . . 209
9.1.2.1 The DMA Implementation in F0/F1/F3/L0/L1/L4 MCUs . . . 210
9.1.2.2 The DMA Implementation in F2/F4/F7 MCUs . . . 214
9.1.2.3 The DMA Implementation in G0/G4/L4+/L5/H7 MCUs . . . 217
9.2 HAL_DMA Module . . . 220
9.2.1 DMA_HandleTypeDef in F0/F1/F3/L0/L1/L4 HALs . . . 220
9.2.2 DMA Configuration in G0/G4/L4+/L5/H7 HALs . . . 223
9.2.3 DMA_HandleTypeDef in F2/F4/F7 HALs . . . 226
9.2.4 How to Perform DMA Transfers in Polling Mode . . . 229
9.2.5 How to Perform DMA Transfers in Interrupt Mode . . . 232
9.2.6 Using the HAL_UART Module with DMA Mode Transfers . . . 233
9.2.7 Programming the DMAMUX With the CubeHAL . . . 236
9.2.8 Miscellaneous Functions From HAL_DMA and HAL_DMA_Ex Modules . . . 237
9.3 Using CubeMX to Configure DMA Requests . . . 238
9.4 Correct Memory Allocation of DMA Buffers . . . 239
9.5 A Case Study: The DMA *Memory-To-Memory* Transfer Performance Analysis . . . 240

10. Clock Tree . . . 245
10.1 Clock Distribution . . . 245
10.1.1 Overview of the STM32 Clock Tree . . . 246
10.1.1.1 The Multispeed Internal RC Oscillator in STM32L/U Families 250
10.1.2 Configuring Clock Tree Using CubeMX . . . 251
10.1.3 Clock Source Options in Nucleo Boards . . . 253
10.1.3.1 Clock Source in Nucleo-64 rev. MB1136 (older ones with ST-LINK V2.1) . . . 254
10.1.3.1.1 OSC Clock Supply . . . 254
10.1.3.1.2 OSC 32kHz Clock Supply . . . 255
10.1.3.2 Clock Source in Nucleo-64 rev. MB1367 (newer ones with ST-LINK v3) . . . 256
10.1.3.2.1 OSC Clock Supply . . . 256
10.1.3.2.2 OSC 32kHz Clock Supply . . . 257

10.2 Overview of the HAL_RCC Module . . . 257
10.2.1 Compute the Clock Frequency at Run-Time . . . 259
10.2.2 Enabling the *Master Clock Output* . . . 260
10.2.3 Enabling the *Clock Security System* . . . 260
10.3 HSI Calibration . . . 261

11. Timers . . . 263
11.1 Introduction to Timers . . . 263
11.1.1 Timer Categories in an STM32 MCU . . . 264
11.1.2 Effective Availability of Timers in the STM32 Portfolio . . . 266
11.2 Basic Timers . . . 269
11.2.1 Using Timers in *Interrupt Mode* . . . 272
11.2.1.1 Time Base Generation in *Advanced* Timers . . . 274
11.2.2 Using Timers in *Polling Mode* . . . 274
11.2.3 Using Timers in *DMA Mode* . . . 275
11.2.4 Stopping a Timer . . . 278
11.2.5 Using CubeMX to Configure a *Basic Timer* . . . 278
11.3 General Purpose Timers . . . 278
11.3.1 Time Base Generator with External Clock Sources . . . 279
11.3.1.1 External Clock Mode 2 . . . 281
11.3.1.2 External Clock Mode 1 . . . 284
11.3.1.3 Using CubeMX to Configure the Source Clock of a *General Purpose* Timer . . . 289
11.3.2 Master/Slave Synchronization Modes . . . 290
11.3.2.1 Enable Trigger-Related Interrupts . . . 296
11.3.2.2 Using CubeMX to Configure the Master/Slave Synchronization . . . 296
11.3.3 Generate Timer-Related Events by Software . . . 297
11.3.4 Counting Modes . . . 299
11.3.5 Input Capture Mode . . . 300
11.3.5.1 Using CubeMX to Configure the Input Capture Mode . . . 307
11.3.6 Output Compare Mode . . . 308
11.3.6.1 Using CubeMX to Configure the Output Compare Mode . . . 313
11.3.7 Pulse-Width Generation . . . 313
11.3.7.1 Generating a Sinusoidal Wave Using PWM . . . 317
11.3.7.2 Using CubeMX to Configure the PWM Mode . . . 322
11.3.8 One Pulse Mode . . . 322
11.3.8.1 Using CubeMX to Configure the OPM Mode . . . 325
11.3.9 Encoder Mode . . . 325
11.3.9.1 Using CubeMX to Configure the *Encoder Mode* . . . 331
11.3.10 Other Features Available in *General Purpose* and *Advanced* Timers . . . 331
11.3.10.1 *Hall Sensor* Mode . . . 332

11.3.10.2 Combined Three-Phase PWM Mode and Other Motor-Control Related Features . . . 332
11.3.10.3 Break Input and Locking of Timer Registers . . . 333
11.3.10.4 Preloading of Auto-Reload Register . . . 333
11.3.11 Debugging and Timers . . . 334
11.4 SysTick Timer . . . 335
11.4.1 Use Another Timer as System Timebase Source . . . 336
11.5 A Case Study: How to Precisely Measure Microseconds with STM32 MCUs . . . 337

12. Analog-To-Digital Conversion . . . 343
12.1 Introduction to SAR ADC . . . 343
12.2 HAL_ADC Module . . . 348
12.2.1 Conversion Modes . . . 350
12.2.1.1 Single-Channel, Single Conversion Mode . . . 350
12.2.1.2 Scan Single Conversion Mode . . . 351
12.2.1.3 Single-Channel, Continuous Conversion Mode . . . 351
12.2.1.4 Scan Continuous Conversion Mode . . . 352
12.2.1.5 Injected Conversion Mode . . . 352
12.2.1.6 Dual Modes . . . 353
12.2.2 Channel Selection . . . 353
12.2.3 ADC Resolution and Conversion Speed . . . 355
12.2.4 A/D Conversions in Polling Mode . . . 356
12.2.5 A/D Conversions in Interrupt Mode . . . 360
12.2.6 A/D Conversions in DMA Mode . . . 361
12.2.6.1 Convert Multiple Times the Same Channel in DMA Mode . 365
12.2.6.2 Multiple and not Continuous Conversions in DMA Mode . 365
12.2.6.3 Continuous Conversions in DMA Mode . . . 365
12.2.7 Errors Management . . . 365
12.2.8 Timer-Driven Conversions . . . 366
12.2.9 Conversions Driven by External Events . . . 370
12.2.10 ADC Calibration . . . 370
12.3 Using CubeMX to Configure ADC Peripheral . . . 371

13. Digital-To-Analog Conversion . . . 373
13.1 Introduction to the DAC Peripheral . . . 373
13.2 HAL_DAC Module . . . 375
13.2.1 Driving the DAC Manually . . . 377
13.2.2 Driving the DAC in DMA Mode Using a Timer . . . 379
13.2.3 Triangular Wave Generation . . . 383
13.2.4 Noise Wave Generation . . . 384

14. I²C . . . 385
14.1 Introduction to the I²C specification . . . 385
14.1.1 The I²C Protocol . . . 387

14.1.1.1 START and STOP Condition . . . 388
14.1.1.2 Byte Format . . . 388
14.1.1.3 Address Frame . . . 388
14.1.1.4 Acknowledge (ACK) and Not Acknowledge (NACK) . . . 389
14.1.1.5 Data Frames . . . 389
14.1.1.6 Combined Transactions . . . 390
14.1.1.7 Clock Stretching . . . 391
14.1.2 Availability of I²C Peripherals in STM32 MCUs . . . 391
14.2 HAL_I2C Module . . . 392
14.2.1 Using the I²C Peripheral in *Master Mode* . . . 396
14.2.1.1 I/O MEM Operations . . . 403
14.2.1.2 Combined Transactions . . . 405
14.2.1.3 A Note About the Clock Configuration in STM32F0/L0/L4 families . . . 406
14.2.2 Using the I²C Peripheral in *Slave Mode* . . . 406
14.3 Using CubeMX to Configure the I²C Peripheral . . . 412

15. SPI . . . 414
15.1 Introduction to the SPI Specification . . . 414
15.1.1 Clock Polarity and Phase . . . 417
15.1.2 Slave Select Signal Management . . . 418
15.1.3 SPI *TI Mode* . . . 418
15.1.4 Availability of SPI Peripherals in STM32 MCUs . . . 419
15.2 HAL_SPI Module . . . 420
15.2.1 Exchanging Messages Using SPI Peripheral . . . 422
15.2.2 Maximum Transmission Frequency Reachable using the CubeHAL . . . 424
15.3 Using CubeMX to Configure SPI Peripheral . . . 424

16. Cyclic Redundancy Check . . . 425
16.1 Introduction to CRC Computing . . . 425
16.1.1 CRC Calculation in STM32F1/F2/F4/L1 MCUs . . . 428
16.1.2 CRC Peripheral in STM32F0/F3/F7/L0/L4/L5/G0/G4 MCUs . . . 430
16.2 HAL_CRC Module . . . 431

17. IWDG and WWDG Timers . . . 434
17.1 The *Independent Watchdog* Timer . . . 434
17.1.1 Using the CubeHAL to Program IWDG Timer . . . 435
17.2 The *System Window Watchdog* Timer . . . 436
17.2.1 Using the CubeHAL to Program WWDG Timer . . . 438
17.3 Detecting a System Reset Caused by a Watchdog Timer . . . 439
17.4 Freezing Watchdog Timers During a Debug Session . . . 440
17.5 Selecting the Right Watchdog Timer for Your Application . . . 440

18. Real-Time Clock . . . 441

18.1 Introduction to the RTC Peripheral . . . 441
18.2 HAL_RTC Module . . . 443
18.2.1 Setting and Retrieving the Current Date/Time . . . 444
18.2.1.1 Correct Way to Read Date/Time Values . . . 446
18.2.2 Configuring Alarms . . . 447
18.2.3 Periodic Wakeup Unit . . . 449
18.2.4 Timestamp Generation and Tamper Detection . . . 451
18.2.5 RTC Calibration . . . 451
18.2.5.1 RTC Coarse Calibration . . . 452
18.2.5.2 RTC Smooth Calibration . . . 452
18.2.5.3 Reference Clock Detection . . . 454
18.3 Using the Backup SRAM . . . 454

III Advanced topics . . . 456

19. Power Management . . . 457
19.1 Power Management in Cortex-M Based MCUs . . . 457
19.2 How Cortex-M MCUs Handle *Run* and *Sleep* Modes . . . 458
19.2.1 Entering/exiting sleep modes . . . 461
19.2.1.1 Sleep-On-Exit . . . 463
19.2.2 *Sleep* Modes in Cortex-M Based MCUs . . . 463
19.3 Power Management in STM32F Microcontrollers . . . 464
19.3.1 Power Sources . . . 464
19.3.2 Power Modes . . . 465
19.3.2.1 Run Mode . . . 466
19.3.2.1.1 Dynamic Voltage Scaling in STM32F4/F7 MCUs . . . 467
19.3.2.1.2 Over/Under-Drive Mode in STM32F4/F7 MCUs . . . 468
19.3.2.2 Sleep Mode . . . 468
19.3.2.3 Stop Mode . . . 469
19.3.2.4 Standby Mode . . . 470
19.3.2.5 Low-Power Modes Example . . . 470
19.3.3 An Important Warning for STM32F1 Microcontrollers . . . 474
19.4 Power Management in STM32L/G Microcontrollers . . . 476
19.4.1 Power Sources . . . 476
19.4.2 Power Modes . . . 478
19.4.2.1 Run Modes . . . 478
19.4.2.2 Sleep Modes . . . 480
19.4.2.2.1 Batch Acquisition Mode . . . 481
19.4.2.3 Stop Modes . . . 481
19.4.2.4 Standby Modes . . . 482
19.4.2.5 Shutdown Mode . . . 483
19.4.3 Power Modes Transitions . . . 483

19.4.4 Low-Power Peripherals . . . 484
19.4.4.1 LPUART . . . 484
19.4.4.2 LPTIM . . . 485
19.4.4.3 LPGPIO . . . 485
19.4.4.4 LPDMA . . . 486
19.5 Power Supply Supervisors . . . 486
19.6 Debugging in Low-Power Modes . . . 487
19.7 Using the CubeMX Power Consumption Calculator . . . 487
19.8 A Case Study: Using Watchdog Timers With Low-Power Modes . . . 489

20. Memory layout . . . 490
20.1 The STM32 Memory Layout Model . . . 490
20.1.1 Flash Memory Typical Organization . . . 490
20.1.2 SRAM Memory Typical Organization . . . 492
20.1.3 Understanding Compilation and Linking Processes . . . 493
20.2 The Really Minimal STM32 Application . . . 495
20.2.1 ELF Binary File Inspection . . . 499
20.2.2 `.data` and `.bss` Sections Initialization . . . 501
20.2.2.1 A Word About the `COMMON` Section . . . 508
20.2.3 `.rodata` Section . . . 509
20.2.4 Stack and Heap Regions . . . 511
20.2.5 Checking the Size of Heap and Stack at Compile-Time . . . 514
20.2.6 Differences With the Tool-Chain Script Files . . . 515
20.3 How to Use the CCM Memory . . . 520
20.3.1 Relocating the *vector table* in CCM Memory . . . 524
20.4 How to Use the MPU in Cortex-M0+/3/4/7 Based STM32 MCUs . . . 527
20.4.1 Programming the MPU With the CubeHAL . . . 531

21. Flash Memory Management . . . 534
21.1 Introduction to STM32 Flash Memory . . . 534
21.2 The `HAL_FLASH` Module . . . 537
21.2.1 Flash Memory Unlocking . . . 538
21.2.2 Flash Memory Erasing . . . 538
21.2.3 Flash Memory Programming . . . 540
21.2.4 Flash Read Access During Programming and Erasing . . . 541
21.3 Option Bytes . . . 541
21.3.1 Flash Memory Read Protection . . . 543
21.4 Optional OTP and True-EEPROM Memories . . . 545
21.5 Flash Read Latency and the ART™ Accelerator . . . 547
21.5.1 The Role of the TCM Memories in STM32F7/H7 MCUs . . . 549
21.5.1.1 How to Access Flash Memory Through the TCM Interface . 555
21.5.1.2 Using CubeMX to Configure Flash Memory Interface . . . 556

22. Booting Process . . . 558

22.1 The Cortex-M Unified Memory Layout and the Booting Process . . . 558
22.1.1 Software *Physical Remap* . . . 559
22.1.2 Vector Table Relocation . . . 560
22.1.3 Running the Firmware From SRAM Using the STM32CubeIDE . . . 562
22.2 Integrated STM32 Bootloader . . . 563
22.2.1 Starting the STM32 Bootloader from the On-Board Firmware . . . 566
22.2.2 The Booting Sequence in the STM32CubeIDE Tool-chain . . . 567
22.3 Developing a Custom Bootloader . . . 568
22.3.1 *Vector Table* Relocation in STM32F0 Microcontrollers . . . 579
22.3.2 How to Use the `flasher.py` Tool . . . 581

23. Running FreeRTOS . . . 584
23.1 Understanding the Concepts Underlying an RTOS . . . 585
23.2 Configuring FreeRTOS and the CMSIS-RTOS v2 Wrapper . . . 591
23.2.1 The FreeRTOS Source Tree . . . 592
23.2.1.1 How to Configure FreeRTOS Using CubeMX . . . 593
23.3 Thread Management . . . 594
23.3.1 Thread States . . . 597
23.3.2 Thread Priorities and Scheduling Policies . . . 598
23.3.3 Voluntary Release of the Control . . . 602
23.3.4 The *idle* Thread . . . 602
23.4 Memory Allocation and Management . . . 604
23.4.1 Dynamic Memory Allocation Model . . . 604
23.4.1.1 `heap_1.c` . . . 605
23.4.1.2 `heap_2.c` . . . 606
23.4.1.3 `heap_3.c` . . . 606
23.4.1.4 `heap_4.c` . . . 607
23.4.1.5 `heap_5.c` . . . 607
23.4.1.6 FreeRTOS Heap Definition . . . 608
23.4.2 Static Memory Allocation Model . . . 608
23.4.2.1 *idle* Thread Allocation with Static Memory Allocation Model 609
23.4.3 FreeRTOS and the C stdlib . . . 609
23.4.3.1 How to Configure `newlib` to Handle Concurrency with FreeRTOS . . . 610
23.4.3.2 How to Use `malloc()` and `malloc()`-dependant `newlib` Functions With FreeRTOS . . . 613
23.4.3.3 STM32CubeMX Approach to Thread-Safety . . . 621
23.4.4 Memory Pools . . . 622
23.4.5 Stack Overflow Detection . . . 625
23.5 Synchronization Primitives . . . 627
23.5.1 Message Queues . . . 627
23.5.2 Semaphores . . . 630
23.5.3 Event and Thread Flags . . . 633

23.6 Resources Management and Mutual Exclusion . . . 637
23.6.1 Mutexes . . . 637
23.6.1.1 The Priority Inversion Problem . . . 639
23.6.1.2 Recursive Mutexes . . . 640
23.6.2 Critical Sections . . . 640
23.6.3 Interrupt Management With an RTOS . . . 641
23.6.3.1 FreeRTOS API and Interrupt Priorities . . . 642
23.7 Software Timers . . . 643
23.7.1 How FreeRTOS Manages Timers . . . 645
23.8 A Case Study: Low-Power Management With an RTOS . . . 645
23.8.1 The *idle* Thread Hook . . . 646
23.8.2 The Tickless Mode in FreeRTOS . . . 647
23.8.2.1 A Schema for the *tickless* Mode . . . 649
23.8.2.2 A Custom *tickless* Mode Policy . . . 652
23.9 Debugging Features . . . 660
23.9.1 `configASSERT()` Macro . . . 660
23.9.2 Run-Time Statistics and Thread State Information . . . 661
23.9.3 FreeRTOS Debugging in STM32CubeIDE . . . 665
23.9.4 FreeRTOS Kernel-Aware Debugging in STM32CubeIDE . . . 668
23.10 Alternatives to FreeRTOS . . . 670
23.10.1 AzureRTOS . . . 670
23.10.2 ChibiOS . . . 670
23.10.3 Contiki OS . . . 671
23.10.4 OpenRTOS . . . 671

24. Advanced Debugging Techniques . . . 673
24.1 Understanding Cortex-M Fault-Related Exceptions . . . 673
24.1.1 The Cortex-M Exception Entrance Sequence and the ARM Calling Convention . . . 675
24.1.1.1 How to Interpret the Content of the LR Register on Exception Entrance . . . 680
24.1.2 Fault Exceptions and Faults Analysis . . . 681
24.1.2.1 *Memory Management* Exception . . . 682
24.1.2.2 *Bus Fault* Exception . . . 682
24.1.2.3 *Usage Fault* Exception . . . 683
24.1.2.4 *Hard Fault* Exception . . . 684
24.1.2.5 *Secure Fault* Exception . . . 685
24.1.2.6 Enabling Optional Fault Handlers . . . 685
24.1.2.7 Fault Analysis in Cortex-M0/0+ Based Processors . . . 686
24.2 STM32CubeIDE Advanced Debugging Features . . . 686
24.2.1 Expressions and Live Expressions . . . 686
24.2.1.1 Memory Monitors . . . 688
24.2.2 Watchpoints . . . 689

24.2.3 Instruction Stepping Mode . . . 690
24.2.4 SFRs View . . . 690
24.2.5 Fault Analyzer . . . 691
24.2.5.1 Tracing Fault-Related Registers Without the IDE Support . 693
24.2.6 Build Analyzer . . . 696
24.2.7 Static Stack Analyzer . . . 697
24.3 Serial Wire Viewer Tracing . . . 698
24.3.1 Enabling SWV Debugging . . . 700
24.3.2 Configuring SWV . . . 701
24.3.3 SWV Views . . . 703
24.3.3.1 SWV Trace Log . . . 704
24.3.3.2 SWV Exception Trace Log . . . 704
24.3.3.3 SWV Data Trace . . . 705
24.3.3.4 SWV Data Trace Timeline Graph . . . 706
24.3.3.5 SWV ITM Data Console . . . 707
24.3.3.6 SWV Statistical Profiling . . . 708
24.4 Debugging Aids from the CubeHAL . . . 709
24.5 External Debuggers . . . 709
24.6 Debugging two Nucleo Boards Simultaneously . . . 711
24.7 ARM Semihosting . . . 712
24.7.1 Enable Semihosting on a Project . . . 713
24.7.2 Semihosting Drawbacks . . . 715
24.7.3 Understanding How Semihosting Works . . . 716

25. FAT Filesystem . . . 720
25.1 Introduction to FatFs Library . . . 720
25.1.1 Adding FatFs Library in Your Projects . . . 723
25.1.1.1 The *Generic Disk Interface* API . . . 724
25.1.1.2 The Implementation of a Driver to Access SD Cards in SPI Mode . . . 725
25.1.2 Relevant FatFs Structures and Functions . . . 725
25.1.2.1 Mounting a Filesystem . . . 725
25.1.2.2 Opening a File . . . 726
25.1.2.3 Reading From/Writing into a File . . . 727
25.1.2.4 Creating and Opening a Directory . . . 728
25.1.3 How to Configure the FatFs Library . . . 731

26. Develop IoT Applications . . . 733
26.1 Solutions Offered by STM to Develop IoT Applications . . . 734
26.2 The W5500 Ethernet Controller . . . 736
26.2.1 How to Use the W5500 Shield and the `ioLibrary_Driver` Module . . . 740
26.2.1.1 Configuring the SPI Interface . . . 742
26.2.1.2 Configuring the Socket Buffers and the Network Interface . 743

26.2.2 Socket APIs . . . 745
26.2.2.1 Handling Sockets in TCP Mode . . . 746
26.2.2.2 Handling Sockets in UDP Mode . . . 747
26.2.3 I/O Retargeting to a TCP/IP Socket . . . 748
26.2.4 Building up an HTTP Server . . . 749
26.2.4.1 A Web-Based Oscilloscope . . . 752

27. Universal Serial Bus . . . 765
27.1 USB 2.0 Specification Overview . . . 766
27.1.1 The "Before-To-Die" Guide to USB . . . 766
27.1.2 USB Physical Architecture Overview . . . 769
27.1.3 USB Logical Architecture Overview . . . 772
27.1.3.1 Device States . . . 772
27.1.3.2 Communication Endpoints . . . 774
27.1.4 USB 2.0 Communication Protocol Overview . . . 777
27.1.4.1 Packet Types . . . 778
27.1.4.2 Transaction Types . . . 780
27.1.4.2.1 Control Transactions . . . 780
27.1.4.2.2 IN/OUT Transactions . . . 784
27.1.4.3 Device and Interface Descriptors . . . 785
27.1.4.3.1 Device Descriptors . . . 786
27.1.4.3.2 Configuration Descriptors . . . 788
27.1.4.3.3 Interface Descriptors . . . 789
27.1.4.3.4 Endpoint Descriptors . . . 789
27.1.4.3.5 String Descriptors . . . 791
27.1.4.4 USB Classes . . . 791
27.2 STM32 USB Device Library . . . 793
27.2.1 Understanding Generated Code . . . 794
27.2.2 USB Initialization Sequence . . . 798
27.2.3 USB Enumeration Sequence . . . 801
27.2.4 The USB CDC Class . . . 804
27.2.4.1 USB CDC Descriptors . . . 805
27.2.4.2 USB CDC Class Initialization . . . 809
27.2.4.3 USB CDC Class Operations . . . 810
27.3 Building Custom USB Devices . . . 815
27.3.1 The USB HID Class . . . 817
27.3.1.1 USB HID Descriptors . . . 818
27.3.1.2 Overview of the *Report Descriptor* . . . 820
27.3.1.3 USB HID Class-Specific Requests . . . 824
27.3.2 Building a Vendor-Specific USB HID Device . . . 825
27.4 Debugging USB Devices . . . 837
27.4.1 Software Sniffers and Analyzers . . . 837
27.4.2 USB Hardware Analyzers . . . 837

27.5 Optimizing the STM32 USB Device Library . . . 838
27.6 Going to the Market . . . 839

28. Getting Started with a New Design . . . 842
28.1 Hardware Design . . . 842
28.1.1 PCB Layer Stack-Up . . . 843
28.1.2 MCU Package . . . 844
28.1.3 Decoupling of Power-Supply Pins . . . 845
28.1.4 Clocks . . . 847
28.1.5 Filtering of RESET Pin . . . 848
28.1.6 Debug Port . . . 848
28.1.7 Boot Mode . . . 850
28.1.8 Pay attention to "pin-to-pin" Compatibility... . . . 851
28.1.9 ...And to Selecting the Right Peripherals . . . 852
28.1.10 The Role of CubeMX During the Board Design Stage . . . 852
28.1.11 Board Layout Strategies . . . 856
28.2 Software Design . . . 857
28.2.1 Generating the binary image for production . . . 857

Appendix . . . 860

A. Miscellaneous HAL functions and STM32 features . . . 861
Force MCU reset from the firmware . . . 861
STM32 96-bit Unique CPU ID . . . 861

B. Troubleshooting Guide . . . 863
STM32CubeIDE Issues . . . 863
Debugging Continuously Breaks at Every Instruction During a Debug Session . . . 863
The Step-by-Step Debugging is Really Slow . . . 863
The Firmware Works Only Under a Debug Session . . . 864
STM32 Related Issues . . . 864
The Microcontroller Does Not Boot Correctly . . . 864
It is Not Possible to Flash or to Debug the MCU . . . 866

C. Nucleo pin-out . . . 868
Nucleo-G474RE . . . 869
Arduino compatible headers . . . 869
Morpho headers . . . 869
Nucleo-F446RE . . . 870
Arduino compatible headers . . . 870
Morpho headers . . . 870
Nucleo-F401RE . . . 871
Arduino compatible headers . . . 871

Morpho headers . . . 871
Nucleo-F303RE . . . 872
Arduino compatible headers . . . 872
Morpho headers . . . 872
Nucleo-F103RB . . . 873
Arduino compatible headers . . . 873
Morpho headers . . . 873
Nucleo-F072RB . . . 874
Arduino compatible headers . . . 874
Morpho headers . . . 874
Nucleo-L476RG . . . 875
Arduino compatible headers . . . 875
Morpho headers . . . 875
Nucleo-L152RE . . . 876
Arduino compatible headers . . . 876
Morpho headers . . . 876
Nucleo-L073R8 . . . 877
Arduino compatible headers . . . 877
Morpho headers . . . 877

D. Differences with the 1st edition . . . 878
Chapter 1 . . . 878
Chapter 2 . . . 878
Chapter 3 and 4 . . . 878
Chapter 5 . . . 878
Chapter 6 . . . 879
Chapter 7 . . . 879
Chapter 8 . . . 879
Chapter 9 . . . 879
Chapter 10 . . . 879
Chapter 11 . . . 879
Chapter 12-22 . . . 880
Chapter 23 . . . 880
Chapter 24 . . . 880
Chapter 25-26 . . . 880
Chapter 27 . . . 880
Chapter 28 . . . 880

Preface

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

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

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

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

Even in the second edition, the book is divided in three parts: an introductory part showing how to setup the STM32CubeIDE and how to work with it; a part that introduces the basics of STM32 programming and the main aspects of the official HAL (Hardware Abstraction Layer); 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, advanced peripherals like the USB.

However, this book does not aim to replace official datasheets from ST Microelectronics. A datasheet is still the main reference about electronic devices, and it is impossible (as well as making little sense) to arrange the content of tens of datasheets in a book. You have to consider that the official datasheet of the one of latest - and not the most complex of the portfolio - STM32G4 MCU alone is almost three thousand pages! 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.

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

So, the question is: "Is Arduino good for professional applications or for those wanting to develop the 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 a development environment based on Eclipse and GCC, you will not have to invest a fortune to start developing STM32 applications. Moreover, if you are building a cost sensitive device, where each PCB square inch makes a difference for you, consider that the STM32F0 value line is also known as the *32-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/
[b] https://www.arduino.cc/en/Main/Products
[c] https://processing.org/

How to Integrate This Book?

This book does not aim to be a 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 Yiu[1] is the best source for every Cortex-M developer.

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

How Is the Book Organized?

The book is divided in twenty-eight 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 the STM32CubeIDE 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.

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

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

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

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

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

Chapter 10 introduces the clock tree of an STM32 microcontroller, showing 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 IWDT and WWDT timers, and it briefly introduces their role and how to use the related CubeHAL modules to program them.

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

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

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

Chapter 21 introduces 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 can upgrade the on-board firmware through the USART peripheral. The bootloader uses the AES algorithm to encrypt the firmware.

Chapter 23 is dedicated to the FreeRTOS Real-Time Operating System. It introduces the reader to the most relevant concepts underlying an RTOS and shows how to use the main FreeRTOS functionalities (like threads, semaphores, mutexes, and so on) using the CMSIS-RTOS v2 layer developed by ST on top of the FreeRTOS API. Moreover, some advanced techniques, like the *tickless mode* in low-power design and the handling of concurrency with C stdlib 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, all STM32CubeIDE advanced debugging tools are presented, such as watchpoints, expressions and SWV-related tools. Finally, a brief introduction to SEGGER J-LINK professional debuggers is given, and 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 introduces to one of the widespread communication protocols: the USB 2.0. The Chapter will guide the reader to the fundamentals of USB specification, both from the hardware and the communication protocol point-of-view. Moreover, the STM32 USB Device Stack is deeply explained, with practical examples about USB-CDC and USB-HID classes.

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

F4

During the book you will find some horizontal rulers with “badges”, like the one above. This means that the instructions in that part of the book are specific 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 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 a warning box. The text contained explains important aspects or gives important instructions. It is strongly recommended to read the text carefully and follow the instructions.



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



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



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



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

Differences With the First Edition

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

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

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

About the Author

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

I began my career in informatics when I was only a young boy with an 80286 PC but, unlike all those who started programming in BASIC, I decided to learn a quite uncommon language: Clipper. Clipper was a language mostly used to write software for banks, and a lot of people suggested that I should start with this programming language (uh?!?). When visual environments, like Windows 3.1, started to 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 of lines of code in Python, ranging from web systems to embedded devices. I think Python is an expressive and productive programming language, and it is always my first choice when I have to code something.

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

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

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

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

Errata and Suggestions

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

On the other 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/[2]

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/[3].

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

How to Help the Author

If you want to help me, you may consider to:

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

Copyright Disclaimer

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

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

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

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

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

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

During the rest of the book, I will mention the copyright of tools and libraries I will introduce. If I have 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[7])

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

Acknowledgments to the First Edition

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

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

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

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

Regards,
Carmine I.D. Noviello

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

I Introduction

1. Introduction to STM32 MCU Portfolio

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

1.1 Introduction to ARM Based Processors

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

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

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

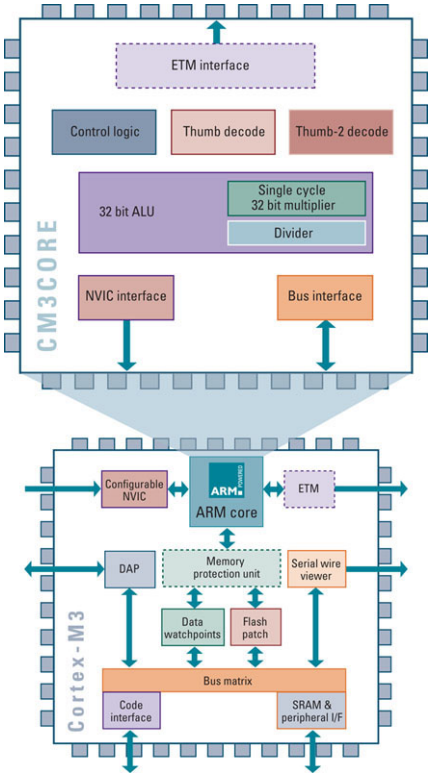


Figure 1.1: The relation between a Cortex-M3 core and a Cortex-M3 based MCU

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

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

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

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

1.1.1 Cortex and Cortex-M Based Processors

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

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

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

1.1.1.1 Core Registers

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

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

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

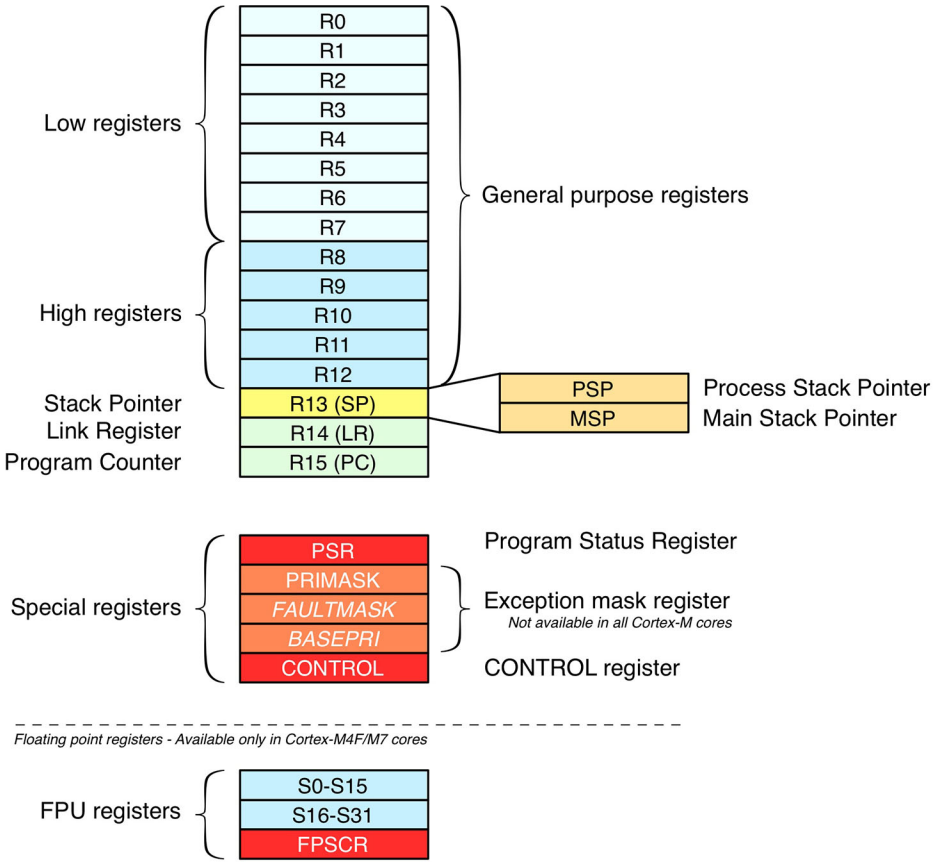


Figure 1.2: ARM Cortex-M core registers

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

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

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

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

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

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

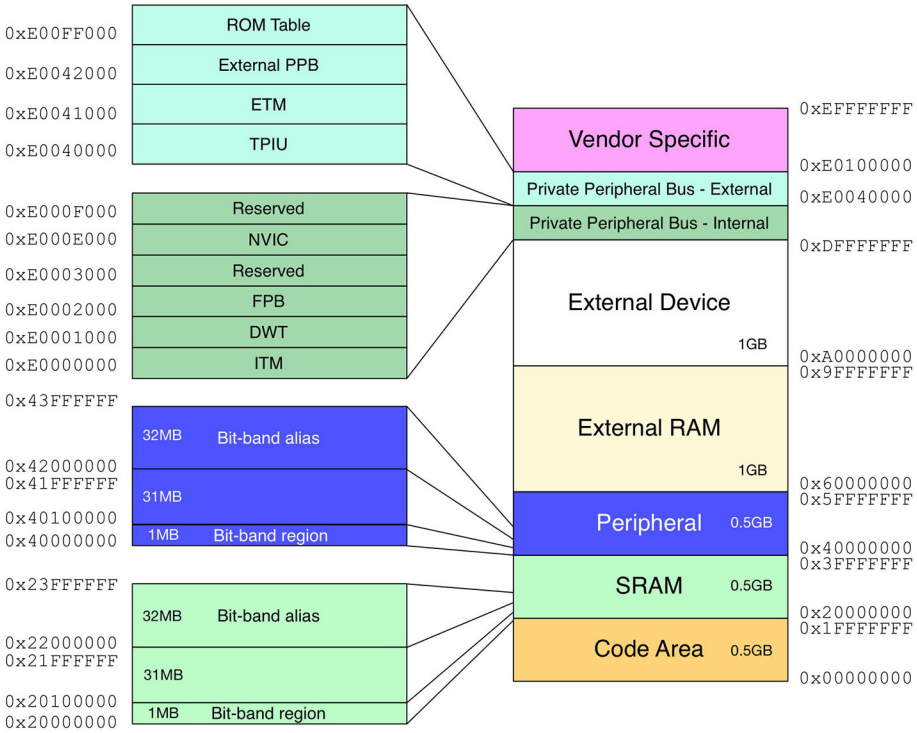


Figure 1.3: Cortex-M fixed memory address space

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

1.1.1.2 Memory Map

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

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

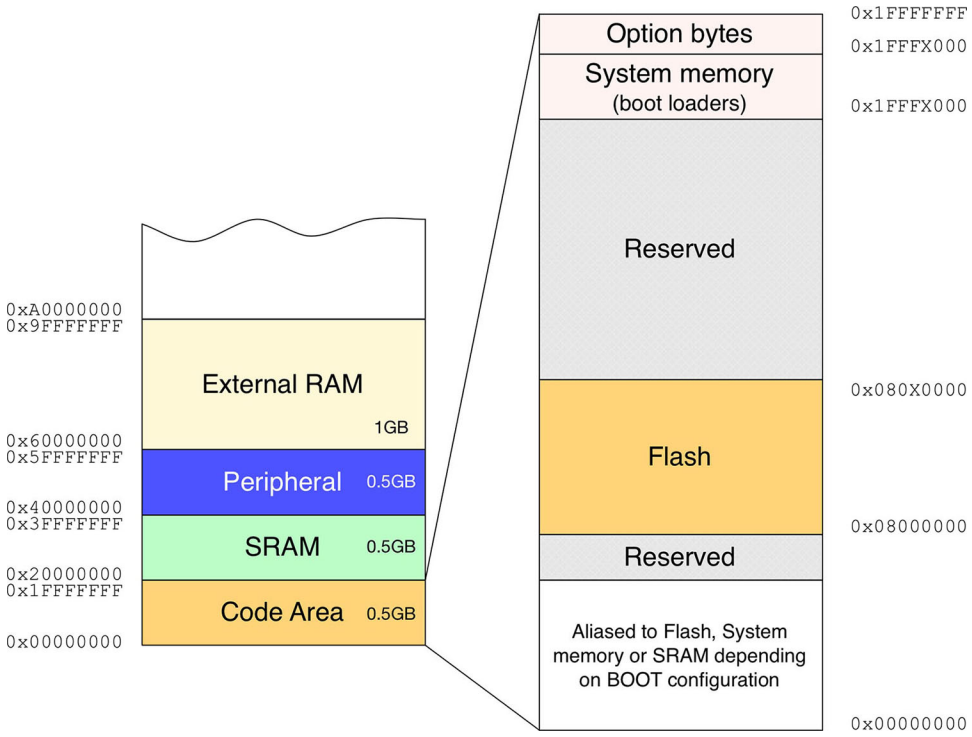


Figure 1.4: Memory layout of Code Area on STM32 MCUs

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

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

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

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

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

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

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

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

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

1.1.1.3 Bit-Banding

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

```
...
uint8_t temp = 0;

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

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

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

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

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

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

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

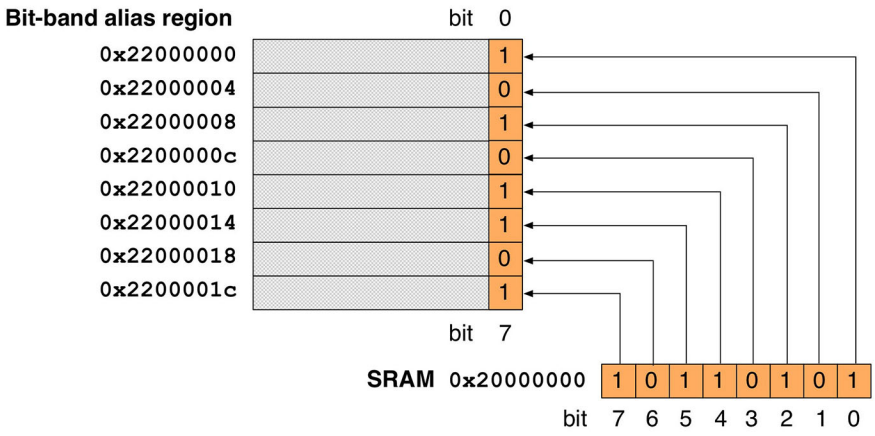


Figure 1.5: Memory mapping of SRAM address `0x2000 0000` in bit-banding region (first 8 of 32 bits shown)

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

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

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

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

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

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

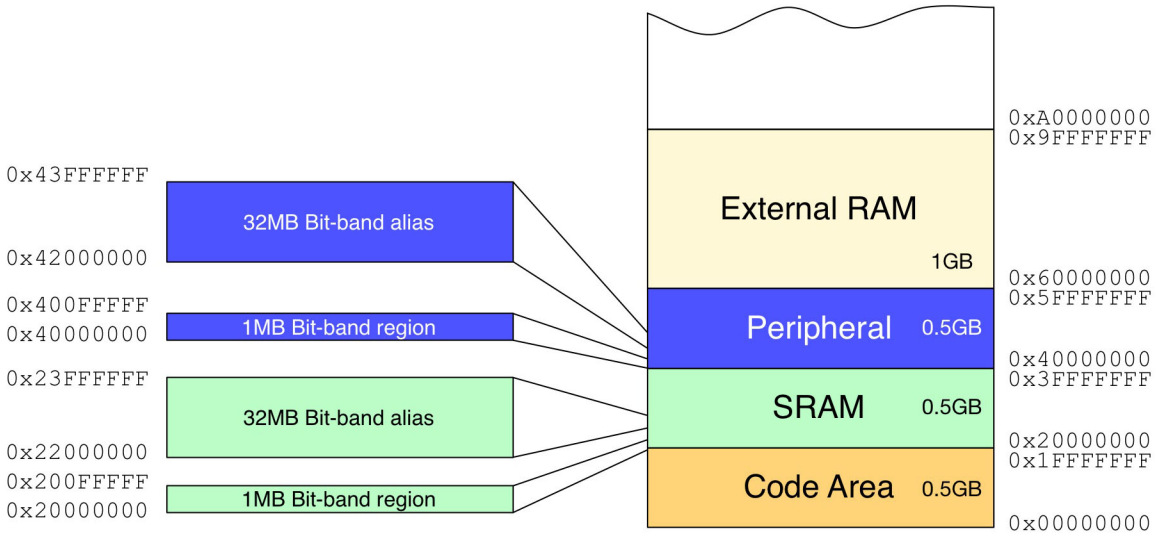


Figure 1.6: Memory map and bit-banding regions

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

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

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

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

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

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

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

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

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

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

1.1.1.4 Thumb-2 and Memory Alignment

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

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

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

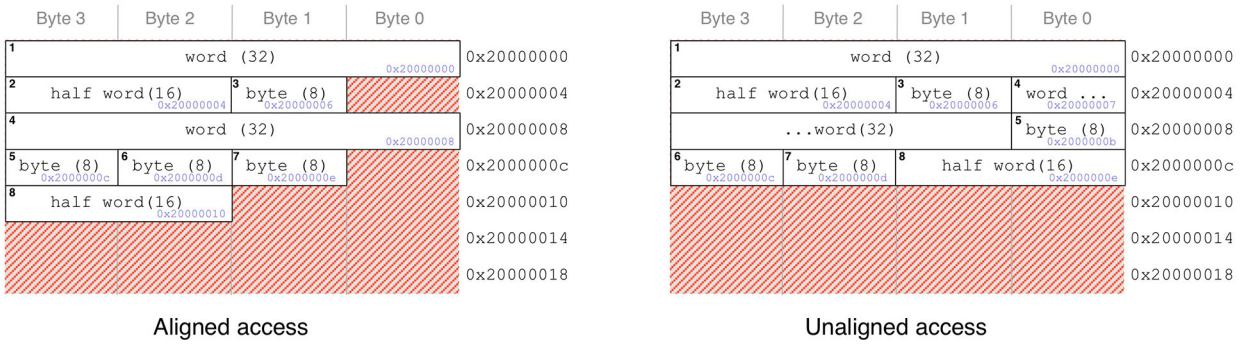


Figure 1.7: Difference between aligned and unaligned memory access

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

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

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

```
STR R2, 0x20000002
```

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

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

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

- STR, STRT
- STRH, STRHT

1.1.1.5 Pipeline

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

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

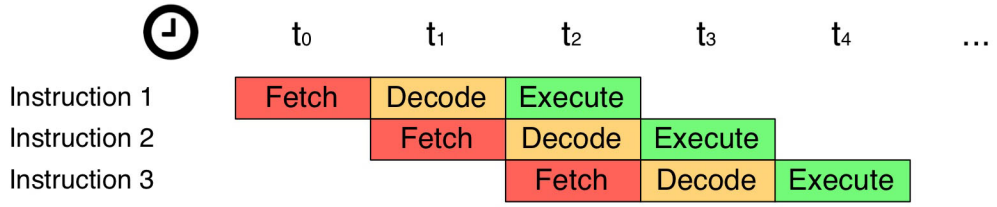


Figure 1.8: Three stage instruction pipeline

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

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

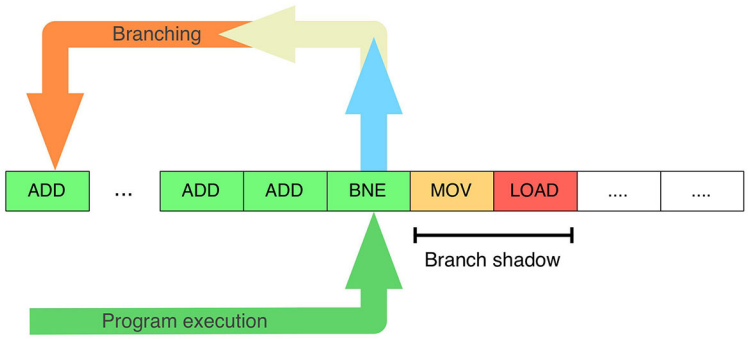


Figure 1.9: Branching in program execution related to pipelining

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

1.1.1.6 Interrupts and Exceptions Handling

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

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

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

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

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

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

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

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

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

Table 1.1: Cortex-M exception types

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

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

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

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

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

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

1.1.1.7 SysTimer

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

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

1.1.1.8 Power Modes

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

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

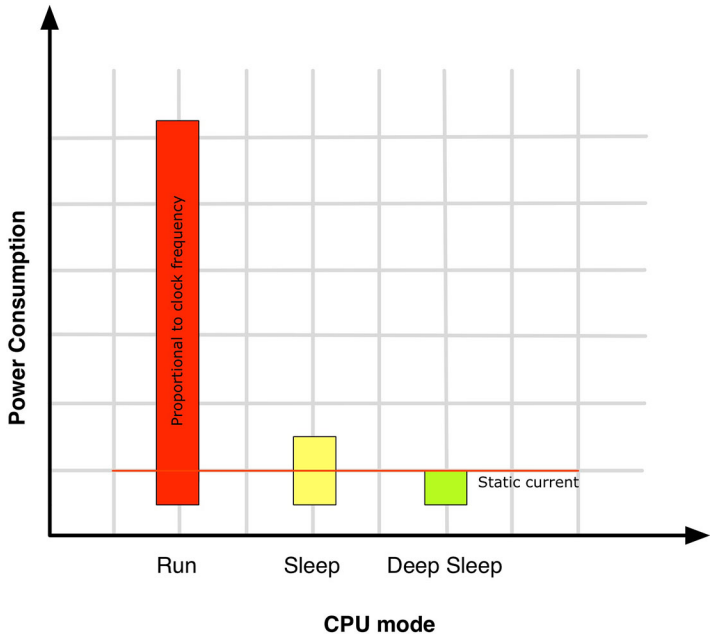


Figure 1.10: Cortex-M power consumption at different power modes

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

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

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

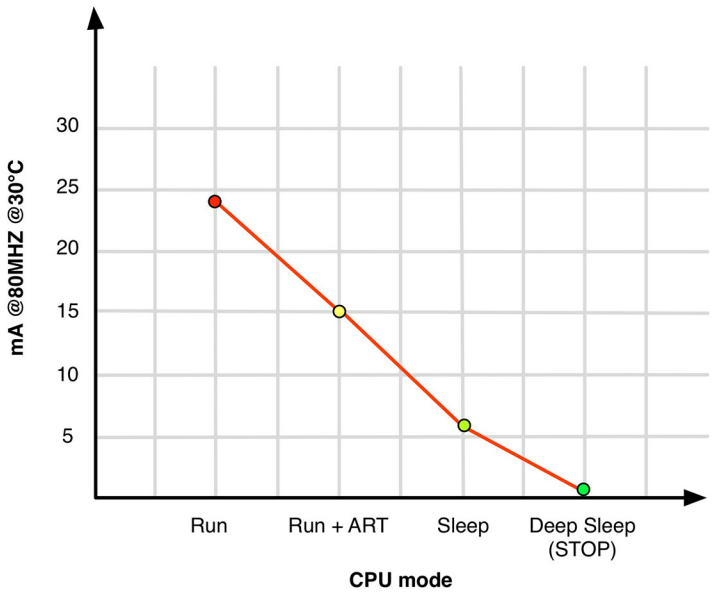


Figure 1.11: STM32F2 power consumption at different power modes

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

1.1.1.9 TrustZone™

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

[11]Source ST AN3430

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

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

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

1.1.1.10 CMSIS

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

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

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

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

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

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

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

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

Optional in ARM specification

Table 1.2: ARM Cortex-M instruction variations

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

Optional in ARM specification

Table 1.3: ARM Cortex-M optional components

1.2 Introduction to STM32 Microcontrollers

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

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

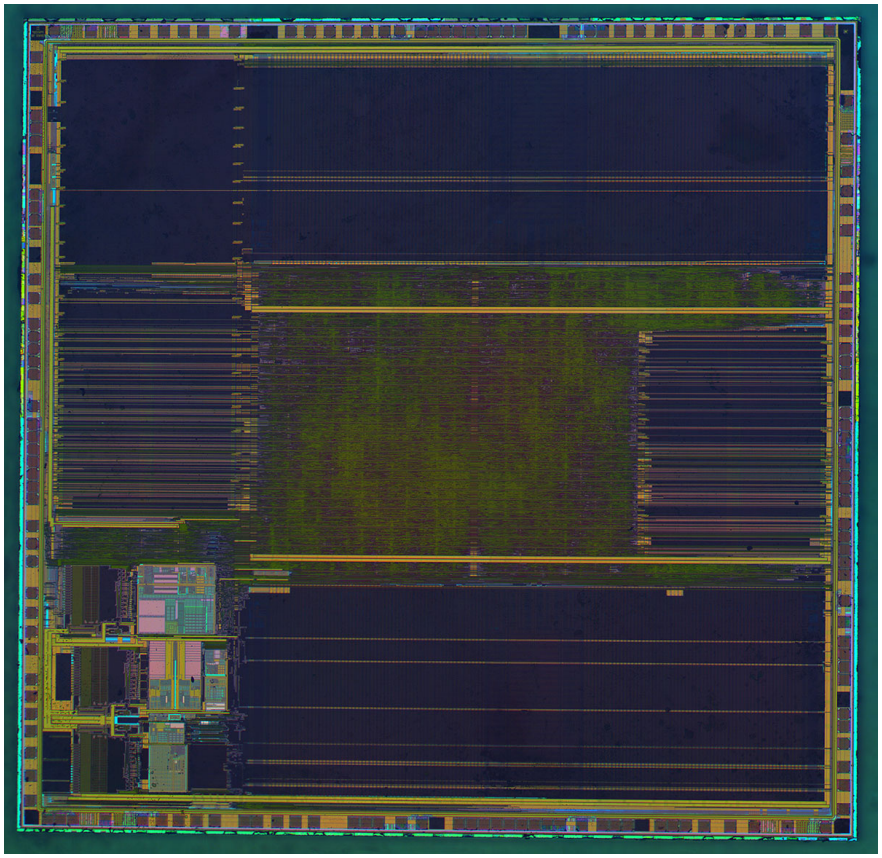


Figure 1.12: Internal die of an STM32F103 MCU

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

1.2.1 Advantages of the STM32 Portfolio....

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

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

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

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

1.2.2 ....And Its Drawbacks

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

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

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

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

1.3 A Quick Look at the STM32 Subfamilies



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

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

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

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

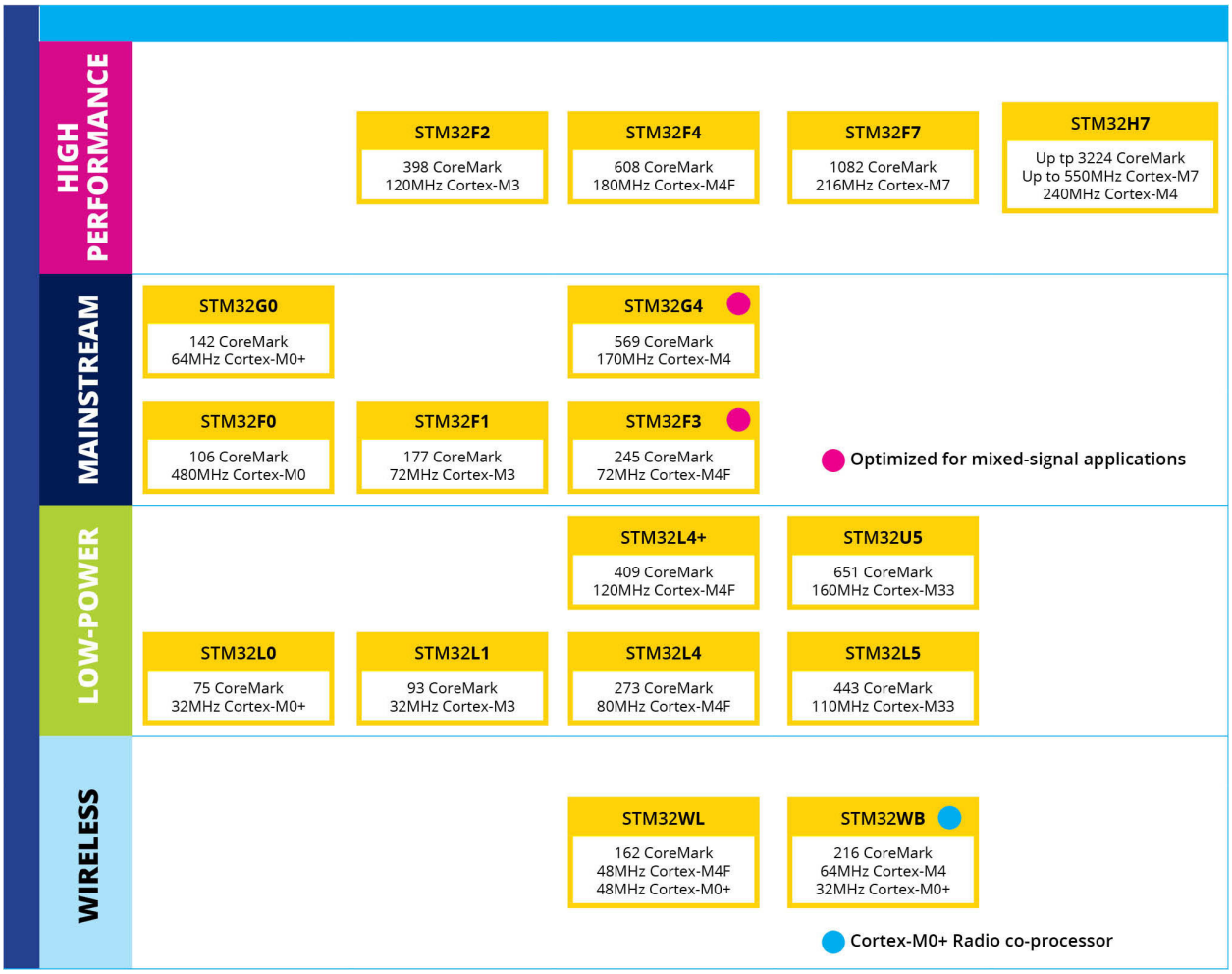


Figure 1.13: STM32 portfolio

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

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

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

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

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

1.3.1 F0

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

Table 1.4: STM32F0 features

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

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

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

1.3.2 F1

Common to all STM32F1 MAINSTREAM

- -40 to +105° range
- USART, SPI, I²C
- 16/32-bit timers
- Temperature sensor
- Up to 3x12-bit DAC
- Dual 12-bit ADC
- Up to 12-channels DMA
- Low voltage 2.0 to 3.6V
- 5V tolerant I/Os
- Up to 80 fast I/Os
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 F1

Core: Cortex-M3
Instruction set: *Thumb*, *Thumb-2*, Saturated Math
Internal RC oscillators: HSI=8MHz, LSI=40KHz
External clocks: HSE=4-24Mhz(F100), 4-16Mhz(F101/2/3), 3-25MHz (F105/7), LSE=32.768 - 1000 KHz
Low power modes: Sleep, Stop and Standby
Year of commercialization: 2007
Available Packages: LFBGA(100,144), LQFP(48,64,100,144), UFBGA(100), UFQFPN(36,48), WLCSP(64)

Product Line	FLASH (KB)	RAM (KB)	F_{CPU} (MHz)	USB 2.0 FS	USB 2.0 OTG FS	FSMC	3-phase MC Timer	I²S	CAN 2.0B	SDIO	Ethernet
STM32F100 Value Line	16 to 512	4 to 32	24			•	•				
STM32F101 Access Line	16 to 1024	4 to 80	36			•					
STM32F102 USB Line	16 to 128	4 to 16	48	•							
STM32F103 Performance Line	16 to 1024	6 to 96	72	•		•	•	•	•	•	
STM32F105 STM32F107 Connectivity Line	64 to 256	64	72		•	•	•	•	•		•

Table 1.5: STM32F1 features

The STM32F1 series was the first ARM based MCU from ST. Introduced in the market in 2007, it is still the most widespread MCU from the STM32 portfolio. Plenty of development boards are available on the market, produced by ST and other vendors, and you will find tons of examples on the web for F1 microcontrollers. If you are new to the STM32 world, probably the F1 line is the best choice to start working with to learn this platform.
The F1-series has evolved over time by increasing speed, size of internal memory, variety of peripherals. There are five F1 lines: *Connectivity* (STM32F105/107), *Performance* (STM32F103), *USB Access* (STM32F102), *Access* (STM32F101), *Value* (STM32F100).
The most important features of this series are:

- **Core**:
  - ARM Cortex-M3 core at a maximum clock rate ranging from 24 to 72 MHz.
- **Memory**:
  - Static RAM from 4 to 96 KB.
  - Flash from 16 to 256 KB.
  - Each chip has a factory-programmed 96-bit unique device identifier number.

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

- **Peripherals**:
  - Each F1-series device features a range of peripherals which vary from line to line (see **Table 1.5** for a quick overview).
- Oscillator source consists of internal RC (8 MHz, 40 kHz), optional external HSE (4-24MHz(F100), 4-16MHz(F101/2/3), 3-25MHz (F105/7), LSE (32.768 - 1000 kHz)).
- IC packages: LFBGA, LQFP, UFBGA, UFQFPN, WLCSP (see **Table 1.5** for more about this).
- Operating voltage range is 2.0V to 3.6V
- Multiple connectivity options, including Ethernet, CAN and USB 2.0 OTG.

1.3.3 F2

HIGH PERFORMANCE — Common to all STM32F2:
- -40 to +105° range
- USART, SPI, I²C
- 16/32-bit timers
- Temperature sensor
- 12-bit DAC up to 24 channels
- Dual 12-bit ADC
- 16-channels DMA
- Low voltage 1.8 to 3.6V
- 5V tolerant I/Os
- Up to 136 fast I/Os
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 F2

Core: Cortex-M3 with ART™ Accelerator
Instruction set: *Thumb*, *Thumb-2*, Saturated Math
Internal RC oscillators: HSI=16MHz, LSI=32KHz
External clocks: HSE=1 - 26MHz, LSE=32.768 - 1000 KHz
Maximum Core Frequency: 120MHz
Low power modes: Sleep, Stop and Standby
Year of commercialization: 2010
Available Packages: LQFP(64,100,144,176), UFBGA(176), WLCSP(66)

Product Line	FLASH (KB)	RAM (KB)	Hardware Crypto/Hash	USB 2.0 OTG FS	FSMC	Camera I/F	SDIO	Ethernet
STM32F205	128 to 1024	Up to 128		•	•		•	
STM32F215			•					
STM32F207	512 to 1024	Up to 128		•	•	•	•	•
STM32F217			•					

Table 1.6: STM32F2 features

The STM32F2 series of STM32 microcontrollers is the cost-effective solution in the **High-performance** segment. It is the most recent and fastest Cortex-M3 based MCU in the STM32 portfolio, with exclusive ART™ Accelerator from ST. The F2 is pin-to-pin compatible with the STM32 F4-series.

The most important features of this series are:

- **Core**:
  - ARM Cortex-M3 core at a maximum clock rate of 120 MHz.
- **Memory**:
  - Static RAM from 64 to 128 KB.
    - 4 KB battery-backed, 80 bytes battery-backed with tamper-detection erase.
  - Flash from 128 to 1024 KB.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each F2-series device features a range of peripherals which vary from line to line (see **Table 1.6** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 32 kHz), optional external HSE (1 to 26 MHz), LSE (32.768 to 1000 kHz).
- IC packages: BGA, LQFP, UFBGA, WLCSP (see **Table 1.6** for more about this).
- Operating voltage range is 1.8V to 3.6V.

1.3.4 F3

MAINSTREAM

Common to all STM32F3:
- -40 to +105° range
- USART, SPI, I²C
- 16/32-bit timers
- Temperature sensor
- Up to 3x12-bit DAC
- 12-bit ADC
- 7-channels DMA
- Low voltage 2.0 to 3.6V
- 5V tolerant I/Os
- Up to 50 fast I/Os
- Reset POR/PDR
- 2xWDT
- SDIO
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- CAN 2.0
- Unique ID

STM32 F3

Core: Cortex-M4F
Instruction set: *Thumb*, *Thumb-2*, Saturated Math, DSP, FPU
Internal RC oscillators: HSI=8MHz, LSI=40KHz
External clocks: HSE=4-32Mhz, LSE=32.768 - 1000 KHz
Maximum Core Frequency: 72MHz
Low power modes: Sleep, Stop and Standby
Year of commercialization: 2012
Available Packages: LQFP(32,48,64,100,144), UFBGA(100), UFQFPN(32), WLCSP(49,66,100)

Product Line	FLASH (KB)	RAM (KB)	CCM SRAM	ADC 12-bit	ADC 16-bit	12-bit DAC	Fast Comparator	OpAmp (PGA)	Advanced 16-bit timer	Hig resolution Timer
STM32F301 - Access	32 to 64	16		Up to 2		1	3	1	1	
STM32F302 - USB & CAN	32 to 512	16 to 64		Up to 2		1	Up to 4	Up to 2	1	
STM32F303 - Performance	32 to 512	16 to 80	•	Up to 4		Up to 3	Up to 7	Up to 4	Up to 3	
STM32F3x4 Digital Power	32 to 512	16	•	2		3	2x Ultra fast	1	1	• 10 ch
STM32F373 Precision measurement	16 to 64	32		1	3	3	2			
STM32F3x8 1.8V ±8%	64 to 512	16 to 64	•	Up to 4		Up to 3	Up to 7	Up to 4	Up to 3	

Table 1.7: STM32F3 features

The STM32F3 it is based on the ARM Cortex-M4F core and it is one of the two families dedicated to mixed-signal application. It is designed to be almost pin-to-pin compatible with the STM32 F1-series, even if it does not offer the same variety of peripherals. STM32F3 was the MCU chosen by the developers of the BB-8 droid[16] toy by Sphero[17].

The distinguishing feature for this series is the presence of integrated analog peripherals leading to cost reduction at application level and simplifying application design, including:

- Ultra-fast comparators (25 ns).
- Op-amp with programmable gain.
- 12-bit DACs.
- Ultra-fast 12-bit ADCs with 5 MSPS (Million Samples Per Second) per channel (up to 18 MSPS in Interleaved mode).
- Precise 16-bit sigma-delta ADCs (21 channels).
- 144 MHz Advanced 16-bit pulse-width modulation timer (resolution < 7 ns) for control applications; high resolution timer (217 picoseconds), self-compensated vs power supply and temperature drift.

Another interesting feature of this series is the presence of a *Core Coupled Memory* (CCM), a specific memory architecture which couples some regions of memory to the CPU core, allowing *0-wait* states.

[16] http://cnet.co/1M2NyJS
[17] http://www.sphero.com/

This can be used to boost time-critical routines, improving performance by up to 40%. For example, OS routines for context switching can be stored in this area to speed up RTOS activities.
The most important features of this series are:



Figure 1.14: The BB-8 droid made with an STM32F3 MCU

- **Core**:
  - ARM Cortex-M4F core at a maximum clock rate of 72 MHz.
- **Memory**:
  - Static RAM from 16 to 80 KB general-purpose with hardware parity check.
    - 64 / 128 bytes battery-backed with tamper-detection erase.
  - Up to 8 KB Core Coupled Memory (CCM) with hardware parity check.
  - Flash from 32 to 512 KB.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each F3-series device features a range of peripherals which vary from line to line (see **Table 1.7** for a quick overview).
- Oscillators consist of internal RC (8 MHz, 40 kHz), optional external HSE (4 to 32 MHz), LSE (32.768 to 1000 kHz).
- IC packages: LQFP, UFBGA, UFQFPN, WLCSP (see **Table 1.7** for more about this). Operating voltage range is 1.8V ±8%. to 3.6V.

1.3.5 F4

Common to all STM32F4
- USART, SPI, I²C
- I²S + audio PLL
- 16/32-bit timers
- Temperature sensor
- Up to 2x12-bit DAC
- Up to 3x12-bit ADC
- 7-channels DMA
- Low voltage 1.7 to 3.6V
- 5V tolerant I/Os
- Up to 136 fast I/Os
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

HIGH PERFORMANCE

STM32 F4

Core: Cortex-M4F with ART™ Accelerator
Instruction set: *Thumb*, *Thumb-2*, Saturated Math, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI=32KHz
External clocks: HSE=4-26Mhz, LSE=32.768KHz
Low power modes: Sleep, Stop and Standby
Year of commercialization: 2011
Available Packages: LQFP(64,100,144,176,208), TFBGA(216), UFBGA(64,100,144,169,176), UFQFPN(48), WLCSP(36,49,64,81,90,143,168)

Product Line	FLASH (KB)	RAM (KB)	F_{CPU} (MHZ)	Ethernet I/F	Camera I/F	SDRAM I/F	SAI[3] I/F	Chrom-ART™	TFT Controller
				2xCAN		2xQuad SPI	SPDIF RX		
Advanced lines									
STM32F469	512 to 2048	384	180	• •	•	• •	•	•	•
STM32F429	512 to 2048	256	180	• •	•	•	•	•	•
STM32F427	1024 to 2048	256	180	• •	•	•	•	•	
Foundation lines									
STM32F446	256 to 512	128	180	•	•	• •	• •		
STM32F407	512 to 1024	192	168	• •	•				
STM32F405	512 to 1024	192	168						

Product Line	FLASH (KB)	RAM (KB)	F_{CPU} (MHZ)	Dynamic Efficiency	Run Current (µA/MHz)	STOP Current (µA)	QSPI	CAN 2.0B	USB 2.0 OTG FS
Access lines									
STM32F401	128 to 512	96	84	•	Down to 128	Down to 10			•
STM32F410	64 to 128	32	100	•	Down to 89	Down to 6			-
STM32F411	256 to 512	128	100	•	Down to 100	Down to 12			•
STM32F412	512 to 1024	256	100	•	Down to 112	Down to 18	•	•	•
STM32F413	1024 to 1536	320	100	•	Down to 115	Down to 18	•	•	•

Table 1.8: STM32F4 features

The STM32F4 series is the most widespread group of Cortex-M4F based MCUs in the *High-performance* segment. The F4-series is also the first STM32 series to have DSP and Floating Point SP instructions. The F4 is pin-to-pin compatible with the STM32 F2-series and adds higher clock speed, 64K CCM static RAM, full duplex I²S, improved real-time clock, and faster ADCs. The STM32F4-series is also targeted to multimedia applications, and some MCUs offer dedicated support for LCD-TFT and extendable memory with flexible memory controller (FMC) and a Dual-mode Quad-SPI interface.
The most important features of this series are:

- **Core**:
  - ARM Cortex-M4F core at a maximum clock ranging from 84 to 180 MHz.
- **Memory**:
  - Static RAM from 128 to 384 KB.
    * 4 KB battery-backed, 80 bytes battery-backed with tamper-detection erase.
  - 64 KB Core Coupled Memory (CCM).
  - Flash from 256 to 2048 KB.
  - Easily extendable memory range with flexible memory controller (FMC) and a Dual-mode Quad-SPI interface. The FMC runs at 90 MHz with a 32-bit parallel interface, and supporting Compact Flash, SRAM, PSRAM, NOR, NAND and SDRAM memories. The Dual-mode Quad-SPI running at 90 MHz enables cost-effective NOR Flash and supports memory-mapped mode.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each F4-series device features a range of peripherals which vary from line to line (see **Table 1.8** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 32 kHz), optional external HSE (4 to 26 MHz), LSE (32.768 to 1000 kHz).
- IC packages: BGA, LQFP, TFBGA, UFBGA, UFQFPN, WLCSP (see **Table 1.8** for more about this).
- Operating voltage range is 1.8V to 3.6V.

1.3.6 F7

HIGH PERFORMANCE

Common to all STM32F7

- Chrom-ART™ Accelerator
- USART, SPI, I²C
- I²S + audio PLL
- 2xSAI³
- SDIO
- 2xCAN
- 2xUSB OTG FS/HS
- HDMI-CEC
- Ethernet
- 16/32-bit timers
- Temperature sensor
- Up to 2x12-bit DAC
- Up to 3x12-bit ADC
- 16-channels DMA
- Low voltage 1.7 to 3.6V
- 5V tolerant I/Os
- Up to 164 fast I/Os
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 F7

Core: Cortex-M7 with ART™ Accelerator
Instruction set: *Thumb*, *Thumb-2*, Saturated Math, DSP, FPU, SIMD
Internal RC oscillators: HSI=16MHz, LSI=40KHz
External clocks: HSE=4-26Mhz, LSE=32.768KHz
Maximum Core Frequency: 216MHz
Low power modes: Sleep, Stop and Standby
Year of commercialization: 2015
Available Packages: LQFP(64,100,144,176,208), TFBGA(100,216), UFBGA(144,176), WLCSP(100,143,180)

Product Line	FLASH (KB)	RAM (KB)	L1 Cache (I/D)	FPU	Ethernet	FMC	CAN	JPEG Codec	TFT Controller
Advanced lines									
STM32F7x9 STM32F7x8	1024 to 2048	512	16K+16K	Double Precision	•	•	3	•	•
STM32F7x7	1024 to 2048	512	16K+16K	Double Precision	•	•	3	•	•
STM32F7x6	512 to 1024	320	4k+4k	Single Precision	•	•	2		•
STM32F765	1024 to 2048	512	16K+16K	Double Precision	•	•	3		
STM32F745	512 to 1024	320	4k+4k	Single Precision	•	•	2		
Foundation lines									
Product Line	**FLASH (KB)**	**RAM (KB)**	**L1 Cache (I/D)**	**FPU**	**PC-ROP**	**FMC**	**CAN**	**USB PHY**	**TFT Controller**
STM32F7x3	512 to 1024	256	8k+8k	Single Precision	•	•	1	•	
STM32F7x2	512 to 1024	256	8k+8k	Single Precision	•	•	1		
Value lines									
STM32F730	64	256	8k+8k	Single Precision	•	•	1	•	
STM32F750	64	320	4k+4k	Single Precision		•	2		•

Table 1.9: STM32F7 features

The STM32F7 series is the latest ultra-performance MCU in the *High-performance* segment, and it was the first Cortex-M7 based MCU introduced on the market. Thanks to ST's ART™ Accelerator as well as an L1 cache, STM32F7 devices deliver the maximum theoretical performance of the Cortex-M7 regardless of code being executed from embedded flash or external memory: 1082 CoreMark/462 DMIPS at 216 MHz. STM32F7 is clearly targeted to heavy multimedia embedded applications. Thanks to the STM32 longevity program (10 years) it is possible to develop powerful embedded applications without worrying about the MCU availability on the market in the far future. Cortex-M7 is backwards compatible with the Cortex-M4 instruction set, and STM32F7 series is pin-to-pin compatible with the STM32F4 series.

The most important features of this series are:

- **Core**:
  - ARM Cortex-M7 core at a maximum clock of 216 MHz.

- **Memory**:
  - Static RAM with scattered architecture:
    * Up to 512 Kbytes of universal data memory, including up to 128 Kbytes of Tightly-Coupled Memory for Data (DTCM) for time critical data handling (stack, heap…)
    * 16 Kbytes of Tightly-Coupled Memory for Instructions (ITCM) for time-critical routines
    * 4 Kbytes of backup SRAM to keep data in the lowest power modes
  - L1 cache (I/D up to 16 KB + 16 KB).
  - Flash from 512 to 2048 KB.
  - Easily extendable memory range with flexible memory controller (FMC) and a Dual-mode Quad-SPI interface. The FMC runs at 90 MHz with a 32-bit parallel interface, and supporting Compact Flash, SRAM, PSRAM, NOR, NAND and SDRAM memories. The Dual-mode Quad-SPI running at 90 MHz enables cost-effective NOR Flash and supports memory-mapped mode.
  - Two general-purpose DMA controllers and dedicated DMA controllers for Ethernet (on some variants), high-speed USB On-The-Go interfaces and the Chrom-ART™ graphic accelerator (on some variants)
  - Peripheral speed is independent from CPU speed (dual clock support) allowing system clock changes without any impact on peripheral operations
  - Protected code execution feature (PC-ROP) on some variants
  - On-chip USB high-speed PHY on some variants
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each F7-series device features a range of peripherals which vary from line to line (see **Table 1.9** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 32 kHz), optional external HSE (4 to 26 MHz), LSE (32.768 to 1000 kHz).
- IC packages: LQFP, TFBGA, UFBGA, WLCSP (see **Table 1.9** for more about this).
- Operating voltage range is 1.7V to 3.6V.

1.3.7 H7

HIGH PERFORMANCE

Common to all STM32H7

- Chrom-ART™ Accelerator
- L1 cache: 16K+16K data and instruction cache
- Double-precision FPU
- 4xDMA
- 2xUSB2.0 OTG FS/HS
- USART, SPI, I²C
- I²S + audio PLL
- 2xSAI³
- 2xSDMMC
- FMC, Dual Model Quad-SPI
- 2xCAN (1xFD and 1xTT)
- Camera-IF
- 16/32-bit timers
- Temperature sensor
- Up to 2x12-bit DAC
- Up to 3x12-bit ADC
- 16-channels DMA
- Low voltage 1.62 to 3.6V
- 5V tolerant I/Os
- Up to 164 fast I/Os
- Crypto/HASH
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 H7

Single-Core: Cortex-M7 with ART™ Accelerator
Dual-Core: Cortex-M7 with ART™ Accelerator + Cortex-M4F
Instruction set: *Thumb*, *Thumb-2*, Saturated Math, DSP, FPU, SIMD, SP-FPU, DP-FPU
Internal RC oscillators: HSI=16MHz, LSI=40KHz
External clocks: HSE=4-26Mhz, LSE=32.768KHz
Maximum Core Frequency for Cortex-M7 core: 550MHz in Single Core / 480MHz in Dual Core MCU
Maximum Core Frequency for Cortex-M4 core: 240MHz
Low power modes: Sleep, Stop and Standby
Year of commercialization: 2017
Available Packages: LQFP(100,144,176,208), TFBGA(240, 100), UFBGA(169, 176), WLCSP(150)

Product Line	FLASH (KB)	RAM (KB)	Ethernet I/F	Camera I/F	Power Supply	JPEG Codec	TFT Controller
Dual-core lines							
STM32F747/757	Up to 2048	1024	•	•	SMPS + LDO	•	MIPI-DSI
STM32F745/755	Up to 2048	1024	•	•	SMPS + LDO	•	•
Single-core lines							
STM32H7A3/7B3	Up to 2048	1024		•	SMPS + LDO	•	•
STM32H743/753	Up to 2048	1024	•	•	LDO	•	•
STM32H742	Up to 2048	1024	•	•	LDO		
STM32H725/735	Up to 1024	564	•	•	SMPS + LDO		•
STM32H723/733	Up to 1024	564	•	•	LDO		•
Value lines							
STM32H7B0	128	1380			SMPS + LDO	•	•
STM32H750	128	1024	•	•	LDO	•	•
STM32H730	128	564	•	•	SMPS + LDO		•

Table 1.10: STM32H7 features

ST released on the market in 2017 this family of single-core Cortex-M7 and dual-core Cortex-M7 and Cortex-M4 microcontrollers, made with a 40nm process able to run up to 550MHz for single-core MCUs. It also provides up to 1MB SRAM with the same scattered architecture found in the STM32F7-series.

The most important features of this series are:

- **Core**:
  - ARM Cortex-M7 core at a maximum clock of 550 MHz for single-core MCUs and ARM Cortex-M7 at a maximum clock of 480 MHz plus ARM Cortex-M4 cores at a maximum clock of 240 MHz for dual-core MCUs.
- **Memory**:
  - Static RAM up to 1024 KB with scattered architecture:

    * 192 Kbytes of TCM RAM (including 64 Kbytes of ITCM RAM and 128 Kbytes of DTCM RAM for time-critical routines and data), 512 Kbytes, 288 Kbytes and 64 Kbytes of user SRAM, and 4 Kbytes of SRAM in backup domain to keep data in the lowest power modes
  - L1 cache (I/D up to 16 KB + 16 KB).
  - Flash from 512 to 2048 KB.
  - Easily extendable memory range with flexible memory controller (FMC) and a Dual-mode Quad-SPI interface. The FMC runs at 90 MHz with a 32-bit parallel interface, and supporting Compact Flash, SRAM, PSRAM, NOR, NAND and SDRAM memories. The Dual-mode Quad-SPI running at 90 MHz enables cost-effective NOR Flash and supports memory-mapped mode.
  - A high-speed master direct memory access (MDMA) controller, two dual-port DMAs with FIFO and request router capabilities for optimal peripheral management, and one additional DMA
  - Chrom-ART Accelerator™ for efficient 2D image copy and double-precision FPU are also part of the acceleration features available. A MIPI-DSI physical layer to drive high-resolution displays is also available on certain devices.
  - Thanks to dual-clock support, the speed of peripherals is independent from the CPU's speed, allowing system clock changes without any impact on peripheral operations
  - Even more peripherals, such as four serial audio interfaces (SAI) with SPDIF output support, three full-duplex I^2S interfaces, a SPDIF input interface supporting four inputs, two USB OTG with dedicated power supply and Dual-mode Quad-SPI interface, two FD-CAN controllers, a high-resolution timer, a TFT-LCD controller, a JPEG codec, two SDIO interfaces and many other analog peripherals including three fast 16-bit ADCs, two comparators and two operational amplifiers.
  - The Secure Firmware Install and Secure Boot – Secure Firmware Upgrade allow the user to perform the initial programming of the device in a secured manner, or to perform a secured field upgrade of the code.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Several new peripherals such as 14-bit ADC and a new SAI.
- IC packages: LQFP, TFBGA, UFBGA, WLCSP
- Pin-to-pin compatible with the STM32F7-series.

This book does not cover the STM32H7 series.

1.3.8 L0

Common to all STM32L0 LOW POWER

- -40 +125°C range
- Low voltage 1.65 to 3.6V
- Dynamic Voltage Scaling
- 5 clock sources
- USART, SPI, I²C
- 16-bit timers
- Temperature sensor
- DMA
- 5V tolerant I/Os
- Up to 51 fast I/Os
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 L0

Core: Cortex-M0+ with MPU
Instruction set: *Thumb* subset, *Thumb-2* subset
Internal RC oscillators: HSI=16MHz, LSI=37KHz
External clocks: HSE=1-24Mhz, LSE=32.768KHz
Maximum Core Frequency: 32MHz
Low power modes: Low-power run, Sleep, Low-power sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2014
Available Packages: LQFP(32,48,64,100), TFBGA(64), TSSOP(14,20),UFBGA(100), UFQFPN(20,28,32), WLCSP(25,36,49)

Product Line	FLASH (KB)	RAM (KB)	EEPROM (KB)	12-bit ADC	Low Power UART	Low Power 16-bit timer	12-bit DAC	Touch Sense	True RNG	USB 2.0 Crystal-less	Segment LCD Driver
STM32L0x0 Value line	Up to 128	Up to 20	Up to 512	•	•	•					
STM32L0x1 Access	Up to 192	Up to 20	Up to 6144	•	•	•					
STM32L0x2 USB	Up to 192	Up to 20	Up to 6144	•	•	•	•	•	•	•	
STM32L0x3 USB & LCD	Up to 192	Up to 20	Up to 6144	•	•	•	•	•	•	•	Up to 8x28 or 4x32

Table 1.11: STM32L0 features

The STM32L0 series is the cost-effective solution of the *Ultra Low-Power* segment. The combination of an ARM Cortex-M0+ core and ultra-low-power features makes STM32L0 the best fit for applications operating on battery or powered by energy harvesting, offering the world's lowest power consumption at 125°C. The STM32L0 offers dynamic voltage scaling, an ultra-low-power clock oscillator, LCD interface, comparator, DAC and hardware encryption. Current consumption reference values:

- Dynamic run mode: down to 49µA/MHz.
- Ultra-low-power mode + full RAM + low power timer: 340 nA (16 wakeup lines).
- Ultra-low-power mode + backup register: 230 nA (3 wakeup lines).
- Wake-up time: 3.5 µs.

The most important features of this series are:

- **Core**:
  - ARM Cortex-M0+ core at a maximum clock rate of 32 MHz.
- **Memory**:
  - Static RAM up to 20 KB.
    * 20-byte battery-backed with tamper-detection erase.
  - Flash from 32 to 192 KB.
  - EEPROM up to 6 KB (with ECC).
  - Each chip has a factory-programmed 96-bit unique device identifier number.

- **Peripherals**:
  - Each L0-series features a range of peripherals which vary from line to line (see **Table 1.11** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 37 kHz), optional external HSE (1 to 24 MHz), LSE (32.768kHz).
- IC packages are LQFP, TFBGA, UFQFPN, WLCSP (see **Table 1.11** for more about this).
- Operating voltage range is 1.65V to 3.6V.

1.3.9 L1

LOW POWER

Common to all STM32L1:
- -40 +105°C range in LP
- Low voltage 1.65 to 3.6V
- Dynamic Voltage Scaling
- 5 clock sources
- USART, SPI, I²C
- 16-bit timers
- Temperature sensor
- DMA
- 5V tolerant I/Os
- Up to 51 fast I/Os
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 L1

Core: Cortex-M3
Instruction set: *Thumb, Thumb-2*, Saturated Math
Internal RC oscillators: HSI=16MHz, LSI=37KHz
External clocks: HSE=1-24Mhz, LSE=32.768KHz
Maximum Core Frequency: 32MHz
Low power modes: Low-power run, Sleep, Low-power sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2010
Available Packages: LQFP(48,64,100,144), TFBGA(64), UFBGA(100,132), UQFPN(48), WLCSP(63,64,104)

Product Line	FLASH (KB)	RAM (KB)	EEPROM (KB)	Memory I/F	OpAmp	Temperature Sensor	AES 128-bit	Touch Sense	Segment LCD Driver
STM32L100 Value line	32 to 256	4 to 16	2						Up to 8x28
STM32L151 STM32L152	32 to 512	16 to 80	4 to 16	SDIO FSMC	•	•		•	Up to 8x28
STM32L162	256 to 512	8 to 16	8 to 16	SDIO FSMC	•	•	•	•	Up to 8x40

Table 1.12: STM32L1 features

The STM32L1 series is the mid-range solution of the *Ultra Low-Power* segment. The combination of an ARM Cortex-M3 core with FPU and ultra-low-power features makes the STM32L1 optimal for applications operating on battery that also demand sufficient computing power. Like the L0-series, The STM32L1 offers dynamic voltage scaling, an ultra-low-power clock oscillator, LCD interface, comparator, DAC and hardware encryption.
Current consumption reference values:

- Ultra-low-power mode: 280 nA with backup registers (3 wakeup pins)
- Ultra-low-power mode + RTC: 900 nA with backup registers (3 wakeup pins)
- Low-power run mode: down to 9 µA
- Dynamic run mode: down to 177 µA/MHz

STM32L1 is pin-to-pin compatible with several MCU from the STM32F series. The most important features of this series are:

- **Core**:

  - ARM Cortex-M3 core with FPU at a maximum clock rate of 32 MHz.
- **Memory**:
  - Static RAM from 4 to 80 KB.
    * 20 bytes battery-backed with tamper-detection erase.
  - Flash from 32 to 512 KB.
  - EEPROM up to 2 KB (with ECC).
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each L1 series device features a range of peripherals which vary from line to line (see **Table 1.12** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 37 kHz), optional external HSE (1 to 24 MHz), LSE (32.768kHz).
- IC packages are LQFP, TFBGA, UFBGA, UFQFPN, WLCSP (see **Table 1.12** for more about this).
- Operating voltage range is 1.65V to 3.6V, including a programmable brownout detector.

1.3.10 L4

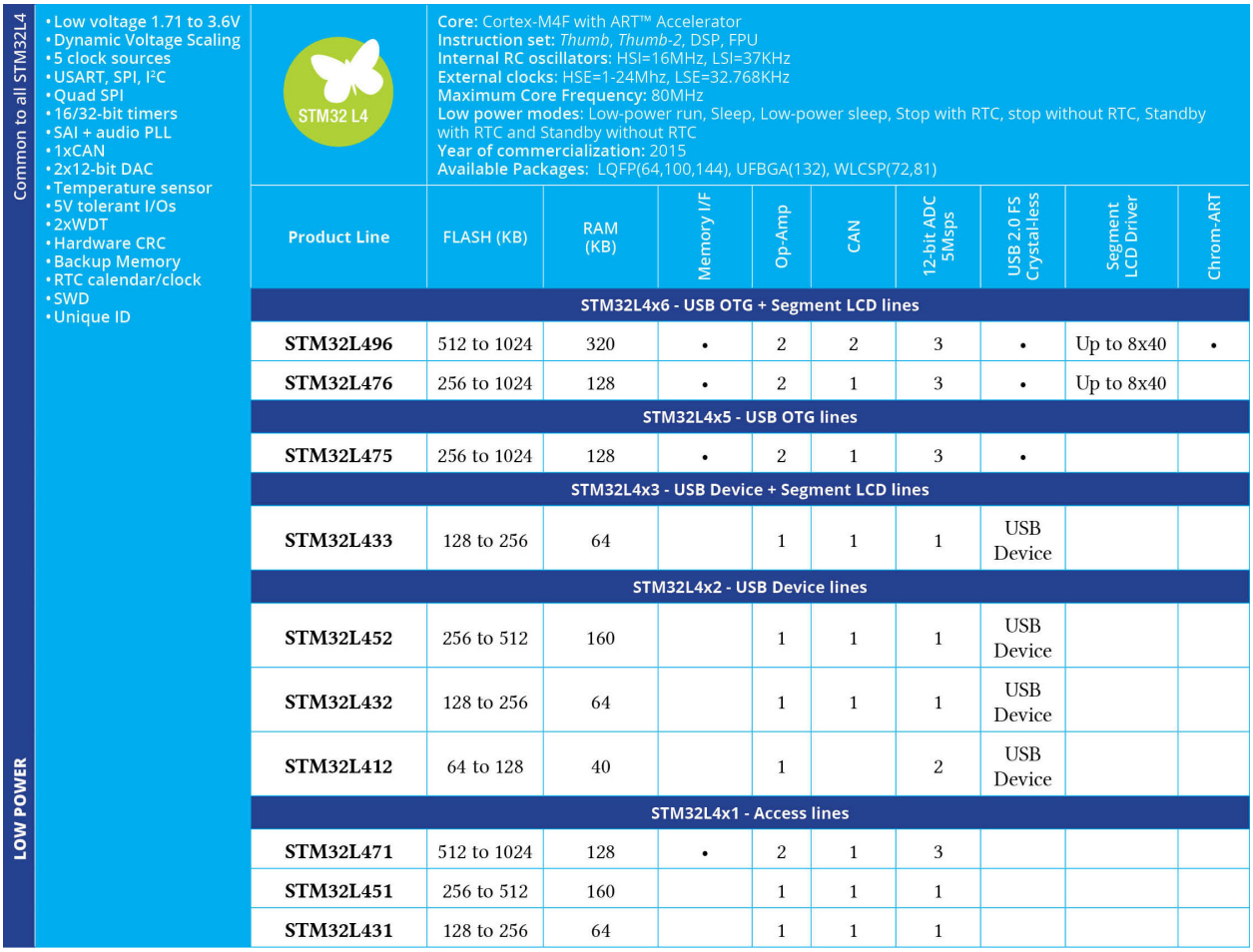


Table 1.13: STM32L4 features

The STM32L4 series is one of the best-in-class MCU series in the *Ultra Low-Power* segment. The combination of an ARM Cortex-M4 core with FPU and ultra-low-power features, makes the STM32L4 the best fit for applications demanding high performance while operating on battery or powered by energy harvesting. Like the L1-series, The STM32L4 offers dynamic voltage scaling and an ultra-low-power clock oscillator.
Current consumption reference values:

- Ultra-low-power mode: 8 nA with backup registers without RTC.
- Ultra-low-power mode + RTC: 200 nA with backup registers (5 wakeup lines).
- Ultra-low-power mode + 8 Kbytes of RAM: 195 nA.
- Ultra-low-power mode + 8 Kbytes of RAM + RTC: 340 nA.
- Dynamic run mode: down to 28 µA/MHz.
- Wake-up time: 5 µs.

STM32L4 is pin-to-pin compatible with several MCU from the STM32F series. The most important features of this series are:

- **Core**:
  - ARM Cortex-M4F core with FPU at a maximum clock rate of 80 MHz.
- **Memory**:
  - Static RAM up to 320 KB.
    * 20 bytes battery-backed with tamper-detection erase.
  - Flash sizes from 256 to 1024 KB.
  - Support to SDMMC and FSMC interfaces.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each L4-series device features a range of peripherals which vary from line to line (see **Table 1.13** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 37 kHz), optional external HSE (1 to 24 MHz), LSE (32.768kHz).
- IC packages are LQFP, UFBGA, WLCSP (see **Table 1.13** for more about this).
- Operating voltage range is 1.7V to 3.6V.

1.3.11 L4+

Common to all STM32L4+ — LOW POWER

- Low voltage 1.71 to 3.6V
- Dynamic Voltage Scaling
- 5 clock sources
- USART, SPI, I²C
- Quad SPI
- TFT and MIPI-DSI
- ART™ and Chrom-ART™
- Camera IF
- 16/32-bit timers
- 1xCAN
- 2x12-bit DAC
- Temperature sensor
- 5V tolerant I/Os
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 L4+

Core: Cortex-M4F with ART™ Accelerator
Instruction set: *Thumb*, *Thumb-2*, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI=37KHz
External clocks: HSE=1-24Mhz, LSE=32.768KHz
Maximum Core Frequency: 120MHz
Low power modes: Low-power run, Sleep, Low-power sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2017
Available Packages: LQFP(100,144), UFBGA(132,144,169), WLCSP(144)

Product Line	FLASH (KB)	RAM (KB)	Memory I/F	Op-Amp	Comp.	12-bit ADC 5Msps	USB 2.0 OTG	TFT Display	MIPI-DSI	AES 128/256
STM32L4x5										
STM32L4P5	512 to 1024	320	•	2	2	3	•	•		
STM32L4Q5	1024	320	•	2	2	3	•	•		•
STM32L4R5	1024 to 2048	640	•	2	2	1	•			
STM32L4S5	2048	640	•	2	2	1	•			•
STM32L4x7										
STM32L4R7	1024 to 2048	640	•	2	1	1	•	•		
STM32L4S7	2048	640	•	2	1	1	•	•		•
STM32L4x9										
STM32L4R9	1024 to 2048	640	•	2	1	1	•	•	•	
STM32L4R9	1024 to 2048	640	•	2	1	1	•	•	•	•

Table 1.14: STM32L4+ features

The STM32L4+ series, introduced on the market at the end of 2017, is the new best-in-class MCU series in the *Ultra Low-Power* segment. The STM32L4+ series shatters processing capabilities limits in the ultra-low-power world by delivering 150 DMIPS/409 CoreMark score while executing from internal Flash memory and by embedding 640 Kbytes SRAM enabling more advanced consumer, medical and industrial low-power applications and devices. STM32L4+ microcontrollers offer dynamic voltage scaling to balance power consumption with processing demand, low-power peripherals (LP UART, LP timers) available in Stop mode, safety and security features, smart and numerous peripherals, advanced and low-power analog peripherals such as op amps, comparators, 12-bit DACs and 16-bit ADC (hardware oversampling). The new STM32L4+ series also embeds advanced graphic features enabling state-of-the-art graphic user interfaces. The Chrom-ART Accelerator™, ST's proprietary 2D hardware graphic accelerator, efficiently handles repetitive graphic operations releasing the main CPU capabilities for real time processing or even more advanced graphic operations. The Chrom-ART Accelerator is coupled with the large embedded SRAM, the Chrom-GRC™ round display memory optimizer, the high-throughput Octo-SPI interface and to the advanced TFT and DSI controllers, allowing you to achieve 'smartphone-like' graphic user interfaces in a single-chip and an ultra-low power solution.

Current consumption reference values:

- Ultra-low-power mode: 20 nA with backup registers without RTC.
- Ultra-low-power mode + RTC: 200 nA with backup registers (5 wakeup lines).
- Ultra-low-power mode + 64 Kbytes of RAM: 800 nA.
- Ultra-low-power mode + 64 Kbytes of RAM + RTC: 1 µA.
- Dynamic run mode: down to 43 µA/MHz.
- Wake-up time: 5 µs.

STM32L4+ is pin-to-pin compatible with several MCU from the STM32F series. The most important features of this series are:

- **Core**:
  - ARM Cortex-M4F core with FPU at a maximum clock rate of 120 MHz.
- **Memory**:
  - Static RAM up to 640 KB.
    * 64 bytes battery-backed with tamper-detection erase.
  - Flash sizes from 1024 to 2048 KB.
  - Support to SDMMC and FSMC interfaces.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each L4+-series device features a range of peripherals which vary from line to line (see **Table 1.14** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 37 kHz), optional external HSE (1 to 24 MHz), LSE (32.768kHz).
- IC packages are LQFP, UFBGA, WLCSP (see **Table 1.14** for more about this).
- Operating voltage range is 1.7V to 3.6V.

1.3.12 L5

In 2018 ST announced the first STM32 family based on the new Cortex-M33 core. STM32L5 comes with a variety of security features, such as TrustZone™, Secure Boot, active IO tamper detection, Secure Firmware Install loader, certified cryptolib etc. According to STM view, STM32L5 family is addressed to IoT devices where security has emerged as one of the 3 key areas that developers of embedded and IoT applications are thriving to improve. The STM32L5 microcontroller series is the solution and provide a new optimal balance between performance, power and security.

Offering up to 512 Kbytes of flash (dual bank) memory and 256 Kbytes of SRAM, the STM32L5 series of microcontrollers reaches an upgraded level of performance (443 CoreMark) thanks to this new core and a new ST ART Accelerator™ (supporting now also external memory). The STM32L5 offers a large portfolio with 7 packages (from 48 to 144 pins) and supports up to 125°C ambient temperature. The most important features of this series are:

- **Core**:
  - ARM Cortex-M33 core with FPU at a maximum clock rate of 110 MHz.
- **Memory**:
  - Static RAM up to 256 KB.
  - Flash sizes from 256 to 512 KB.
  - Support to SDMMC and Octa-SPI interfaces.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each L5-series device features a range of peripherals which vary from line to line (see **Table 1.15** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 37 kHz), optional external HSE (1 to 24 MHz), LSE (32.768kHz).
- IC packages are LQFP, UFBGA, WLCSP (see **Table 1.15** for more about this).
- Operating voltage range is 1.71V to 3.6V.

Common to all STM32L5 LOW POWER											
• Low voltage 1.71 to 3.6V • Dynamic Voltage Scaling • 6 clock sources • USART, SPI, I²C • Octo SPI • TFT and MIPI-DSI • ART™ • ARM TrustZone™ • Up to 16 timers • 1xCAN • 2x12-bit DAC • 2x12-bit ADC • 1x USB Type-C™ PD • 1x USB 2.0 • Up to 22 cap-sens CH • Temperature sensor • 5V tolerant I/Os • 2xWDT • Hardware CRC • Backup Memory • RTC calendar/clock • SWD • Unique ID	STM32 L5	**Core:** Cortex-M33 with ART™ Accelerator **Instruction set:** *Thumb*, *Thumb-2*, DSP, FPU **Internal RC oscillators:** HSI=16MHz, LSI=37KHz **External clocks:** HSE=1-24Mhz, LSE=32.768KHz **Maximum Core Frequency:** 110MHz **Low power modes:** Low-power run, Sleep, Low-power sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC **Year of commercialization:** 2018 **Available Packages:** LQFP(48,100,144), QFN(48), UFBGA(132), WLCSP(81)									
	Product Line	FLASH (KB)	RAM (KB)	Memory I/F	2xOp-Amp	2xComp.	12-bit ADC 5Msps	USB 2.0 OTG	CAN FD	4ch ΣΔ Modulator	AES 128/256
	STM32L5x2										
	STM32L552	256 to 512	256	SDIO FSMC OctoSPI	•	•	2	•	•	•	
	STM32L562	256 to 512	256	SDIO FSMC OctoSPI	•	•	2	•	•	•	•

Table 1.15: STM32L5 features

1.3.13 U5

In 2021 ST announced an important evolution to the low-power portfolio introducing the STM32U5 family, where the "**U**" stands for *Ultra low-power*. Based on Cortex®-M33, STM32U5 meets the most demanding power/performance requirements for smart applications, including wearables, personal medical devices, home automation, and industrial sensors. Offering up to 2 Mbytes of Flash (dual bank) memory and 786 Kbytes of SRAM, the STM32U5 series of microcontrollers takes performance to the next level.

The most important features of this series are:

- **Core**:
  - ARM Cortex-M33 core with FPU at a maximum clock rate of 160 MHz.
- **Memory**:
  - Static RAM up to 786 KB.
  - Flash sizes up to 2048 KB.
  - Support to SDMMC, FSMC and Octa-SPI interfaces.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each L5-series device features a range of peripherals which vary from line to line (see **Table 1.16** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 37 kHz), optional external HSE (1 to 24 MHz), LSE (32.768kHz).
- IC packages are LQFP, UFBGA, WLCSP (see **Table 1.16** for more about this).
- Operating voltage range is 1.71V to 3.6V.
- Embedded SMPS (Switched-Mode Power Supply) step-down converter to improve power management.
- *Low Power Background Autonomous Mode* (LPBAM), an innovative autonomous power mode, saves power by enabling direct memory access *Low Power Direct Memory Access* (LPDMA) and ensuring the peripherals keep working, while most of the device is in stop mode.
- Key performance indicators include:
  - 110 nA in shutdown mode.
  - 300 nA in standby mode.
  - 1.7 µA in stop mode 3 with 16 Kbytes of SRAM.
  - 6.6 µA in stop mode 2 with 786 Kbytes of SRAM.
  - Down to 19 µA/MHz in active mode.

LOW POWER

Common to all STM32U5:
- Low voltage 1.71 to 3.6V
- Dynamic Voltage Scaling
- 6 clock sources
- USART, SPI, I²C
- Octo SPI
- ART™
- ARM TrustZone™
- 16/32-bit timers
- 1xCAN
- 2x12-bit DAC
- 2x12-bit ADC
- 1x USB Type-C™ PD
- 1x USB 2.0
- Up to 22 cap-sens CH
- Temperature sensor
- 5V tolerant I/Os
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 U5

Core: Cortex-M33 with ART™ Accelerator
Instruction set: *Thumb*, *Thumb-2*, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI=37KHz
External clocks: HSE=1-24Mhz, LSE=32.768KHz
Maximum Core Frequency: 160MHz
Low power modes: Sleep, Stop 0, 1, 2, 3 , Standby, Shutdown
Year of commercialization: 2021
Available Packages: LQFP(48,64,100,144), QFN(48), UFBGA(132), WLCSP(90)

Product Line	FLASH (KB)	RAM (KB)	Memory I/F	2xOp-Amp	2xComp.	1x14-bit 1x12-bit ADC	USB -C FS Dual Role	CAN FD	On-the-fly Decryption	AES 128/256
STM32L5x2										
STM32U575	1024 to 2048	786	SDIO FSMC 2xOctoSPI	•	•	2	•	•		
STM32U585	2048	786	SDIO FSMC 2xOctoSPI	•	•	2	•	•	•	•

Table 1.16: STM32U5 features

1.3.14 G0

Common to all STM32G0 MAINSTREAM

- Low voltage 1.7 to 3.6V
- Dynamic Voltage Scaling
- USART, SPI, I²C
- DMA
- Temperature sensor
- 5V tolerant I/Os
- WDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 G0

Cores: Cortex-M0+ with MPU
Instruction set: *Thumb*, *Thumb-2*, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI1/LSI2=32KHz, USB=100Khz to 48MHz
External clocks: HSE=1-16Mhz, LSE=32.768KHz
Maximum Core Frequency: 64MHz
Low power modes: Low-power run, Sleep, Low-power sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2019
Available Packages: LQFP(32,48,64), SO-8, TSSOP(20), UFQFPN(28,32, 48), WLCSP(25)

Product Line	FLASH (KB)	RAM (KB)	12-bit ADC	12-bit DAC	1x32-bit timer	1x16-bit timer @128Mhz	Low-Power Timer	Low-Power UART	USB-PD w/ DEV-HOST 2.0	Securable Memory Area	CAN FD	AES 256 + TRNG
STM32G0x0 Value Line	32 to 512	Up to 128	•									
STM32G0x1 Access Line	16 to 512	Up to 128	•	•	•	•	•	•	•	•	•	•

Table 1.17: STM32G0 features

In 2019 ST introduced on the market what it is essentially a modern replacement for the STM32F0 series: the STM32G0 family. This new series is based on a 90nm lithographic process, instead of the quite old 180nm process used for the STM32F0 family. This means less space needed for the same peripherals, smaller packages, lower costs at higher volumes due to the less demand in silicon, less power consumption. Traditionally, STM32 MCUs came in larger packages, compared to other Cortex-M manufacturers that offer even smaller packages like the SO-8 one. Now ST finally fixes this gap, allowing to reduce the PCB space and design. In fact, instead of having the traditional multiple power lines and their necessary decoupling capacitors, the STM32G0 uses only one line, which will make the PCB design a lot simpler and cheaper. This also means that some STM32G0 MCUs offer nine more I/Os than the corresponding STM32F0 MCUs that have a similar pin count. Moreover, a multi-bonding approach is used on STM32G03X and STM32G04X MCUs with small packages (SO8, TSSOP20) in order to offer a maximum of alternate functions and analog inputs. This approach results in multiple die pads connected internally to a single package pin.

This new generation offers a Cortex-M0+ core at 64 MHz as well as a Securable Memory Area, more RAM and Flash, more I/Os, and a more extensive integration of components to simplify the overall PCB design significantly. 32 pins and 48 pins models go from the simplest configuration with 8 KB of RAM and 16 KB of Flash to most powerful 128 KB RAM / 512 KB Flash architecture. The adoption of a Cortex-M0+ core will facilitate application programming is the low power modes. Despite being a mainstream device, the STM32G0 requires less than 100 µA / MHz when running at 64 MHz, thanks to its 90nm process that has some similarities with the STM32L4. Comparatively, the STM32F0 requires 250 µA / MHz. The new architecture also has a STOP mode that can go as low as 3 µA with only the Flash and Real-Time Clock off, while its STANDBY mode just demands 200 nA, which is drastically less than anything on the previous generation. Despite such a low power state, the architecture can still wake up in about 5 µs in STOP and 14 µs in STANDBY, making them even more advantageous.

Additionally, STM32G0 offers a VBAT pin, which means that engineers can put a capacitor to only power the RTC and backup registers, which allows the rest of the system to shut down and drops

the power consumption to 10 nA. With such a low consumption, it is possible to create a design that keeps basic information in memory while the user replaces a battery, for instance, thus improving the overall experience.

Finally, STM32G0 includes a crypto-core capable of accelerating AES 256 bit computations, as well as a *True Random Number Generator* (TRNG) to optimize cryptographic keys. However, the most interesting security feature is the implementation of a programmable securable memory area. Developers can define a portion of the Flash that will be inaccessible from the rest of the system once they log out of it, which enables them to store root keys and critical routines to implement features such as Secure Boot and Secure Firmware Upgrade.

STM32G0 is pin-to-pin compatible with a lot of MCU from the STM32F0 series. The most important features of this series are:

- **Core**:
  - ARM Cortex-M0+ core with MPU at a maximum clock rate of 64 MHz.
- **Memory**:
  - Static RAM up to 128 KB.
  - Flash sizes from 16 to 512 KB.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each G0-series features a range of peripherals which vary from line to line (see **Table 1.17** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 37 kHz), optional external HSE (1 to 48 MHz), LSE (32.768kHz).
- IC packages are LQFP, TSSOP, SO-8, UFQFPN, WLCSP (see **Table 1.11** for more about this).
- Operating voltage range is 1.7V to 3.6V.

1.3.15 G4

In 2019, ST also announced a new series that combines the performance of the STM32F4 family with some of the distinctive features of the STM32F3 series: STM32G4. The STM32G4 is the first is dedicated to the "mixed-signal" segment, since it is the first STM32 to include five analog-to-digital converters, seven digital-to-analog converters, six operational amplifiers, and seven comparators, while also integrating a USB-C Power Delivery controller, a 184 picoseconds high-resolution timer, CAN interfaces with flexible data rate, and math units accelerating certain trigonometric functions. Its Cortex-M4 core will reach up to 170 MHz to obtain 213 DMIPS, and 550 points in CoreMark while its numerous architectural improvements enable the creation of simpler yet more robust designs.

MAINSTREAM

Common to all STM32G4:
- Low voltage 1.7 to 3.6V
- Dynamic Voltage Scaling
- ART™ accelerator
- Flash memory with ECC
- Securable memory area
- Quad-SPI
- USART, SPI, I²C
- CAN-FD
- Advanced motor control timers
- 2xDMA
- Temperature sensor
- 5V tolerant I/Os
- WDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 G4

Cores: Cortex-M4F with MPU
Instruction set: *Thumb*, *Thumb-2*, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI1/LSI2=32KHz, USB=100Khz to 48MHz
External clocks: HSE=1-16Mhz, LSE=32.768KHz
Maximum Core Frequency: 170MHz
Low power modes: Low-power run, Sleep, Low-power sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2019
Available Packages: LQFP(32,48,64,80,100,128), UFQFPN(32, 48), WLC-SP(64,81), UFBGA(100)

Product Line	FLASH (KB)	RAM (KB)	CCM SRAM (KB)	12-bit ADC	12-bit DAC	Ulta-fast comparator	Op-amp (PGA)	FSMC	HRTIM
STM32G4x1 Access Line	32 to 512	32 to 96	Up to 16	3	4	3	4		
STM32G4x3 Performance Line	128 to 512	128	Up to 32	5	7	7	6	•	
STM32G4x4 Hi-Resolution Line	128 to 512	128	Up to 32	5	7	7	6	•	•

Table 1.18: STM32G4 features

The STM32G4 is the first STM32 with two math accelerators, one for trigonometric calculations (named by ST *COordinate Rotation DIgital Computer* or CORDIC) and the other for filtering functions (named by ST *Filter Mathematical ACcelerator* or FMAC). Very simply, the CORDIC accelerator provides hardware acceleration for the trigonometric functions often present in motor control, metering, signal processing, and many other applications. On the other hand, the FMAC enables the implementation of two major primary filters in signal processing: the finite impulse response (FIR) and infinite impulse response (IIR) digital filters. The STM32G4 series is also the first ST MCU architecture to include a timer with a resolution below 200 picoseconds. Its first and most obvious benefit is its ability to drive a highly precise power supply in LLC resonant topologies. The timer also offers a highly flexible pulse-width modulation (PWM) thanks to the presence of seven timebases that developers can combine to obtain very granular modulations. The high-resolution timer also benefits from an event handler to help engineers more easily configure and call the timer or use it to generate interrupts, for example.

The STM32G47x MCUs also have a critical feature in the form of a dual bank flash, like in some STM32F4 and in all STM32F7/H7 series. Very simply, the MCU organizes the flash memory in two physical banks with Read While Write (RWW) capability. As a result, it becomes possible to download, install, then run a new firmware without any interruptions. The system runs on one bank while the other receives the new firmware. The system can then hot swap the bank and seamlessly switch to the second one to run the new code. Developers can even secure the download operation by using new security features available on the STM32G4 like the securable memory area: a section of memory that can store secret keys or execute software routines and that only runs once after reset before becoming invisible to the user code. Moreover, the STM32G4 also borrows other features available in the STM32F3, like the *Core Coupled Memory* (CCM) described later in the book.

- **Core**:

  - ARM Cortex-M4F core with MPU at a maximum clock rate of 170 MHz.
- **Memory**:
  - Static RAM up to 128 KB.
  - Flash sizes from 32 to 512 KB.
  - Rich advanced analog peripherals (comparator, op-amps, DAC)
  - ADC with hardware oversampling (16-bit resolution)
  - Dual-bank Flash memory with error-correcting code (ECC) (supports in-field firmware upgrades)
  - Securable memory area
  - High-resolution timerUSB Type-C interface with Power Delivery including physical layer (PHY)
  - AES hardware encryption
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Peripherals**:
  - Each G0-series features a range of peripherals which vary from line to line (see **Table 1.17** for a quick overview).
- Oscillators consist of internal RC (16 MHz, 37 kHz), optional external HSE (1 to 48 MHz), LSE (32.768kHz).
- IC packages are LQFP, UFBGA, WLCSP, UQFN (see **Table 1.18** for more about this).
- Operating voltage range is 1.71V to 3.6V.

1.3.16 STM32WB

Common to all STM32WB
- Low voltage 1.71 to 3.6V
- Dynamic Voltage Scaling
- Multiprotocol 2.4GHz radio
- USART, LPUART, SPI, I²C
- ART™ accelerator
- 7x 16/32-bit timers
- 2xDMA
- Built-in DC/DC converter
- Temperature sensor
- 5V tolerant I/Os
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

WIRELESS

Cores: Cortex-M4F with ART™ Accelerator (*Application Processor*) + Cortex-M0+ (*Radio Processor*)
Instruction set: *Thumb*, *Thumb-2*, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI1/LSI2=32KHz, USB=100Khz to 48MHz
External clocks: HSE=1-24Mhz, LSE=32.768KHz
Maximum Core Frequency for Application Processor: 64MHz
Maximum Core Frequency for Radio Processor: 32MHz
Low power modes: Sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2018
Available Packages: UFQFPN(48), VFQFPN(64), WLCSP(100)

Product Line	FLASH (KB)	RAM (KB)	BLE 5.2	IEEE802.15.4 ZIGBEE 3.0	Concurrent Mode	FW OTA	External PA Support	HW Security	Quad-SPI	16-bit ADC	USB 2.0 FS Crystal-less	Segment LCD	AES 128/256
Dual Cortex-M4 and M0+ Core Lines													
STM32WB55	256 to 1024	Up to 256	•	•	•	•	•	•	•	•	•	Up to 8x40	•
STM32WB35	256 to 512	96	•	•		•	•	•	•	•	•		•
STM32WB15	320	48	•			•	•	•		•			•
Value Lines													
STM32WB50	1024	198	•	•		•		•		•			
STM32WB30	512	96	•	•		•		•		•			
STM32WB10	320	48	•			•		•		•			
Module Line													
STM32WB5M	1024	256	•	•	•	•		•	•	•	•	Up to 8x40	•

Table 1.19: STM32WB features

The STM32WB series, introduced on the market at the beginning of 2018, is the MCU series in the *2.4GHz Wireless* segment. STM32WB is a lineup of dual-core STM32 microcontrollers with integrated 2.4GHz radio fronted suitable for wireless and Bluetooth 5.x applications. These MCUs feature a Cortex-M0+ core running at 32 MHz (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 running at 64 MHz (named *Application Processor*) for the main embedded application.

The STM32WB platform is an evolution of the STM32L4 *Ultra Low-Power* series of MCUs. It provides the same digital and analog peripherals suitable for applications requiring extended battery life and complex functionalities. STM32WB integrate several communication peripherals, a convenient crystal-less USB2.0 FS interface, audio support, an LCD driver, up to 72 GPIOs, an integrated SMPS for power consumption optimization and multiple *low-power* modes to maximize battery life.

On top of wireless and *low-power* features, a particular focus was placed on embedding security hardware functions such as a 256-bit AES, PCROP, JTAG Fuse, PKA (elliptic curve encryption engine), and Root Secure Services (RSS). The RSS allows authenticating OTA communications, regardless of the radio stack or application.

The STM32WB55 is a Bluetooth 5.2 certified device, and it offers Mesh 1.0 software support, multiple profiles and flexibility to integrate proprietary BLE stacks. OpenThread-certified software stack is also available. The radio can also run BLE/OpenThread protocols concurrently. The embedded generic MAC allows the usage of other IEEE 802.15.4 proprietary stacks like ZigBee®, or proprietary protocols, giving even more options for connecting devices to the Internet of Things (IoT).

- **Core**:
  - ARM Cortex-M4F core with FPU at a maximum clock rate of 64 MHz (*Application Processor*). * ARM Cortex-M0+ core a maximum clock rate of 32 MHz (*Network Processor*).
- **Memory**:
  - Static RAM up to 256 KB.
  - Flash sizes up to 1024 KB.
  - Support to Quad-SPI interface.
  - Each chip has a factory-programmed 96-bit unique device identifier number.
- **Radio**:
  - BLE 5.0 compatible radio front-end and stack.
  - IEEE 802.15.4 compatible radio front-end.
  - Over-The-Air Firmware upgradability.
  - Support for an external Power Amplifier.
- **Peripherals**:
  - Each WB-series device features a range of peripherals which vary from line to line (see **Table 1.19** for a quick overview).
- Oscillators consist of several internal RC (16 MHz, 32 kHz), optional external HSE (1 to 24 MHz), LSE (32.768kHz).
- IC packages are WLCSP, UFQFPN, VFQFPN (see **Table 1.19** for more about this).
- Operating voltage range is 1.7V to 3.6V.

1.3.17 STM32WL

Common to all STM32WL

- Low voltage 1.71 to 3.6V
- Dynamic Voltage Scaling
- Multiprotocol 2.4GHz radio
- USART, LPUART, SPI, I²C
- ART™ accelerator
- 7x 16/32-bit timers
- 2xDMA
- Built-in DC/DC converter
- 1xADC
- 1xDAC
- Temperature sensor
- 5V tolerant I/Os
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

WIRELESS

Cores: Cortex-M4F with ART™ Accelerator (*Application Processor*) + Cortex-M0+ (*Radio Processor*)
Instruction set: *Thumb*, *Thumb-2*, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI1/LSI2=32KHz, USB=100Khz to 48MHz
External clocks: HSE=1-24Mhz, LSE=32.768KHz
Maximum Core Frequency for Application Processor: 48MHz
Maximum Core Frequency for Radio Processor: 32MHz
Low power modes: Sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2020
Available Packages: UFQFPN(48), VFQFPN(64), WLCSP(100)

Product Line	FLASH (KB)	RAM (KB)	LoRa	(G)FSK	(G)MSK	BPSK	Dual Power Output	HW Security
Dual Cortex-M4 and M0+ Core Lines								
STM32WL55	256	64	•	•	•	•	1 x 22dBm 1 x 15dBm	•
STM32WB54	256	64		•	•	•	1 x 22dBm 1 x 15dBm	•
Value Lines								
STM32WLE5	Up to 256	Up to 64	•	•	•	•	1 x 22dBm	
STM32WLE4	Up to 256	Up to 64		•	•	•	1 x 15dBm	

Table 1.20: STM32WL features

The STM32WL family was introduced in 2020 and it a sub-GHz radio. Built on Cortex-M4 and Cortex-M0+ cores (single- and dual-core architectures available), STM32WL microcontrollers support multiple modulations: LoRa, (G)FSK, (G)MSK, BPSK to ensure flexibility in wireless applications with LoRaWAN, Sigfox, W-MBUS, mioty or any other suitable protocol in a fully open way. STM32WL microcontrollers feature a sub-GHz radio based-on Semtech SX126x to meet the requirements of a wide range of Low-Power Wide Area Network (LPWAN) wireless applications in industrial and consumer Internet-of-Things (IoT). The radio is suitable for systems targeting compliance with radio regulations including, but not limited to, ETSI EN 300 220, FCC CFR 47 Part 15, China regulatory requirements and the Japanese ARIB T-108. Continuous frequency coverage from 150 to 960 MHz enables the support of all major sub-GHz ISM bands around the world.

Core: * ARM Cortex-M4F core with FPU at a maximum clock rate of 48 MHz (*Application Processor*). * ARM Cortex-M0+ core a maximum clock rate of 32 MHz (*Network Processor*). * **Memory**: * Static RAM up to 64 KB. * Flash sizes up to 256 KB. * Each chip has a factory-programmed 96-bit unique device identifier number. * **Radio**: * 150 to 960 MHz radio front-end. * LoRa, (G)FSK, (G)MSK, BPSK compatible modulations. * Dual-output power to balance between long-rage, battery life and regulatory compliance. * **Peripherals**: * Each WB-series device features a range of peripherals which vary from line to line (see **Table 1.20** for a quick overview). * Oscillators consist of several internal RC (16 MHz, 32 kHz), optional external HSE (1 to 24 MHz), LSE (32.768kHz). * IC packages are WLCSP, UFQFPN, VFQFPN (see **Table 1.20** for more about this). * Operating voltage range is 1.7V to 3.6V.

1.3.18 How to Select the Right MCU for You?

Selecting a microcontroller for a new project is never a trivial task, unless you are reusing a previous design. First of all, there are tens of MCU manufacturers on the market, each one with its market share and audience. ST, Microchip, TI, Atmel, Renesas, NXP and so on[18]. In our case we are very lucky: we have already picked a brand.
As we have seen in the previous paragraphs, the STM32 is really an extensive portfolio. We can choose an MCU from more than 1200 devices (if we also consider package variants). So, where to start?

In an ideal world, the first step of the selection process involves the understanding of needed computing power. If we are going to develop a CPU intensive application, focused on multimedia and graphic applications, then we have to shift our attention to the **High-Performance** group of STM32 microcontrollers. If, on the other hand, the computing power is not the main requirement of our electronic device, we can focus on the **Mainstream** segment, giving a close look at the STM32F1 series which offers the most extensive selection to choose from.

The next step is about connectivity requirements. If we need to interact with the external world through an Ethernet connection or other industrial protocols such as a CAN bus, and our application has to be responsive and able to deal with several Internet Protocols, then the STM32F4 portfolio is probably your best option; otherwise, the STM32F105/7 connectivity line is a better choice.

If we are going to develop a battery-powered device (maybe the new bestseller on the wearable market), then we have to look at the STM32L selection, choosing amongst the various sub-families according to the computing power we need.

As stated at the beginning of this paragraph, this is the selection process as it happens in an ideal world. But what about the real world? In the everyday development process, we probably have to answer the following questions before we begin selecting the right MCU for our project:

- **Is this device targeted for mass-market or a niche?**
  If you are developing a device that will be produced in small quantities, then the price difference amongst the STM32 microcontrollers will not affect your project too much. You may also consider the brand new STM32F7 and put little attention to software optimization (when dealing with low performance MCUs you have to do your best to optimize your code - keep in mind that this is also a cost which increases the final investment). On the other hand, if you are going to build a mass-market device, then the price of a single IC is really important: how much you will save during production often outweighs the initial investment.
- **What is the allowed budget for the total BOM?**
  This is a corollary to the previous point. If you already have the target price of your board, then you must carefully select the right MCU in the early stages.
- **What about space constraints?** Does your board have to fit the latest wearable device, or do you have sufficient room to use the IC package you prefer? The answer to this question deeply

[18]A good list of MCU manufacturers can be found here (http://bit.ly/1VUkN2e). Please, take note that in the last years several of the mentioned companies have merged to try to survive in a market that has become very crowded.

affects the selection process of an MCU and what we can demand of it in terms of performance and peripherals capabilities.

- **Which production technology can my company afford?**
  This is another non-trivial question. LQFP packages are still really popular in the MCU market thanks to the fact that they do not require complex production costs and they can be easily assembled even on old production lines. BGA and WLCSP packages require X-Ray inspection equipment and could affect your selection process.
- **Is time-to-market critical for you?**
  Time-to-market is always a key factor for anybody doing business, but sometimes you are required to have a firmware ready the day before you start the development process. This could lead to non-optimized firmware, at least at an early stage. This means that probably an MCU with more computing power is the best choice for you.
- **Can you reuse board layouts or code?**
  Every embedded developer has a portfolio of libraries and well-known ICs. Software development is a complex task which involves several stages before we can consider our firmware stable and ready for production. Sometimes (this is happening really frequently nowadays), you have to deal with undocumented hardware bugs or, at least, with their unpredictable behavior. This implies that you have to be really careful in deciding to switch to another architecture or even another MCU in the same series.
- **Is the candidate MCU available?**
  Well, in 2022 this is not a trivial question. Thanks to the global silicon shortage crisis, almost all silicon vendors have lead-times that span between 8 months and 2 years. And this is dramatically true for STM. A friend of mine, to meet a project deadline, had to buy 200 Nucleo and Discovery boards to desolder an STM32F1 MCU in order to deliver 200 custom boards. And now all distributors for electronic components (RS, Farnell, etc.) are placing caps on the maximum number of development boards a company can buy. So, before starting design a new product on an STM32 MCU, I strongly suggest buying the wanted MCU and keep the material in stock before even opening the IDE.

One of the key features of the STM32 platform could help a lot during the selection process: the pin-to-pin compatibility. This allows you to choose a more powerful (or cheaper) MCU during the selection process, giving you the freedom to change it at a more advanced development stage. For example, for a recent board I have developed, I started by choosing an STM32F1 MCU, but I downgraded it to a cheaper STM32F0 when I reached the conclusion that it would satisfy my requirements. However, keep in mind that this process always involves adapting the code to the different sub-family.

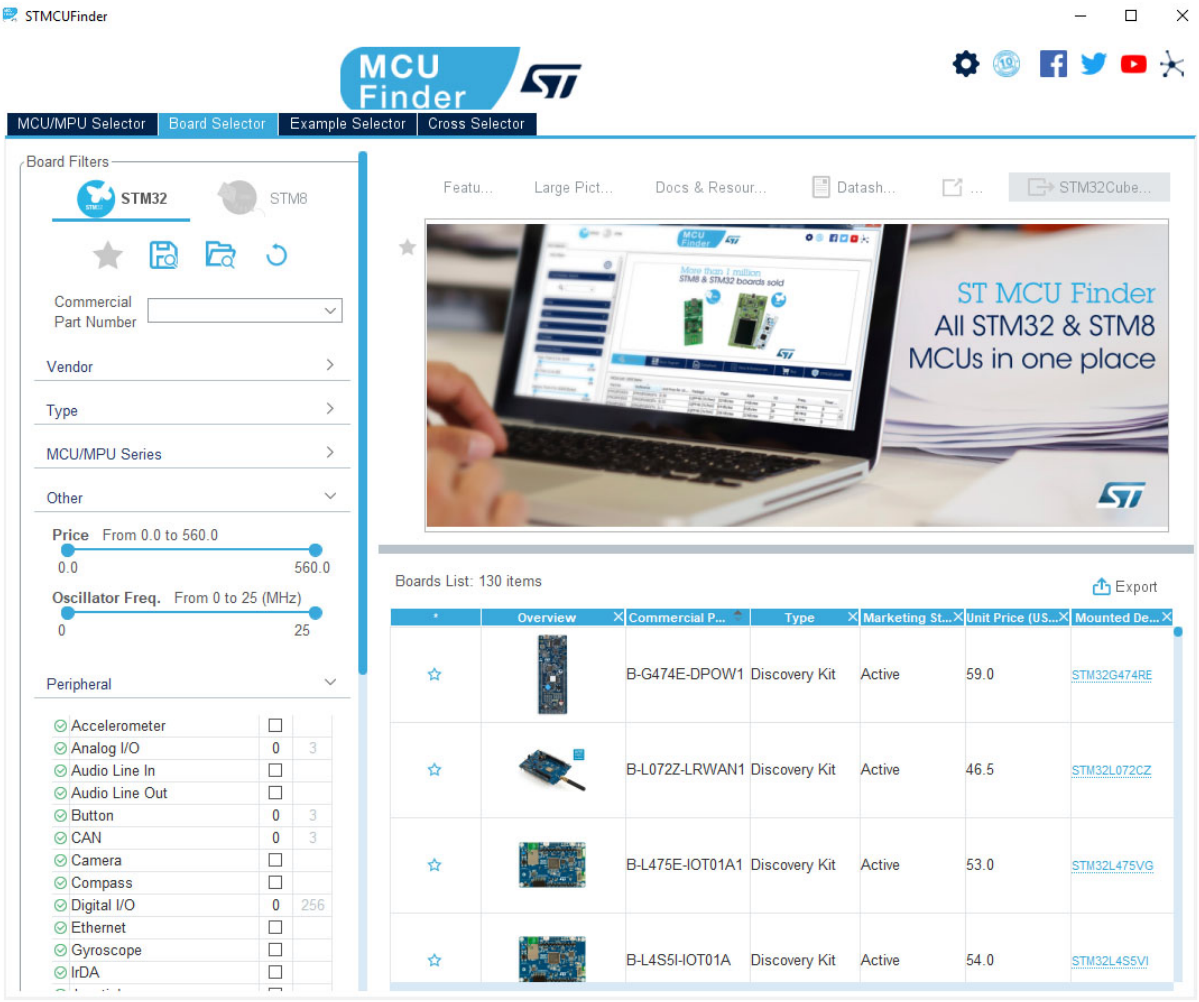


Figure 1.15: The MCU Finder tool for desktop

ST offers two convenient tools to help you in the MCU selection process. The first one is the STM32 and STM8 MCU Finder for Windows, Mac and Linux (see **Figure 1.15**) and its available on the ST website[19]. Before installing it, be patient: as we will see later, this tool is already integrated in the STM32CubeIDE tool-chain. So, there is no need to eat space on your PC.

[19] https://bit.ly/34BWSW3



Figure 1.16: The MCU Finder App for Android OS

The second tool is a useful mobile app (see **Figure 1.16**) available for iOS[20] and Android[21].

1.4 The Nucleo Development Board

Every practical text about an electronic device requires a development board (also known as *kit*) to start working with it. In the STM32 world the most widespread development board is the STM32 Discovery. ST has developed more than 48 different discovery boards useful to test STM32 MCUs and their capabilities.

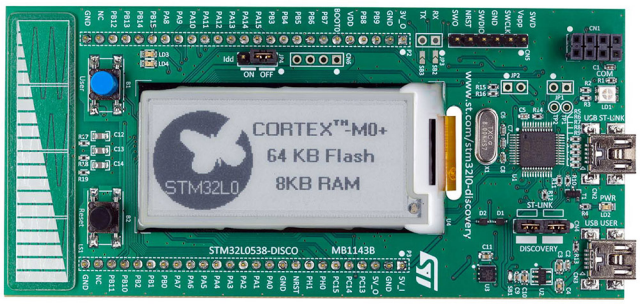


Figure 1.17: The STM32L0538 Discovery kit introduced by ST in 2015

For example, the STM32L0538DISCOVERY board (**Figure 1.17**) allows to test both the STM32L053 MCU and an e-paper display. You can find a lot of tutorials around the Internet covering boards from the Discovery line.

[20] http://apple.co/Uf20WR
[21] http://bit.ly/1Pvo8EV

ST introduced in 2015 a completely new range of development boards: the Nucleo. The Nucleo line-up is divided in three main groups: Nucleo-32, Nucleo-64 and Nucleo-144 (see **Figure 1.18**). The name of each group comes from the MCU package type used: Nucleo-32 uses an STM32 in an LQFP-32 package; Nucleo-64 uses an LQFP-64; Nucleo-144 an LQFP-144. The Nucleo-64 was the first line introduced to the market and there are currently 32 different boards, each one with a given STM32 microcontroller. The Nucleo-144 has been introduced in January 2016, and it is the first low-cost kit equipping the powerful STM32F746. It also provides an Ethernet phyther[22] and a LAN port. Since the Nucleo-64 is the most complete range, this book will cover only the most relevant boards from the Nucleo-64 line-up (see **Table 1.21** for the complete list). In the remaining parts of this book, we refer to the Nucleo-64 simply with the term "Nucleo".

The Nucleo is composed of two parts, as shown in **Figure 1.19**. The part with the mini-USB connector is an ST-LINK 2.1 integrated debugger, which is used to upload the firmware on the target MCU and to do step-by-step debugging. The ST-LINK interface also provides a *Virtual COM Port* (VCP), which can be used to exchange data and messages with the host PC. One key feature of Nucleo boards is that the ST-LINK interface can be easily separated from the rest of the board (two red scissors in **Figure 1.19** show where to break). This way it can be used as stand-alone ST-LINK programmer (a stand-alone ST-LINK programmer cost about $25). However, the ST-LINK provides an optional SWD interface that can be used to program another board without detaching the ST-LINK interface from the Nucleo (as it already happens with the Discovery boards) by removing the two jumpers labeled ST-LINK. The rest of the board contains the target MCU (the microcontroller we will use to develop our applications), a RESET button (the black one), a user programmable tactile button (the blue one) and an LED. The board also contains one pad to mount an external high-speed crystal (HSE). All recent Nucleo boards already provide a low-speed crystal. Finally, the board has several pin headers we will look at in a while.

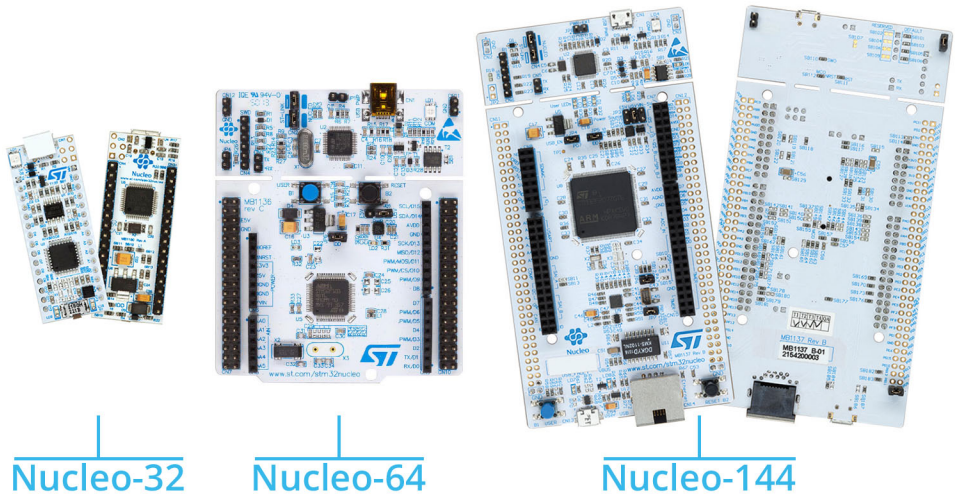


Figure 1.18: A Nucleo development board

ST introduced this new kit to attract people from the Arduino world. In fact, Nucleo boards provide pin headers to accept *Arduino shields*, expansion boards specifically built to expand the Arduino

[22] The *Ethernet phyther* (also called *Ethernet PHY*) is a device which translates messages exchanged over a LAN network in electrical signals.

UNO and all other Arduino boards. **Figure 1.20**[23] shows the STM32 peripherals and GPIOs associated with the Arduino compatible connector.

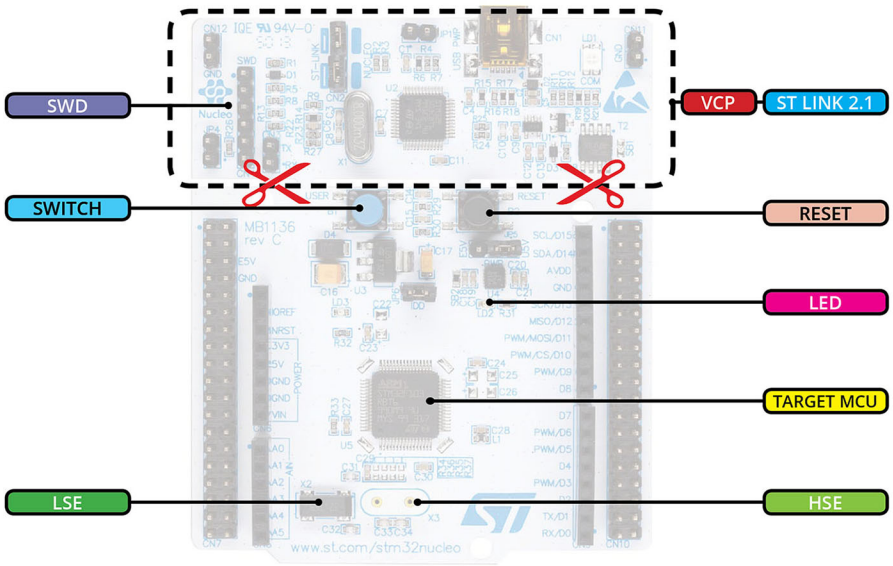


Figure 1.19: The relevant parts of a Nucleo board

To be honest, the Nucleo boards have other interesting advantages compared to the Discovery ones. First of all, ST sells them at a really aggressive price (probably for the aforementioned reasons). A Nucleo costs between $10 and $15, depending on where you buy it, and if you think about what you can do with this architecture, you have to agree that it is really underpriced compared to an Arduino DUE board (which is also equipped with a 32-bit ARM processor from Microchip). Another interesting feature is that Nucleo boards are designed to be pin-to-pin compatible with each other. This means that you can develop the firmware for the STM32Nucleo-F103RB board (equipped with the popular STM32F103 MCU) and later adapt it to a more powerful Nucleo (e.g. STM32Nucleo-F401RE) if you need more computing power.

[23]Figure 1.20 and 22 are taken from the mbed.org website and they refer to the Nucleo-F401RE board. Please, refer to Appendix C for the right pin-out of your Nucleo board.

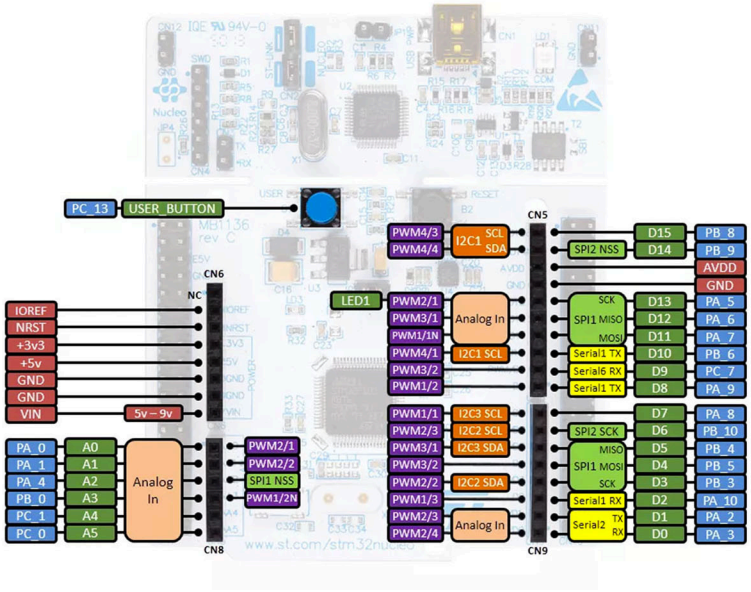


Figure 1.20: Peripherals and GPIOs associated to Arduino headers

In addition to Arduino compatible pin headers, the Nucleo provides its own expansion connectors. They are two 2x19, 2.54mm spaced male pin headers. They are called *Morpho* connectors and are a convenient way to access most of the MCU pins. **Figure 1.21** shows the STM32 peripherals and GPIOs associated with the *Morpho* connector.

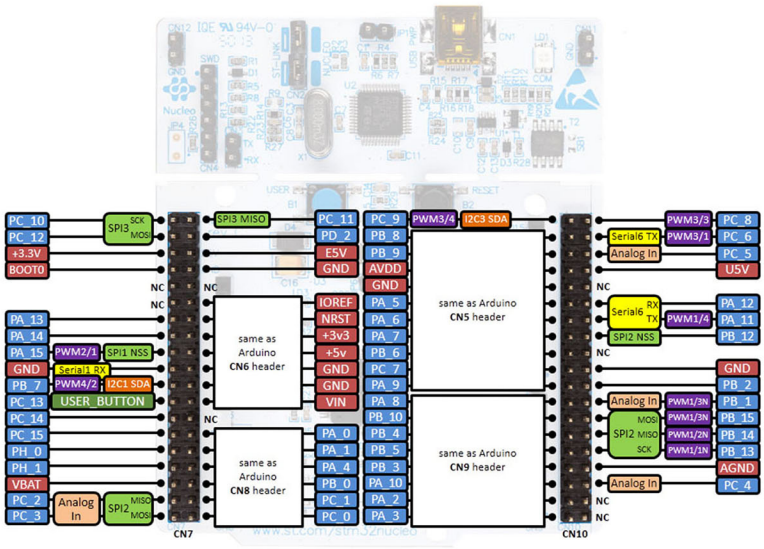


Figure 1.21: Peripherals and GPIOs associated to *Morpho* headers

ST is releasing several expansion shields for the Nucleo that are compatible with the Arduino UNO or the ST Morpho header. For example, **Figure 1.22** shows a Nucleo-F302R8 board with a X-NUCLEO-IHM07M1 expansion board, a shield which features the ST L6230 DMOS driver, a motor control

solution for low voltage 3-phase DC brushless motors.

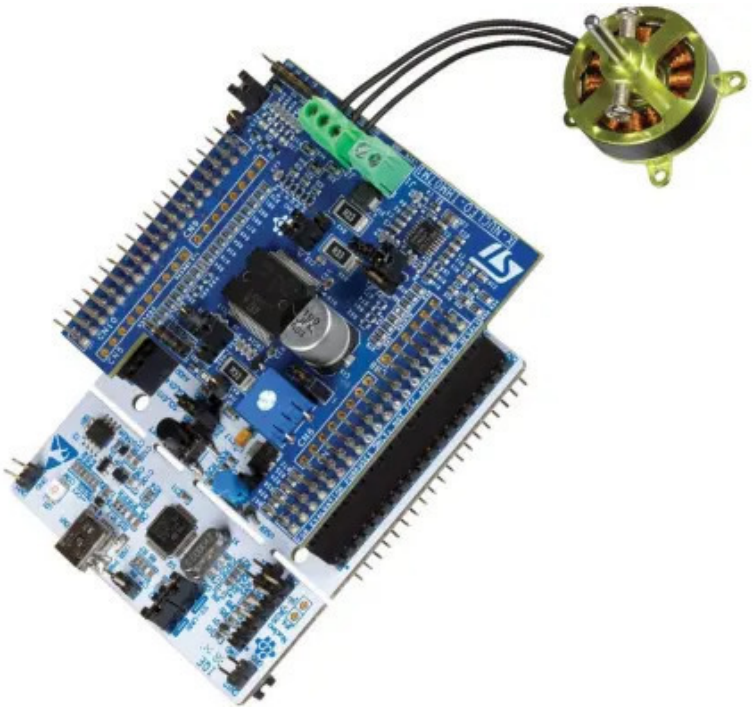


Figure 1.22: The X-NUCLEO-IHM07M1 expansion shield

There are third-two Nucleo boards available at the time of writing this chapter (February 2022). **Table 1.21** summarizes the main features of the Nucleo boards used to make the samples in this book, together with the ones common to all Nucleo-64 boards[24].

[24]Please, take note that ST places the STM32G4 family in the *Mainstream* segment. However, with a core running at 170MHz it can be considered an *High Performance* MCU. So, in this book, you will find the Nucleo-G474RE classified in the *High Performance* series.

Common to all Nucleo-64 boards:
Integrated ST-Link/V2-1 debugger that can be also used as stand-alone debugger
Virtual COM port integrated in ST-Link interface
64-pin LQFP target MCU
2x(2x19) 2.54mm Morpho extension headers
Arduino UNO extension headers
1 LED and 1 Tactile switch freely available to programmer

	Nucleo P/N	STM32 MCU	RAM (KB)	FLASH (KB)	F_{CPU} (MHz)
HIGH PERFORMANCE	**NUCLEO-F446RE**	STM32F446RET6	128	512	180
	NUCLEO-G474RE	STM32G474RET6	128	512	170
	NUCLEO-F401RE	STM32F401RET6	96	512	84
MAINSTREAM	**NUCLEO-F303RE**	STM32F303RET6	64	512	72
	NUCLEO-F103RB	STM32F103RBT6	20	128	72
	NUCLEO-F072RB	STM32F072RBT6	16	128	48
LOW POWER	**NUCLEO-L476RG**	STM32L476RGT6	96	1024	80
	NUCLEO-L152RE	STM32L152RET6	80	512 + 16KB EEPROM	32
	NUCLEO-L073RZ	STM32L073RZT6	20	192 + 6KB EEPROM	32

Table 1.21: The list of Nucleo boards used in this text and their specific features



Why Use the Nucleo as Example Board for This Book?

The answers to this question are almost all contained in the previous paragraphs. First of all, Nucleo boards are cheap, and allow you to start learning the STM32 platform at nearly no cost. Second, they greatly simplify the instructions and examples contained in this book. You are completely free to use the Nucleo you like. The book will show all the steps required to easily adapt examples to your specific Nucleo.



Keep in mind that the whole book is designed to give the reader all the necessary tools to start working with any board, even custom ones. It will be easy to adapt the examples to your needs.

2. Get In Touch With SM32CubeIDE

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

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

To accomplish these activities, we essentially need:

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

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

2.1 Why Choose STM32CubeIDE as Tool-Chain for STM32

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

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

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

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

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

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

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

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

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

2.1.1 Two Words About Eclipse...

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

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

2.1.2 ... and GCC

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

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

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



What Is a Cross-Compiler?

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

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

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

2.2 Downloading and Installing the STM32CubeIDE

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

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

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

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

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

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

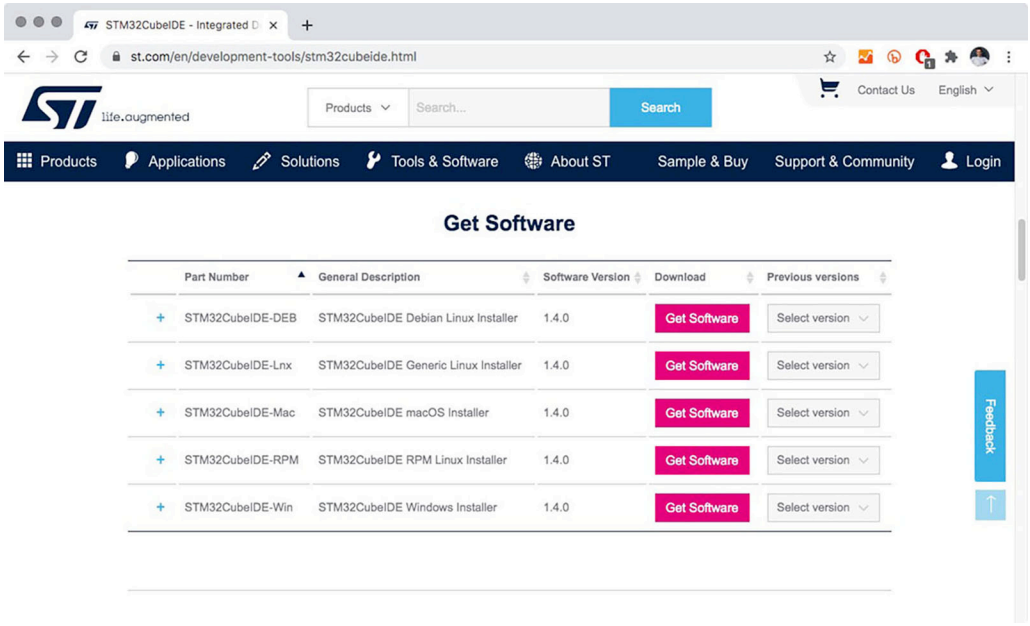


Figure 2.1: The STM32CubeIDE download page on the official STM website



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

2.2.1 Windows - Installing the Tool-Chain

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

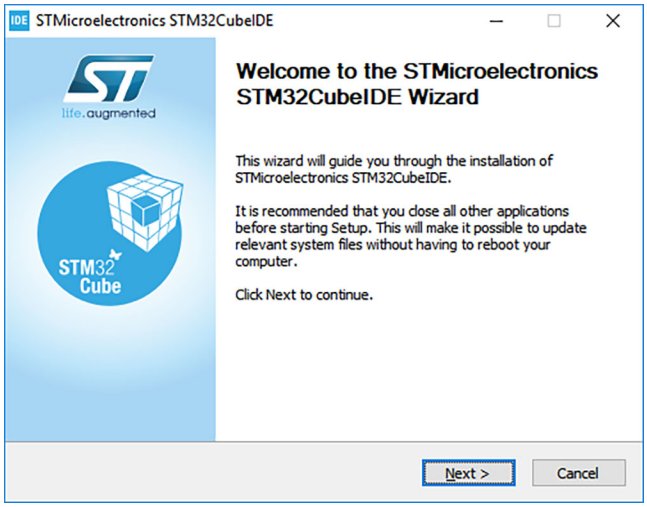


Figure 2.2: Windows installer welcome page

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

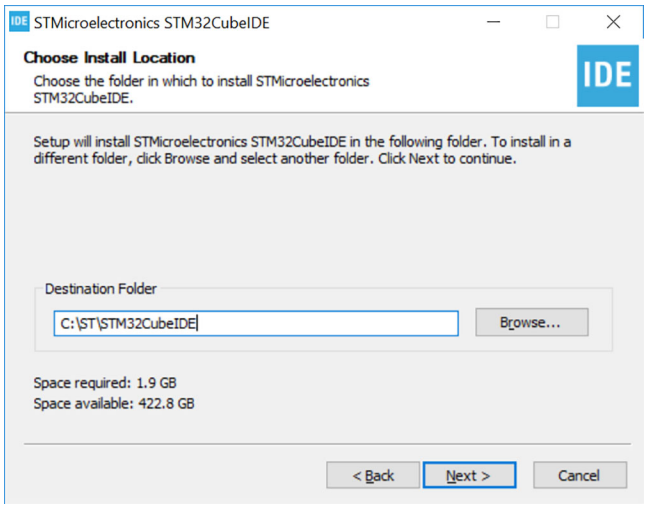


Figure 2.3: *Chose Install Location* dialog



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

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

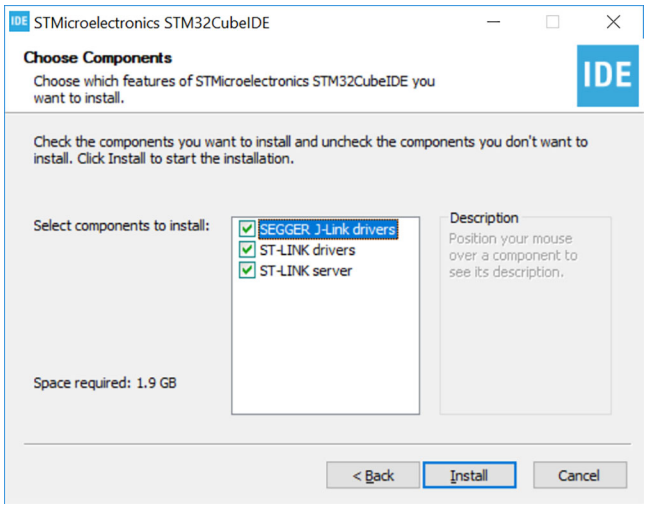


Figure 2.4: *Choose Components* dialog

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

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

You can download STM32CubeProgrammer from the official ST page[7] (the download link is at the bottom of the page in the "**Get Software**" section). Once the download is completed, extract the

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

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

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

2.2.2 Linux - Installing the Tool-Chain

The whole installation procedure will assume these requirements:

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

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

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

Proceed as follows:

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

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

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

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

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

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

- `libusb-1.0.0-dev`

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

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

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

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

2.2.3 Mac - Installing the Tool-Chain

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

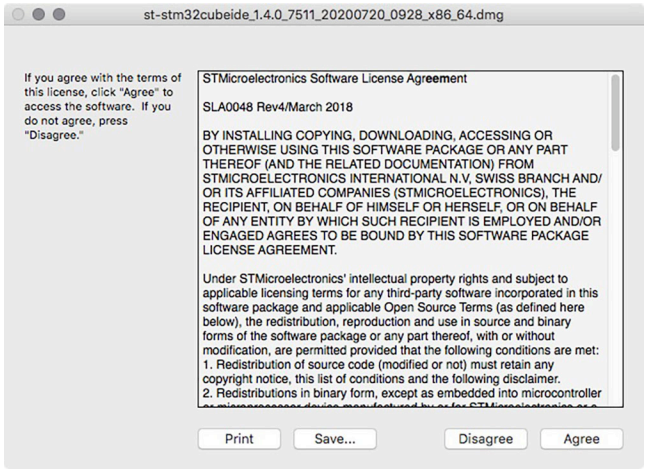


Figure 2.5: ***License Agreement*** **dialog**

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

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



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

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

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

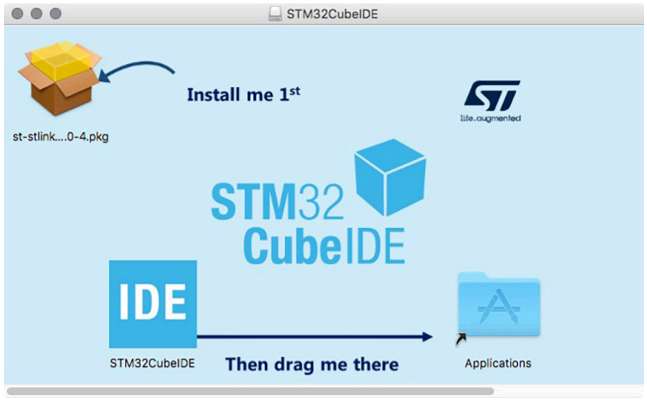
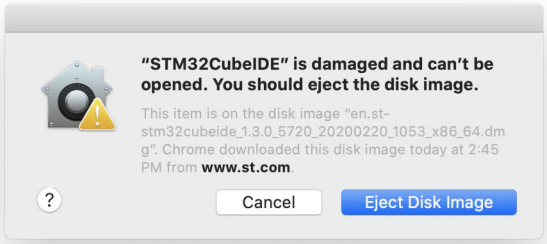


Figure 2.6: ***Install page*** dialog



Read Carefully!

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



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

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

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

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

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

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

2.3 STM32CubeIDE overview

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

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

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

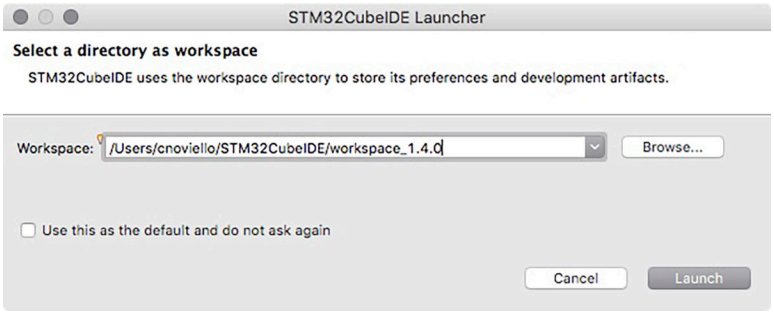


Figure 2.7: Eclipse workspace selection dialog

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

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



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

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

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

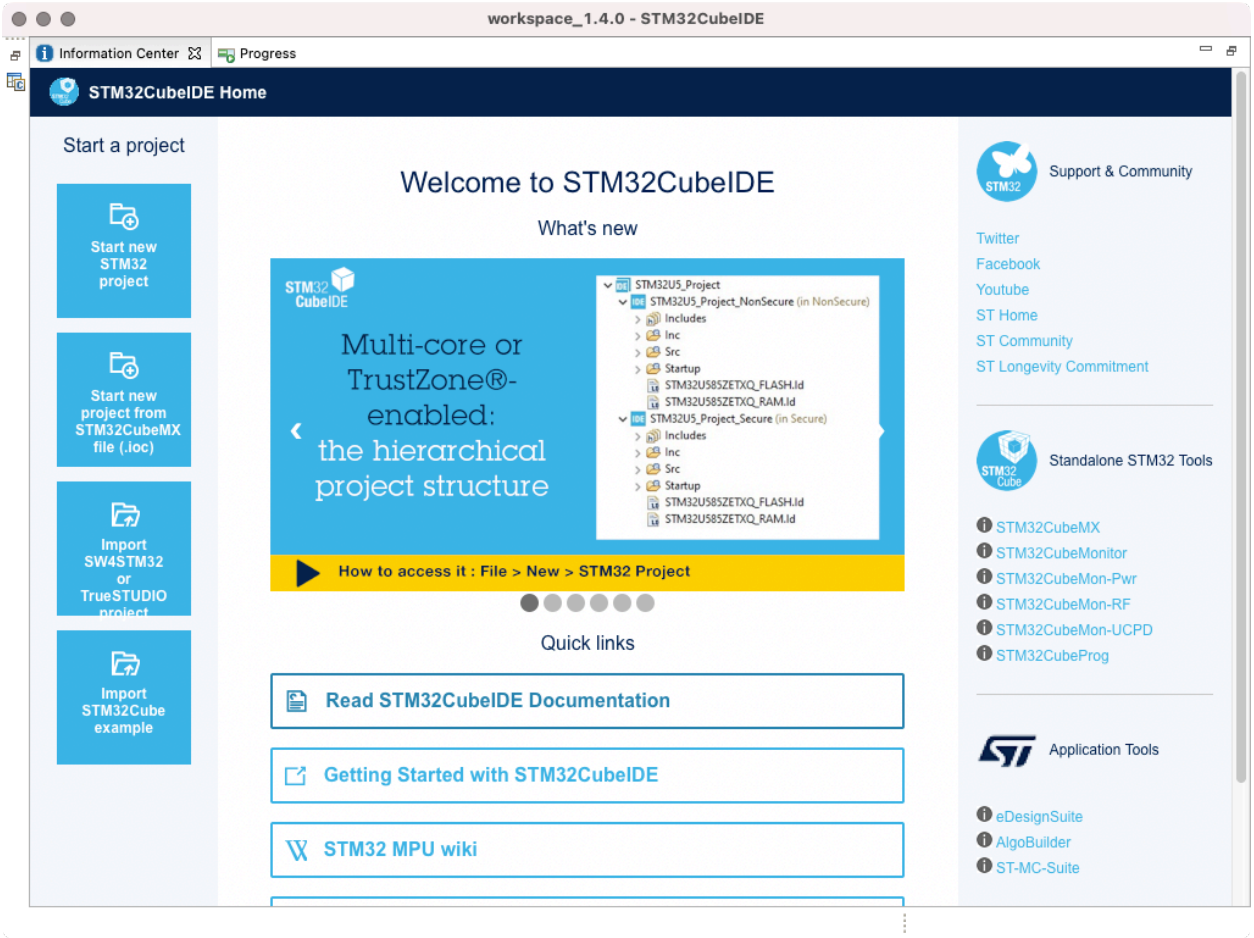


Figure 2.8: The Eclipse interface once started for the first time

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

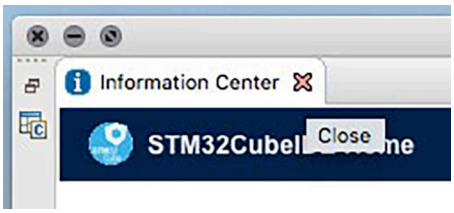


Figure 2.9: How to close the *Welcome view* by clicking on the *X*.

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

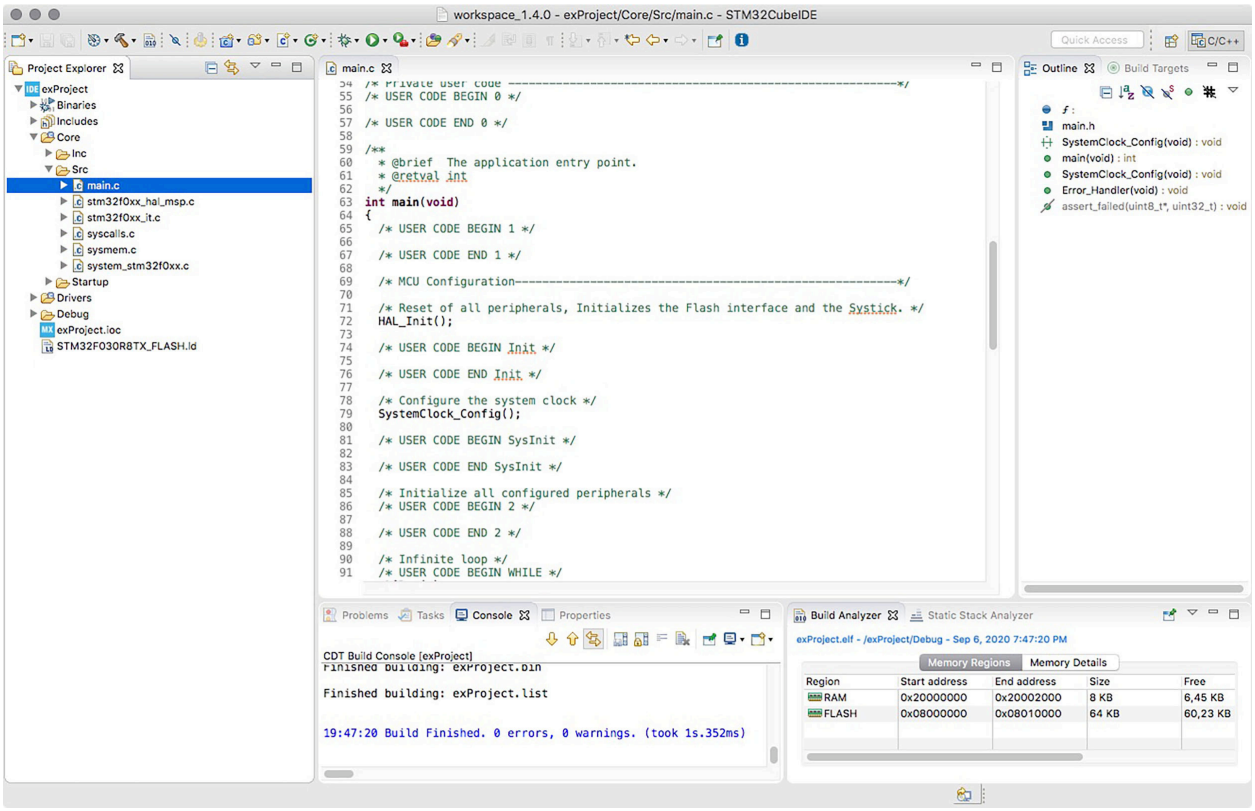
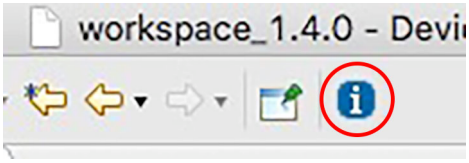


Figure 2.10: The C/C++ perspective view in eclipse (with a main.c file loaded later)

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



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

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

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

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

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

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



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

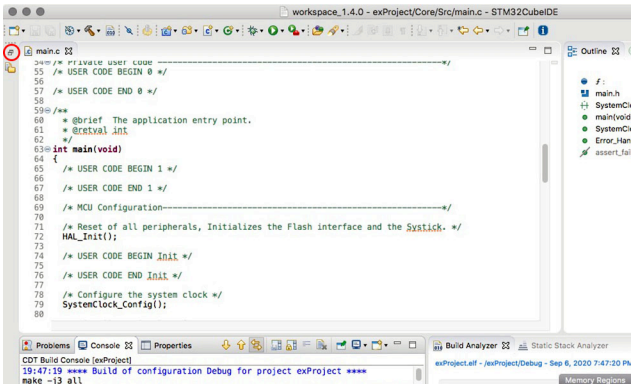


Figure 2.11: *Project Explorer* view minimized

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

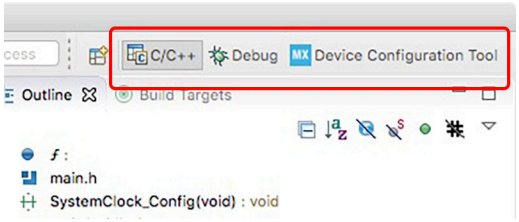
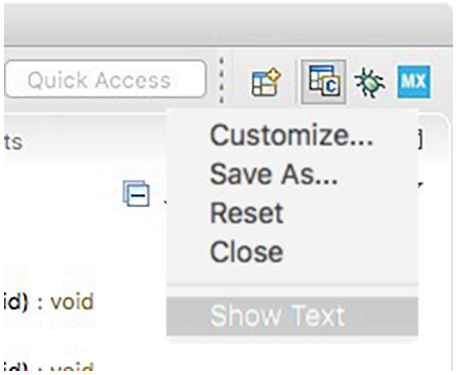


Figure 2.12: Perspective switcher toolbar

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



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



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

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

Table 2.1: Main Eclipse's toolbar icons

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

3. Hello, Nucleo!

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

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

3.1 Create a Project

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



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

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

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

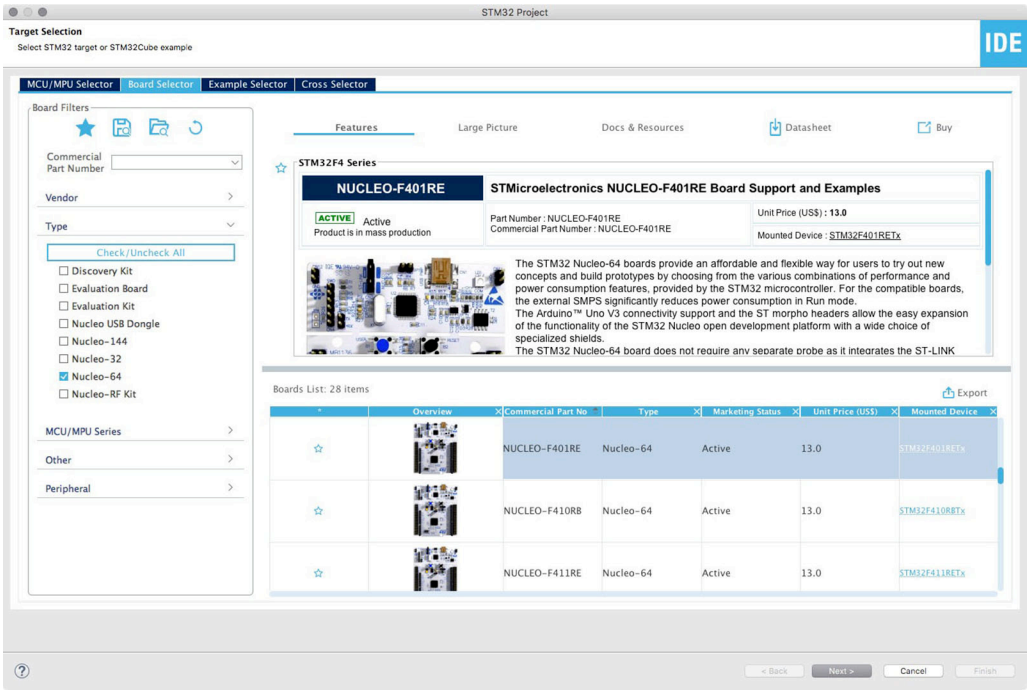


Figure 3.1: The Target Selection wizard - STEP 1



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

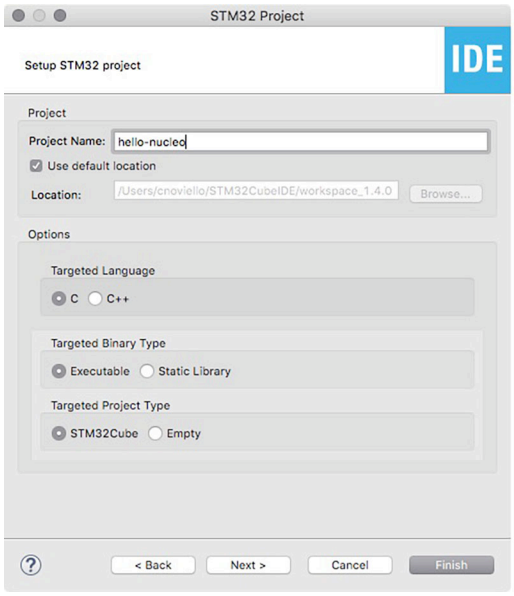


Figure 3.2: Project wizard - STEP 2

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

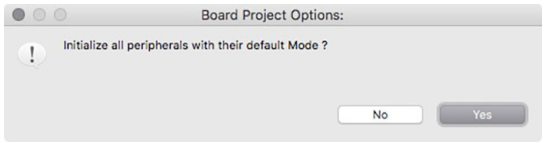


Figure 3.3: Project wizard - STEP 3

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

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

These packages contain several relevant components:

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

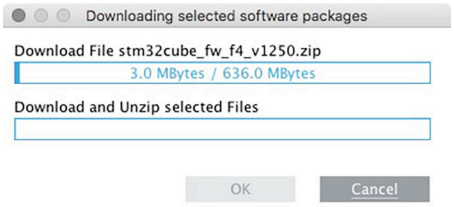


Figure 3.4: Cube Firmware Package download dialog

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

3.2 Adding Something Useful to the Generated Code

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

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

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

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

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

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

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

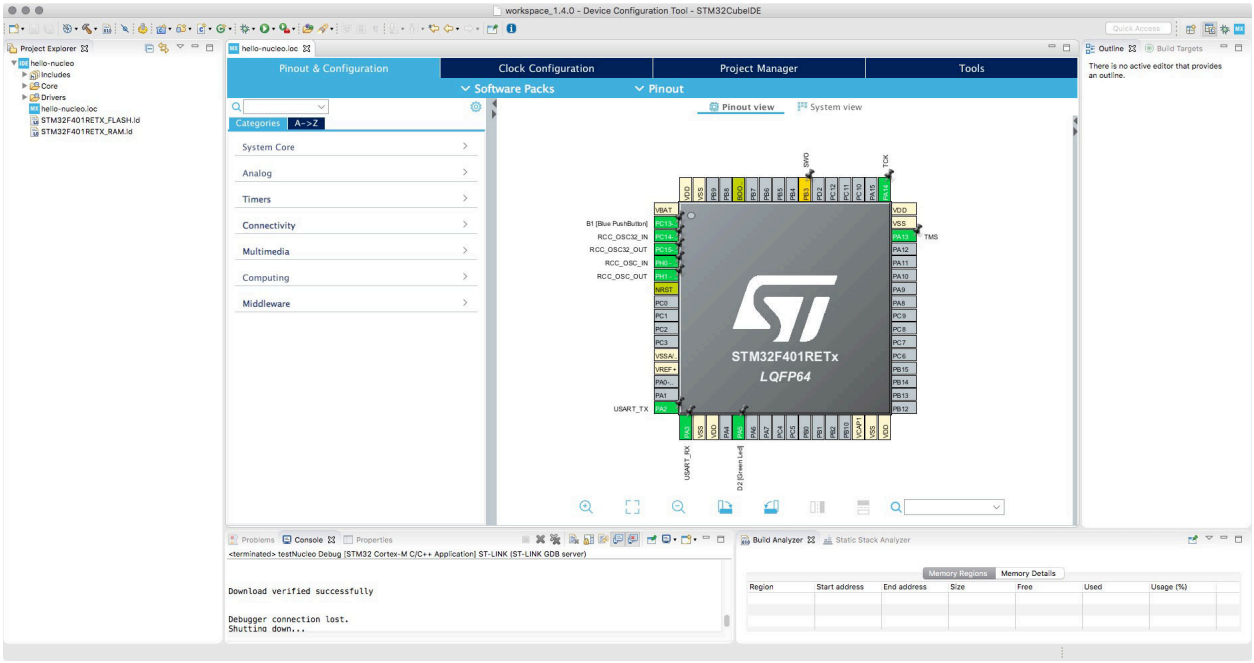


Figure 3.5: The STM32CubeIDE interface after complete project generation

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

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

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

the invocation of four routines[4]:

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

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

Filename: `src/main.c`

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

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

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



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

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

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

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



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

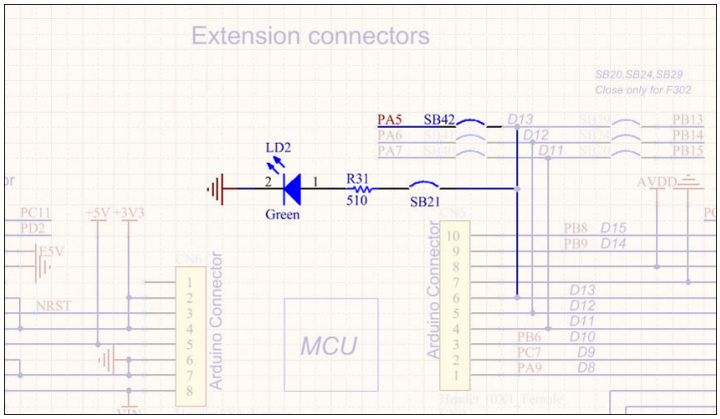


Figure 3.6: `LD2` connection to PA5

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

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

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

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

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

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

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

3.3 Connecting the Nucleo to the PC

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



Read Carefully

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

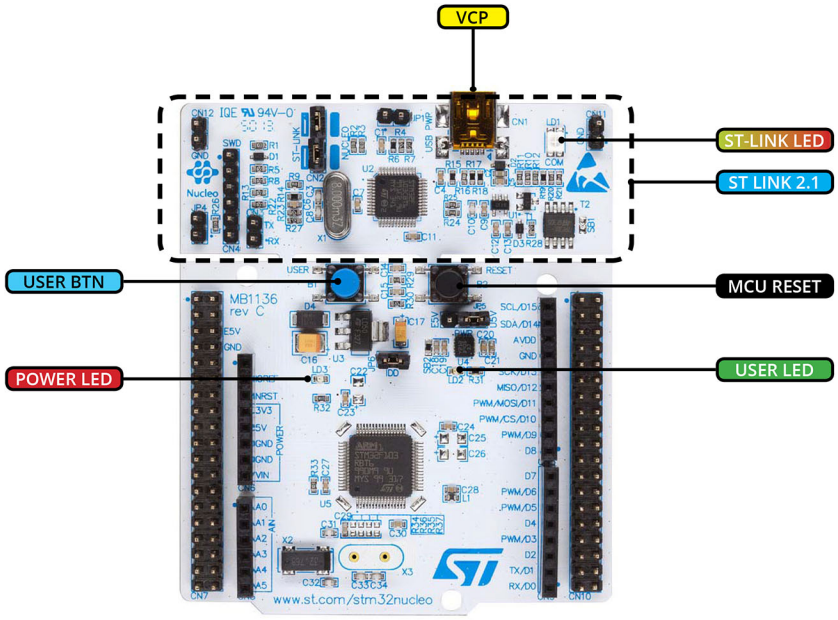


Figure 3.7: A Nucleo board and its main interfaces

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

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

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

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

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

3.3.1 ST-LINK Firmware Upgrade



Warning

Read this paragraph carefully. Do not skip this step!

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

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

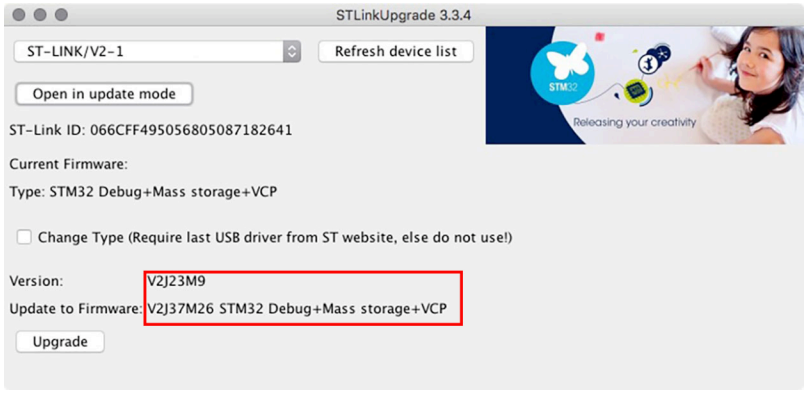


Figure 3.8: The ST-LINK Upgrade program

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



Error in upgrading the ST-LINK firmware

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

3.4 Flashing the Nucleo using STM32CubeProgrammer

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

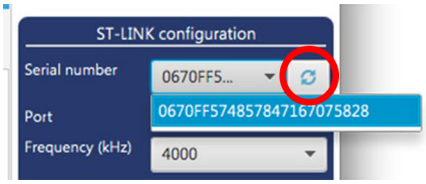


Figure 3.9: The ST-LINK interface serial number as shown by STM32CubeProgrammer tool



Read Carefully

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

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

Congratulations: welcome to the STM32 world ;-)

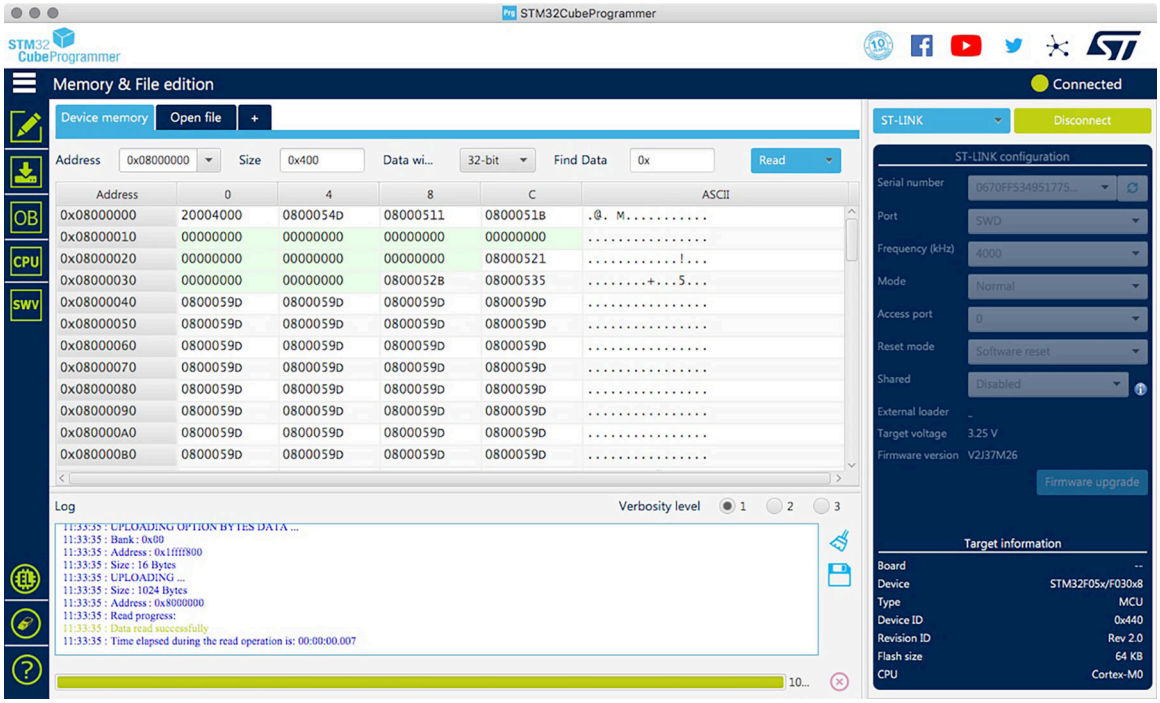
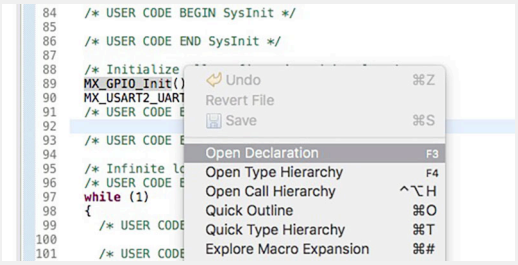


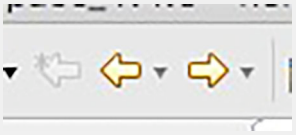
Figure 3.10: The STM32CubeProgrammer interface when connected to the Nucleo board

Eclipse intermezzo

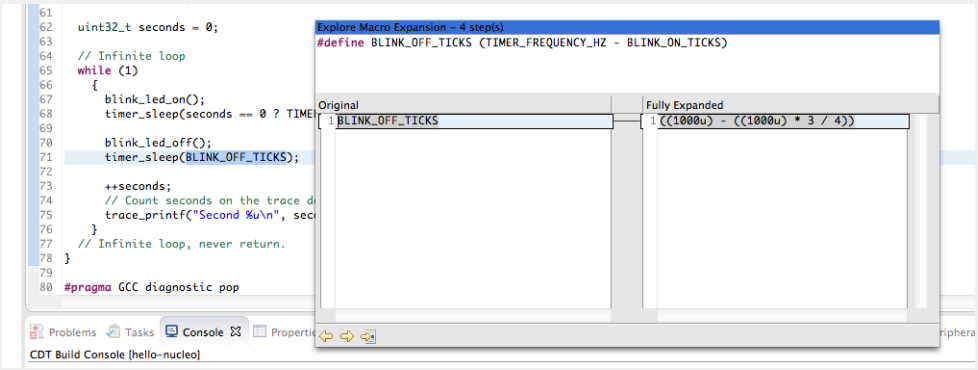
Eclipse allows us to easily navigate inside the source code, without jumping between source files manually looking for where a function is defined. For example, let us suppose that we want to see how the function **`MX_GPIO_Init()`** is coded. To go to its definition, highlight the function call, click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



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



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



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

4. STM32CubeMX Tool

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

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

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

4.1 Introduction to CubeMX Tool

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

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

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

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

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

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

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

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

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

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

4.1.1 Target Selection Wizard

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

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

[5]ch3-hello-nucleo-project

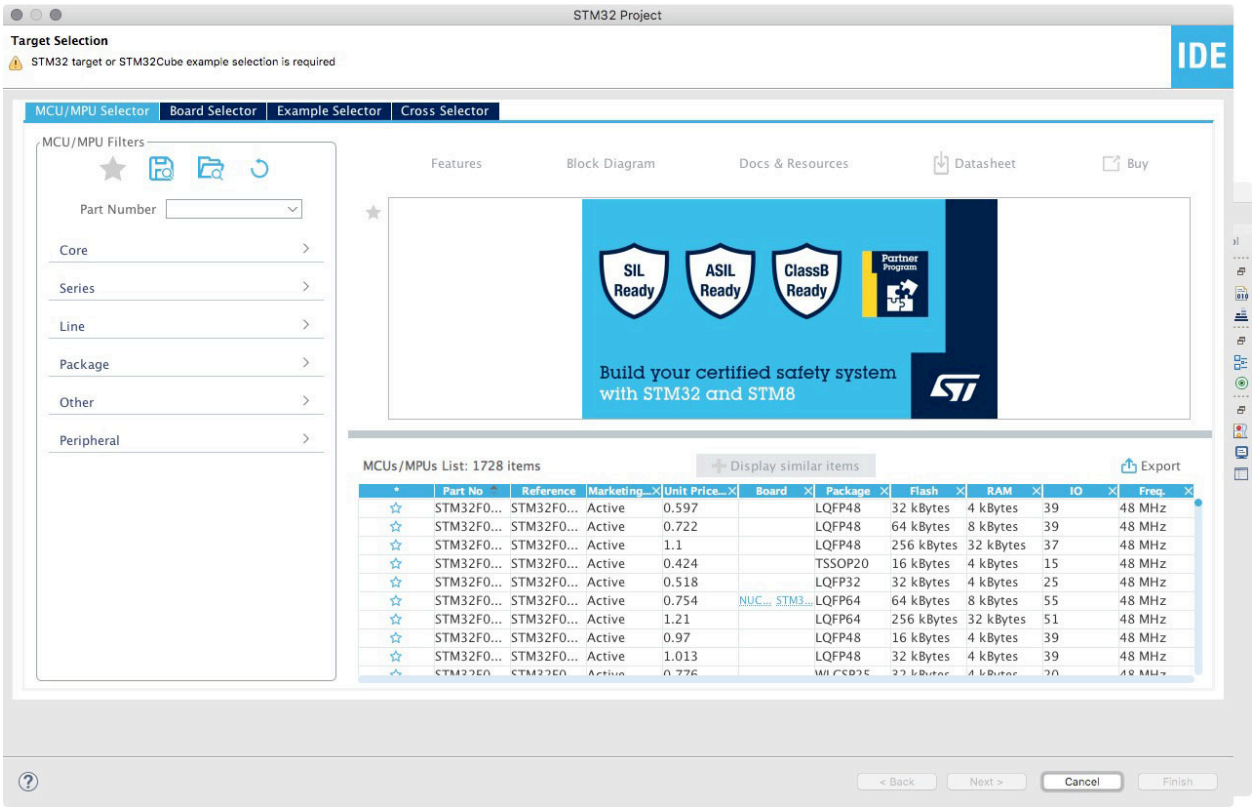


Figure 4.1: CubeMX *MCU/MPU Selector*

4.1.1.1 MCU/MPU Selector

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

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

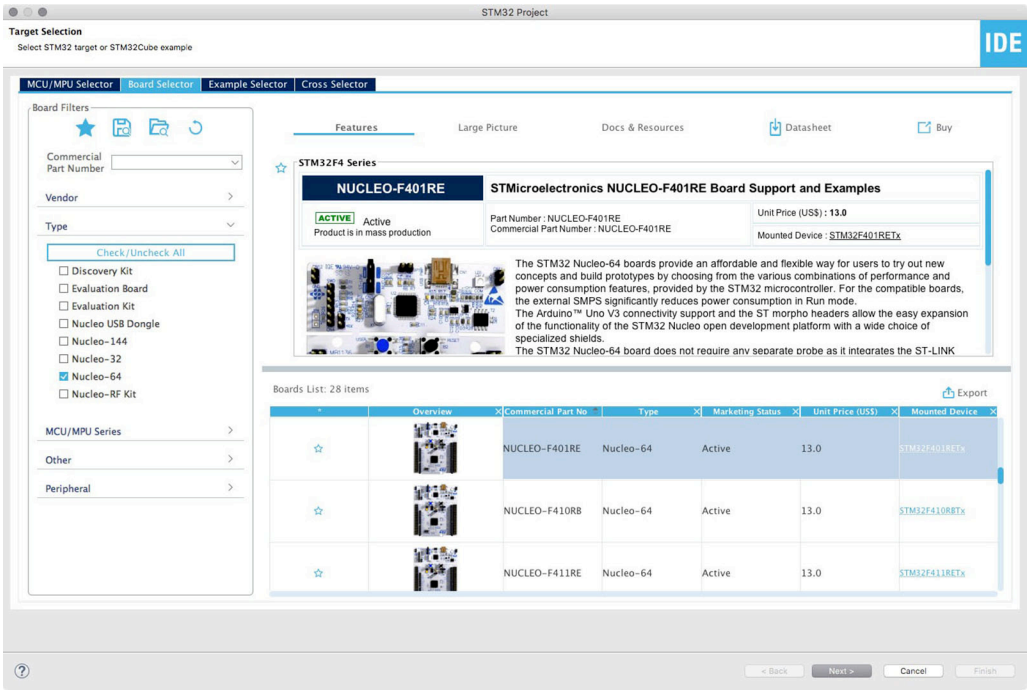


Figure 4.2: CubeMX *Board Selector*

4.1.1.2 Board Selector

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

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

4.1.1.3 Example Selector

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

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

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

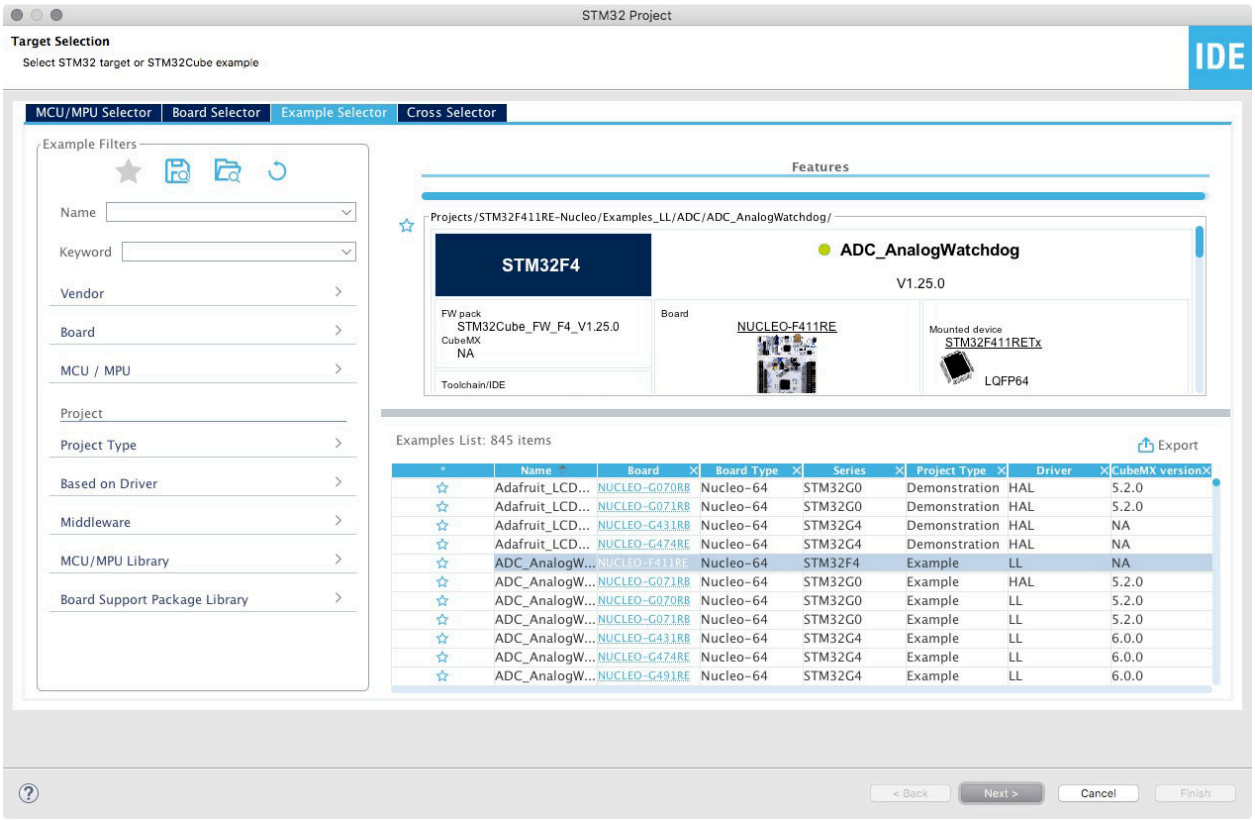


Figure 4.3: CubeMX *Example Selector*

4.1.1.4 Cross Selector

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

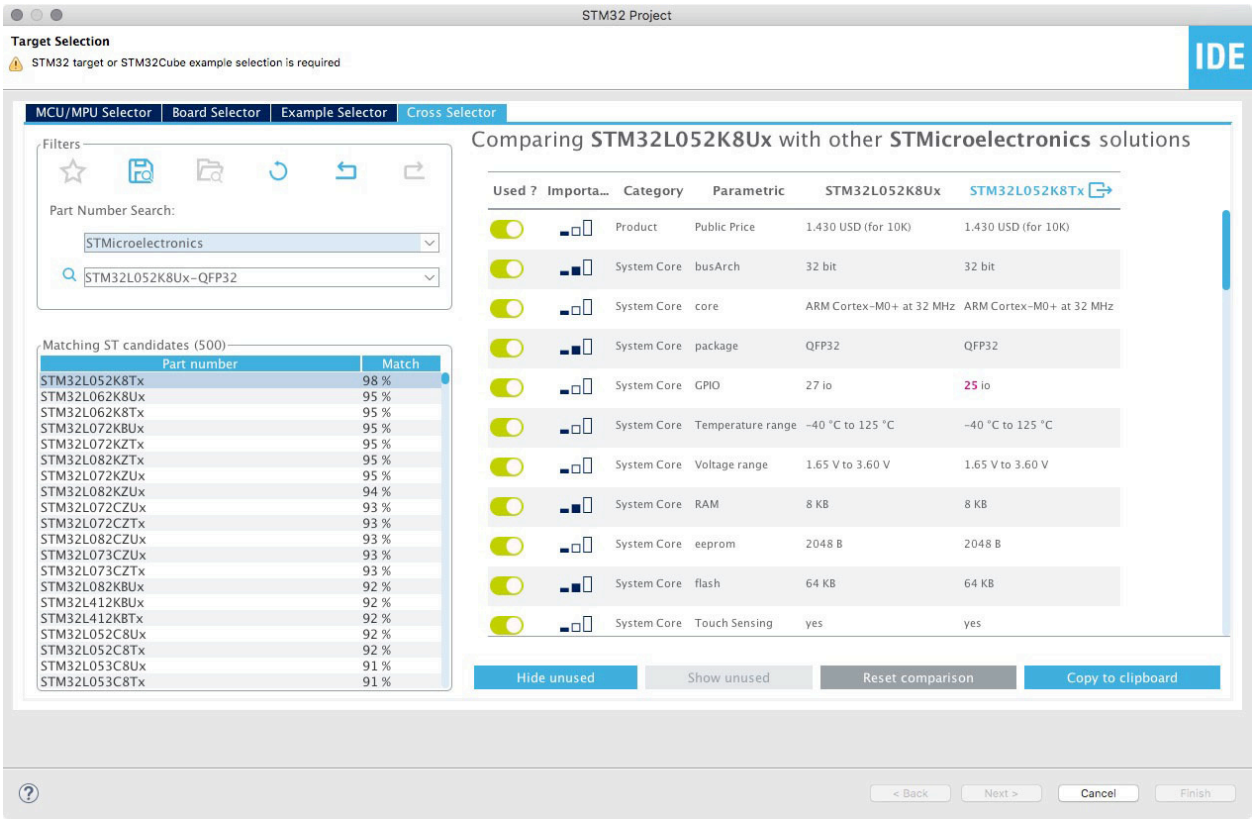


Figure 4.4: CubeMX *Cross Selector*

4.1.2 MCU and Middleware Configuration

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

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

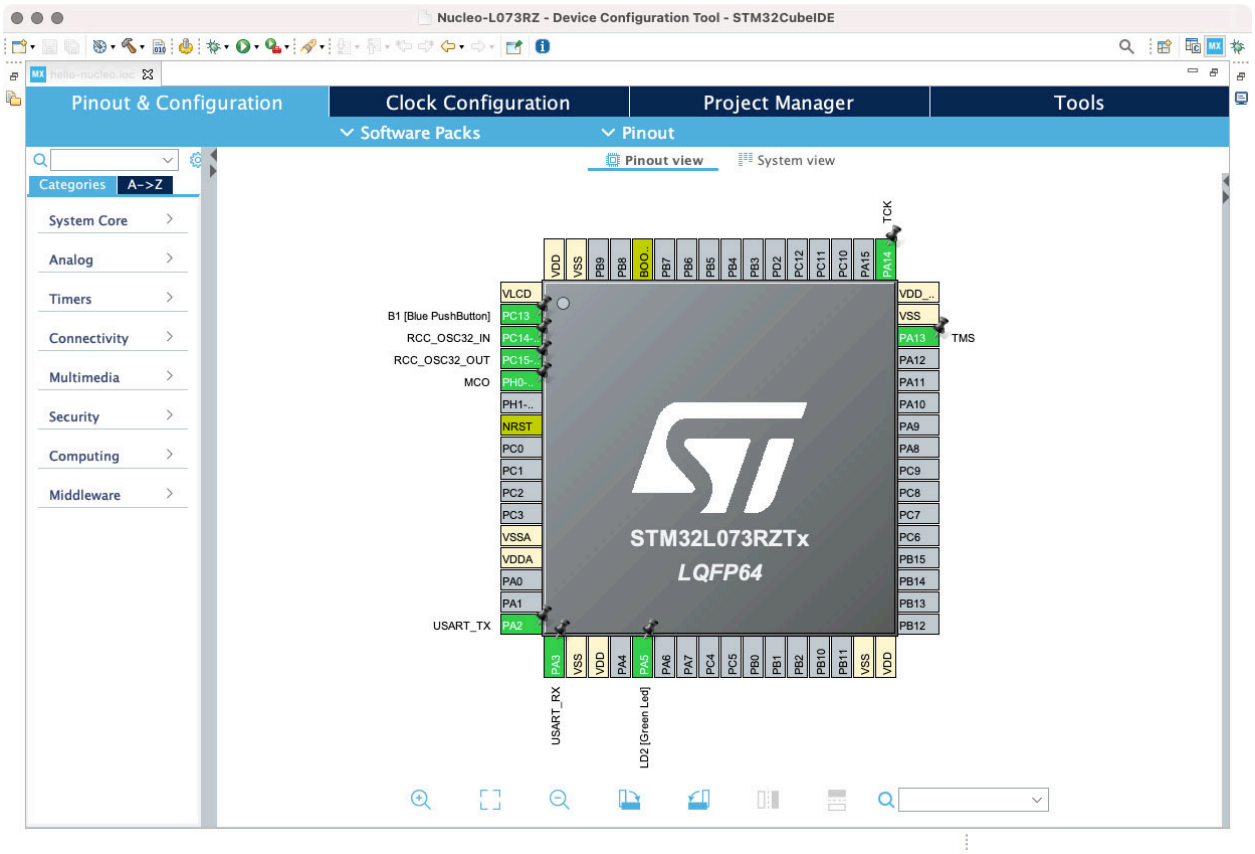


Figure 4.5: CubeMX *Device Configuration Tool* view

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

4.1.2.1 Pinout View & Configuration

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

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

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

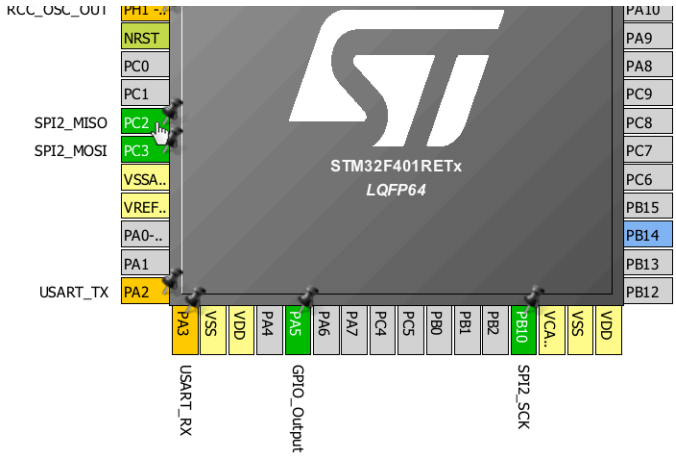


Figure 4.6: Alternate mapping of peripherals

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

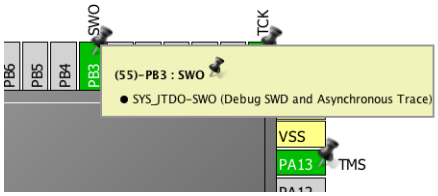


Figure 4.7: Contextual tool-tips help understanding signal usage

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

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

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

pin may simplify the board.

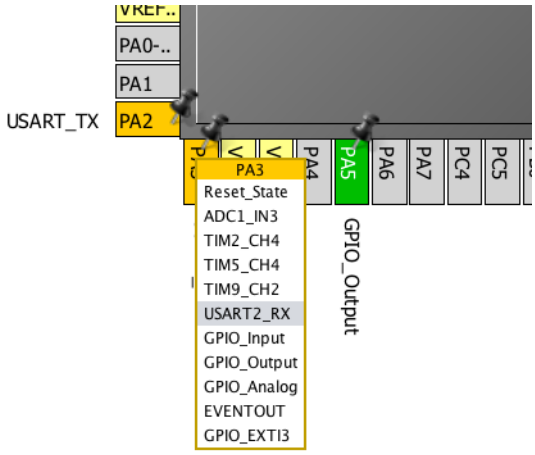


Figure 4.8: Alternate function of a pin

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

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

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

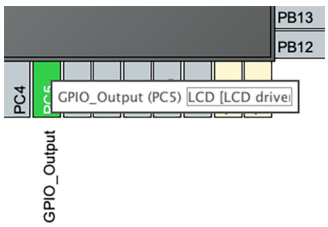


Figure 4.9: How to add custom label to MCU I/Os

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

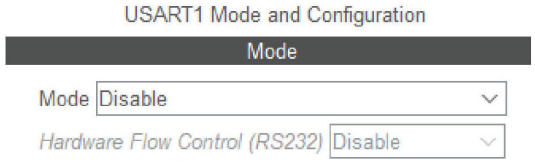
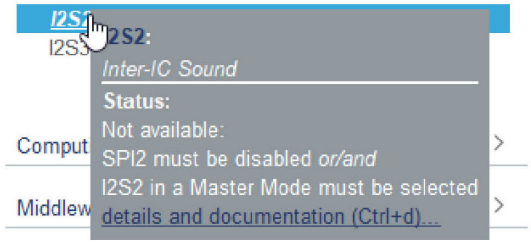
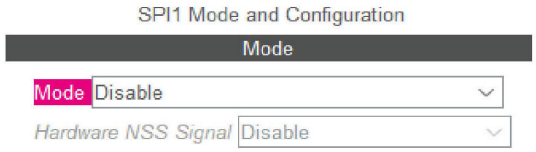
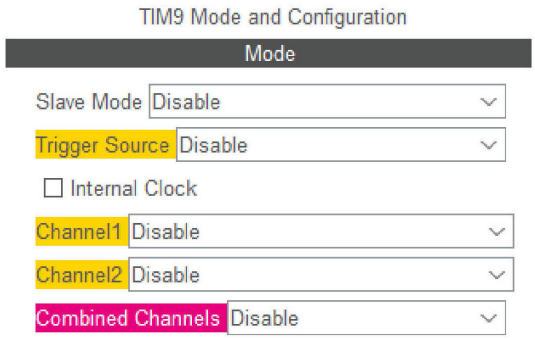
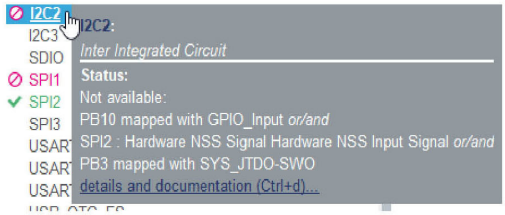
Case	Display	Component status	Corresponding Mode view / Tooltips
1	USART1 (Plain Black Text)	The peripheral is not configured (no mode is set) and all modes are available.	
2	*I2S2* (Gray Italic Text)	Peripheral is not available because some constraints are not solved. See tooltip.	
3	✔ CAN1 ⊗ CAN2	The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

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

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

initialization parameters that will be included in the generated code.

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

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

4.1.2.2 Clock Configuration View

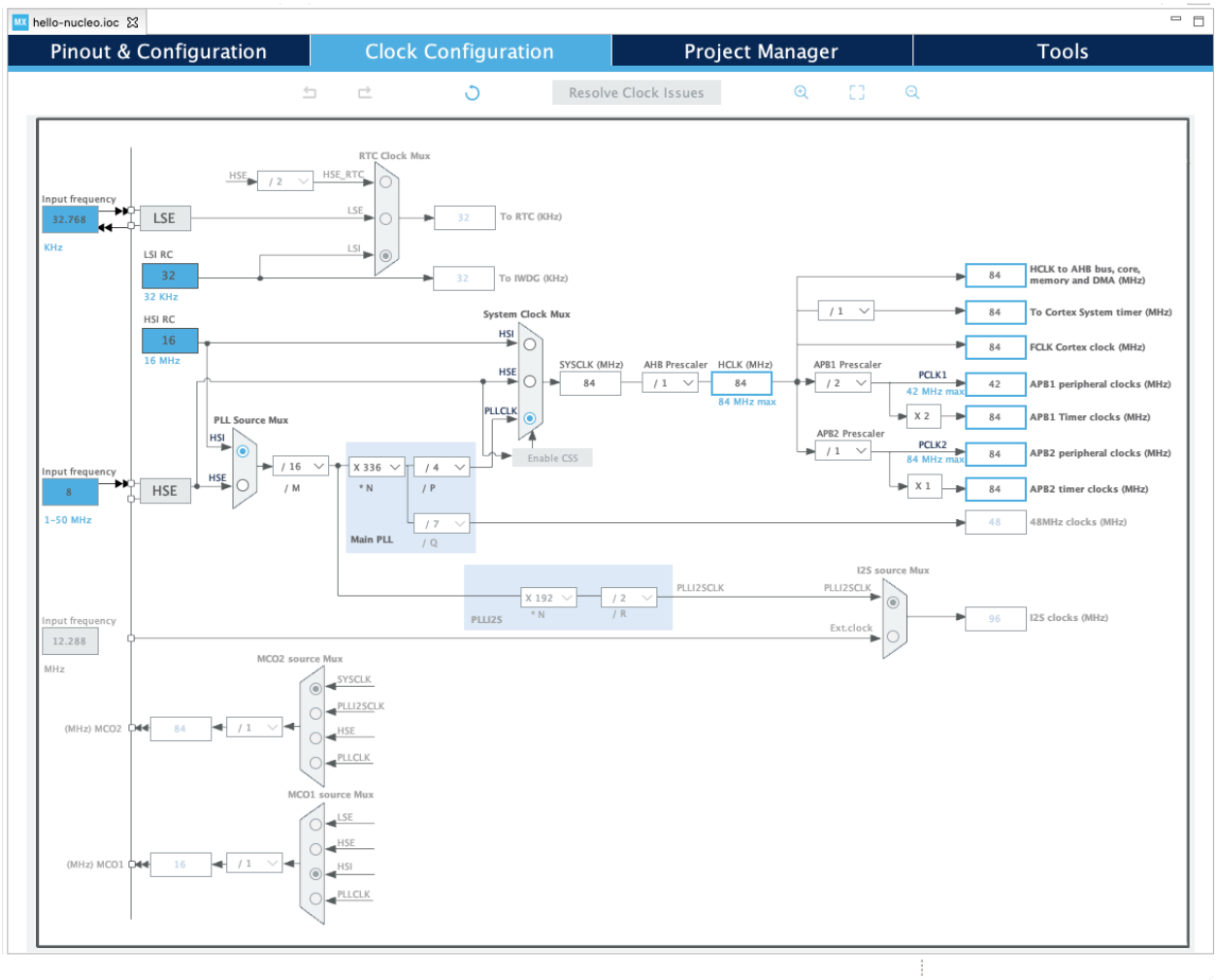


Figure 4.10: The CubeMX clock view

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

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

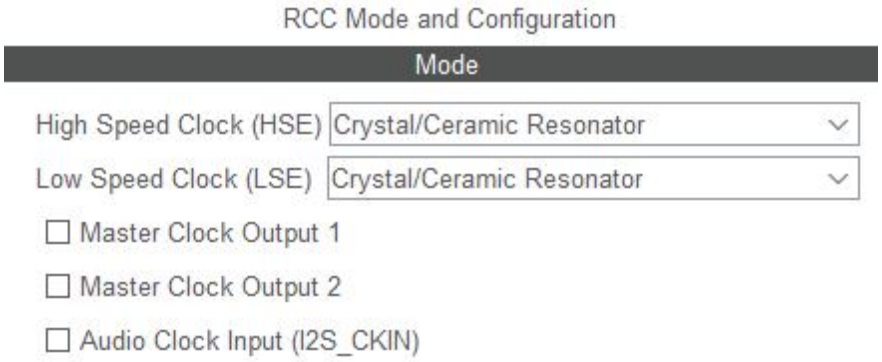


Figure 4.11: HSE and LSE enabling in CubeMX

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

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



Overclocking

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

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

4.1.3 Project Manager

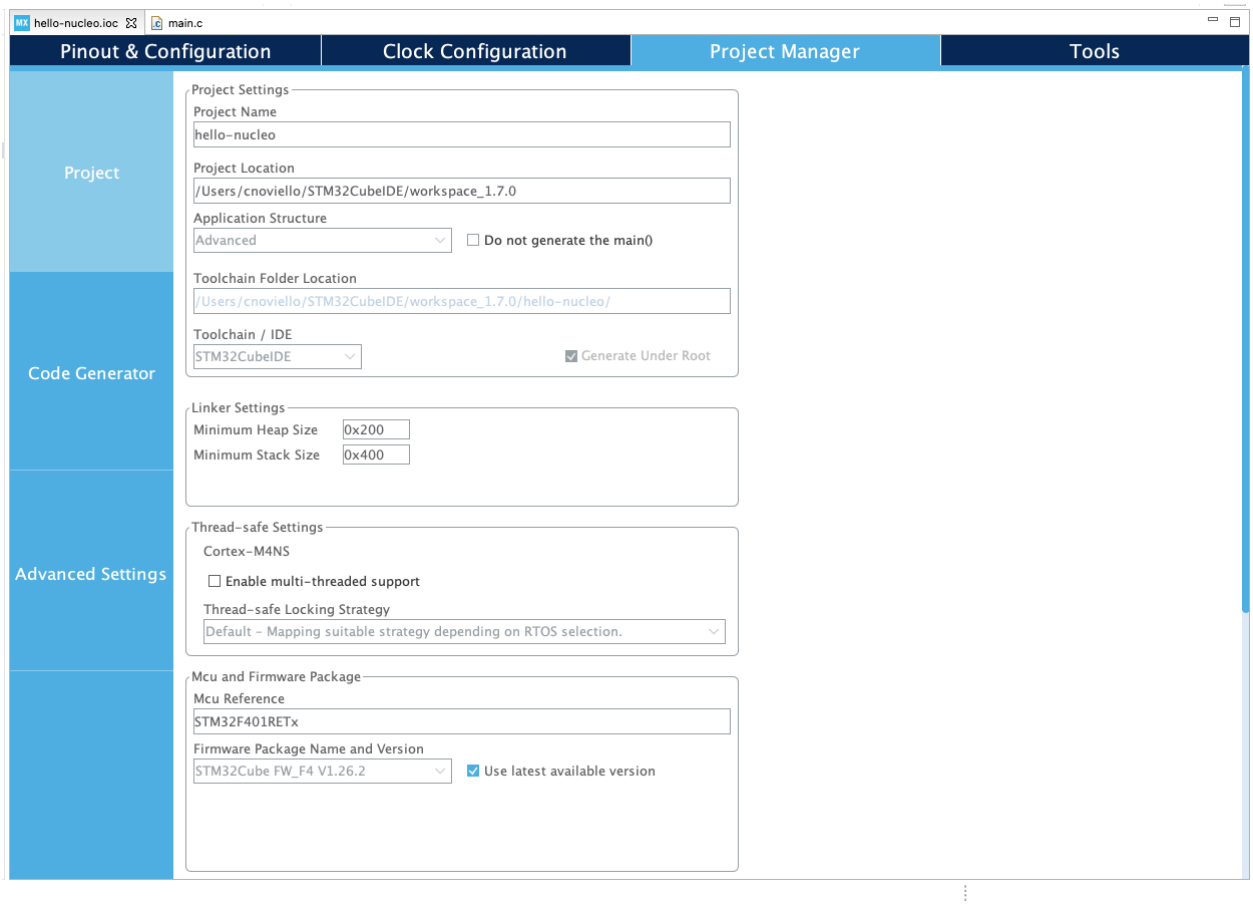


Figure 4.12: The Project Manager view

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

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

What is the Cube Low-Layer API?

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

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

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

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

4.1.4 Tools View

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

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

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

PCC view will be analyzed in a following chapter.

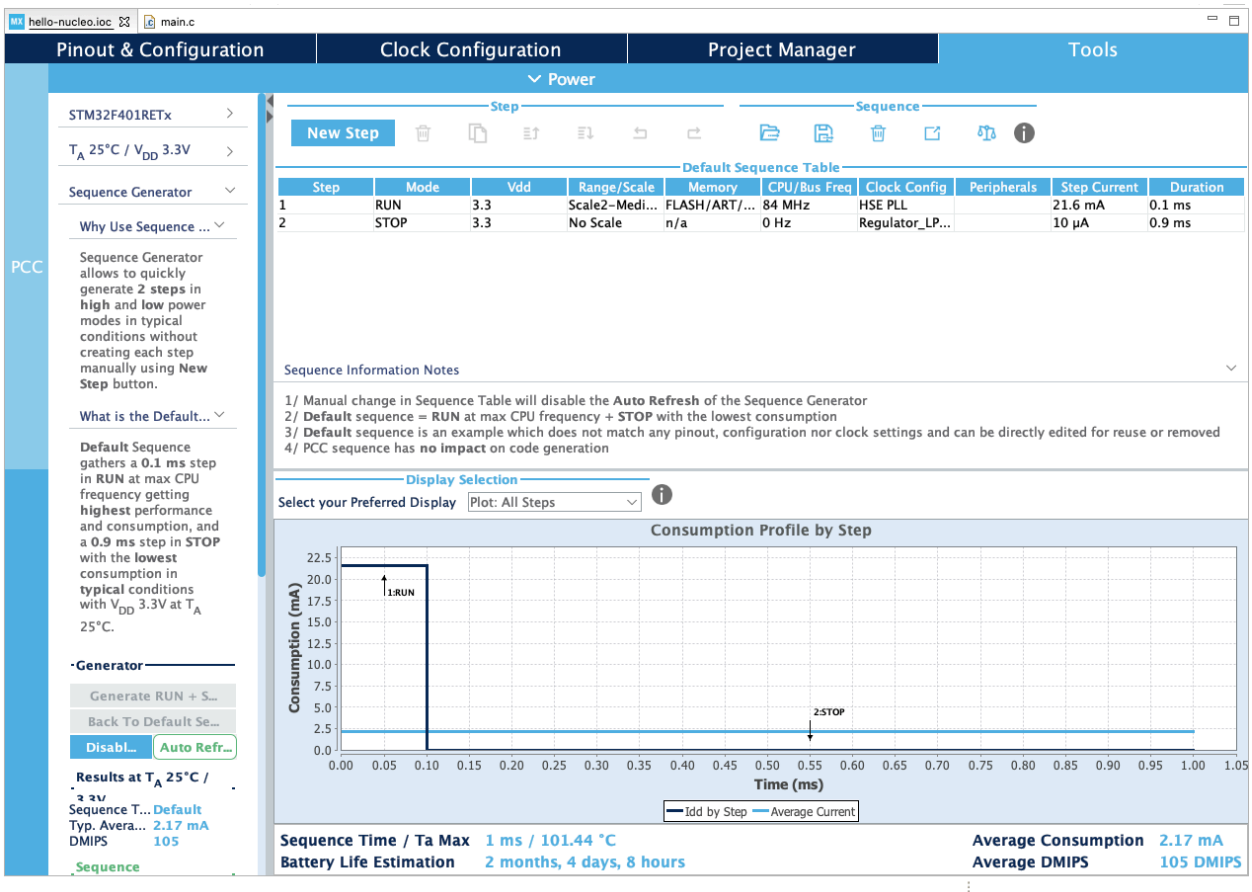


Figure 4.13: The CubeMX Tools view

4.2 Understanding Project Structure

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

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

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

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

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

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

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

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

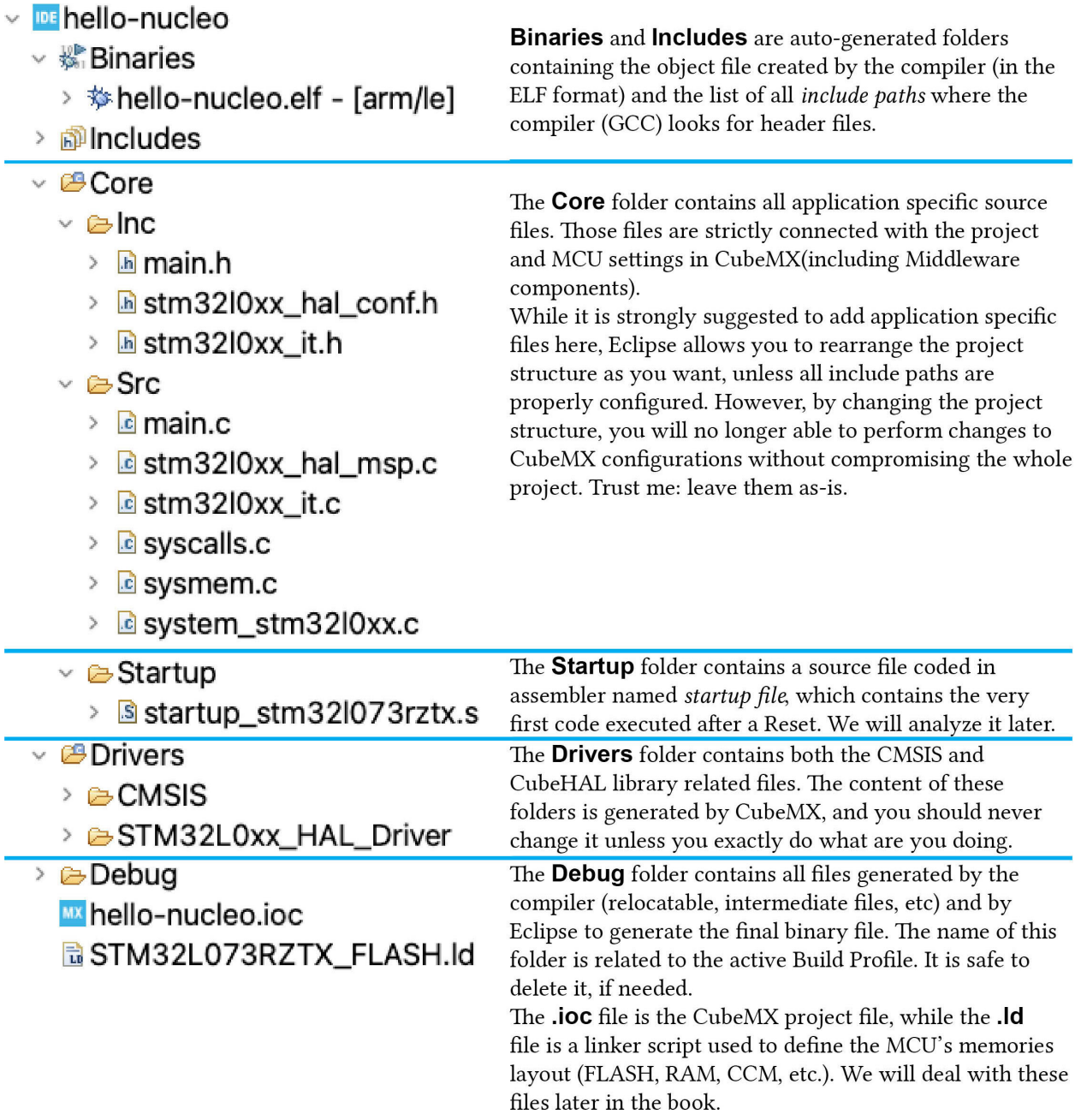


Figure 4.14: The typical structure of CubeMX project

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

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

Filename: `Core/Inc/stm32XXxx_hal_conf.h`

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

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

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

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

Filename: `Core/Src/stm32XXxx_it.c`

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

[11] ch3-hello-nucleo-project

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

Filename: `Core/Startup/startup_stmXXxx.s`

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

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



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

`Core/Src/stm32XXxx_hal_msp.c`

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

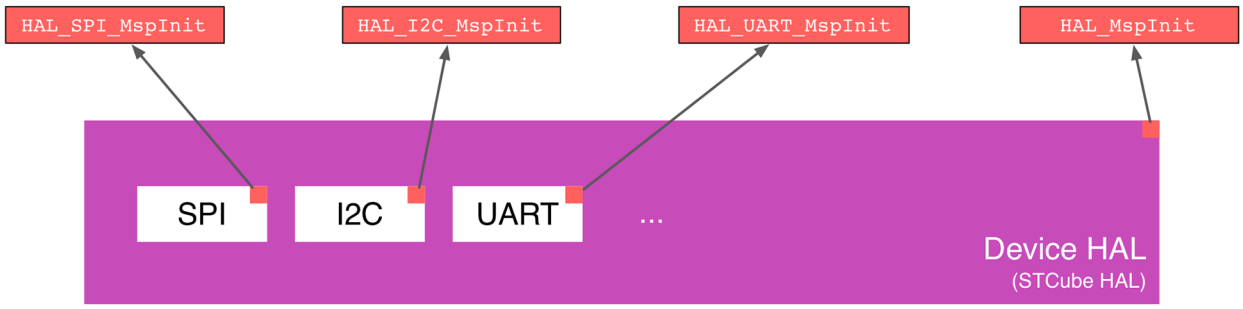


Figure 4.15: The relation between MSP files and the HAL

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

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

Filename: `Core/Src/stm32XXxx_hal_msp.c`

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

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

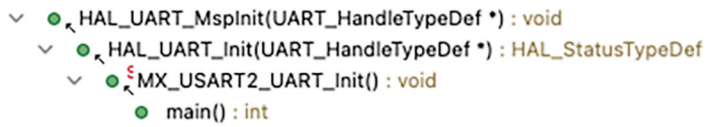


Figure 4.16: The Call Hierarchy of the function `HAL_UART_MspInit()`

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

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

Filename: `Core/Src/main.c`

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

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



Peripheral Initialization and Deinitialization

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

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

`Core/Src/syscalls.c`

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

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

`Core/Src/sysmem.c`

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

`Drivers`

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

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

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

Debug

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

`STM32XXxx_FLASH.ld` and `STM32XXxx_RAM.ld`

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

4.3 Downloading Book Source Code Examples

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

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

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

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

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

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

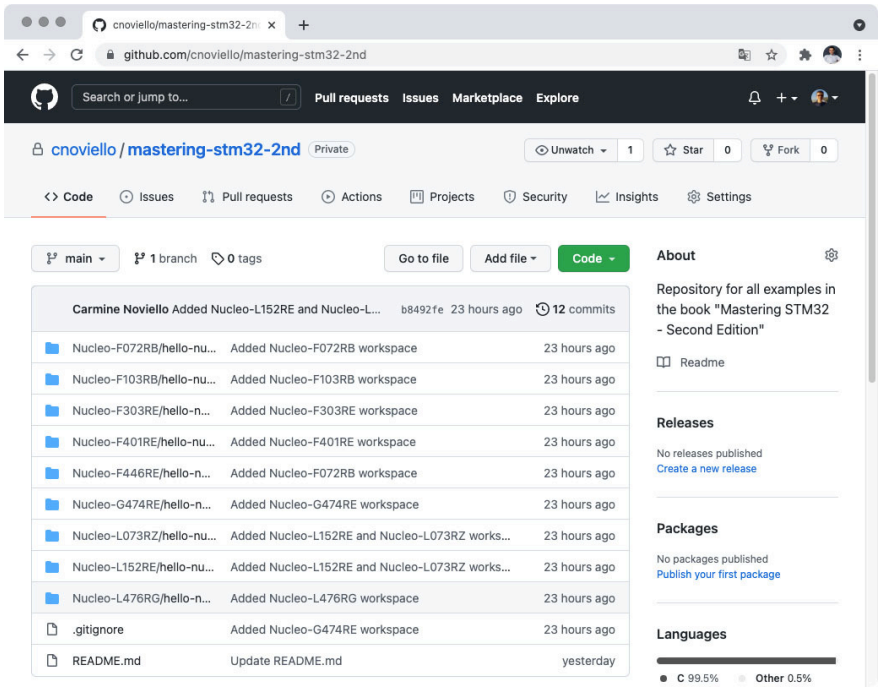


Figure 4.17: The content of the GitHub repository containing all the book examples

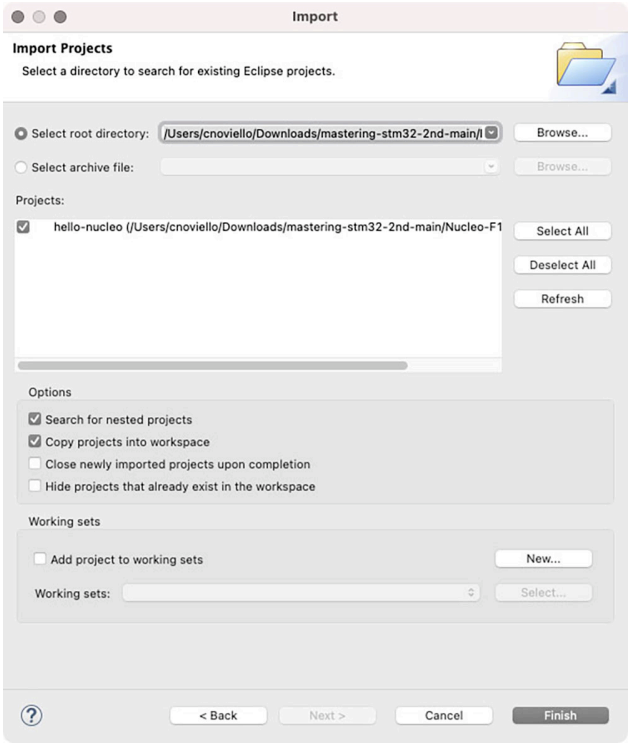


Figure 4.18: Eclipse project import wizard

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

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

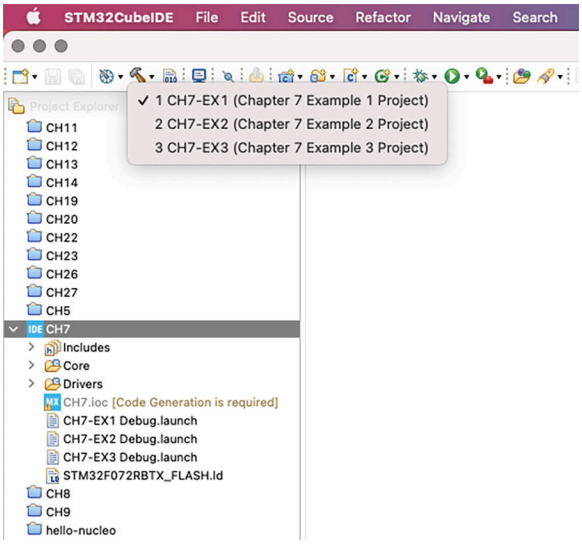


Figure 4.19: How to switch to a different project configuration to select other chapter's examples

4.4 Management of STM32Cube Packages

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

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

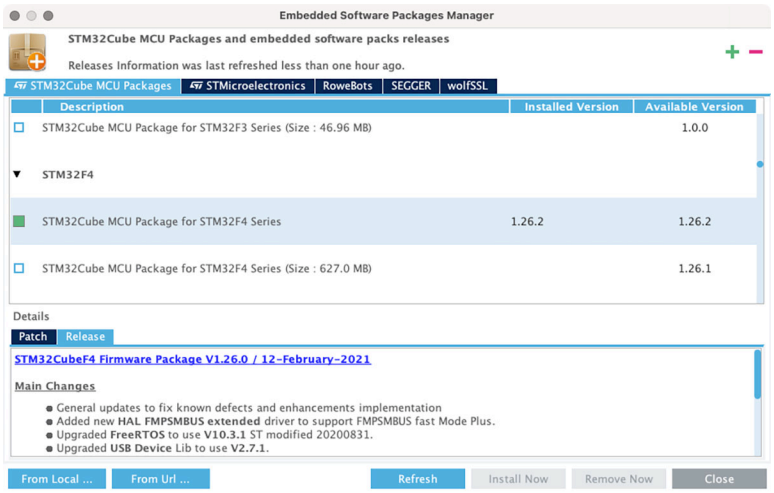


Figure 4.20: The Embedded Software Package Manager

5. Introduction to Debugging

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

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

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

5.1 What is Behind a Debug Session

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

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

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

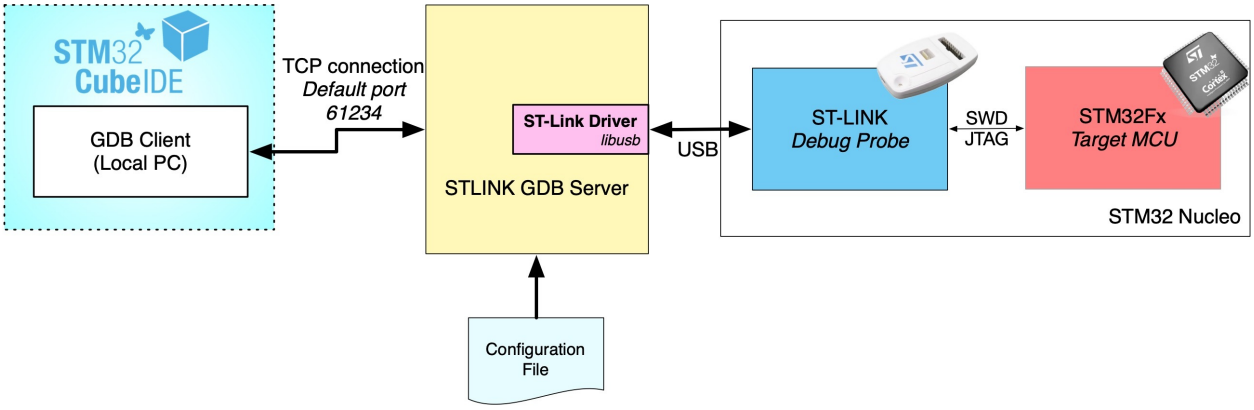


Figure 5.1: How OpenOCD interacts with a Nucleo board

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

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

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

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

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

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

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

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

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

5.2 Debugging With STM32CubeIDE

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

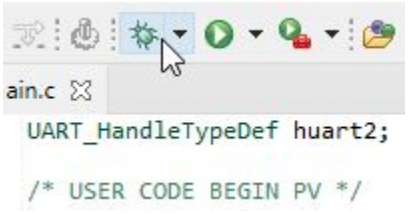


Figure 5.2: The *Debug* icon to start debugging in Eclipse

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

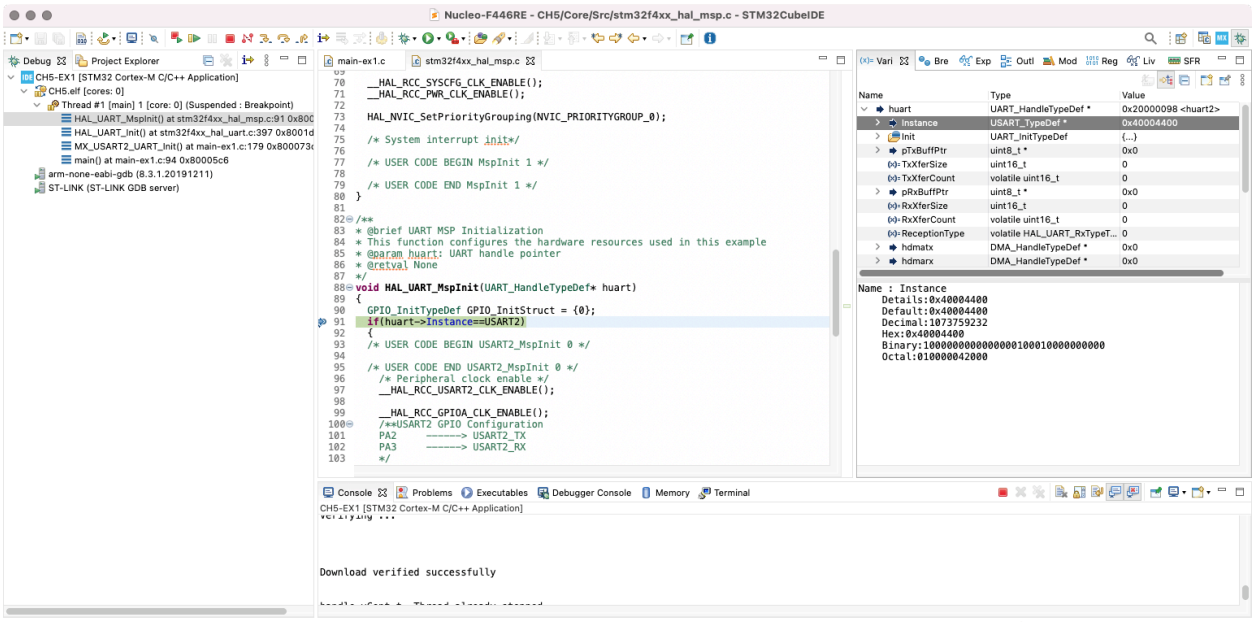


Figure 5.3: The *Debug Perspective*

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

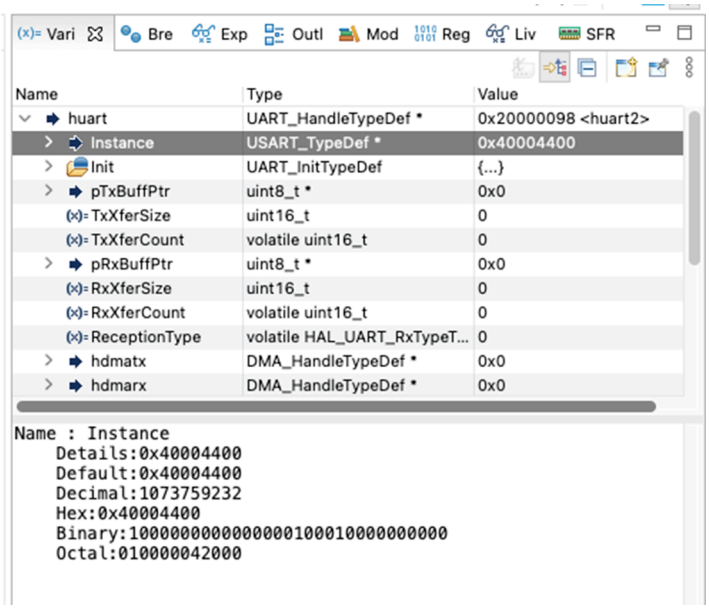


Figure 5.4: The variables inspection pane in the *debug perspective*

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

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

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

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

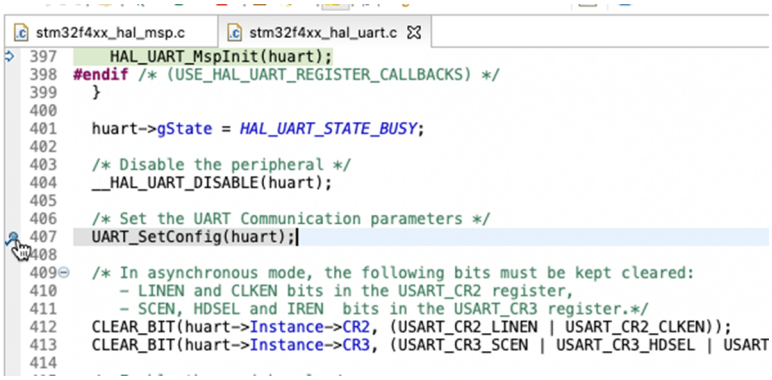


Figure 5.5: How to add a breakpoint at a given line number

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

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

Table 5.2: Most relevant icons on the Eclipse debug toolbar

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

Table 5.2: Most relevant icons on the Eclipse debug toolbar

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

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

5.2.1 Debug Configurations

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

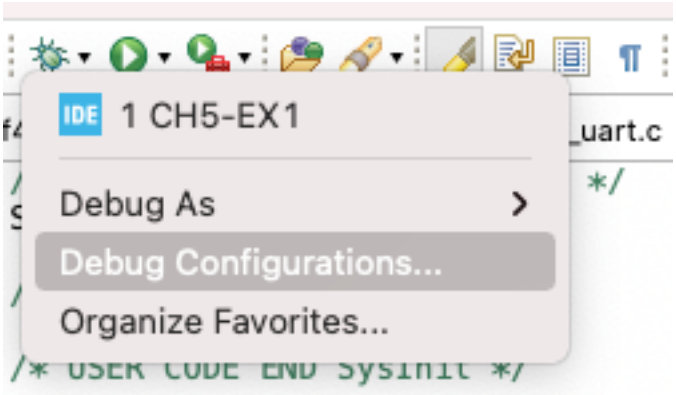


Figure 5.6: Debug Contextual Menu

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

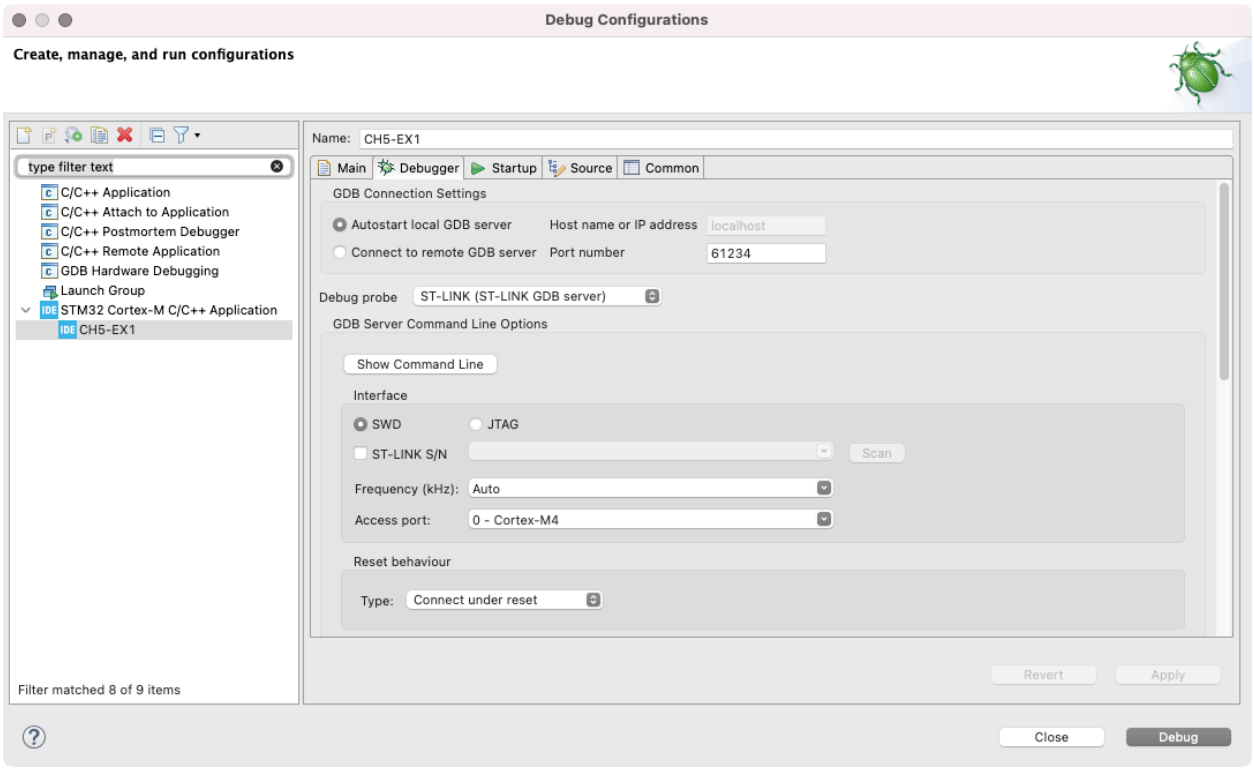


Figure 5.7: Debug Configurations Dialog

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

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

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

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

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

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

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

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

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

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

5.3 I/O Retargeting

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

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

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

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

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

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

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

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

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

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

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

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

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



`printf()` and `float` datatypes.

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

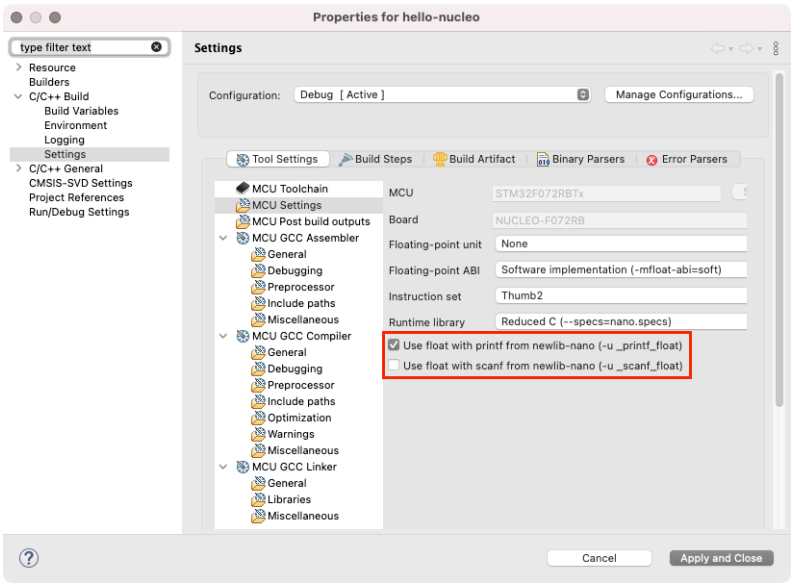


Figure 5.8: How to enable `float` support in `printf()` and `scanf()`

II Diving into the HAL

6. GPIO Management

All STM32 microcontrollers have a variable number of *General Programmable I/Os* (GPIO) . The exact number depends on:

- The type of package chosen (LQFP48, BGA176, and so on).
- The family of microcontroller (F0, F1, etc.).
- The usage of external crystals for HSE and LSE.

GPIOs are the way an MCU communicates with the external world. Every electronic board uses a variable number of I/Os to drive external peripherals (e.g. an LED) or to exchange data through several types of communication peripherals (UART, USB, SPI, etc.).

This chapter starts our journey inside the CubeHAL by looking to one of its simplest modules: `HAL_GPIO`. We have already used several functions from this module in the early examples in this book, but now it is the right time to understand all possibilities offered by a so simple and commonly used peripheral. However, before we can start describing HAL features, it is best to give a quick look at how the STM32 peripherals are mapped to logical addresses and how they are represented inside the HAL library.

6.1 STM32 Peripherals Mapping and HAL *Handlers*

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

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

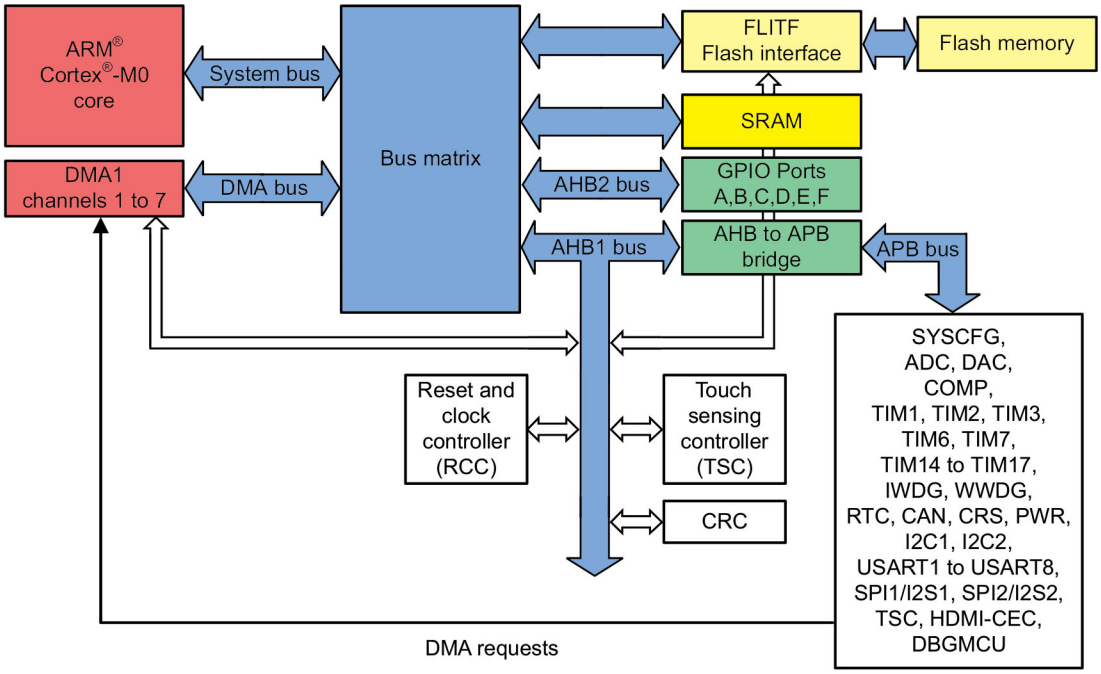


Figure 6.1: Bus architecture of an STM32F072 microcontroller

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

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

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

[2]For some of you the above description may be unclear and too complex to understand. Don't worry and keep reading the next content in this chapter. They will become clear once you reach the chapter dedicated to the DMA.

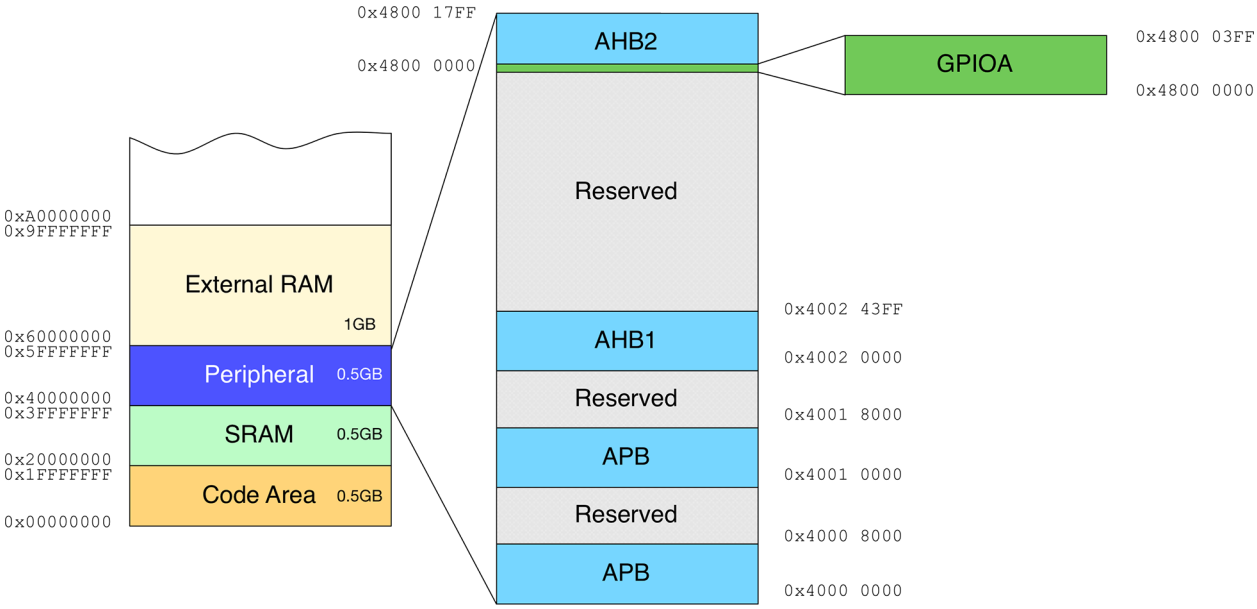


Figure 6.2: Memory map of peripheral regions for an STM32F072 microcontroller

The way this space is organized, and hence how peripherals are mapped, is specific of a given STM32 microcontroller. For example, in an STM32F072 microcontroller the AHB2 bus is mapped to the region ranging from `0x4800 0000` to `0x4800 17FF`. This means that the region is 6144 bytes wide. This region is further divided in several sub-regions, each one corresponding to a specific peripheral. Following the previous example, the `GPIOA` peripheral (which manages all pins connected to the PORT-A) is mapped from `0x4800 0000` to `0x4800 03FF`, which means that it occupies 1KB of aliased peripheral memory. This memory-mapped space is in turn organized depends on the specific peripheral. **Table 6.1**[3] shows the memory layout of a GPIO peripheral.

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

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

Figure 6.3: GPIO MODER register memory layout

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

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**GPIOA_MODER**	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x00	**GPIOx_MODER** (where x = **B..F**)	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x04	**GPIOx_OTYPER** (where x = **A..F**)	Res.		Res.		Res.		Res.		Res.		Res.		Res.		Res.		OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	**GPIOx_OSPEEDR** (where x = **B..F**)	OSPEEDR15[1:0]		OSPEEDR14[1:0]		OSPEEDR13[1:0]		OSPEEDR12[1:0]		OSPEEDR11[1:0]		OSPEEDR10[1:0]		OSPEEDR9[1:0]		OSPEEDR8[1:0]		OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1[1:0]		OSPEEDR0[1:0]	
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	**GPIOA_PUPDR**	PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]		PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
	Reset value	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	**GPIOx_IDR** (where x = **A..F**)	Res.		Res.		Res.		Res.		Res.		Res.		Res.		Res.		IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
	Reset value																	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0x14	**GPIOx_ODR** (where x = **A..F**)	Res.		Res.		Res.		Res.		Res.		Res.		Res.		Res.		ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x18	**GPIOx_BSRR** (where x = **A..F**)	BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0	BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x1C	**GPIOx_LCKR** (where x = **A..B**)	Res.		Res.		Res.		Res.		Res.		Res.		Res.		Res.	LCKK	LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
	Reset value																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x20	**GPIOx_AFRL** (where x = **A.., B**)	AFRLAFR7[3:0]				AFRLAFR6[3:0]				AFRLAFR5[3:0]				AFRLAFR4[3:0]				AFRLAFR3[3:0]				AFRLAFR2[3:0]				AFRLAFR1[3:0]				AFRLAFR0[3:0]			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x24	**GPIOx_AFRH** (where x = **A..B**)	AFRHAFR15[3:0]				AFRHAFR14[3:0]				AFRHAFR13[3:0]				AFRHAFR12[3:0]				AFRHAFR11[3:0]				AFRHAFR10[3:0]				AFRHAFR9[3:0]				AFRHAFR8[3:0]			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x28	**GPIOx_BRR** (where x = **A..F**)	Res.		Res.		Res.		Res.		Res.		Res.		Res.		Res.		BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.1: GPIO peripheral memory map for an STM32F072 microcontroller

A peripheral is controlled by modifying and reading each register of these mapped regions. For example, continuing the example of the `GPIOA` peripheral, to enable PA5 pin as output pin we

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

The following minimal example shows how to use pointers to access to the GPIOA peripheral mapped memory in an STM32F72 MCU.

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

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

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

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

It is important to clarify once again that every STM32 family (F0, F1, etc.) and every member of the given family (STM32F072, STM32F103, etc.) provides its subset of peripherals, which are mapped to specific addresses. Moreover, the way peripherals are implemented differs between STM32-series.

One of the HAL roles is to abstract from the specific peripheral mapping. This is done by defining several *handlers* for each peripheral. A handler is nothing more than a C `struct`, whose references are used to point to real peripheral address. Let us see one of them.

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

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

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

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

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

```
GPIO_TypeDef *GPIOA = 0x48000000;

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

6.2 GPIOs Configuration

As seen before, the HAL is designed so that it abstracts from the specific peripheral memory mapping. But it also provides a general and more user-friendly way to configure the peripheral, without forcing the programmers to know how to configure its registers in detail.

To configure a GPIO we use the `HAL_GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_Init)` function. `GPIO_InitTypeDef` is the C `struct` used to configure the GPIO, and it is defined in the following way:

```
typedef struct {
        uint32_t Pin;
        uint32_t Mode;
        uint32_t Pull;
        uint32_t Speed;
        uint32_t Alternate;
} GPIO_InitTypeDef;
```

This is the role of each field of the `struct`:

- `Pin`: it is the number, starting from 0, of the pins we are going to configure. For example, for PA5 pin it assumes the value `GPIO_PIN_5`[5]. We can use the same `GPIO_InitTypeDef` instance

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

[5] Take note that the `GPIO_PIN_x` is a bit mask, where the *i-th* pin corresponds to the *i-th* bit of a `uint16_t` datatype. For example, the `GPIO_PIN_5` has a value of `0x0020`, which is 32 in base 10.

to configure several pins at once, doing a bitwise `OR` (e.g., `GPIO_PIN_1 | GPIO_PIN_5 | GPIO_PIN_6`).

- `Mode`: it is the operating mode of the pin, and it can assume one of the values in **Table 6.2**. More about this field soon.
- `Pull`: specifies the Pull-up or Pull-Down activation for the selected pins, according to **Table 6.3**.
- `Speed`: defines the pin switching speed, and it can assume a value from a range of constants specific of the given STM32 family. In every STM32 MCU the GPIOs have a maximum switching frequency. Please, consult the datasheet of your MCU in the section *Input/output AC characteristics* of the *Absolute maximum ratings* paragraph.
- `Alternate`: specifies which peripheral to associate to the pin. More about this later.

Table 6.2: Available `GPIO_InitTypeDef.Mode` for a GPIO

Pin Mode	Description
`GPIO_MODE_INPUT`	Input Floating Mode [6]
`GPIO_MODE_OUTPUT_PP`	Output Push Pull Mode
`GPIO_MODE_OUTPUT_OD`	Output Open Drain Mode
`GPIO_MODE_AF_PP`	Alternate Function Push Pull Mode
`GPIO_MODE_AF_OD`	Alternate Function Open Drain Mode
`GPIO_MODE_ANALOG`	Analog Mode
`GPIO_MODE_IT_RISING`	External Interrupt Mode with Rising edge trigger detection
`GPIO_MODE_IT_FALLING`	External Interrupt Mode with Falling edge trigger detection
`GPIO_MODE_IT_RISING_FALLING`	External Interrupt Mode with Rising/Falling edge trigger detection
`GPIO_MODE_EVT_RISING`	External Event Mode with Rising edge trigger detection
`GPIO_MODE_EVT_FALLING`	External Event Mode with Falling edge trigger detection
`GPIO_MODE_EVT_RISING_FALLING`	External Event Mode with Rising/Falling edge trigger detection

Table 6.3: Available `GPIO_InitTypeDef.Pull` modes for a GPIO

Pin Mode	Description
`GPIO_NOPULL`	No Pull-up or Pull-down activation
`GPIO_PULLUP`	Pull-up activation
`GPIO_PULLDOWN`	Pull-down activation

6.2.1 GPIO Mode

STM32 MCUs provide a flexible GPIOs management. **Figure 6.4**[7] shows the hardware structure of a single I/O of an STM32F072 microcontroller.

[6] During and just after reset, the alternate functions are not active, and all the I/O ports are configured in *Input Floating* mode.
[7] The figure is taken from the ST STM32F072 Reference Manual (https://bit.ly/2XzzJ3s).

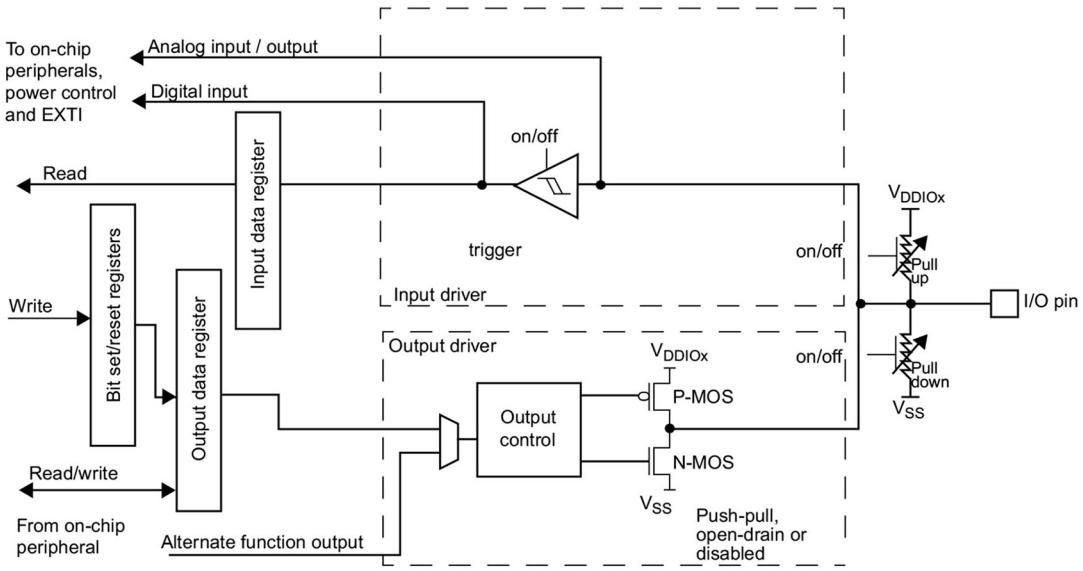


Figure 6.4: Basic structure of an I/O port bit

Depending on the GPIO `GPIO_InitTypeDef.Mode` field, the MCU changes the way the hardware of an I/O works. Let us have a look at the main modes.

When the I/O is configured as `GPIO_MODE_INPUT`:

- The output buffer is disabled.
- The Schmitt trigger input is activated.
- The pull-up and pull-down resistors are activated depending on the value of the `Pull` field.
- The data present on the I/O pin are sampled into the input data register every AHB clock cycle.
- A read access to the input data register provides the I/O state.

When the I/O port is programmed as `GPIO_MODE_ANALOG`:

- The output buffer is disabled.
- The Schmitt trigger input is deactivated, providing zero consumption for every analog value of the I/O pin.
- The weak pull-up and pull-down resistors are disabled by hardware.
- Read access to the input data register gets the value `0`.

When the I/O port is programmed as output:

- The output buffer is enabled as follow:
  - if mode is `GPIO_MODE_OUTPUT_OD`: A `0` in the Output register (ODR) activates the N-MOS whereas a `1` leaves the port in Hi-Z (the P-MOS is never activated);
  - if mode is `GPIO_MODE_OUTPUT_PP`: A `0` in the ODR activates the N-MOS whereas a `1` activates the P-MOS.

- The Schmitt trigger input is activated.
- The pull-up and pull-down resistors are activated depending on the value of the `Pull` field.
- The data present on the I/O pin are sampled into the input data register every AHB clock cycle.
- A read access to the input data register gets the I/O state.
- A read access to the output data register gets the last written value.

When the I/O port is programmed as alternate function:

- The output buffer can be configured in open-drain or push-pull mode.
- The output buffer is driven by the signals coming from the peripheral (transmitter enable and data).
- The Schmitt trigger input is activated.
- The weak pull-up and pull-down resistors are depending on the value of the `Pull` field.
- The data present on the I/O pin are sampled into the input data register every AHB clock cycle.
- A read access to the input data register gets the I/O state.

The GPIO modes `GPIO_MODE_EVT_*` are related to sleep modes. When an I/O is configured to work in one of these modes, the CPU will be woken up (when placed in sleep mode with a `WFE` instruction) if the corresponding I/O is triggered, without generating the corresponding interrupt (more about this topic in Chapter 19). The GPIO modes `GPIO_MODE_IT_*` modes are related to interrupts management, and they will be analyzed in the next chapter.

However, keep in mind that this implementation scheme can vary between the STM32-families, especially for the low-power series. Always refer to the reference manual of your MCU, which exactly describes I/O modes and their impact on the MCU working and power consumption.

It is also important to remark that this flexibility represents an advantage for the hardware design too. For example, if you need external pull-up resistors in your application there is no need to use external and dedicated ones, since the corresponding GPIOs can be configured setting `GPIO_InitTypeDef.Mode = GPIO_MODE_OUTPUT_PP` and `GPIO_InitTypeDef.Pull = GPIO_PULLUP`. This saves space on the PCB and simplifies the BOM.

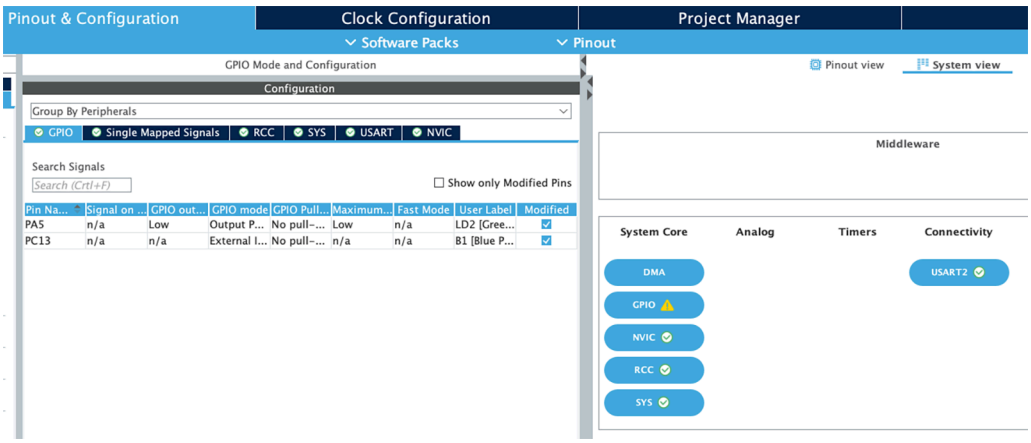


Figure 6.5: Pin Configuration dialog can be used to configure I/O mode

I/O mode can be eventually configured using the CubeMX tool, as shown in **Figure 6.5**. Pin Configuration dialog can be reached inside the *Configuration* view, clicking on the **GPIO** button.

6.2.2 GPIO Alternate Function

Most of GPIOs have "alternate functions", that is they can be used as I/O pin for at least one internal peripheral. However, keep in mind that an I/O can be associated to only one peripheral at a time.

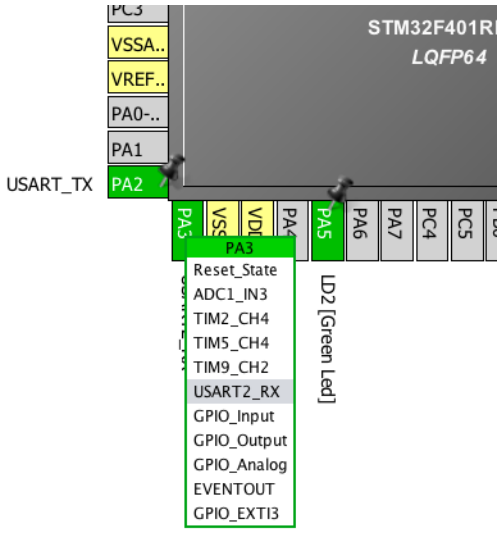


Figure 6.6: CubeMX can be easily used to discover alternate functions of an I/O

To discover which peripherals can be bound to an I/O, you can refer to the MCU datasheet or simply by using the CubeMX tool. Clicking on a pin in the `Pin View` causes a pop-up menu to appear. In this menu we can set the wanted alternate function. For example, in **Figure 6.6** you can see that `PA3` can be used as `USART2_RX` (that is, it can be used as RX pin for USART/UART2 peripheral, and this is possible for every STM32 MCU with LQFP64 package). CubeMX will automatically generate the right initialization code for us, as shown below:

```
/* Configure GPIO pins : PA2 PA3 */
GPIO_InitStruct.Pin = GPIO_PIN_2|GPIO_PIN_3;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
GPIO_InitStruct.Alternate = GPIO_AF1_USART2;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

F1

Those of you working on an STM32F1 MCU will notice that the `GPIO_InitTypeDef.Alternate` field is missed in the CubeF1 HAL. This happens because STM32F1 MCUs have a less flexible way to

define alternate functions of a pin. While other STM32 microcontrollers define the possible alternate functions at GPIO level (by configuring dedicated registers `GPIOx_AFRL` and `GPIOx_AFRH`), allowing to have up to sixteen different alternate functions associated to a pin (this only happens in packages with high pin count), GPIOs of an STM32F1 MCU have really limited remapping capabilities. For example, in an STM32F103RB MCU only the USART3 can have two couple of pins that can be used as peripheral I/O alternatively. Usually, two dedicated peripheral registers, `AFIO_MAPR` and `AFIO_MAPR2` "remap" signal I/Os of those peripherals allowing this operation.

This is essentially the reason why that field is not available in CubeF1 HAL.

6.3 Driving a GPIO

CubeHAL provides four manipulation routines to read, change and lock the state of an I/O.
To read the status of an I/O we can use the function:

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
```

which accepts the GPIO descriptor and the pin number. It returns `GPIO_PIN_RESET` when the I/O is low or `GPIO_PIN_SET` when high. Conversely, to change the I/O state, we have the function:

```
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)
```

which accepts the GPIO descriptor, the pin number and the desired state. If we want to simply invert the I/O state, then we can use this convenient routine:

```
void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin).
```

Finally, one feature of the GPIO peripheral is that we can lock the configuration of an I/O. Any subsequent attempt to change its configuration will fail, until a reset occurs. To lock a pin configuration, we can use this routine:

```
HAL_StatusTypeDef HAL_GPIO_LockPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin).
```

6.4 De-initialize a GPIO

It is possible to set a GPIO pin to its default reset status (that is in *Input Floating Mode*). The function:

```
void HAL_GPIO_DeInit(GPIO_TypeDef  *GPIOx, uint32_t GPIO_Pin).
```

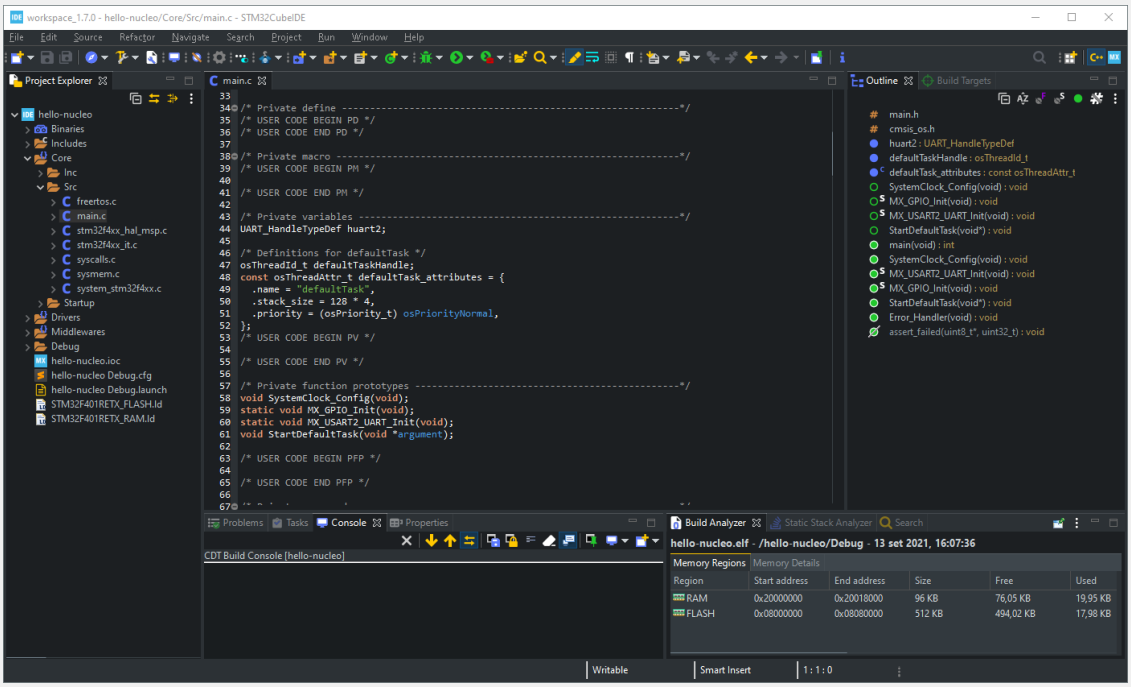
does this job automatically for us.

This function comes in handy if we no longer need a given peripheral, or to avoid waste of power when the CPU goes in sleep mode.

Eclipse Intermezzo

It is possible to heavily customize the Eclipse interface by installing custom themes. A theme essentially allows to change the appearance of the Eclipse user interface. This may seem a non-essential feature, but nowadays a lot of programmers prefer to customize colors, fonts type and size and so on of their favorite development environment. That is one of the success reasons of some minimal yet highly customizable source code editors, like TextMate, Sublime Text or Visual Studio Code.

Instead of customizing the interface with regular Eclipse settings, there are several theme packs available for Eclipse on the Eclipse Marketplace. This author prefers dark themes, instead of light ones. A recent theme pack is the Darkest Dark Theme[a] included with DevStyle theming pack. The STM32CubeIDE interface will change quite a lot, with a UX similar to recent releases of Android Studio, as you can see in the following capture.



[a] https://marketplace.eclipse.org/content/darkest-dark-theme-devstyle

7. Interrupts Management

Hardware management is all about dealing with asynchronous events. The most of these come from hardware peripherals. For example, a timer reaching a configured period value, or a UART that warns about the arrival of data. Others are originated by the "world outside" our board. For example, the user presses that damned switch that causes your board to hang, and you are going to spend a whole day understanding what is going wrong.

All microcontrollers provide a feature called *interrupts*. An interrupt is an asynchronous event that causes stopping the execution of the current code on a priority basis (the more important the interrupt is, the higher its priority; this will cause that a lower-priority interrupt is suspended). The code that services the interrupt is called *Interrupt Service Routine* (ISR).

Interrupts are a source of multiprogramming: the hardware knows about them and it is responsible of saving the current execution context (that is, the stack frame, the current *Program Counter* (PC) and few other things) before switching to the ISR. Interrupts are exploited by *Real Time Operating Systems* (RTOS) to introduce the notion of *tasks*: without the help by the hardware it is impossible to have a true preemptive system, which allows switching between several execution contexts without irreparably losing the current execution flow.

Interrupts can originate both by the hardware and the software itself. ARM architecture distinguishes between the two types: *interrupts* originate by the hardware, *exceptions* by the software (e.g., an access to invalid memory location). In ARM terminology, an interrupt is a type of exception.

Cortex-M processors provide a unit dedicated to exceptions management. This is called *Nested Vectored Interrupt Controller* (NVIC) and this chapter is about programming this fundamental hardware component. However, here we will deal only with interrupts management. Exceptions handling will be treated in Chapter 24 about advanced debugging.

7.1 NVIC Controller

NVIC is a dedicated hardware unit inside the Cortex-M based microcontrollers that is responsible of the exceptions handling. **Figure 7.1** shows the relation between the NVIC unit, the Processor Core and peripherals. Here we have to distinguish two types of peripherals: those external to the Cortex-M core, but internal to the STM32 MCU (e.g., timers, UARTS, and so on), and those peripherals external to the MCU at all. The source of the interrupts coming from the last kind of peripherals are the MCU I/O, which can be both configured as general purpose I/O (e.g., a tactile switch connected to a pin configured as input) or to drive an external advanced peripheral (e.g., I/Os configured to exchange data with an *ethernet phyther* through the RMII interface). A dedicated programmable controller, named *External Interrupt/Event Controller* (EXTI), is responsible of the interconnection between the external I/O signals and the NVIC controller, as we will see next.

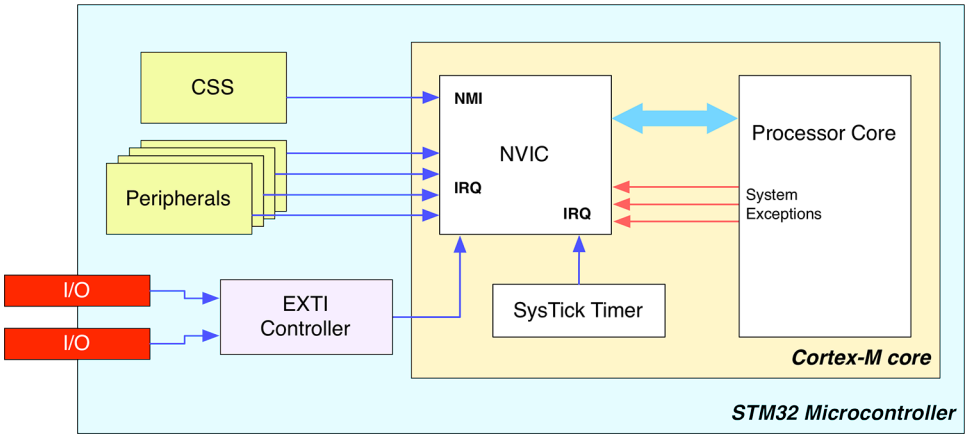


Figure 7.1: the relation between the NVIC controller, the Cortex-M core and the STM32 peripherals

As stated before, ARM distinguishes between system exceptions, which originate inside the CPU core, and hardware exceptions coming from external peripherals, also called *Interrupt Requests* (IRQ). Programmers manage exceptions using specific ISRs, which are coded at higher level (most often using C language). The processor knows where to locate these routines thanks to an indirect table containing the addresses in memory of Interrupt Service Routines. This table is commonly called *vector table*, and every STM32 microcontrollers defines its own. Let us analyze this in depth.

7.1.1 Vector Table in STM32

All Cortex-M processors reserve a fixed set of fifteen exceptions common to all Cortex-M families. However, not all these exceptions are currently defined (they are marked as **RESERVED** in the ARM official documentation) and just a subset is available in Cortex-M0/0+ cores. We already encountered them in Chapter 1. Here, you can find the same table (**Table 7.1**) for your convenience. It is a good idea to take a quick look at these exceptions (we will study fault exceptions better in Chapter 24 dedicated to advanced debugging).

- **Reset**: this exception is raised just after the CPU resets. Its handler is the real entry point of the running firmware. In an STM32 application all starts from this exception. The handler contains some assembly-coded functions designed to initialize the execution environment, such as the main stack, the `.bss` area, etc. Chapter 22 dedicated to the booting process will explain in depth.
- **NMI**: this is a special exception, which has the highest priority after the *Reset* one. Like the *Reset* exception, it cannot be masked, and it can be associated to critical and non-deferrable activities. In all STM32 microcontrollers it is linked to the *Clock Security System* (CSS). CSS is a self-diagnostic peripheral that detects the failure of the external clock, called HSE. If this happens, HSE is automatically disabled (this means that the internal HSI is automatically enabled) and an NMI interrupt is raised to inform the software that something is wrong with the HSE. More about this feature in Chapter 10.
- **Hard Fault**: is the generic fault exception, and hence related to software interrupts. When the other fault exceptions are disabled, it acts as a collector for all types of exceptions (e.g., a

memory access to an invalid location raised the Hard Fault exceptions if the Bus Fault one is not enabled).

- **Memory Management Fault**[1]: it occurs when executing code attempts to access an illegal location or violates a rule of the Memory Protection Unit (MPU). More about this in Chapter 20.
- **Bus Fault**[1]: it occurs when AHB interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch, or *data abort* if it is a data access). Can also be caused by other illegal accesses (e.g., an access to a non-existent SRAM memory location).
- **Usage Fault**[1]: it occurs when there is a program error such as an illegal instruction, alignment problem, or attempt to access a non-existent co-processor.
- **SVCCall**: this is not a fault condition, and it is raised when the *Supervisor Call* (SVC) instructions is called. This is used by *Real Time Operating Systems* to execute instructions in privileged state (a task needing to execute privileged operations executes the SVC instruction, and the OS performs the requested operations - this is the same behavior of a system call in other OSes).
- **Debug Monitor**[1]: this exception is raised when a software debug event occurs while the processor core is in Monitor Debug-Mode. It is also used as exception for debug events like breakpoints and watchpoints when software-based debug solution is used.
- **PendSV**: this is another exception related to RTOS. Unlike the SVCall exception, which is executed immediately after an SVC instruction is executed, the PendSV can be delayed. This allows the RTOS to complete tasks with higher priorities.
- **SysTick**: this exception is also usually related to RTOS activities. Every RTOS needs a timer to periodically interrupt the execution of current code and to switch to another task. All STM32 microcontrollers provide a *SysTick* timer, internal to the Cortex-M core. Even if every other timer may be used to schedule system activities, the presence of a dedicated timer ensures portability among all STM32 families (due to optimization reasons related to the internal die of the MCU, not all timers could be available as external peripheral). Moreover, even if we aren't using an RTOS in our firmware, it is important to keep in mind that the ST CubeHAL uses the *SysTick* timer to perform internal time-related activities (**and it assumes that the SysTick timer is configured to generate an interrupt every 1ms**).

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

Where can we find the list of usable interrupts for a given STM32 microcontrollers? The datasheet of that MCU is certainly the main source about available interrupts. However, we can simply refer to the *vector table* provided by ST in its HAL. This table is defined inside the startup file for our MCU, the assembly file ending with `.s` inside the `Core/Startup` folder of our project (for example, for an STM32F446RET MCU the file name is `startup_stm32f446retx.s`). Opening that file, we can find the whole vector table for that MCU, starting about at line 128 (see example in Chapter 4).

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

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

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

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

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

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

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

Table 7.1: Cortex-M exception types

Even if the *vector table* contains the addresses of the handler routines (it is, in fact, an indirect table), the Cortex-M core needs a way to find the *vector table* in memory. By convention, the *vector table* starts at the hardware address `0x0000 0000` in all Cortex-M based processors.
If our firmware is designed so that the *vector table* resides in the internal flash memory (a quite common scenario), then the *vector table* will be placed starting from the `0x0800 0000` address in all STM32 MCUs. However, in Chapter 1 we saw that the `0x0800 0000` address is automatically aliased

to `0x0000 0000` when the CPU boots up[2].

Figure 7.2 shows how the *vector table* is organized in memory. The first entry of this array is the address of the *Main Stack Pointer* (MSP) inside the SRAM. Usually, this address corresponds to the end of the SRAM, that is its base address + its size (more about memory layout of an STM32 application in Chapter 20). Starting from the second entry of this table, we can find all exceptions and interrupts handler. This means that the *vector table* has a length equal to 48 for Cortex-M0/0+ based microcontrollers and a length equal to 256 for Cortex-M3/4/7.

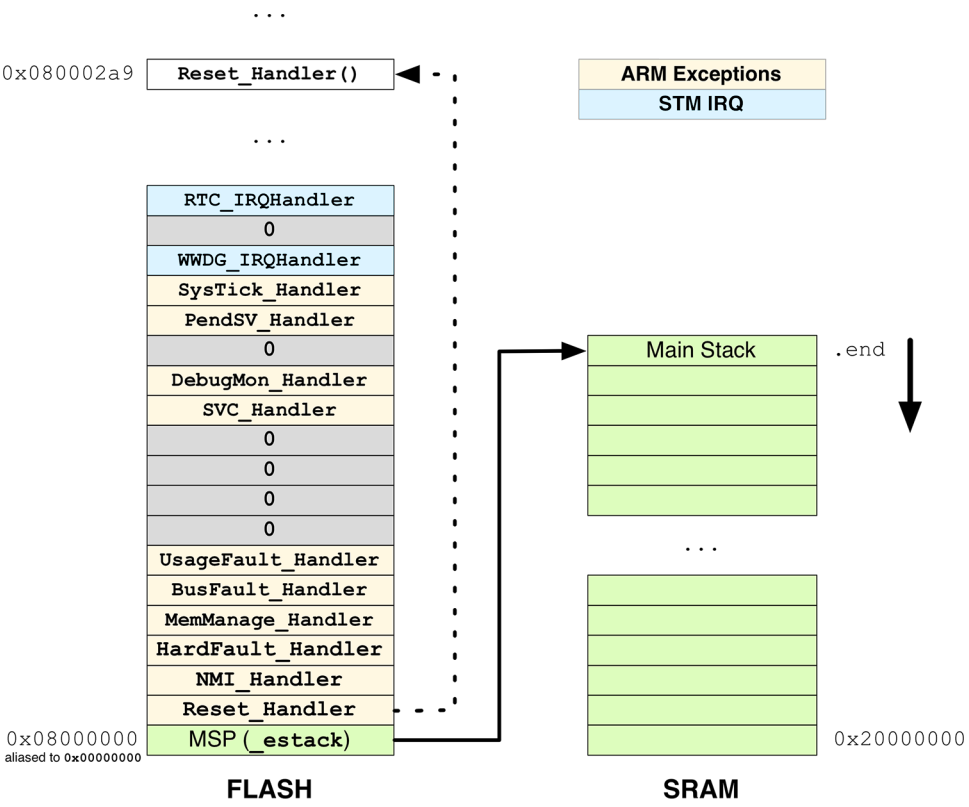


Figure 7.2: The minimal layout of the *vector table* in an STM32 MCU based on a Cortex-M3/4/7 core

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

1. The name of the exception handlers is just a convention, and you are totally free to rename them if you like a different one. They are just *symbols* (like variables and functions inside a program). However, keep in mind that the CubeMX software is designed to generate ISR with those names, which are an ST convention. So, you have to rename the ISR name too.
2. As said before, the *vector table* must be placed at the beginning of the flash memory, where the processor expects to find it. This is a GCC Linker job that places the *vector table* at the beginning of the flash data during the generation of the *absolute file*, which is the binary file

[2]Apart from the Cortex-M0, the rest of Cortex-M cores allow to *relocate* the position in memory of the *vector table*. Moreover, it is possible to force the MCU to boot up from different memories than the internal flash one. These are advanced topics that will be covered in Chapter 20 about memory layout and another one dedicated to booting process. To avoid confusion in unexperienced readers it is best to consider the *vector table* position fixed and bound to the `0x0000 0000` address.

we upload to the flash. In Chapter 20 we will study the content of `STM32XXxx_FLASH.ld` file, which contains the directives to instruct GNU LD about this.

7.2 Enabling Interrupts

When an STM32 MCU boots up, only *Reset*, *NMI* and *Hard Fault* exceptions are enabled by default. The rest of exceptions and peripheral interrupts are disabled, and they have to be enabled on request. To enable an IRQ, the CubeHAL provides the following function:

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
```

where the `IRQn_Type` is an enumeration of all exceptions and interrupts defined for that specific MCU. The `IRQn_Type enum` is part of the ST Driver HAL, and it is defined inside a header file specific for the given STM32 MCU in the Eclipse folder `Drivers/CMSIS/Device/ST/STM32XXxx/Include`. These files are named `stm32XXxx.h`. For example, for an STM32F446RE MCU the right filename is `stm32f446xx.h`.

Filename: `Drivers/CMSIS/Device/ST/STM32XXxx/Include/stm32XXxx.h`

```
66  typedef enum {
67  /******  Cortex-M4 Processor Exceptions Numbers ***************************/
68    NonMaskableInt_IRQn   = -14,    /*!< 2 Non Maskable Interrupt                */
69    MemoryManagement_IRQn = -12,    /*!< 4 Cortex-M4 Memory Management Interrupt */
70    BusFault_IRQn         = -11,    /*!< 5 Cortex-M4 Bus Fault Interrupt         */
71    UsageFault_IRQn       = -10,    /*!< 6 Cortex-M4 Usage Fault Interrupt       */
72    SVCall_IRQn           = -5,     /*!< 11 Cortex-M4 SV Call Interrupt          */
73    DebugMonitor_IRQn     = -4,     /*!< 12 Cortex-M4 Debug Monitor Interrupt    */
74    PendSV_IRQn           = -2,     /*!< 14 Cortex-M4 Pend SV Interrupt          */
75    SysTick_IRQn          = -1,     /*!< 15 Cortex-M4 System Tick Interrupt      */
76    ...
```

The corresponding function to disable an IRQ is the:

```
void HAL_NVIC_DisableIRQ(IRQn_Type IRQn);
```

It is important to remark that the previous two functions enable/disable an interrupt at the NVIC controller level. Looking a **Figure 7.1**, you can see that an interrupt line is asserted by the peripheral connected to that line. For example, the USART2 peripheral asserts the interrupt line that corresponds to the `USART2_IRQn` interrupt line inside the NVIC controller. This means that the single peripheral must be properly configured to work in interrupt mode. As we will see in the rest of the book, the majority of STM32 peripherals are designed to work, among the others, in *interrupt mode*. By using specific HAL routines, we can enable the interrupt at *peripheral level*. For example, using the `HAL_USART_Transmit_IT()` we implicitly configure the USART peripheral in *interrupt mode*. Clearly, it is also required to enable the corresponding interrupt at NVIC level by calling the `HAL_NVIC_EnableIRQ()`.

Now it is a good time to start playing with interrupts.

7.2.1 External Lines and NVIC

As we have seen in **Figure 7.1**, STM32 microcontrollers provide a variable number of external interrupt sources connected to the NVIC through the EXTI controller, which in turn is capable to manage several *EXTI lines*. The number of interrupt sources and lines depends on the specific STM32 family.

GPIO are connected to the EXTI lines, and it is possible to enable interrupts for every MCU GPIO, even if the most of them share the same interrupt line. For example, for an STM32F4 MCU, up to 114 GPIOs are connected to 16 EXTI lines. However, only 7 of these lines have an independent interrupt associated with them.

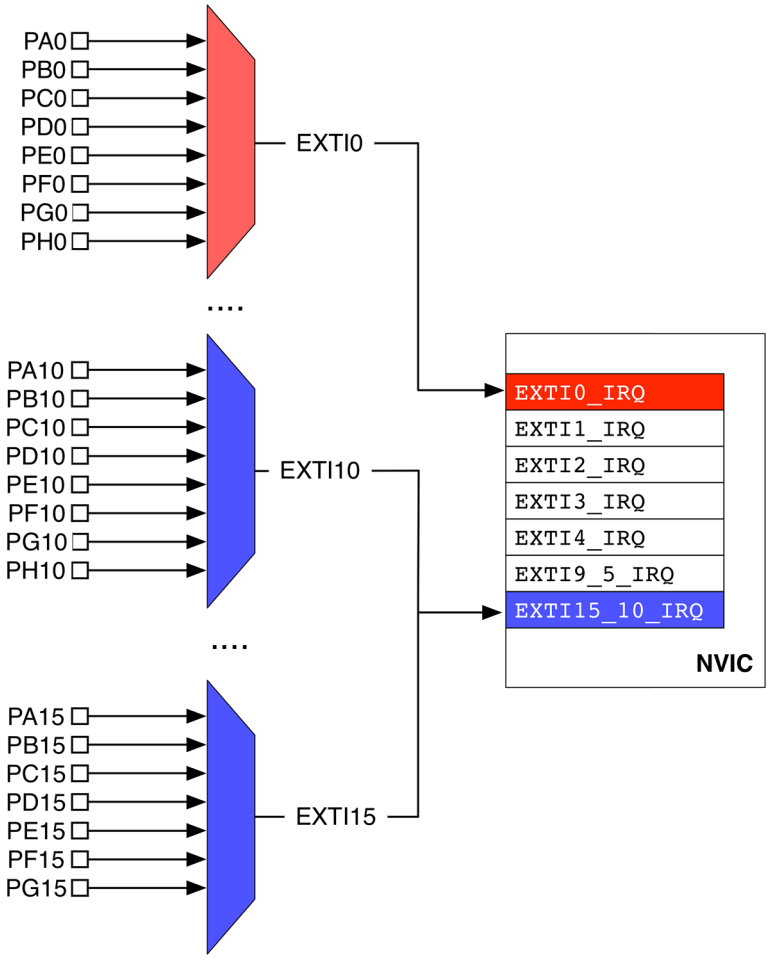


Figure 7.3: The relation between GPIO, EXTI lines and corresponding ISR in an STM32F4 MCU

Figure 7.3 shows EXTI lines 0, 10 and 15 in an STM32F4 MCU. All Px0 pins are connected to EXTI0, all Px10 pins are connected to EXTI10 and all Px15 pins are connected to EXTI15. However, EXTI lines 10 and 15 share the same IRQ inside the NVIC (and hence are serviced by the same ISR)[3].

[3]Sometimes, it also happens that different peripherals share the same request line, even in Cortex-M3/4/7 based MCUs where up to 240 configurable request lines are available. For example, in an STM32F446RE MCU, timer TIM6 shares its global IRQ with DAC1 and DAC2 under-run error interrupts.

This means that:

- Only one PxY pin can be a source of interrupt. For example, we cannot define both PA0 and PB0 as input interrupt pins.
- For EXTI lines sharing the same IRQ inside the NVIC controller, we have to code the corresponding ISR so that we must be able to discriminate which lines generated the interrupt.

The following example[4] shows how to use interrupts to toggle the LD2 LED every time we press the user-programmable button, which is connected to the PC13 pin. First, we configure the GPIO PC13 to fire an interrupt every time it goes from the low level to the high one (line 79:82). This is accomplished by setting GPIO `.Mode` to `GPIO_MODE_IT_RISING` (for the complete list of available interrupt related modes, refer to Table 6.2). Next, we enable the interrupt of the EXTI line associated with the Px13 pins, that is `EXTI15_10_IRQn`.

Filename: `src/main-ex1.c`

```
64  int main(void) {
65    GPIO_InitTypeDef GPIO_InitStruct = {0};
66
67    /* MCU Configuration--------------------------------------------------------*/
68    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
69    HAL_Init();
70    /* Configure the system clock */
71    SystemClock_Config();
72
73    /* GPIOA and GPIOC Configuration---------------------------------------------*/
74    /* GPIO Ports Clock Enable */
75    __HAL_RCC_GPIOC_CLK_ENABLE();
76    __HAL_RCC_GPIOA_CLK_ENABLE();
77
78    /*Configure GPIO pin : PC13 */
79    GPIO_InitStruct.Pin = GPIO_PIN_13;
80    GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
81    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
82    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
83
84    /*Configure GPIO pin : PA5 */
85    GPIO_InitStruct.Pin = GPIO_PIN_5;
86    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
87    GPIO_InitStruct.Pull = GPIO_NOPULL;
88    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
89    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
90
91    /* Configure GPIO pin Output Level */
92    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
```

[4] The example is designed to work with a Nucleo-F446RE board. Please, refer to other book examples if you have a different Nucleo board.

```
93
94    /* EXTI interrupt init*/
95    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
96
97    while (1);
98  }
99
100 void EXTI15_10_IRQHandler(void) {
101   if(__HAL_GPIO_EXTI_GET_IT(GPIO_PIN_13) != RESET) {
102     __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_13);
103     HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
104   }
```

Finally, we need to define the function `void EXTI15_10_IRQHandler()`[5], which is the ISR routine associated to the IRQ for the `EXTI15_10` line inside the *vector table* (lines 100:104). The content of the ISR is quite simple. Since the `EXTI15_10` line is connected to different pins, we need to check if PC13 is the pin that fired the interrupt. If so, we toggle the PA5 I/O and we clear the pending bit associated to the EXTI line (the reason why we need to perform this operation will be explained later).

Fortunately, the ST HAL provides an abstraction mechanism that avoids us to deal with all those details, unless we need to take care of them. The previous example can be rewritten in the following way:

Filename: **`src/main-ex2.c`**

```
64    GPIO_InitTypeDef GPIO_InitStruct = {0};
65
66    /* MCU Configuration--------------------------------------------------------*/
67    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
68    HAL_Init();
69    /* Configure the system clock */
70    SystemClock_Config();
71
72    /* GPIOA and GPIOC Configuration--------------------------------------------*/
73    /* GPIO Ports Clock Enable */
74    __HAL_RCC_GPIOC_CLK_ENABLE();
75    __HAL_RCC_GPIOA_CLK_ENABLE();
76
77    /*Configure GPIO pin : PC13 */
78    GPIO_InitStruct.Pin = GPIO_PIN_12 | GPIO_PIN_13;
79    GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
80    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
```

[5] Another feature of the ARM architectures is the ability to use conventional C functions as ISRs. When an interrupt fires, the CPU switches from the *Threaded mode* (that is, the main execution flow) to the *Handler mode*. During this switching process, the current execution context is saved thanks to a procedure named *stacking*. The CPU itself is responsible of storing the previous saved context when the ISR terminates the execution (*unstacking*). The explanation of this procedure is outside from the scope of this book. For more information about these aspects, refer to one of the Joseph Yiu books.

```
81    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
82
83    /*Configure GPIO pin : PA5 */
84    GPIO_InitStruct.Pin = GPIO_PIN_5;
85    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
86    GPIO_InitStruct.Pull = GPIO_NOPULL;
87    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
88    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
89
90    /* Configure GPIO pin Output Level */
91    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
92
93    /* EXTI interrupt init*/
94    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
95
96    while (1);
97  }
98
99  void EXTI15_10_IRQHandler(void) {
100   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_12);
101   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
102 }
103
104 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
105   if(GPIO_Pin == GPIO_PIN_13)
106     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, SET);
107   else if(GPIO_Pin == GPIO_PIN_12)
108     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, RESET);
109 }
```

This time we have configured as interrupt source both pin PC13 and PC12. When the `EXTI15_10_IRQHandler()` ISR is called, we transfer the control to the `HAL_GPIO_EXTI_IRQHandler()` function inside the HAL. This will perform for us all the interrupt related activities, and it will call the `HAL_GPIO_EXTI_Callback()` routine passing the actual GPIO that generated the IRQ (remember that PC12 and PC13 are connected to the same EXTI interrupt line). **Figure 7.4** clearly shows the call sequence that generates from the IRQ[6].

[6] Don't consider those time intervals related to the CPU cycles, they are just used to indicate "subsequent" events.

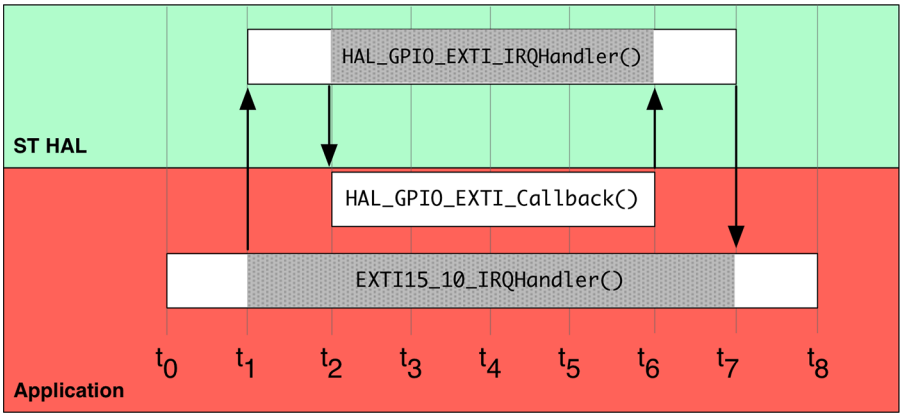


Figure 7.4: How an IRQ is processed by the HAL

This mechanism is used by almost all IRQ handler routines inside the HAL.

7.2.2 Enabling Interrupts with CubeMX

CubeMX can be used to easily enable IRQs and to automatically generate the ISR code. The first step is to enable the corresponding EXTI line using the *Chip view*, as shown in **Figure 7.5**.

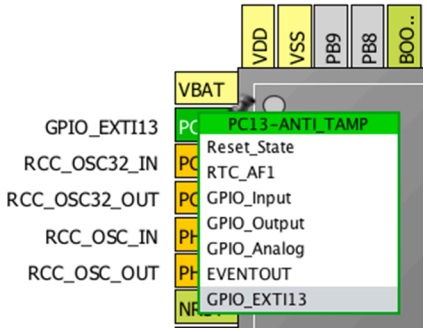


Figure 7.5: How a GPIO can be bound to EXTI line using CubeMX

Once we have enabled an IRQ, we need to instruct CubeMX to generate the corresponding ISR. This configuration is done through the *System view*, clicking on the NVIC button. A list of ISRs that can be enabled appears, as shown in **Figure 7.6**.

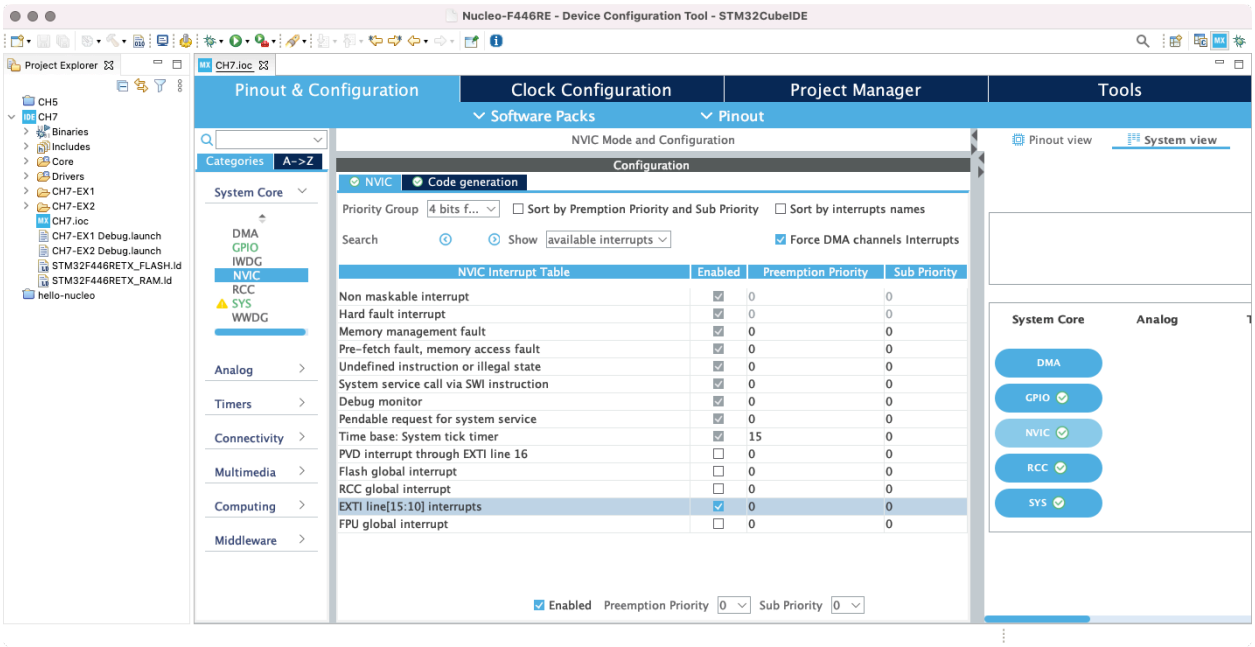


Figure 7.6: The NVIC configuration view allows to enable the corresponding ISR

CubeMX automatically adds the enabled ISRs inside the `Core/Src/stm32XXxx_it.c` file, and it takes care of enabling the IRQs. Moreover, it adds for us the corresponding HAL handler routine to call, as shown below:

```
/**
* @brief This function handles EXTI line[15:10] interrupts.
*/
void EXTI15_10_IRQHandler(void) {
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */

  /* USER CODE END EXTI15_10_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  /* USER CODE BEGIN EXTI15_10_IRQn 1 */

  /* USER CODE END EXTI15_10_IRQn 1 */
}
```

We only need to add the corresponding callback function (for example the `HAL_GPIO_EXTI_Callback()` routine) inside our application code.



What Belongs to What

When starting to deal with the ST HAL, a lot of confusion arises from its relationship with the ARM CMSIS package. The `stm32XX_hal_cortex.c` module clearly shows the interaction between the ST HAL and the CMSIS package, since it completely relies on the official ARM package to deal with the underlying Cortex-M NVIC controller. Every `HAL_NVIC_xxx()` function is a wrap of the corresponding CMSIS `NVIC_xxx()` function. This means that we may use the CMSIS API to program the NVIC controller. However, since this book is about the CubeHAL, we will use the ST API to manage interrupts.

7.3 Interrupt Lifecycle

Once dealing with interrupts, it is really important to understand their lifecycle. Although the Cortex-M core automatically performs most of the work for us, we have to pay attention to some aspects that could be a source of confusion during the interrupt management. However, this paragraph gives a look at the interrupt lifecycle from the "HAL point of view". If you are interested in looking deeper into this matter, the book series from Joseph Yiu[7] it is again the best source.

An interrupt can:

1. either be disabled (default behavior) or enabled;
   - we enable/disable it calling the `HAL_NVIC_EnableIRQ()`/`HAL_NVIC_DisableIRQ()` function;
2. either be pending (a request is waiting to be served) or not pending;
3. either be in an active (being served) or inactive state.

We have already seen the first case in the previous paragraph. Now it is important to study what happens when an interrupt occurs.

When an interrupt fires, it is marked as *pending* until the processor can serve it. If no other interrupts are currently being processed, its pending state is automatically cleared by the processor, which almost immediately starts serving it.

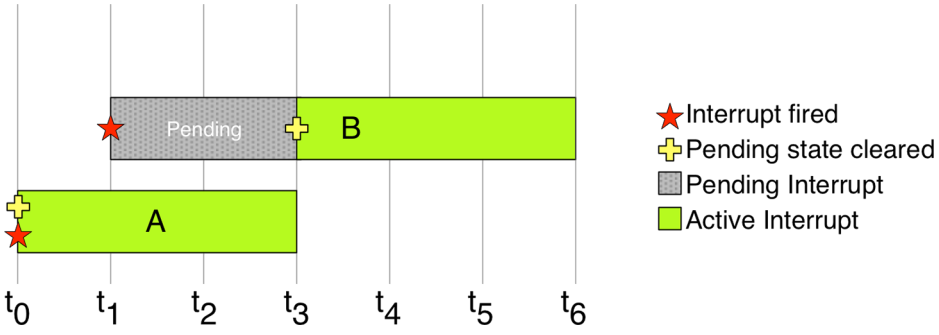


Figure 7.7: The relation between the pending bit and the interrupt active status

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

Figure 7.7 shows how this works. Interrupt A fires at the time t_0 and, since the CPU is not servicing another interrupt, its pending bit is cleared and its execution starts immediately[8] (the interrupt becomes *active*). At the time t_1 the B interrupt fires, but here we suppose that it has a lower priority than A. So, it is leaved in pending state until the A ISR concludes its operations. When this happens, the pending bit is automatically cleared and the ISR become *active*.

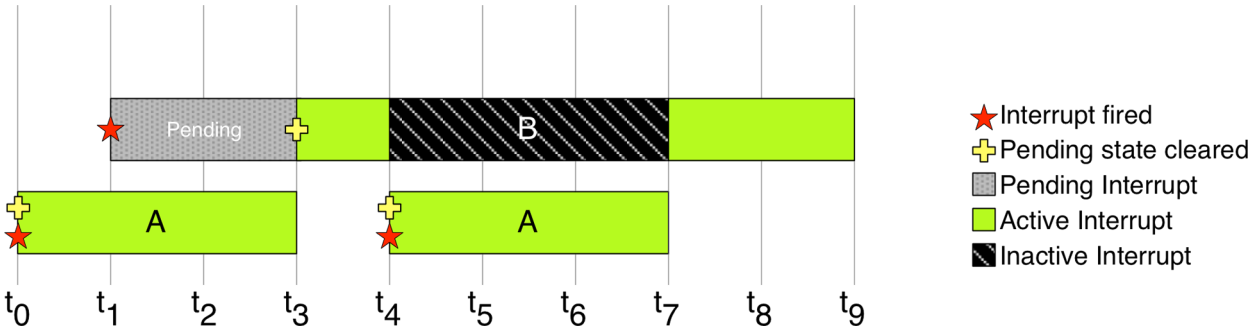


Figure 7.8: The relation between the active status and interrupts priority

Figure 7.8 shows another important case. Here we have that the A interrupt fires, and the CPU can immediately serve it. The interrupt B fires while A is serviced, so it remains in pending state until A finishes. When this happens, the pending bit of B interrupt is cleared, and it becomes active. However, after a while, A interrupt fires again, and since it has a higher priority, B interrupt is suspended (becomes *inactive*) and the execution of A starts immediately. When this finishes, the B interrupt becomes active again, and it completes its job.

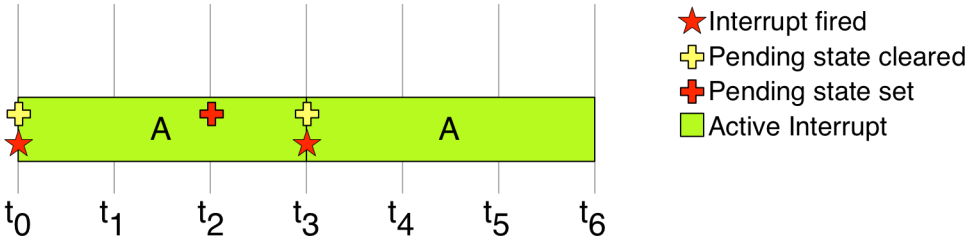


Figure 7.9: How an interrupt can be forced to fire again setting its pending bit

NVIC provides a high degree of flexibility for programmers. An interrupt can be forced to fire again during its execution, simply setting its pending bit again, as shown in **Figure 7.9**[9]. In the same way, the execution of an interrupt can be canceled clearing its pending bit while it is in pending state, as shown in **Figure 7.10**.

[8] Here, it is important to understand the with the word "immediately" we are not saying that the interrupt execution starts without delay. If no other interrupts are running, Cortex-M3/4/7/33 cores serve an interrupt in 12 CPU cycles, while Cortex-M0 does it in 15 cycles and Cortex-M0+ in 16 cycles.

[9] For the sake of completeness, it is important to specify that Cortex-M architecture is designed so that if an interrupt fires while the processor is already servicing another interrupt, this will be serviced without restoring the previous application doing the *unstacking* (refer to note 3 in this chapter for the definition of *stacking/unstacking*). This technique is called *tail chaining* and it allows to speed up interrupt management and to reduce power consumption.

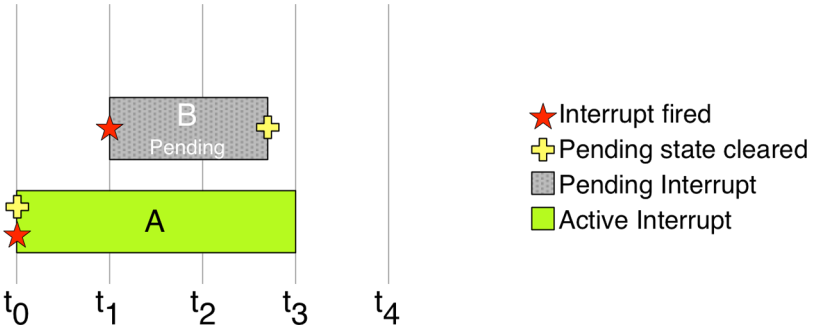


Figure 7.10: IRQ servicing can be canceled clearing its pending bit before it is executed

Here it is important to clarify an important aspect related to how peripherals warn the NVIC controller about the interrupt request. When an interrupt takes place, the most of STM32 peripherals assert a specific signal connected to the NVIC, which is mapped in the peripheral memory through a dedicated bit. This peripheral *Interrupt Request* bit will be held high until it is manually cleared by the application code. For example, in the Example 1 we had to expressly clear the EXTI line IRQ pending bit using the macro `__HAL_GPIO_EXTI_CLEAR_IT()`. If we do not de-assert that bit, a new interrupt will be fired until it is cleared.

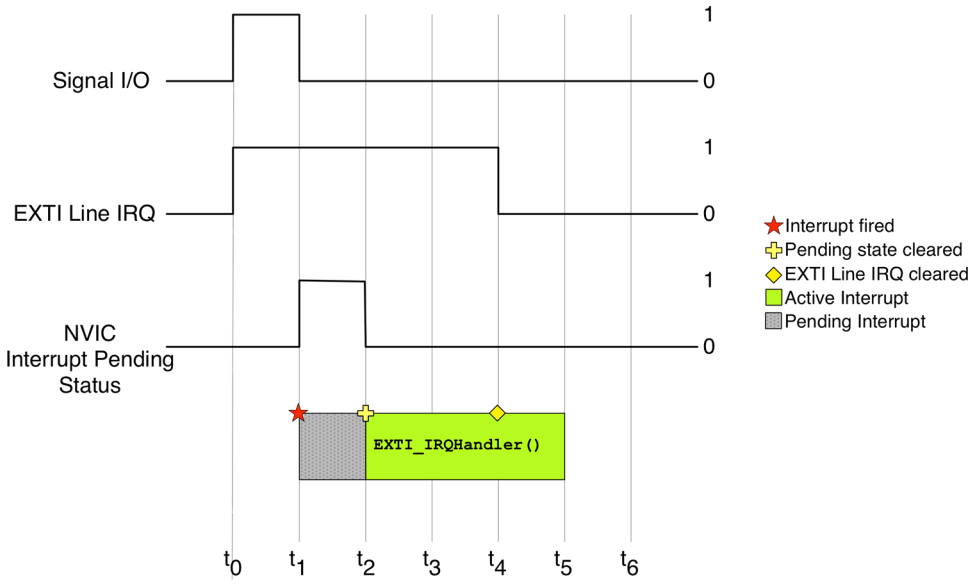


Figure 7.11: The relation between the peripheral IRQ and the corresponding interrupt

The **Figure 7.11** clearly shows the relation between the peripheral IRQ pending state and the ISR pending state. Signal I/O is the external peripheral driving the I/O (e.g., a tactile switch connected to a pin). When the signal level changes, the EXTI line connected to that I/O generates an IRQ and the corresponding pending bit is asserted. As consequence, the NVIC generates the interrupt. When the processor starts servicing the ISR, the ISR pending bit is cleared automatically, but the peripheral IRQ pending bit will be held high until it is cleared by the application code.

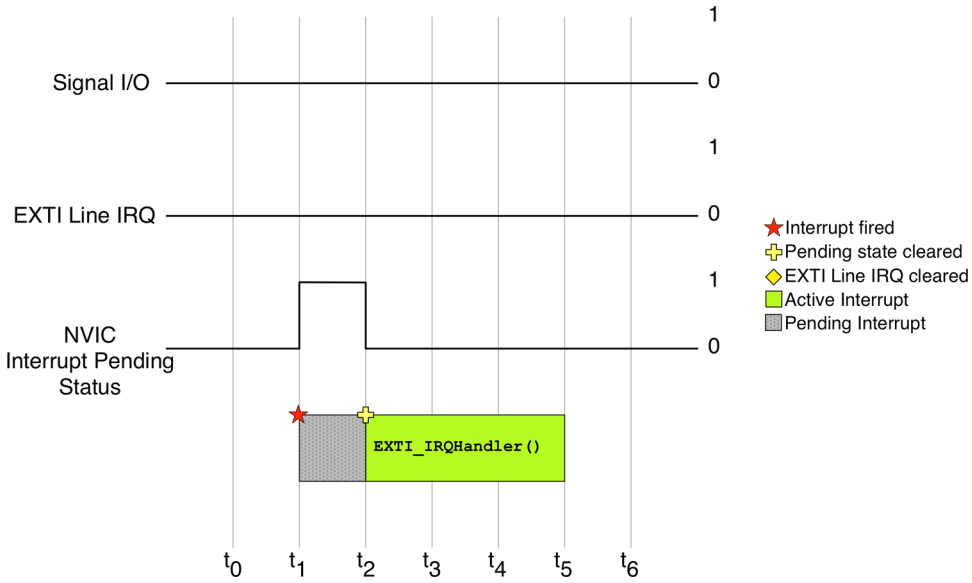


Figure 7.12: When an interrupt is forced setting its pending bit, the corresponding peripheral IRQ remains unset

The **Figure 7.12** shows another case. Here we force the execution of the ISR setting its pending bit. Since this time the external peripheral is not involved, there is no need to clear the corresponding IRQ pending bit.

Since the presence of the IRQ pending bit is peripheral dependent, it is always opportune to use the ST HAL functions to manage interrupts, leaving all the underlying details to the HAL implementation (unless we want to have full control, but this is not case of this book). However, take in mind that to avoid losing important interrupts, it is a good design practice to clear peripherals IRQ pending status bit as their ISR start to be serviced. The processor core does not keep track of multiple interrupts (it does not queue interrupts), so if we clear the peripheral pending bit at the end of an ISR, we may lose important IRQs that fire in the middle.

To see if an interrupt is pending (that is, fired but not running), we can use the HAL function:

```
uint32_t HAL_NVIC_GetPendingIRQ(IRQn_Type IRQn);
```

which returns `0` if the IRQ is not pending, `1` otherwise.
To set the pending bit of an IRQ we can use the HAL function:

```
void HAL_NVIC_SetPendingIRQ(IRQn_Type IRQn);
```

This will cause the interrupt to fire, as it would be generated by the hardware. A distinctive feature of Cortex-M processors it that it is possible to programmatically fire an interrupt inside the ISR routine of another interrupt.
Instead, to clear the pending bit of an IRQ, we can use the function:

```
void HAL_NVIC_ClearPendingIRQ(IRQn_Type IRQn);
```

Once again, it is also possible to clear the execution of a pending interrupt inside the ISR servicing another IRQ.

To check if an ISR is active (IRQ being serviced), we can use the function:

```
uint32_t HAL_NVIC_GetActive(IRQn_Type IRQn);
```

which returns `1` if the IRQ is active, `0` otherwise.

7.4 Interrupt Priority Levels

A distinctive feature of the ARM Cortex-M architecture is the ability to prioritize interrupts (except for the first three software exceptions that have a fixed priority, as shown in **Table 7.1**). Interrupt priority allows to define two things:

- the ISRs that will be executed first in case of concurrent interrupts;
- those routines that can be optionally preempted to start executing an ISR with a higher priority.

NVIC priority mechanism is substantially different between Cortex-M0/0+ and Cortex-M3/4/7/33 cores. For this reason, we are going to explain them in two separated subparagraphs.

7.4.1 Cortex-M0/0+

Cortex-M0/0+ based microcontrollers have a simpler interrupt priority mechanism. This means that STM32F0/G0/L0 MCUs have a different behavior from the rest of STM32 microcontrollers. And you have to pay special attention if you are porting your code between the STM32 series.

In Cortex-M0/0+ cores the priority of each interrupt is defined through an 8-bit register, called `IPR`. In the ARMv6-M core architecture only 4 bits of this register are used, allowing up to 16 different priority levels. However, in practice, STM32 MCUs implementing these cores use only the two upper bits of this register, seeing all other bits equal to zero.

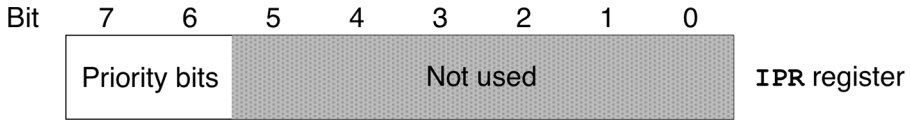


Figure 7.13: The content of IPR register on an STM32 MCU based on Cortex-M0

Figure 7.13 shows how the content of IPR is interpreted. This means that we have only four maximum priority levels: 0x00, 0x40, 0x80, 0xC0. The lower this number is, the higher the priority is. That is, an IRQ having a priority equal to 0x40 has a higher priority than an IRQ with a priority

level equal to 0xC0. If two interrupts fire at the same time, the one with the higher priority will be served first. If the processor is already servicing an interrupt and a higher priority interrupt fires, then the current interrupt is suspended and the control passes to the higher priority interrupt. When this is completed, the execution goes back to the previous interrupt, if no other interrupt with higher priority occurs in the meantime. This mechanism is called *interrupt preemption*.

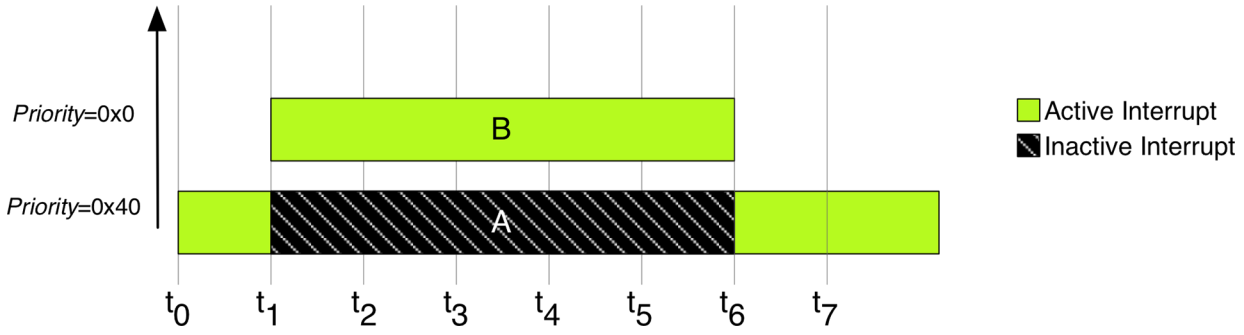


Figure 7.14: Preemption of interrupts in case of concurrent execution

Figure 7.14 shows an example of interrupt preemption. A is an IRQ with lower priority that fires at time t_0. The ISR starts the execution but the IRQ B, which has a higher priority (lower priority level), fires at time $t_1$and the execution of A ISR is stopped. When B finishes its job, the execution of A ISR is resumed until it finishes. This "nested" mechanism induced by interrupt priorities leads to the name of the NVIC controller, which is *Nested Vectored Interrupt Controller*.

Cortex-M0/0+ has an important difference compared to Cortex-3/4/7/33 cores. The interrupt priority is static. This means that once an interrupt is enabled its priority can no longer be changed, until we disable the IRQ again.

The CubeHAL provides the following function to assign a priority to an IRQ:

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority);
```

The `HAL_NVIC_SetPriority()` function accepts the IRQ we are going to configure and the `PreemptPriority`, which is the preemption priority we are going to assign to the IRQ. The CMSIS API, and hence the CubeHAL library, is designed so that `PreemptPriority` is specified with a priority level number ranging from 0 to 4. The value is internally shifted to the most significant bits automatically. This simplifies the porting of code to other MCU with a different number of priority bits (this is the reason why only the left part of `IPR` register is used by silicon vendors).



As you can see, the `HAL_NVIC_SetPriority()` function accepts also the additional parameter `SubPriority`, which is simply ignored in CubeF0 and CubeL0 HALs since the underlying Cortex-M processor does not support interrupt sub-priority. Here ST engineers have decided to use the same API available in the other HALs for Cortex-M3/4/7/33 based processors. Probably they decided to do so to simplify porting code between the different STM32 MCUs. Curiously, they have decided to define the corresponding function to retrieve the priority of an IRQ in the following way:

```
uint32_t HAL_NVIC_GetPriority(IRQn_Type IRQn);
```

which is completely different from the one defined in the HALs for Cortex-M3/4/7/33 based processors.

The following example[10] shows how the interrupt priority mechanism works.

Filename: **`src/main-ex3.c`**

```
39 uint8_t blink = 0;
40
41 int main(void) {
42   GPIO_InitTypeDef GPIO_InitStruct;
43
44   HAL_Init();
45
46   /* GPIO Ports Clock Enable */
47   __HAL_RCC_GPIOC_CLK_ENABLE();
48   __HAL_RCC_GPIOB_CLK_ENABLE();
49   __HAL_RCC_GPIOA_CLK_ENABLE();
50
51   /*Configure GPIO pin : PC13 - USER BUTTON */
52   GPIO_InitStruct.Pin = GPIO_PIN_13 ;
53   GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
54   GPIO_InitStruct.Pull = GPIO_PULLDOWN;
55   HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
56
57   /*Configure GPIO pin : PB2 */
58   GPIO_InitStruct.Pin = GPIO_PIN_2 ;
59   GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
60   GPIO_InitStruct.Pull = GPIO_PULLUP;
61   HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
62
63   /*Configure GPIO pin : PA5 - LD2 LED */
64   GPIO_InitStruct.Pin = GPIO_PIN_5;
65   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
66   GPIO_InitStruct.Pull = GPIO_NOPULL;
67   GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
```

[10] The example is designed to work with a Nucleo-F072RB board. Please, refer to other book examples if you have a different Nucleo board.

```
68    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
69
70    HAL_NVIC_SetPriority(EXTI4_15_IRQn, 0x1, 0);
71    HAL_NVIC_EnableIRQ(EXTI4_15_IRQn);
72
73    HAL_NVIC_SetPriority(EXTI2_3_IRQn, 0x0, 0);
74    HAL_NVIC_EnableIRQ(EXTI2_3_IRQn);
75
76    while(1);
77  }
78
79  void EXTI4_15_IRQHandler(void) {
80    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
81  }
82
83  void EXTI2_3_IRQHandler(void) {
84    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_2);
85  }
86
87  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
88    if(GPIO_Pin == GPIO_PIN_13) {
89      blink = 1;
90      while(blink) {
91        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
92        for(volatile int i = 0; i < 100000; i++) {
93          /* Busy wait */
94        }
95      }
96    }
97    else {
98      blink = 0;
```

The code should be easy to understand if my previous explanation is clear. Here we have two IRQs associated to EXTI lines 2 and 13. The corresponding ISRs call the HAL `HAL_GPIO_EXTI_IRQHandler()` which in turn calls the `HAL_GPIO_EXTI_Callback()` callback passing the GPIO involved in the interrupt. When the user button connected to PC13 signal is pushed, the ISR starts an infinite loop until the `blink` global variables is `>0`. This loop makes the LD2 LED blinking quickly. When the PB2 pin is asserted low (use the pinout diagram for your Nucleo from Appendix C to identify PB2 pin position), the `EXTI2_3_IRQHandler()`[11] fires and this causes the `HAL_GPIO_EXTI_IRQHandler()` to set the blink variable to `0`. The `EXTI4_15_IRQHandler()` can now end. The priority of each interrupt is configured at lines 70 and 73: as you can see, since the interrupt priority is static in Cortex-M0/0+ based MCUs, we have to set it before we enable the corresponding interrupt.

[11]Please, take note that for STM32F302 MCUs the default name of the IRQ associated to EXTI line 2 is `EXTI2_TSC_IRQHandler`. Refer to book examples if you are working with this MCU.



Please, take note that this is a really bad way to deal with interrupts. Locking the MCU inside an interrupt is a poor programming style, and it is the root of all evil in embedded programming. Unfortunately, this is the only example that came up to the author's mind, considering that at this point the book still covers few topics. Every ISR must be designed to last as little as possible, otherwise other fundamental ISRs could be masked for a long time loosing important information coming from other peripherals.



As exercise, try to play with interrupt priorities, and see what happens if both interrupts have the same priority.



A good question might arise: why not using the `HAL_Delay()` function instead of the *busy-wait* at lines 87:89? The answer is simple, and it is directly connected to interrupt priorities. The `HAL_Delay()` performs a busy-wait while waiting for the *SysTick* timer to increment the global `uwTick` variable (an `unsigned 32-bit integer` incremented at 1KHZ frequency). However, as we will see in Chapter 11, the *SysTick* timer is configured to work in interrupt mode and the interrupt priority is set by CubeMX - by default - at the lowest possible priority (take a look to the macro `TICK_INT_PRIORITY` inside the `Core/Inc/stm32XXxx_hal_conf.h` file). By setting the priority level of the `EXTI15_10_IRQn` interrupt to `1` will cause that the ISR of *SysTick* timer will never fire until the `blink` variable goes to `0` and the ISR can terminate.



You may notice that often the interrupt fires by simply touching the wire, even if it is not tied to the ground. Why does this happen? There are essentially two reasons that cause the interrupt to "accidentally" trigger. First of all, modern microcontrollers try to minimize the power leakages connected with the usage of internal pull-up/down resistors. So, the value of these resistors is chosen high (something around 50kΩ). If you play with the voltage divider equation, you can figure out that it is really easy to pull an I/O low or high when a pull-up/down resistor has a high resistance value. Secondly, here we are not doing adequate *debouncing* of the input pin. *Debouncing* is the process of minimizing the effect of *bounces* produced by "unstable" sources (e.g., a mechanical switch). Usually, debouncing is performed in hardware[12] or in software, by counting how much time is elapsed from the first variation of the input state: in our case, if the input remains low for more than a given period (usually something between 100ms and 200ms is sufficient), then we can say that the input has been effectively tied to the ground). As we will see in Chapter 11, we can also use one channel of a timer configured to work in input capture mode to detect when a GPIO changes state. This gives us the ability to automatically count how much time is elapsed from the first event. Moreover, timer channels support integrated and programmable hardware filters, which allow us to reduce the number of external components to debounce the I/Os.

[12]Usually, a capacitor and a resistor in parallel with the switch contacts are sufficient in most cases. For example, you can take a look at schematics of the Nucleo board to see how ST engineers have debounced the USER button connected to PC13 GPIO.

7.4.2 Cortex-M3/4/7/33

Interrupt priority mechanism in Cortex-M3/4/7/33 is more advanced than the one available in Cortex-M0/0+ based microcontrollers. Developers have a higher degree of flexibility, and this is often source of several headaches for novices. Moreover, the way interrupt priority is presented both in the ARM and ST documentation is a little bit counterintuitive.

In Cortex-M3/4/7/33 cores the priority of each interrupt is defined through the `IPR` register. This is an 8bit register in the ARMv7-M core architecture that allows up to 255 different priority levels. However, in practice, STM32 MCUs implementing Cortex-M3/4/7 cores use only the four upper bits of this register, while STM32 MCUs based on Cortex-M33 cores use only three upper bits.

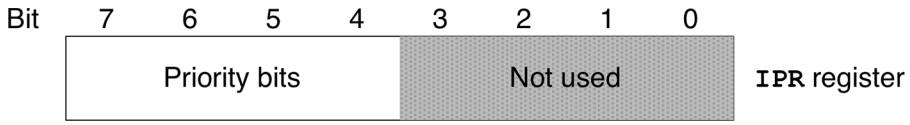


Figure 7.15: The content of IPR register on an STM32 MCU based on Cortex-M3/4/7 core

Figure 7.15 clearly shows how the content of IPR is interpreted. This means that we have the only sixteen maximum priority levels: 0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90, 0xA0, 0xB0, 0xC0, 0xD0, 0xE0, 0xF0. The lower this number is, the higher the priority is. That is, an IRQ having a priority equal to 0x10 has a higher priority than an IRQ with a priority level equal to 0xA0. If two interrupts fire at the same time, the one with the higher priority will be served first. If the processor is already servicing an interrupt and a higher priority interrupts fires, then the current interrupt is suspended, and the control passes to the higher priority interrupt. When this is completed, the execution goes back to the previous interrupt, if no other interrupts with higher priority occurs in the meantime.

So far, the mechanism is substantially the same of Cortex-M0/0+. The complication arises from the fact that the IPR register can be logically subdivided in two parts: a series of bits defining the *preemption priority*[13] and a series of bits defining the *sub-priority*. The first priority level rules the preemption priorities between ISRs. If an ISR has a priority higher than another one, it will preempt the execution of the lower priority ISR in case it fires. The *sub-priority* determines what ISR will be executed first, in case of multiple pending ISR, but it will not act on ISR preemption.

[13] What complicates the understanding of interrupt priorities is the fact that in the official documentation sometimes the *preemption priority* is also called *group priority*. This leads to a lot of confusion, since novices tends to imagine that these bits define a sort of *Access Control List* (ACL) privileges. Here, to simplify the understanding of this matter, we will only speak about *preemption priority* level.

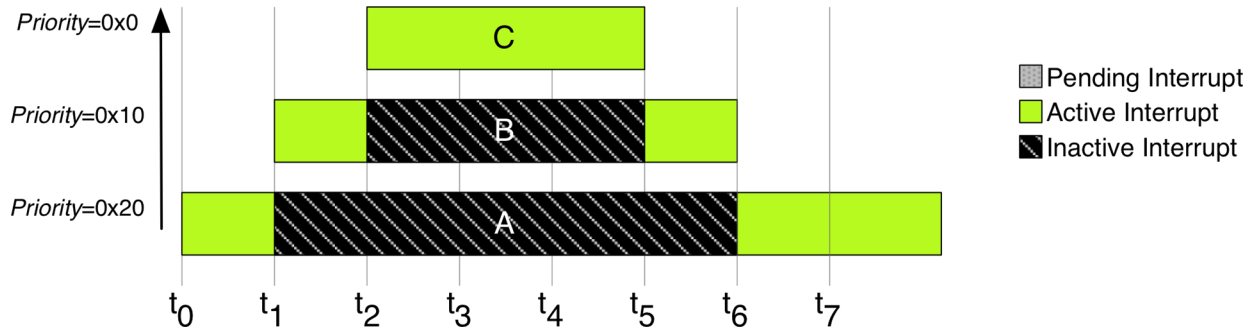


Figure 7.16: Preemption of interrupts in case of concurrent execution

Figure 7.16 shows an example of interrupt preemption. A is an IRQ with the lowest priority that fires at time t_0. The ISR starts the execution but the IRQ B, which has a higher priority (lower priority level), fires at time $t_1$and the execution of A ISR is stopped. After a while, C IRQ fires at time t_2 and the B ISR is stopped and the C ISR starts execution. When this finishes, the execution of B ISR is resumed until it finishes. When this happens, the execution of A ISR is resumed. This "nested" mechanism induced by interrupt priorities leads to the name of the NVIC controller, which is *Nested Vectored Interrupt Controller*.

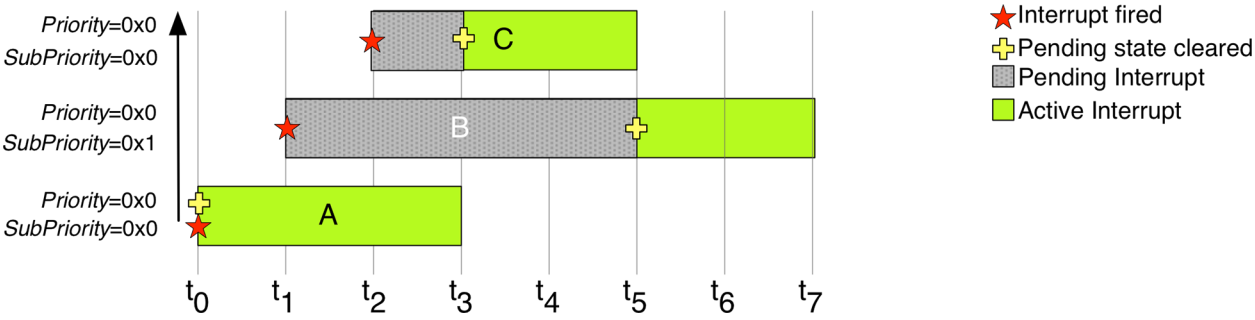


Figure 7.17: If two interrupts with the same priority are pending, the one with the higher sub-priority is executed first

Figure 7.17 shows how the *sub-priority* affects the execution of multiple pending ISRs. Here we have three interrupts, all with the same maximum priority. At time t_0 the IRQ A fires and it is serviced immediately. At the time t_1 B IRQ fires, but since it has the same priority level of other IRQs, it is leaved in pending state. At time t_2 also C IRQ fires, but for the same reason as before it is leaved in pending state by the processor. When The A ISR finishes, the C IRQ is served first, since it has a higher sub-priority than B. Only when the C ISR finishes the B IRQ can be served.

The way how `IPR` bits are logically subdivided is defined by the `SCB->AIRCR` register (a sub-group of bits of the *System Control Block* (SCB) register), and it is important to stress right from the start that this way to interpret the content of the `IPR` register **is global to all ISRs**. Once we have defined a priority scheme (also called *priority grouping* in the HAL), this is common to all interrupts used in the system.

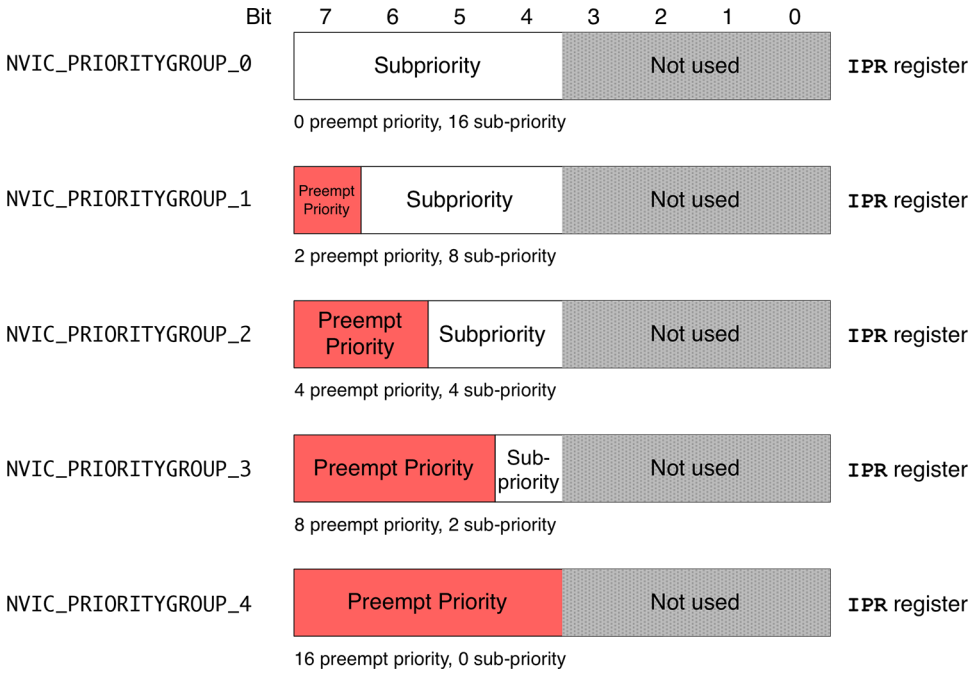


Figure 7.18: The subdivision of IPR bits between preemption priority and sub-priority

Figure 7.18 shows all five possible subdivisions of `IPR` register, while **Table 2** shows the maximum number of preemption priority levels and sub-priority levels that each subdivision scheme allows[14].

Table 2: The number of preemption priority level available based on the current *priority grouping* schema

NVIC Priority Group	Number of preemption priority levels	Number of sub-priority levels
`NVIC_PRIORITYGROUP_0`	0	16
`NVIC_PRIORITYGROUP_1`	2	8
`NVIC_PRIORITYGROUP_2`	4	4
`NVIC_PRIORITYGROUP_3`	8	2
`NVIC_PRIORITYGROUP_4`	16	0

The CubeHAL provides the following function to assign a priority to an IRQ:

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority);
```

The HAL library is designed so that the `PreemptPriority` and `SubPriority` can be configured with a priority level number ranging from 0 to 16. The value is internally shifted to the most significant bits automatically. This simplifies the porting of code to other MCU with a different number of priority bits (this is the reason why only the left part of `IPR` register is used by silicon vendors).

Instead, to define the *priority grouping*, that is how to subdivide the IPR register between the *preemption priority* and *sub-priority*, the following function can be used:

[14] As stated before, the STM32 MCUs based on Cortex-M33 cores use just the three upper bits of the `IPR` register. This means that there is a maximum of eight priority level and three *priority groups*. Thus, `NVIC_PRIORITYGROUP_4` is not available at all.

```
void HAL_NVIC_SetPriorityGrouping(uint32_t PriorityGroup);
```

where the `PriorityGroup` parameter is one of the macros from the column **NVIC Priority Group** in **Table 2**.

The following example[15] shows how the interrupt priority mechanism works.

Filename: **src/main-ex3.c**

```
23 void SystemClock_Config(void);
24 uint8_t blink=0;
25
26 int main(void) {
27   GPIO_InitTypeDef GPIO_InitStruct = {0};
28
29   /* MCU Configuration--------------------------------------------------------*/
30   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
31   HAL_Init();
32   /* Configure the system clock */
33   SystemClock_Config();
34
35   /* GPIOA, GPIOB and GPIOC Configuration---------------------------------------------*/
36   /* GPIO Ports Clock Enable */
37   __HAL_RCC_GPIOA_CLK_ENABLE();
38   __HAL_RCC_GPIOC_CLK_ENABLE();
39   __HAL_RCC_GPIOB_CLK_ENABLE();
40
41   /*Configure GPIO pin : PC13 */
42   GPIO_InitStruct.Pin = GPIO_PIN_13;
43   GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
44   GPIO_InitStruct.Pull = GPIO_PULLDOWN;
45   HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
46
47   /*Configure GPIO pin : PB2 */
48   GPIO_InitStruct.Pin = GPIO_PIN_2;
49   GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
50   GPIO_InitStruct.Pull = GPIO_PULLUP;
51   HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
52
53
54   /*Configure GPIO pin : PA5 */
55   GPIO_InitStruct.Pin = GPIO_PIN_5;
56   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
57   GPIO_InitStruct.Pull = GPIO_NOPULL;
58   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
59   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
60
```

[15]The example is designed to work with a Nucleo-F401RE board. Please, refer to other book examples if you have a different Nucleo board.

```
61    /* Configure GPIO pin Output Level */
62    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
63
64    /* EXTI interrupt init*/
65    HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0x1, 0x0);
66    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
67
68    HAL_NVIC_SetPriority(EXTI2_IRQn, 0x0, 0x0);
69    HAL_NVIC_EnableIRQ(EXTI2_IRQn);
70
71    while (1);
72  }
73
74  void EXTI15_10_IRQHandler(void) {
75    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
76  }
77
78  void EXTI2_IRQHandler(void) {
79    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_2);
80  }
81
82  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
83    if (GPIO_Pin == GPIO_PIN_13) {
84      blink = 1;
85      while (blink) {
86        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
87        for (volatile int i = 0; i < 100000; i++) {
88          /* Busy wait */
89        }
90      }
91    } else {
92      blink = 0;
93    }
94  }
```

The code should be easy to understand if my previous explanation is clear. Here we have two IRQs associated to EXTI lines 2 and 13. The corresponding ISRs call the HAL `HAL_GPIO_EXTI_IRQHandler()` which in turn calls the `HAL_GPIO_EXTI_Callback()` callback passing the GPIO involved in the interrupt. When the user button connected to PC13 signal is pushed, the ISR starts an infinite loop until the `blink` global variables is `>0`. This loop makes the LD2 LED blinking quickly. When the PB2 pin is asserted low (use the pinout diagram for your Nucleo from Appendix C to identify its position), the `EXTI2_IRQHandler()` fires and this causes the `HAL_GPIO_EXTI_IRQHandler()` to set the blink variable to `0`. The `EXTI15_10_IRQHandler()` can now end.



Please, take note that this is a really bad way to deal with interrupts. Locking the MCU inside an interrupt is a poor programming style, and it is the root of all evil in embedded programming. Unfortunately, this is the only example that came up to the author's mind, considering that at this point the book still covers few topics. As we will see soon, every ISR must be designed to last as little as possible, otherwise other fundamental ISRs could be masked for a long time loosing important information coming from other peripherals.



As exercise, try to play with interrupt priorities, and see what happens if both interrupts have the same priority.



A good question might arise: why not using the `HAL_Delay()` function instead of the *busy-wait* at lines 87:89? The answer is simple, and it is directly connected to interrupt priorities. The `HAL_Delay()` performs a busy-wait while waiting for the *SysTick* timer to increment the global `uwTick` variable (an `unsigned 32-bit integer` incremented at 1KHZ frequency). However, as we will see in Chapter 11, the *SysTick* timer is configured to work in interrupt mode and the interrupt priority is set by CubeMX - by default - at the lowest possible priority (take a look to the macro `TICK_INT_PRIORITY` inside the `Core/Inc/stm32XXxx_hal_conf.h` file). By setting the priority level of the `EXTI15_10_IRQn` interrupt to `1` will cause that the ISR of *SysTick* timer will never fire until the `blink` variable goes to `0` and the ISR can terminate.



You may notice that often the interrupt fires by simply touching the wire, even if it is not tied to the ground. Why does this happen? There are essentially two reasons that cause the interrupt to "accidentally" trigger. First of all, modern microcontrollers try to minimize the power leakages connected with the usage of internal pull-up/down resistors. So, the value of these resistors is chosen really high (something around 50kΩ). If you play with the voltage divider equation, you can figure out that it is really easy to pull an I/O low or high when a pull-up/down resistor has a high resistance value. Secondly, here we are not doing adequate *debouncing* of the input pin. *Debouncing* is the process of minimizing the effect of *bounces* produced by "unstable" sources (e.g., a mechanical switch). Usually, debouncing is performed in hardware[16] or in software, by counting how much time is elapsed from the first variation of the input state: in our case, if the input remains low for more than a given period (usually something between 100ms and 200ms is sufficient), then we can say that the input has been effectively tied to the ground). As we will see in Chapter 11, we can also use one channel of a timer configured to work in input capture mode to detect when a GPIO changes state. This gives us the ability to automatically count how much time is elapsed from the first event. Moreover, timer channels support integrated and programmable hardware filters, which allow us to reduce the number of external components to debounce the I/Os.

It is important to remark some fundamental things. First of all, different from Cortex-M0/0+ based microcontrollers, Cortex-M3/4/7/33 cores allow to dynamically change the priority of an interrupt,

[16]Usually, a capacitor and a resistor in parallel with the switch contacts are sufficient in most cases. For example, you can look at schematics of the Nucleo board to see how ST engineers have debounced the USER button connected to PC13 GPIO.

even if this is already enabled. Secondly, care must be taken when the *priority grouping* is lowered dynamically. Let us consider the following example.
Suppose that we have three ISRs with three decreasing priorities (the priority is specified inside the parenthesis): A(0x0), B(0x10), C(0x20). Suppose that we have defined these priorities when the *priority grouping* was equal to `NVIC_PRIORITYGROUP_4`. If we lower it to the `NVIC_PRIORITYGROUP_1` level, the current preemption levels will be interpreted as sub-priorities. This will cause that interrupt service routines A, B and C have the same preemption level (that is, 0x0), and it will not be possible to preempt them. For example, looking at **Figure 7.20** we can see what happens to the priority of the ISR C when the *priority grouping* is lowered from 4 to 1. When the *priority grouping* is set to 4, the priority of C ISR is just two levels under the maximum priority level, which is 0 (the next highest level is 0x10, which is the B's priority). This means that C can be preempted both by A and B. However, if we lower the *priority grouping* to 1, then the priority of C becomes 0x0 (only bit 7 acts as priority) and the remaining bits are **interpreted** by the NVIC controller as sub-priority. This can lead to the following scenario:

1. all interrupts will not be able to preempt each other;
2. if C interrupt is triggered, and the CPU is not servicing another interrupt, C is serviced immediately;
3. if CPU is servicing C ISR and then after a short while A and B are triggered, CPU will service A and then B after it completes to service C;
4. if CPU is servicing another ISR, if C triggers and then after a short while A and B are triggered, A will be serviced firstly, followed by B then C.



Figure 7.20: What happens to the C ISR priority when the *priority gruping* is lowered from 4 to 1



Before that the interrupt priority mechanism becomes clear, you will have to do several experiments by yourself. So, try to modify the Example 3 so that changing the *priority grouping* causes that the preemption priority is the same for both the IRQs.

To obtain the priority of an interrupt, the HAL defines the following function:

```
void HAL_NVIC_GetPriority(IRQn_Type IRQn, uint32_t PriorityGroup, uint32_t* pPreemptPriority, \
uint32_t* pSubPriority);
```

I have to admit that the signature of this function is a little bit fuzzy, since it differs from the `HAL_NVIC_SetPriority()`: here we have to specify also the `PriorityGroup`, while the `HAL_NVIC_SetPriority()` function computes it internally. I do not know why ST has decided to use this signature, and I cannot see the reason to make it different from the `HAL_NVIC_SetPriority()`.

The current priority grouping can be obtained using the following function:

```
uint32_t HAL_NVIC_GetPriorityGrouping(void);
```

7.4.3 Setting Interrupt Priority in CubeMX

CubeMX can be also used to set the IRQ priority and the priority grouping schema. This configuration is done through the *Configuration view*, clicking on the NVIC button. The list of enableable ISRs appears, as shown in **Figure 7.21**.

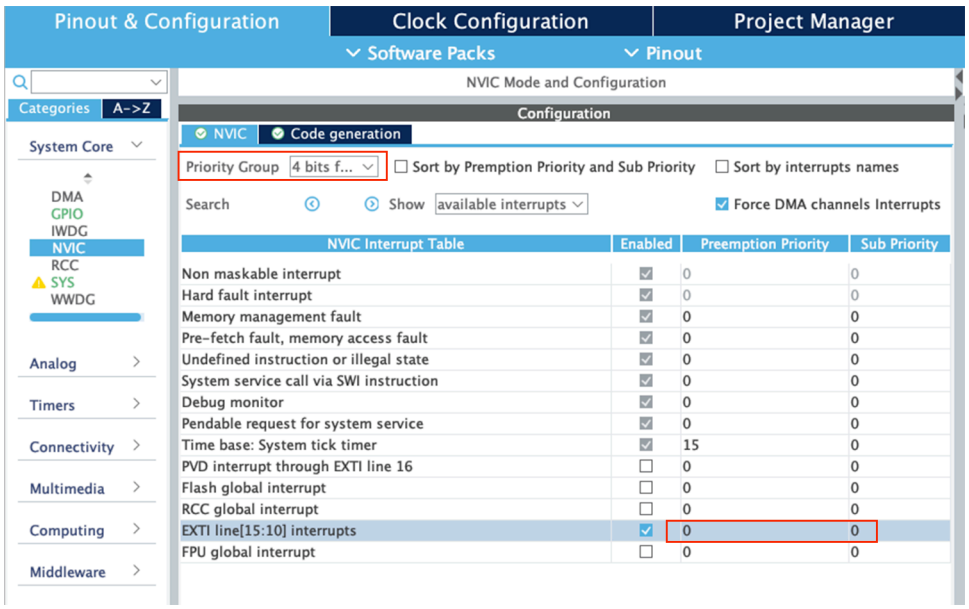


Figure 7.21: The NVIC configuration view allows to set the ISR priority

Using the **Priority Group** combo box, we can set the priority grouping schema, and then assign the individual priority and sub-priority to each interrupt. CubeMX will automatically generate the corresponding C code to setup the IRQ priority inside the `MX_GPIO_Init()` function. Instead, the global priority grouping schema is configured inside the `HAL_MspInit()` function.

7.5 Interrupt Re-Entrancy

Let us suppose rearranging the Example 3 so that it uses pin PC12 instead of PB2. In this case, since EXTI12 and EXTI13 share the same IRQ, our Nucleo would never stop blinking. Due to the way

the priority mechanism is implemented in Cortex-M processors (that is, an exception with a given priority cannot be preempted by another one with same priority), exceptions and interrupts are not re-entrant. So, they cannot be called recursively[17].

However, in most of cases our code can be rearranged to address this limitation. In the following example the blinking code is executed inside the `main()` function, leaving to the ISR only the responsibility to setup the global `blink` variable.

```
50    /*Configure GPIO pin : PC12 & PC13 */
51    GPIO_InitStruct.Pin = GPIO_PIN_12 | GPIO_PIN_13;
52    GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
53    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
54    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
55
56    /*Configure GPIO pin : PA5 */
57    GPIO_InitStruct.Pin = GPIO_PIN_5;
58    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
59    GPIO_InitStruct.Pull = GPIO_NOPULL;
60    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
61    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
62
63    HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_1);
64    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
65    HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0x0, 0);
66
67    while(1) {
68      if(blink) {
69        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
70        for(int i = 0; i < 100000; i++);
71      }
72    }
73  }
74
75  void EXTI15_10_IRQHandler(void) {
76    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_12);
77    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
78  }
79
80  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
81    if(GPIO_Pin == GPIO_PIN_13)
82      blink = 1;
83    else
84      blink = 0;
85  }
```

[17]Joseph Yiu shows a way to bypass this limitation in his books. However, I strongly discourage from using these tricky techniques unless you **really** need interrupt re-entrancy in your application.

7.6 Mask All Interrupts at Once or an a Priority Basis

Sometimes we want to be sure that our code is not preempted to allow the execution of interrupts or more privileged code. That is, we want to ensure that our code is thread-safe. Cortex-M based processors allow to temporarily mask the execution of all interrupts and exceptions, without disabling one by one. Two special registers, named `PRIMASK` and `FAULTMASK` allow to disable all interrupts and exceptions respectively.

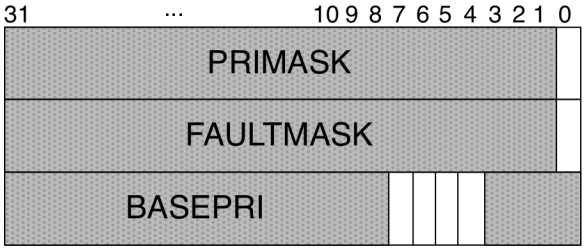


Figure 7.22: `PRIMASK`, `FAULTMASK` and `BASEPRI` registers

Even if these registers are 32-bit wide, just the first bit is used to enable/disable interrupts and exceptions. The ARM assembly instruction `CPSID i` disables all interrupt by setting the `PRIMASK` bit to 1, while the `CPSIE i` instructions enables them by setting `PRIMASK` to zero. Instead, the instruction `CPSID f` disables all exceptions (except for the NMI one) by setting the `FAULTMASK` bit to 1, while the `CPSIE f` instructions enables them.

The CMSIS-Core package provides several macros that we can use to perform these operation: `__disable_irq()` and `__enable_irq()` automatically set and clear the `PRIMASK`. Any critical task can be placed between these two macros, as shown below:

```
...
__disable_irq();
/* All exceptions with configurable priority are temporarily disabled.
   You can place critical code here */
...
__enable_irq();
```

However, take in mind that, as general rule, interrupt must be masked only for really short time, otherwise you could lose important interrupts. Remember that interrupts are not queued.

Another macro we can use is the `__set_PRIMASK(x)` one, where `x` is the content of the `PRIMASK` register (0 or 1). The macro `__get_PRIMARK()` returns the content of the `PRIMASK` register. Instead, the macros `__set_FAULTMASK(x)` and `__get_FAULTMASK()` allow to manipulate the `FAULTMASK` register.

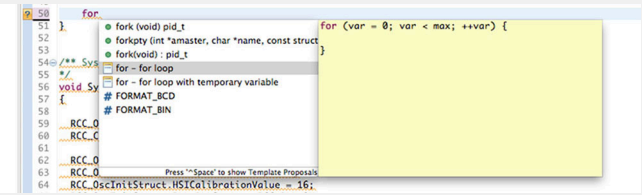
It is important to remark that, once the `PRIMASK` register is again set to zero, all pending interrupts are serviced according to their priority: `PRIMASK` causes that the the interrupt pending bit is set but the ISR is not serviced. This is the reason why we say that interrupts are *masked* and not disabled. Interrupts start to be serviced **as soon as** the `PRIMASK` is cleared.

Cortex-M3/4/7/33 cores allow to selectively mask interrupts on a priority basis. The `BASEPRI` register masks exceptions or interrupts on a priority level. The width of the `BASEPRI` register is the same of the `IPR` one, which lasts for the upper `4` bits in STM32 MCUs based on Cortex-M3/4/7 cores and `3` bits in STM32 MCUs based on Cortex-M33. When BASEPRI is set to `0`, it is disabled. When it is set to a non-zero value, it blocks exceptions (including interrupts) that have the same or lower priority level, while still allowing exceptions with a higher priority level to be accepted by the processor. For example, if the `BASEPRI` register is set to `0x60`, then all interrupts with a priority between `0x60-0xFF` are disabled. Remember that in Cortex-M cores the higher is the priority number the lower is the interrupt priority level. The `__set_BASEPRI(x)` macro allows to set the content of the `BASEPRI` register: remember, again, that the HAL automatically shifts the priority levels to the MSB bits. So, if we want to disable all interrupts with a priority higher than `2`, then we have to pass to the `__set_BASEPRI()` macro the value `0x20`. Alternatively, we can use the following code:

```
__set_BASEPRI(2 << (8 - __NVIC_PRIO_BITS));
```

Eclipse Intermezzo

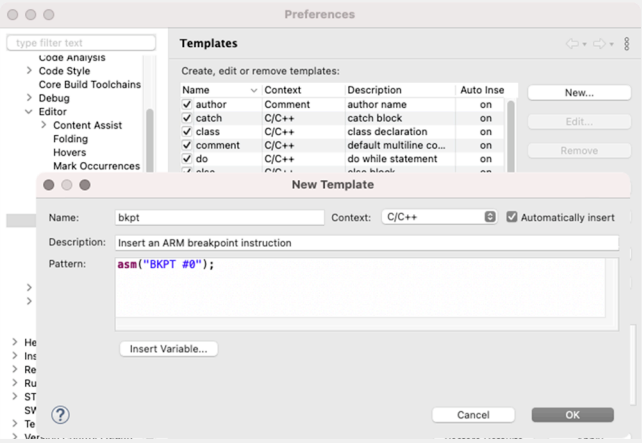
When coding, productivity is important for every developer. Modern source code editors allow to define custom code snippets, that is fragments of source code that are automatically inserted by the editor when a given "keyword" is typed. Eclipse calls this functionality "code templates", and they can be invoked by issuing a **Ctrl+Space** right after a keyword is written. For example, open a source file and write the keyword "`for`" and right after it hit **Ctrl+Space**. A contextual menu pops up, as shown in the following picture.



By choosing the entry "**for - for loop**", Eclipse will automatically place a new `for` loop inside the code. Now note a thing: the loop variable `var` is highlighted, as shown in the following picture.



If you write the new name for the loop variable Eclipse will automatically change it is name in all three places. Eclipse defines its set of code templates, but the good news is that you can define yours! Go inside Eclipse preferences, and then into **C/C++->Editor->Templates**. Here you can find all pre-defined code snippets and you can eventually add yours.



For example, we can add a new code template that insert a software breakpoint instruction (`asm("BKPT #0");`) when we write the keyword `bkpt`, as shown in the previous picture. Code templates are highly customizable, thanks to the usage of variables and other pattern constructs. For more information, refer to the Eclipse documentation[a].

[a] http://bit.ly/2c3Vm1K

8. Universal Asynchronous Serial Communications

Nowadays there are many serial communication protocols and hardware interfaces available in the electronics industry. The most of them are focused on high transmission bandwidths, like the more recent USB 2.0 and 3.x standards, the Firewire (IEEE 1394) and so on. Some of these standards come from the past but are still widespread especially as communication interface between modules on the same board. One of this is the *Universal Synchronous/Asynchronous Receiver/Transmitter* interface, also simply known as USART.

Almost every microcontroller provides at least one UART peripheral. Almost all STM32 MCUs provide at least two UART/USART interfaces, but the most of them provide more than two interfaces (some up to eight interfaces) according to the number of I/O supported by the MCU package.

In this Chapter we will see how to program this useful peripheral using the CubeHAL. Moreover, we will study how to develop applications using the UART both in *polling* and *interrupt* modes, leaving the third operative mode, the *DMA*, to the next chapter.

8.1 Introduction to UARTs and USARTs

Before we start diving into the analysis of the functions provided by the HAL to manipulate universal serial devices, it is best to take a brief look at the UART/USART interface and its communication protocol.

When we want two exchange data between two (or even more) devices, we have two alternatives: we can transmit it in parallel, that is using a given number of communication lines equal to the size of each data word (e.g., eight independent lines for a word made of eight bits), or we can transmit each bit constituting our word one by one. A UART/USART is a device that translates a parallel sequence of bits (usually grouped in a byte) in a continuous stream of signals flowing on a single wire.

When the information flows between two devices inside a common channel, both devices (here, for simplicity, we will refer to them as *the sender* and *the receiver*) have to agree on the *timing*, that this how long it takes to transmit each individual bit of the information. In a **synchronous transmission**, the sender and the receiver share a common clock generated by one of the two devices (usually the device that acts as *the master* of this interconnection system).

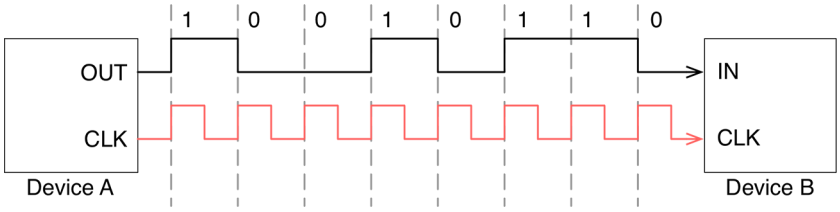


Figure 8.1: A serial communication between two devices using a shared clock source

In **Figure 8.1** we have a typical timing diagram[1] showing the *Device A* sending one byte (`0b01101001`) serially to the *Device B* using a common reference clock. The common clock is also used to agree on when to start *sampling* the sequence of bits: when the master device starts *clocking* the dedicated line, it means that it is going to send a sequence of bits.

In a **synchronous transmission** the transmission speed and duration are defined by the clock: its frequency determines how fast we can transmit a single byte on the communication channel[2]. But if both devices involved in data transmission agree on how long it takes to transmit a single bit and when to start and finish to sample transmitted bits, then we can avoid using a dedicated clock line. In this case we have an **asynchronous transmission**.

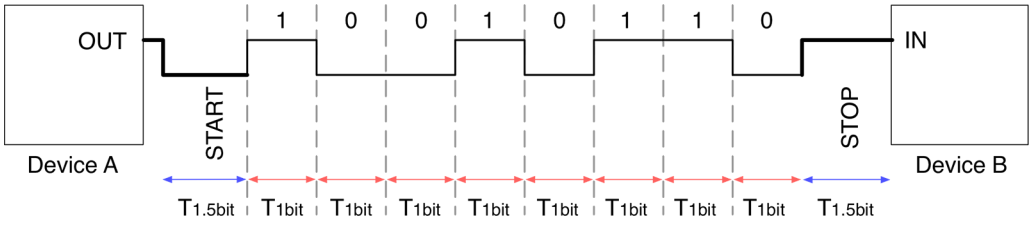


Figure 8.2: The timing diagram of a serial communication without a dedicated clock line

Figure 8.2 shows the timing diagram of an asynchronous transmission. The idle state (that is, no transmission occurring) is represented by the high signal. Transmission begins with a **START** bit, which is represented by the low level. The negative edge is detected by the receiver and 1.5 bit periods after this (indicated in Figure 8.7.1s $T_{1.5bit}$), the sampling of bits begins. Eight data bits are sampled. The least significant bit (LSB) is typically transmitted first. An optional parity bit is then transmitted (for error checking of the data bits). Often this bit is omitted if the transmission channel is assumed to be noise free or if there are error checking higher up in the protocol layers. The transmission is ended by a STOP bit, which last 1.5 bits.

[1] A Timing Diagram is a representation of a set of signals in the time domain.

[2] However, keep in mind that the maximum transmission speed is determined by a lot of other things, like the characteristics of the electrical channel, the ability of each device involved in transmission to sample fast signals, and so on.

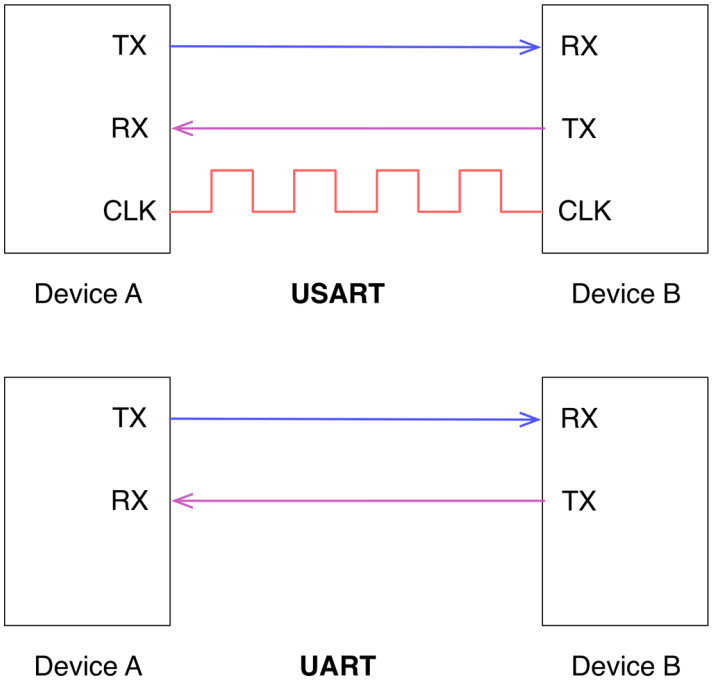


Figure 8.3: The signaling difference between a USART and a UART

A *Universal Synchronous Receiver/Transmitter* interface is a device able to transmit data word serially using two I/Os, one acting as transmitter (TX) and one as receiver (RX), plus one additional I/O as one clock line, while a *Universal Asynchronous Receiver/Transmitter* uses only two RX/TX I/Os (see **Figure 8.3**). Traditional we refer to the first interface with the term **USART** and to the second one with the term **UART**.

A UART/USART defines the signaling method, but it says nothing about the voltage levels. This means that an STM32 UART/USART will use the voltage levels of the MCU I/Os, which is almost equal to VDD (it is also common to refer to these voltage levels as *TTL voltage levels*). The way these voltage levels are translated to allow serial communication outside the board is demanded to other communication standards. For example, the EIA-RS232 or EIA-RS485 are two popular standards that define signaling voltages, in addition to their timing and meaning, and the physical size and pinout of connectors. Moreover, UART/USART interfaces can be used to exchange data using other physical and logical serial interfaces. For example, the FT232RL is a popular IC that allows to map a UART to a USB interface, as shown in **Figure 8.4**.

The presence of a dedicated clock line, or a common agreement about transmission frequency, does not guarantee that the receiver of a byte stream is able to process them at the same transmission rate of the master. For this reason, some communication standards, like the RS232 and the RS485, provide the possibility to use a dedicated *Hardware Flow Control* line. For example, two devices communicating using the RS232 interface can share two additional lines, named *Request To Send*(RTS) and *Clear To Send*(CTS): the sender sets its RTS, which signals the receiver to begin monitoring its data input line. When ready for data, the receiver will raise its complementary line, CTS, which signals the sender to start sending data, and for the sender to begin monitoring the slave's data output line.

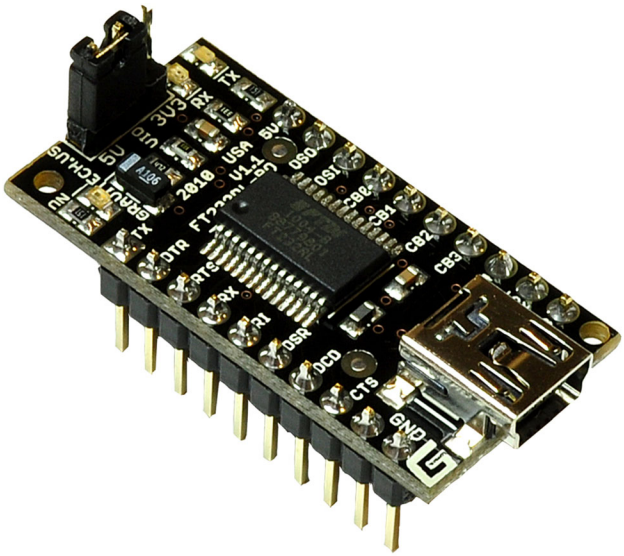


Figure 8.4: A typical circuit based on FT232RL used to convert a 3.3V TTL UART interface to USB

STM32 microcontrollers provide a variable number of USARTs, which can be configured to work both in *synchronous* and *asynchronous* mode. Some STM32 MCUs also provide interfaces only able to act as UART. **Table 8.1** lists the UART/USARTs provided by STM32 MCUs equipping all Nucleo boards used in this text. The most of USARTs are also able to automatically implement *Hardware Flow Control*, both for the RS232 and the RS485 standards.

All Nucleo-64 boards are designed so that the USART2 of the target MCU is linked to the ST-LINK interface[3]. When we install the ST-LINK drivers, an additional driver for the *Virtual COM Port*(VCP) is also installed: this allows us to access to the target MCU USART2 using the USB interface, without using a dedicated TTL/USB converter. Using a terminal emulation program, we can exchange messages and data with our Nucleo.

The CubeHAL separates the API for the management of UART and USART interfaces. All functions and C type handlers used for the handling of USARTs start with the `HAL_USART` prefix and are contained inside the files `stm32xxx_hal_usart.{c,h}`, while those related to UARTs management start with the `HAL_UART` prefix and are contained inside the files `stm32xxx_hal_uart.{c,h}`. Since both the modules are conceptually identical, and since the UART is the most common form of serial interconnection between different modules, this book will only cover the features of the `HAL_UART` module.

[3]Please take note that this statement may not be true if you are using a Nucleo-32 or Nucleo-144 board. Check the ST documentation for more about this.

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

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

8.2 UART Initialization

Like all STM32 peripherals, even the USARTs[4] are mapped in the memory mapped peripheral region, which starts from `0x4000 0000`. The CubeHAL abstracts the effective location of each USART for a given STM32 MCU thanks to the `USART_TypeDef`[5] descriptor. For example, we can simply use the `USART2` macro to refer to the second USART peripheral provided by all STM32 microcontrollers with LQFP64 package.

However, all the HAL functions related to UART management are designed so that they accept as first parameter an instance of the C `struct UART_HandleTypeDef`, which is defined in the following way[6]:

[4]Starting from this paragraph, the terms USART and UART are used interchangeably, unless different noticed.
[5]The analysis of the fields of this C `struct` is outside of the scope of this book.
[6]Please, take note that the list of fields of the `struct UART_HandleTypeDef` is not complete. Several fields, not relevant to the topics covered in this chapter, are omitted here. For more information, refer to the CubeHAL.

```
typedef struct {
  USART_TypeDef                 *Instance;        /* UART registers base address        */
  UART_InitTypeDef              Init;             /* UART communication parameters      */
  UART_AdvFeatureInitTypeDef    AdvancedInit;     /* UART Advanced Features initialization
                                                     parameters */
  uint8_t                       *pTxBuffPtr;      /* Pointer to UART Tx transfer Buffer */
  uint16_t                      TxXferSize;       /* UART Tx Transfer size              */
  uint16_t                      TxXferCount;      /* UART Tx Transfer Counter           */
  uint8_t                       *pRxBuffPtr;      /* Pointer to UART Rx transfer Buffer */
  uint16_t                      RxXferSize;       /* UART Rx Transfer size              */
  uint16_t                      RxXferCount;      /* UART Rx Transfer Counter           */
  DMA_HandleTypeDef             *hdmatx;          /* UART Tx DMA Handle parameters      */
  DMA_HandleTypeDef             *hdmarx;          /* UART Rx DMA Handle parameters      */
  HAL_LockTypeDef               Lock;             /* Locking object                     */
  __IO HAL_UART_StateTypeDef    gState;           /* UART communication state           */
  __IO HAL_UART_ErrorTypeDef    ErrorCode;        /* UART Error code                    */
} UART_HandleTypeDef;
```

Let us see more in depth the most important fields of this `struct`.

- `Instance`: is the pointer to the USART descriptor we are going to use (that is, the base address in memory where the peripheral is mapped). For example, `USART2` is the descriptor of the UART associated to the ST-LINK interface of every Nucleo board.
- `Init`: is an instance of the C `struct UART_InitTypeDef`, which is used to configure the UART interface. We will study it more in depth in a while.
- `AdvancedInit`: this field is used to configure more advanced UART features like the automatic *BaudRate* detection and the TX/RX pin swapping. Some HALs do not provide this additional field. This happens because USART interfaces are not equal for all STM32 MCUs. This is an important aspect to keep in mind while choosing the right MCU for your application. The analysis of this field is outside the scope of this book.
- `pTxBuffPtr` and `pRxBuffPtr`: these fields point to the transmit and receive buffer respectively. They are used as source to transmit `TxXferSize` bytes over the UART and to receive `RxXferSize` when the UART is configured in Full Duplex Mode. The `TxXferCount` and `RxXferCount` fields are used internally by the HAL to take count of transmitted and received bytes.
- `Lock`: this field is used internally by the HAL to lock concurrent accesses to UART interfaces.



As said above, the `Lock` field is used to rule concurrent accesses in almost all HAL routines. If you take a look at the HAL code, you can see several uses of the `__HAL_LOCK()` macro, which is expanded in this way:

```
#define __HAL_LOCK(__HANDLE__)                \
    do{                                       \
       if((__HANDLE__)->Lock == HAL_LOCKED)   \
       {                                      \
          return HAL_BUSY;                    \
       }                                      \
       else                                   \
       {                                      \
          (__HANDLE__)->Lock = HAL_LOCKED;    \
       }                                      \
    }while (0)
```

It is not clear why ST engineers decided to take care of concurrent accesses to the HAL routines. Probably they decided to have a *thread safe* approach, freeing the application developer from the responsibility of managing multiple accesses to the same hardware interface in case of multiple threads running in the same application.

However, this has an annoying side effect for all HAL users: even if my application does not perform concurrent accesses to the same peripheral, my code will be poor optimized by a lot of checks about the state of the `Lock` field. Moreover, that way to lock is intrinsically thread unsafe, because there is no critical section used to prevent race conditions in case a more privileged ISR preempts the running code. Finally, if my application uses an RTOS, it is much better to use native OS locking primitives (like semaphores and mutexes which are not only *atomic*, but also correctly manages the task scheduling avoiding the *busy waiting*) to handle concurrent accesses, without the need to check for a particular return value (`HAL_BUSY`) of the HAL functions.

A lot of developers have disapproved[7] this way to lock peripherals since the first release of the HAL. ST engineers announced several years ago that they are actively working on a better solution. However, there are no news at the moment.

All the UART configuration activities are performed by using an instance of the C `struct UART_InitTypeDef`, which is defined in the following way:

[7] https://bit.ly/3nOo63u

```
typedef struct {
  uint32_t BaudRate;
  uint32_t WordLength;
  uint32_t StopBits;
  uint32_t Parity;
  uint32_t Mode;
  uint32_t HwFlowCtl;
  uint32_t OverSampling;
} UART_InitTypeDef;
```

- `BaudRate`: this parameter refers to the connection speed, expressed in bits per seconds. Even if the parameter can assume an arbitrary value, usually the *BaudRate* comes from a list of well-known and standard values. This because it is a function of the peripheral clock associated to the USART (that is derived from the main HSI or HSE clock by a chain of PLLs and multipliers in some STM32 MCU), and not all *BaudRate*s can be easily achieved without introducing sampling errors, and hence communication errors. **Table 8.2** shows the list of common *BaudRates*, and the related error calculation, for an STM32F072 MCU. Always consult the reference manual for your MCU to see which peripheral clock frequency best fits the needed *BaudRate* on the given STM32 microcontroller.

	Baud rate		Oversampling by 16		Oversampling by 8	
	S.No	Desired (Bps)	Actual	%Error	Actual	%Error
	2	**2400**	2400	0	2400	0
	3	**9600**	9600	0	9600	0
	4	**19200**	19200	0	19200	0
	5	**38400**	38400	0	38400	0
	6	**57600**	57620	0.03	57590	0.02
	7	**115200**	115110	0.08	115250	0.04
	8	**230400**	230760	0.16	230210	0.8
	9	**460800**	461540	0.16	461540	0.16
	10	**921600**	923070	0.16	923070	0.16
	11	**2000000**	2000000	0	2000000	0
	12	**3000000**	3000000	0	3000000	0
	13	**4000000**	N.A.	N.A.	4000000	0
	14	**5000000**	N.A.	N.A.	5052630	1.05
	15	**6000000**	N.A.	N.A.	6000000	0

Table 8.2: Error calculation for programmed baud rates at 48 MHz in both cases of oversampling by 16 or by 8

- `WordLength`: it specifies the number of data bits transmitted or received in a frame. This field can assume the value `UART_WORDLENGTH_8B` or `UART_WORDLENGTH_9B`, which means that we can transmit over a UART packets containing `8` or `9` data bits. This number does not include the overhead bits transmitted, such as the start and stop bits.
- `StopBits`: this field specifies the number of stop bits transmitted. It can assume the value `UART_STOPBITS_1` or `UART_STOPBITS_2`, which means that we can use one or two stop bits to signal the end of the frame.
- `Parity`: it indicates the parity mode. This field can assume the values from **Table 8.3**. Take note that, when parity is enabled, the computed parity is inserted at the MSB position of the transmitted data (9th bit when the word length is set to `9` data bits; 8th bit when the word length is set to `8` data bits). Parity is a very simple form of error checking. It comes in two flavors: *odd* or *even*. To produce the parity bit, all data bits are added up, and the evenness of the sum decides whether the bit is set or not. For example, assuming parity is set to *even* and was being added to a data byte like `0b01011101`, which has an odd number of 1's (5), the parity bit would be set to `1`. Conversely, if the parity mode was set to odd, the parity bit would be `0`. Parity is optional, and not very widely used. It can be helpful for transmitting across noisy mediums, but it will also slow down data transfer a bit and requires both sender and receiver to implement error-handling (usually, received data that fails must be re-sent). When a *parity error* occurs, all STM32 MCUs generate a specific interrupt, as we will see next.
- `Mode`: it specifies whether the RX or TX mode is enabled or disabled. This field can assume one of the values from **Table 8.4**.
- `HwFlowCtl`: it specifies whether the RS232[8] Hardware Flow Control mode is enabled or disabled. This parameter can assume one of the values from **Table 8.5**.

Table 8.3: Available parity modes for a UART connection

Parity Mode	Description
`UART_PARITY_NONE`	No parity check enabled
`UART_PARITY_EVEN`	The parity bit is set to `1` if the count of bits equal to `1` is odd
`UART_PARITY_ODD`	The parity bit is set to `1` if the count of bits equal to `1` is even

Table 8.4: Available UART modes

UART Mode	Description
`UART_MODE_RX`	The UART is configured only in receive mode
`UART_MODE_TX`	The UART is configured only in transmit mode
`UART_MODE_TX_RX`	The UART is configured to work bot in receive an transmit mode

[8] this field is only used to enable the RS232 flow control. To enable the RS485 flow control, the HAL provides a specific function, `HAL_RS485Ex_Init()`, defined inside the `stm32XXxx_hal_uart_ex.c` file.

Table 8.5: Available flow control mode for a UART connection

Flow Control Mode	Description
UART_HWCONTROL_NONE	The Hardware Flow Control is disabled
UART_HWCONTROL_RTS	The *Request To Send*(RTS) line is enabled
UART_HWCONTROL_CTS	The *Clear To Send*(CTS) line is enabled
UART_HWCONTROL_RTS_CTS	Both RTS and CTS lines enabled

- OverSampling: when the UART receives a frame from the remote peer, it samples the signals in order to compute the number of 1 and 0 constituting the message. *Oversampling* is the technique of sampling a signal with a sampling frequency significantly higher than the Nyquist rate. The receiver implements different user-configurable oversampling techniques (except in synchronous mode) for data recovery by discriminating between valid incoming data and noise. This allows a trade-off between the maximum communication speed and noise/clock inaccuracy immunity. The OverSampling field can assume the value UART_OVERSAMPLING_16 to perform 16 samples for each frame bit or UART_OVERSAMPLING_8 to perform 8 samples. **Table 8.2** shows the error calculation for programmed baud rates at 48 MHz in an STM32F072 MCU in both cases of oversampling by 16 or by 8.

Now it is a good time to start writing down a bit of code. Let us see how to configure the USART2 of the MCU equipping our Nucleo to exchange messages through the ST-LINK interface.

```
int main(void) {
  UART_HandleTypeDef huart2;

  /* Initialize the HAL */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Configure the USART2 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 38400;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  HAL_UART_Init(&huart2);
  ...
}
```

The first step is to configure the USART2 peripheral. Here we are using this configuration: 38400, N, 1. That is, a *BaudRate* equal to 38400 Bps, no parity check and just one stop bit. Next, we disable any

form of Hardware Flow Control, and we choose the highest oversampling rate, that is 16 clock ticks for each transmitted bit. The call to the `HAL_UART_Init()` function ensures that the HAL initializes the USART2 according the given options.

However, the above code is still not sufficient to exchange messages through the Nucleo Virtual COM Port. Don't forget that every peripheral designed to exchange data with the outside world must be properly bound to corresponding GPIOs, that is we have to configure the USART2 TX and RX pins. Looking to the Nucleo schematics, we can see that USART2 TX and RX pins are PA2 and PA3 respectively. Moreover, we have already seen in Chapter 4 that the HAL is designed so that `HAL_UART_Init()` function automatically calls the `HAL_UART_MspInit()` (see **Figure 4.15** in Chapter 4) to properly initialize the I/Os: it is our responsibility to write this function in our application code, which we will be automatically called by the HAL.



Is It Mandatory to Define This Function?

The answer is simply no. This is just a practice enforced by the HAL and by the code automatically generated by CubeMX. The `HAL_UART_MspInit()`, and the corresponding function `HAL_UART_MspDeInit()` which is called by the `HAL_UART_DeInit()` function, are declared inside the HAL in this way:

```
__weak void HAL_UART_MspInit(UART_HandleTypeDef *huart);
```

The function attribute `__weak` is a GCC way to declare a symbol (here, a function name) with a weak scope visibility, which we will be overwritten if another symbol with the same name with a global scope (that is, without the __weak attribute) is defined elsewhere in the application (that is, in another relocatable file). The linker will automatically substitute the call to the function `HAL_UART_MspInit()` defined inside the HAL if we implement it in our application code.

The code below shows how to correctly code the `HAL_UART_MspInit()` function.

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart) {
  GPIO_InitTypeDef GPIO_InitStruct;
  if(huart->Instance==USART2) {
    /* Peripheral clock enable */
    __HAL_RCC_USART2_CLK_ENABLE();

    /**USART2 GPIO Configuration
    PA2     ------> USART2_TX
    PA3     ------> USART2_RX
    */
    GPIO_InitStruct.Pin = USART_TX_Pin|USART_RX_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
```

```
    GPIO_InitStruct.Alternate = GPIO_AF1_USART2; /* WARNING: this depends on
                                                    the specific STM32 MCU */
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
  }
}
```

As you can see, the function is designed so that it is common for every USART used inside the application. The `if` statement disciplines the initialization code for the given USART (in our case, USART2). The remaining of code configures the PA2 and PA3 pins. **Please, take note that the alternate function may change for the MCU equipping your Nucleo**. Consult the book examples to see the right initialization code for your Nucleo.

Once we have configured the USART2 interface, we can start exchanging messages with our PC.

F334 F303



Please, take note that the code presented before could not be sufficient to correctly initialize the USART peripheral for some STM32 MCUs. Some STM32 microcontrollers, like the STM32F334R8, allow the developer to choose the clock source for a given peripheral (for example, the USART2 in an STM32F334R8 MCU can be optionally clocked from `SYSCLK`, `HSI`, `LSE` or `PCLK1`). It is strongly suggested to use CubeMX the first time you configure the peripherals for your MCU and to check carefully the generated code looking for this kind of exceptions. Otherwise, the datasheet is the only source for this information.

8.2.1 UART Configuration Using CubeMX

As said before, the first time we configure the USART2 for our Nucleo it is best to use CubeMX. The first step is enabling the USART2 peripheral inside the *Pinout* view: click on **USART2** entry inside the **Connectivity** section and then select the *Asynchronous* entry from the *Mode* combo box inside the **USART2 Mode and Configuration** pane, as shown in **Figure 8.5**. Both PA2 and PA3 pins will be automatically highlighted in green. Then, go inside the *Configuration* section and click on the **USART2** button. By using the Configuration pane, you can set additional options such as the *BaudRate*, word length and so on[9].

Once we have configured the USART interface, we can generate the C code. You will notice that CubeMX places all the USART2 initialization code inside the `MX_USART2_UART_Init()` (which is contained in the `main.c` file). Instead, all the code related to GPIO configuration is placed into the `HAL_UART_MspInit()` function, which is contained inside the `stm32XXxx_hal_msp.c` file.

[9]Some of you, especially those having a Nucleo-F3, will notice that the configuration pane may contain a more settings compared to those shown in **Figure 8.5**. Please, refer to the reference manual for your target MCU for more information.

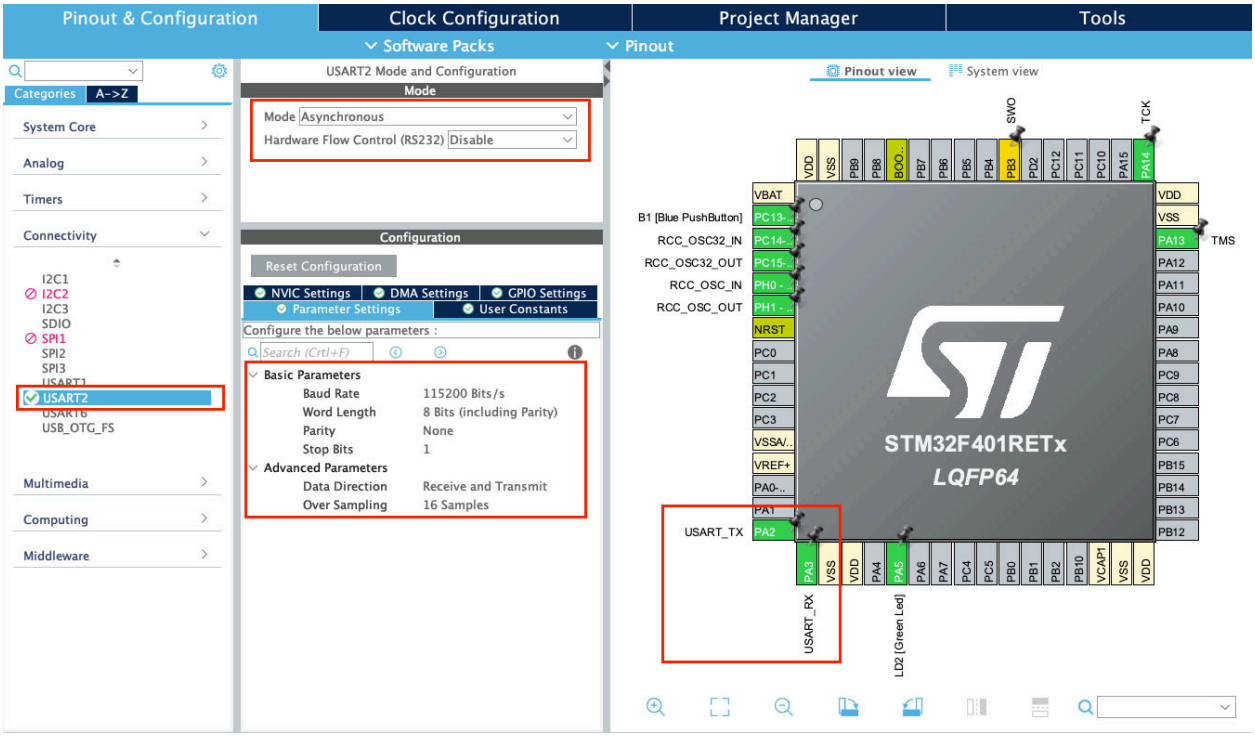


Figure 8.5: CubeMX can be used to configure the UART2 interface easily

8.3 UART Communication in *Polling Mode*

STM32 microcontrollers, and hence the CubeHAL, offer three ways to exchange data between peers over a UART communication: *polling*, *interrupt* and *DMA* mode. It is important to stress right from now that these modes are not only three different flavors to handle UART communications. They are three different programming approach to the same task, which introduce several benefits both from the design and performance point of view. Let us introduce them briefly.

- In *polling mode*, also called *blocking mode*, the main application, or one of its threads, synchronously waits for the data transmission and reception. This is the simplest form of data communication using this peripheral, and it can be used when the transmit rate is not too much low and when the UART is not used as critical peripheral in our application (the classical example is the usage of the UART as output console for debug activities).
- In *interrupt mode*, also called *non-blocking mode*, the main application is freed from waiting for the completion of data transmission and reception. The data transfer routines terminate as soon as they complete to configure the peripheral. When the data transmission ends, a subsequent interrupt will signal the main code about this. This mode is more suitable when communication speed is low (below 38400 Bps) or when it happens “rarely”, compared to other activities performed by the MCU, and we do not want to stick it waiting for data transmission.
- *DMA mode* offers the best data transmission throughput, thanks to the direct access of the UART peripheral to MCU internal RAM. This mode is best for high-speed communications and

when we totally want to free the MCU from the overhead of data transmission. Without the *DMA mode*, it is almost impossible to reach the fastest transfer rates that the USART peripheral is capable to handle. In this chapter we will not see this USART communication mode, leaving it to the next chapter dedicated to DMA management.

To transmit a sequence of bytes over the USART in *polling mode* the HAL provides the function

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData,
                                    uint16_t Size, uint32_t Timeout);
```

where:

- `huart`: it is the pointer to an instance of the `struct UART_HandleTypeDef` seen before, which identifies and configures the UART peripheral;
- `pData`: is the pointer to an array, with a length equal to the `Size` parameter, containing the sequence of bytes we are going to transmit;
- `Timeout`: is the maximum time, expressed in milliseconds, we are going to wait for the transmit completion. If the transmission does not complete in the specified timeout time, the function aborts and returns the `HAL_TIMEOUT` value; otherwise, it returns the `HAL_OK` value if no other errors occur. Moreover, we can pass a timeout equal to `HAL_MAX_DELAY` (`0xFFFF FFFF`) to wait indefinitely for the transmit completion.

Conversely, to receive a sequence of bytes over the USART in polling mode the HAL provides the function

```
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart, uint8_t *pData,
                                   uint16_t Size, uint32_t Timeout);
```

where:

- `huart`: it is the pointer to an instance of the `struct UART_HandleTypeDef` seen before, which identifies and configures the UART peripheral;
- `pData`: is the pointer to an array, with a length at least equal to the `Size` parameter, containing the sequence of bytes we are going to receive. The function will block until all bytes specified by the `Size` parameter are received.
- `Timeout`: is the maximum time, expressed in milliseconds, we are willing to wait for the receive completion. If the transmission does not complete in the specified timeout time, the function aborts and returns the `HAL_TIMEOUT` value; otherwise, it returns the `HAL_OK` value if no other errors occur. Moreover, we can pass a timeout equal to `HAL_MAX_DELAY` (`0xFFFF FFFF`) to wait indefinitely for the receive completion.



Read Carefully

It is important to remark that the timeout mechanism offered by the two functions works only if the `HAL_IncTick()` routine is called every 1ms, as done by the code generated by CubeMX (the function that increments the HAL tick counter is called inside the SysTick timer ISR).

Ok. Now it is the right time to see an example.

Filename: `src/main-ex1.c`

```
21  int main(void) {
22    uint8_t opt = 0;
23
24    /* Reset of all peripherals, Initializes the Flash interface and the SysTick. */
25    HAL_Init();
26
27    /* Configure the system clock */
28    SystemClock_Config();
29
30    /* Initialize all configured peripherals */
31    MX_GPIO_Init();
32    MX_USART2_UART_Init();
33
34  printMessage:
35
36    printWelcomeMessage();
37
38    while (1)  {
39      opt = readUserInput();
40      processUserInput(opt);
41      if(opt == 3)
42        goto printMessage;
43    }
44  }
45
46  void printWelcomeMessage(void) {
47    HAL_UART_Transmit(&huart2, (uint8_t*)"\033[0;0H", strlen("\033[0;0H"), HAL_MAX_DELAY);
48    HAL_UART_Transmit(&huart2, (uint8_t*)"\033[2J", strlen("\033[2J"), HAL_MAX_DELAY);
49    HAL_UART_Transmit(&huart2, (uint8_t*)WELCOME_MSG, strlen(WELCOME_MSG), HAL_MAX_DELAY);
50    HAL_UART_Transmit(&huart2, (uint8_t*)MAIN_MENU, strlen(MAIN_MENU), HAL_MAX_DELAY);
51  }
52
53  uint8_t readUserInput(void) {
54    char readBuf[1];
55
56    HAL_UART_Transmit(&huart2, (uint8_t*)PROMPT, strlen(PROMPT), HAL_MAX_DELAY);
57    HAL_UART_Receive(&huart2, (uint8_t*)readBuf, 1, HAL_MAX_DELAY);
```

```
58    return atoi(readBuf);
59  }
60
61  uint8_t processUserInput(uint8_t opt) {
62    char msg[30];
63
64    if(!opt || opt > 3)
65      return 0;
66
67    sprintf(msg, "%d", opt);
68    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
69
70    switch(opt) {
71    case 1:
72      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
73      break;
74    case 2:
75      sprintf(msg, "\r\nUSER BUTTON status: %s",
76              HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET ? "PRESSED" : "RELEASED");
77      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
78      break;
79    case 3:
80      return 2;
81    };
82
83    return 1;
84  }
```

The example is a sort of bare-bone management console. The application starts printing a welcome message (lines 36) and then entering in a loop waiting for the user choice. The first option allows to toggle the LD2 LED, while the second to read the status of the USER button. Finally, the option 3 causes that the welcome screen is printed again.



The two strings `"\033[0;0H"` and `"\033[2J"` are *escape sequences*. They are standard sequences of chars used to manipulate the terminal console. The first one places the cursor in the top-left part of the available console screen, and the second one clears the screen.

To interact with this simple management console, we need a serial communication program. There are several options available. The easy one is to use a standalone program like `putty`[10] for the Windows platform (if you have an old Windows version, you can also consider to use the classical `HyperTerminal` tool), or `kermit`[11] for Linux and MacOS. However, we are now going to introduce a solution to have an integrated serial communication tool inside the STM32CubeIDE.

[10] http://bit.ly/1jsQjnt
[11] https://www.kermitproject.org/

8.3.1 Installing a Terminal Emulator in Eclipse

Eclipse provides a convenient and multi-platform plug-in that adds a terminal emulator inside the IDE without the need of external tools. The plug-in is freely available on the Eclipse Marketplace.

To install the plug-in, go to **Help->Eclipse Marketplace**.... In the **Find** text box write "terminal". After a while, the **TM Terminal** plug-in should appear, as shown in **Figure 8.6**. Click on the **Install** button and follow the instructions. Restart Eclipse when requested.

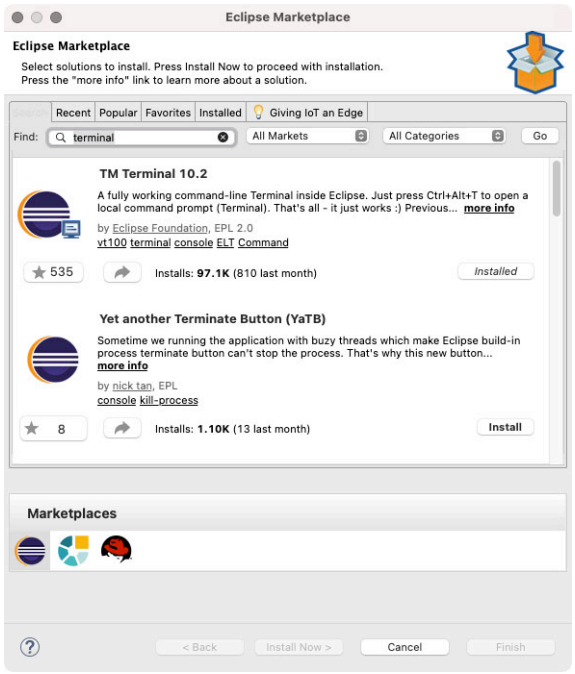


Figure 8.6: The Eclipse Marketplace

To open the Terminal panel, you can simply press **Ctrl+Alt+Shift+T**, or you can click the dedicated icon on the Eclipse's toolbar, as shown in **Figure 8.7**.

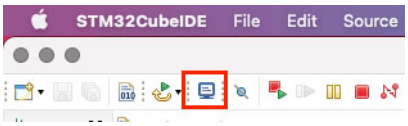


Figure 8.7: How to start a new terminal

The **Launch Terminal** dialog appears and select **Serial Terminal** as terminal type (see **Figure 8.8**), and then select the COM Port corresponding to the Nucleo VCP and set the same *Baud Rate.* configured in CubeMX. Click on the **OK** button.

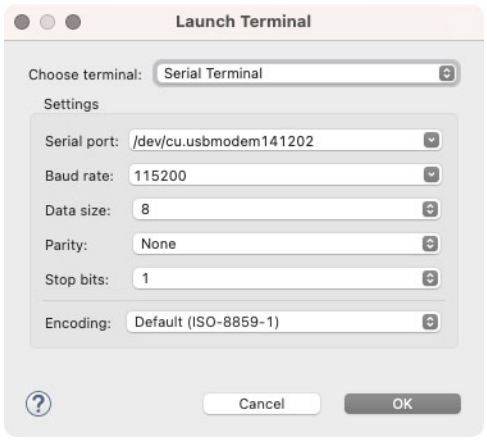


Figure 8.8: Terminal type selection dialog

Now you can reset the Nucleo. The management console we have programmed using the `HAL_UART` library should appear in the serial console window, as shown in **Figure 8.9**.

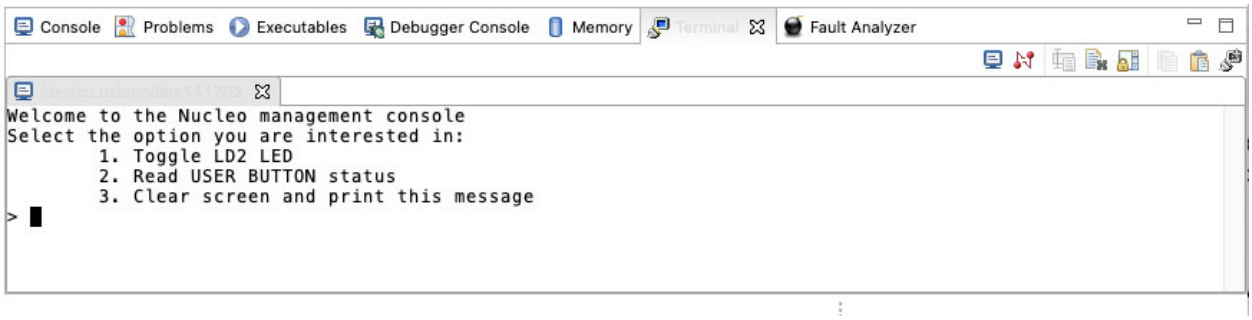


Figure 8.9: The Nucleo management console shown in the terminal view

8.4 UART Communication in *Interrupt Mode*

Let us consider again the first example of this chapter. What's wrong with it? Since our firmware is all committed to this simple task, there is nothing wrong by using the UART in polling mode. The MCU is essentially blocked waiting for the user input (the `HAL_MAX_DELAY` timeout value blocks the `HAL_UART_Receive()` until one char is sent over the UART). But what if our firmware has to do other CPU-intensive activities in real-time?

Suppose to rearrange the `main()` from the first example in the following way:

```
38   while (1)  {
39     opt = readUserInput();
40     processUserInput(opt);
41     if(opt == 3)
42       goto printMessage;
43
44     performCriticalTasks();
45   }
```

In this case we cannot block the execution of function `processUserInput()` waiting for the user choice, but we have to specify a much more short timeout value to the `HAL_UART_Receive()` function, otherwise `performCriticalTasks()` is never executed. However, this could cause the loss of important data coming from the UART peripheral (remember that the UART interface has a one-byte wide buffer).

To address this issue the HAL offers another way to exchange data over a UART peripheral: the *interrupt mode*. To use this mode, we have to do the following tasks:

- To enable the `USARTx_IRQn` interrupt and to implement the corresponding `USARTx_IRQHandler()` ISR.
- To call `HAL_UART_IRQHandler()` inside the `USARTx_IRQHandler()`: this will perform all activities related to management of interrupts generated by the UART peripheral[12].
- To use the functions `HAL_UART_Transmit_IT()` and `HAL_UART_Receive_IT()` to exchange data over the UART. These functions also enable the *interrupt mode* of the UART peripheral: in this way the peripheral will assert the corresponding line in the NVIC controller so that the ISR is raised when an event occurs.
- To design our application code to deal with asynchronous events.

Before we rearrange the code from the first example, it is best to look at the available UART interrupts and to the way HAL routines are designed.

8.4.1 UART Related Interrupts

Every STM32 USART peripheral provides the interrupts listed in **Table 8.6**. These interrupts include both IRQs related to data transmission and to communication errors. They can be divided in two groups:

- *IRQs generated during transmission*: **Transmission Complete**, **Clear to Send** or **Transmit Data Register Empty** interrupt.
- *IRQs generated while receiving*: **Idle Line detection**, **Overrun error**, **Receive Data register not empty**, **Parity error**, **LIN break detection**, **Noise Flag** (only in multi buffer communication) and **Framing Error** (only in multi buffer communication).

[12] If we use CubeMX to enable the `USARTx_IRQn` from the NVIC configuration section (as shown in Chapter 7), it will automatically place the call to the `HAL_UART_IRQHandler()` from the ISR.

Table 8.6: The list of USART related interrupts

Interrupt Event	Event Flag	Enable Control Bit
Transmit Data Register Empty	TXE	TXEIE
Clear To Send (CTS) flag	CTS	CTSIE
Transmission Complete	TC	TCIE
Received Data Ready to be Read	RXNE	RXNEIE
Overrun Error Detected	ORE	RXNEIE
Idle Line Detected	IDLE	IDLEIE
Parity Error	PE	PEIE
Break Flag	LBD	LBDIE
Noise Flag, Overrun error and Framing Error in multi buffer communication	NF or ORE or FE	EIE

These events generate an interrupt if the corresponding *Enable Control Bit* is set (third column of **Table 8.6**). However, STM32 MCUs are designed so that all these IRQs are bound to just one ISR for every USART peripheral (see **Figure 8.10**[13]). For example, the USART2 defines only the `USART2_IRQn` as IRQ for all interrupts generated by this peripheral. It is up to the user code to analyze the corresponding *Event Flag* to infer which interrupt has generated the request.

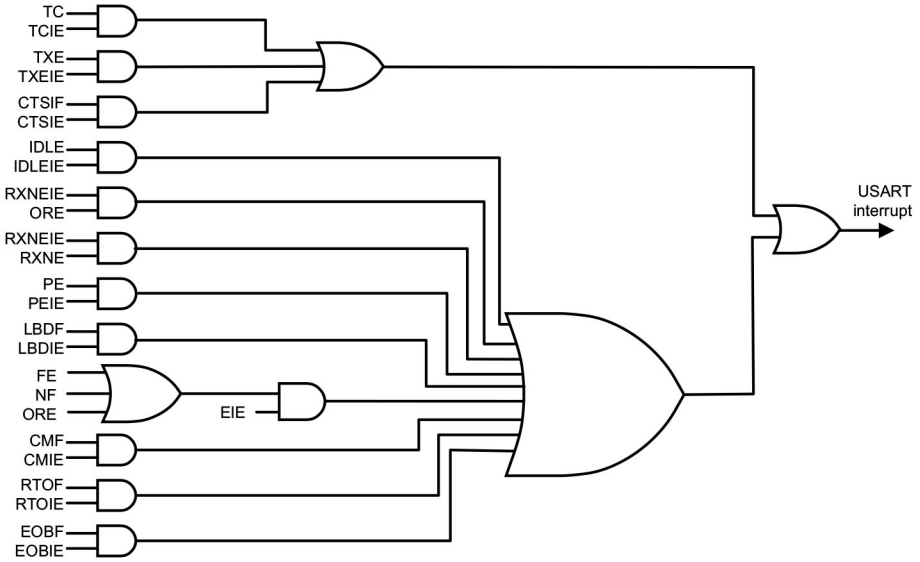


Figure 8.10: How the USART interrupt events are connected to the same interrupt vector

The CubeHAL is designed to automatically do this job for us. The user is warned about the interrupt generation thanks to a series of callback functions invoked by the `HAL_UART_IRQHandler()`, which must be invoked inside the ISR as stated before.

From a technical point of view, there is not so much difference between UART transmission in polling and in interrupt mode. Both the methods transfer an array of bytes using the UART *Data Register* (DR) with the following algorithm:

[13]The Figure 8.10 is taken from the STM32F030 Reference Manual (RM0390).

- For data transmission, place a byte inside the `USART->DR` register and wait until the *Transmit Data Register Empty*(TXE) flag is asserted true.
- For data reception, wait until the *Received Data Ready to be Read*(RXNE) is not asserted true, and then store the content of the `USART->DR` register inside the application memory.

The difference between the two methods consists in how they wait for the completion of data transmission. In polling mode, the `HAL_UART_Receive()/HAL_UART_Transmit()` functions are designed so that it waits for the corresponding event flag to be set, for every byte we want to transmit. In interrupt mode, the function `HAL_UART_Receive_IT()/HAL_UART_Transmit_IT()` are designed so that they do not wait for data transmission completion, but the dirty job to place a new byte inside the `DR` register, or to load its content inside the application memory, is accomplished by the ISR routine when the RXNEIE/TXEIE interrupt is generated[14].

To transmit a sequence of bytes in interrupt mode, the HAL defines the function:

```
HAL_StatusTypeDef HAL_UART_Transmit_IT(UART_HandleTypeDef *huart,
                                       uint8_t *pData, uint16_t Size);
```

where:

- `huart`: it is the pointer to an instance of the `struct UART_HandleTypeDef` seen before, which identifies and configures the UART peripheral;
- `pData`: it is the pointer to an array, with a length equal to the `Size` parameter, containing the sequence of bytes we are going to transmit; the function will not block waiting for the data transmission, and it will pass the control to the main flow as soon as it completes to configure the UART.

Conversely, to receive a sequence of bytes over the USART in interrupt mode the HAL provides the function:

```
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart,
                                      uint8_t *pData, uint16_t Size);
```

where:

- `huart`: it is the pointer to an instance of the `struct UART_HandleTypeDef` seen before, which identifies and configures the UART peripheral;
- `pData`: it is the pointer to an array, with a length at least equal to the `Size` parameter, containing the sequence of bytes we are going to receive. The function will not block waiting for the data reception, and it will pass the control to the main flow as soon as it completes to configure the UART.

Now we can proceed rearranging the first example.

[14] This is the reason why transferring a sequence of bytes in interrupt mode is not a smart thing when the communication speed is too high, or when we have to transfer a great amount of data very often. Since the transmission of each byte happens quickly, the CPU will be "flooded" by the interrupts generated by the UART for every byte transmitted. For continuous transmission of great sequences of bytes at high speed is best to use the DMA mode, as we will see in the next chapter.

Filename: `src/main-ex2.c`

```
55    /* Enable USART2 interrupt */
56    HAL_NVIC_SetPriority(USART2_IRQn, 0, 0);
57    HAL_NVIC_EnableIRQ(USART2_IRQn);
58
59  printMessage:
60
61    printWelcomeMessage();
62
63    while (1)  {
64      opt = readUserInput();
65      if(opt > 0) {
66        processUserInput(opt);
67        if(opt == 3)
68          goto printMessage;
69      }
70      performCriticalTasks();
71    }
72  }
73
74  void USART2_IRQHandler(void) {
75    HAL_UART_IRQHandler(&huart2);
76  }
77
78  void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle) {
79   /* Set transmission flag: transfer complete*/
80   UartReady = SET;
81  }
82
83  void printWelcomeMessage(void) {
84    char *strings[] = {"\033[0;0H", "\033[2J", WELCOME_MSG, MAIN_MENU, PROMPT};
85
86    for (uint8_t i = 0; i < 5; i++) {
87      HAL_UART_Transmit_IT(&huart2, (uint8_t*)strings[i], strlen(strings[i]));
88      while (HAL_UART_GetState(&huart2) == HAL_UART_STATE_BUSY_TX ||
89             HAL_UART_GetState(&huart2) == HAL_UART_STATE_BUSY_TX_RX);
90    }
91  }
92
93  int8_t readUserInput(void) {
94    int8_t retVal = -1;
95
96    if(UartReady == SET) {
97      UartReady = RESET;
98      HAL_UART_Receive_IT(&huart2, (uint8_t*)readBuf, 1);
99      retVal = atoi(readBuf);
100   }
```

```
101   return retVal;
102 }
```

As you can see in the above code, the first step is to enable the `USART2_IRQn` and to assign it a priority[15]. Next, we define the corresponding ISR and we add the call to the `HAL_UART_IRQHandler()`. The remaining part of the example file is all about restructuring the `printWelcomeMessage()` and `readUserInput()` functions to deal with asynchronous events.

The function `readUserInput()` now checks for the value of the global variable `UartReady`. If it is equal to `SET`, it means that the user has sent a char to the management console. This character is contained inside the global array `readBuf`. The function then calls the `HAL_UART_Receive_IT()` to receive next character in interrupt mode. When `readUserInput()` returns a value greater than `0`, the function `processUserInput()` is called. Finally, the function `HAL_UART_RxCpltCallback()`, which is automatically called by the HAL when one byte is received, is defined: it simply sets the global `UartReady` variable, which in turn is used by the `readUserInput()` as seen before.



It is important to clarify that the function `HAL_UART_RxCpltCallback()` is called only when all the bytes specified with the `Size` parameter, passed to the `HAL_UART_Receive_IT()` function, are received.

What about the `HAL_UART_Transmit_IT()` function? It works in a way similar to the `HAL_UART_Receive_IT()`: it transfers the next byte in the array every time the *Transmit Data Register Empty*(TXE) interrupt is generated. However, special care must be taken when calling it multiple times. Since the function returns the control to the caller as soon as it finishes to setup the UART, a subsequent call of the same function will fail, and it will return the `HAL_BUSY` value.

Suppose to rearrange the function printWelcomeMessage() from the previous example in the following way:

```
void printWelcomeMessage(void) {
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)"\033[0;0H", strlen("\033[0;0H"));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)"\033[2J", strlen("\033[2J"));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)WELCOME_MSG, strlen(WELCOME_MSG));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)MAIN_MENU, strlen(MAIN_MENU));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)PROMPT, strlen(PROMPT));
}
```

The above code will never work correctly, since each call to one of the functions `HAL_UART_Transmit_IT()` is much faster than the UART transmission, and the next call would fail messing up the UART flow.

If speed is not a strict requirement for your application, and the use of the `HAL_UART_Transmit_IT()` is limited to few parts of your application, the above code could be rearranged as shown in Example

[15]The example is designed for an STM32F4. Please, refer to the book examples for your specific Nucleo.

2 (see above code at lines 83:91). In that implementation, we transfer each string using the `HAL_UART_Transmit_IT()` but, before we transfer the next string, we wait to the transmission completion. However, this is just a variant of the `HAL_UART_Transmit()` in *polling* mode, since we have a busy wait for every UART transfer.

A more elegant and performing solution is to use a temporary memory area where to store the byte sequences and to let the ISR to execute the transfer. A queue is the best options to handle FIFO events. There are several ways to implement a queue, both using static and dynamic data structure. If we decide to implement a queue with a predefined area of memory, a circular buffer is the data structure suitable for this kind of applications.

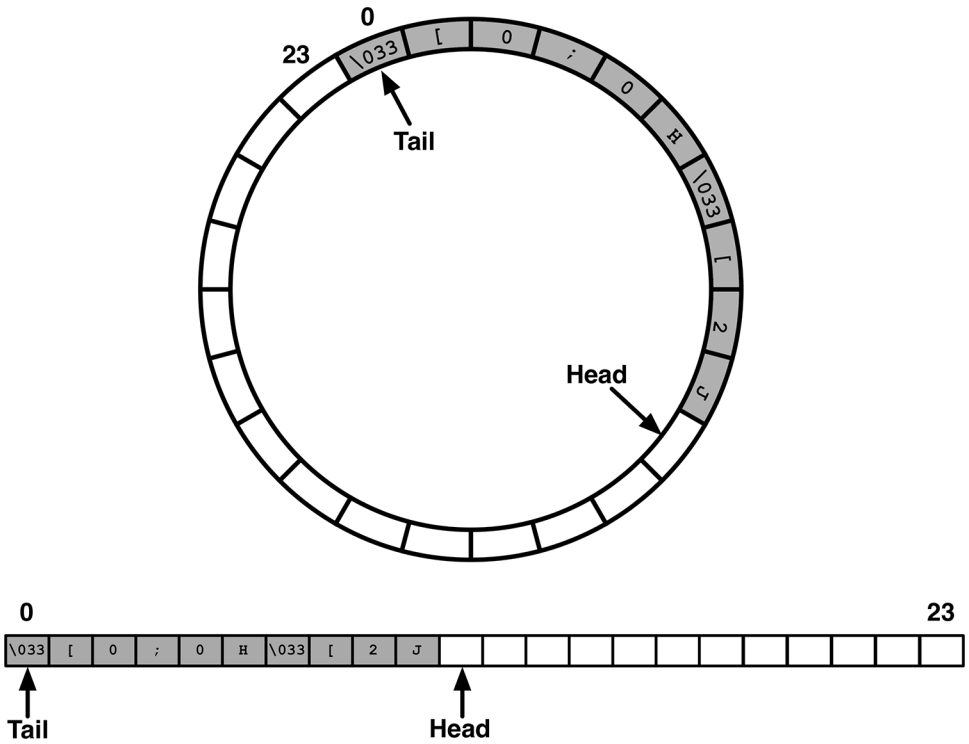


Figure 8.11: A circular buffer implemented using an array and two pointers

A circular buffer is nothing more than an array with a fixed size where two pointers are used to keep track of the *head* and the *tail* of data that still needs to be processed. In a circular buffer, the first and the last position of the array are seen "contiguous" (see **Figure 8.11**). This is the reason why this data structure is called *circular*. Circular buffers have an important feature too: unless our application has up to two concurrent execution streams (in our case, the main flow that places chars inside the buffer and the ISR routine that sends these chars over the UART), they are intrinsically thread safe, since the "consumer" thread (the ISR in our case) will update only the *tail* pointer and the producer (the main flow) will update only the *head* one.

Circular buffers can be implemented in several ways. Some of them are faster, others are safer (that is, they add an extra overhead ensuring that we handle the buffer content correctly). You will find a simple and quite fast implementation in the book examples. Explaining how it is coded is outside

the scope of this book.

Using a circular buffer, we can define a new UART transmit function in the following way:

Filename: src/main-ex3.c

```
77  uint8_t UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t len) {
78    if(HAL_UART_Transmit_IT(huart, pData, len) != HAL_OK) {
79      if(RingBuffer_Write(&txBuf, pData, len) != RING_BUFFER_OK)
80        return 0;
81    }
82    return 1;
83  }
```

The function does just two things: it tries to send the buffer over the UART in interrupt mode; if the `HAL_UART_Transmit_IT()` function fails (which means that the UART is already transmitting another message), then the byte sequence is placed inside a circular buffer.
It is up to the `HAL_UART_TxCpltCallback()` to check for pending bytes inside the circular buffer:

Filename: src/main-ex3.c

```
94  void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
95    if(RingBuffer_GetDataLength(&txBuf) > 0) {
96      RingBuffer_Read(&txBuf, &txData, 1);
97      HAL_UART_Transmit_IT(huart, &txData, 1);
98    }
99  }
```

The `printWelcomeMessage()` and `processUserInput()` functions can be now arranged without performing the *busy-wait*, as shown below:

Filename: src/main-ex3.c

```
105 void printWelcomeMessage(void) {
106   char *strings[] = {"\033[0;0H", "\033[2J", WELCOME_MSG, MAIN_MENU, PROMPT};
107
108   for (uint8_t i = 0; i < 5; i++)
109     UART_Transmit(&huart2, (uint8_t*)strings[i], strlen(strings[i]));
110 }
111
112 uint8_t processUserInput(uint8_t opt) {
113   char msg[30];
114
115   if(!opt || opt > 3)
116     return 0;
117
118   sprintf(msg, "%d", opt);
119   UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg));
```

```
120
121   switch(opt) {
122   case 1:
123     HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
124     break;
125   case 2:
126     sprintf(msg, "\r\nUSER BUTTON status: %s",
127             HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET ? "PRESSED" : "RELEASED");
128     UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg));
129     break;
130   case 3:
131     return 2;
132   };
133
134   UART_Transmit(&huart2, (uint8_t*)PROMPT, strlen(PROMPT));
135   return 1;
136 }
```



The `RingBuffer_Read()` it is not that fast as it could be with a more performant implementation. For some real world situations, the whole overhead of the `HAL_UART_TxCpltCallback()` routine (that is called from the ISR routine) could be too high. If this is your case, you can consider to create a function like the following one:

```
void processPendingTXTransfers(UART_HandleTypeDef *huart) {
   if(RingBuffer_GetDataLength(&txBuf) > 0) {
       RingBuffer_Read(&txBuf, &txData, 1);
       HAL_UART_Transmit_IT(huart, &txData, 1);
    }
}
```

Then, you could call this function from the main application code or in a lower privileged task if you are using an RTOS.

8.5 Error Management

When dealing with external communications, the error management is an aspect that we must strongly take in consideration. An STM32 UART peripheral offers some error flags related to communication errors. Moreover, it is possible to enable a corresponding interrupt to be noticed when the error occurs.

The CubeHAL is designed to automatically detect error conditions, and to warn us about them. We only need to implement the `HAL_UART_ErrorCallback()` function inside our application code. The `HAL_UART_IRQHandler()` will automatically invoke it in case an error occurs. To understand which error has been occurred, we can check the value of the `UART_HandleTypeDef->ErrorCode` field. The list of error codes is reported in **Table 8.7**.

Table 8.7: List of `UART_HandleTypeDef->ErrorCode` possible values

UART Error Code	Description
`HAL_UART_ERROR_NONE`	No error occurred
`HAL_UART_ERROR_PE`	Parity check error
`HAL_UART_ERROR_NE`	Noise error
`HAL_UART_ERROR_FE`	Framing error
`HAL_UART_ERROR_ORE`	Overrun error
`HAL_UART_ERROR_DMA`	DMA Transfer error

The `HAL_UART_IRQHandler()` is designed so that we should not care with the implementation details of UART error management. The HAL code will automatically perform all needed steps to handle the error (like clearing event flags, pending bit and so on), leaving to us the responsibility to handle the error at application level (for example, we may ask to the other peer to resend a corrupted frame).

8.6 List of Available Callbacks in the `HAL_UART` Module

So far, we saw how to exploit the callback mechanism to be notified by the CubeHAL library of the transfer complete event. The CubeHAL is designed with an event-driven approach and almost every `HAL_XXX` module provides a set of callbacks that can be used to catch specific peripheral events.

The **Table 8.8** lists all available callbacks related to the UART module in an STM32G4 library. The first eight callbacks are related to events generated during a transmission. `HAL_UART_MspInitCallback` and `HAL_UART_MspDeInitCallback` are the two callbacks automatically generated by CubeMX inside the **Core/Src/stm32XXxx_hal_msp.c** file. Finally, the last two callbacks are related the UART FIFO, which is a feature available in recent STM32 MCUs like the G4 ones.

Table 8.8: List of callbacks in `HAL_UART` module in the STM32F4 library

`HAL_UART` Callbacks	Description
`HAL_UART_TxHalfCpltCallback`	Tx Half Complete Callback
`HAL_UART_TxCpltCallback`	Tx Complete Callback
`HAL_UART_RxHalfCpltCallback`	Rx Half Complete Callback.
`HAL_UART_RxCpltCallback`	Rx Complete Callback.
`HAL_UART_ErrorCallback`	Error Callback.
`HAL_UART_AbortCpltCallback`	Abort Complete Callback.
`HAL_UART_AbortTransmitCpltCallback`	Abort Transmit Complete Callback.
`HAL_UART_AbortReceiveCpltCallback`	Abort Receive Complete Callback.
`HAL_UART_MspInitCallback`	UART MspInit.
`HAL_UART_MspDeInitCallback`	UART MspDeInit.
`HAL_UARTEx_WakeupCallback`	Wakeup from Stop Mode Callback.
`HAL_UARTEx_RxFifoFullCallback`	Rx Fifo Full Callback.
`HAL_UARTEx_TxFifoEmptyCallback`	Tx Fifo Empty Callback.



Differences Between *HAL_PPP* and *HAL_PPPEx* Modules

We have encountered several HAL modules until here, each one covering one specific peripheral or core feature. Every HAL module is contained in a file named **stm32XXxx_hal_ppp.{c,h}**, where the "XX" represents the STM32 family, and "ppp" the peripheral type. For example, the **stm32f4xx_hal_uart.c** file contains all the function definitions for the `HAL_DMA` module, all those functions have an API common to all STM32 families. This enforces the portability of code in STM32 lineup.

However, some peripheral functions are specific of a given family, and cannot be abstracted in a general way common to all STM32 portfolio. In this cases, the HAL provide an extension module named `HAL_PPPEX` and implemented in a file named **stm32XXxx_hal_ppp_ex.{c,h}**. For example, the previous `HAL_UARTEx_RxFifoFullCallback()` function is defined in the `HAL_UARTEx` module, implemented in **stm32f4xx_hal_uart_ex.c** file.

The implementation of the APIs in an extension module is specific of the corresponding STM32 series, or even of a given part number in that series, and the usage of those APIs leads to a less portable code.

Recent CubeHAL provides two ways to define callbacks. The standard one, is by defining the callback function in the application source files and let the compiler to overwrite the placeholder functions defined with the modifier `__weak`. The second method consists in using a dedicated set of APIs that allow to define at run-time the callback routines. For the majority of HAL modules, it is possible to set the macro `USE_HAL_PPP_REGISTER_CALLBACKS` to 1 (where `PPP` is the corresponding peripheral name) inside the `Core/Inc/stm32f4xx_hal_conf.h`. This will enable the functions:

```
HAL_StatusTypeDef HAL_UART_RegisterCallback(UART_HandleTypeDef *huart,
                        HAL_UART_CallbackIDTypeDef CallbackID, pUART_CallbackTypeDef pCallback);
```

and

```
HAL_StatusTypeDef HAL_UART_UnRegisterCallback(UART_HandleTypeDef *huart,
                        HAL_UART_CallbackIDTypeDef CallbackID, pUART_CallbackTypeDef pCallback);
```

These functions accept the `CallbackID` corresponding to the callbacks in **Table 8.8** and the pointer to the callback function.
The ability to configure callbacks at run-time gives to the programmer the possibility to change the callback behavior at run-time, at the cost of the increase of FW size.

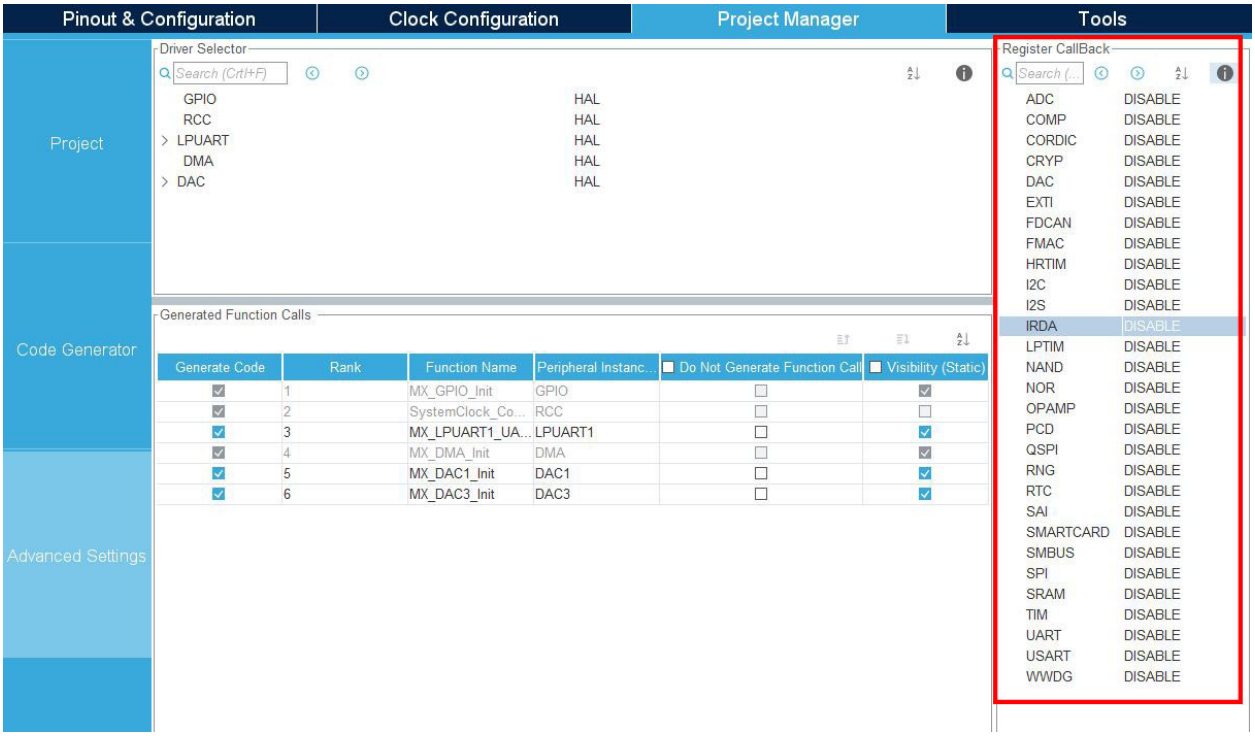


Figure 8.12: How-to enable register callbacks mechanism in CubeMX

CubeMX provides a convenient way to enable run-time callback mechanism. In the **Project Manager** pane, click on the **Advanced Settings** section. The **Register Callback** pane appears on the right side, as shown in **Figure 8.12**. For a given peripheral, you can enable run-time callbacks by choosing **ENABLE** in the corresponding field.

9. DMA Management

Every embedded application needs to exchange data with the outside world or to drive external peripherals. For example, our microcontroller may exchange messages with other modules on the PCB using an UART, or it may store data in an external flash using one of the available SPI interfaces. This involves the transfer of a given amount of data between the internal SRAM or flash memory and the peripheral registers, and it requires a certain number of CPU cycles to accomplish the transfer. This leads to a loss of computing power (the CPU is occupied in the transfer process), to a reduction of the overall performances and eventually to a loss of important asynchronous events.

The *Direct Memory Access* (DMA) controller is a dedicated and programmable hardware unit that allows MCU peripherals to access to internal memories without the intervention of the Cortex-M core. The CPU is completely freed from the overhead generated by the data transfer (except for the overhead related to the DMA configuration), and it can perform other activities in parallel[1]. The DMA is designed to work in both the ways (that is, it allows data transfer from memory to peripheral and *vice versa*), and all STM32 microcontrollers provide at least one DMA controller, but the most of them implement two independent DMAs.

The DMA is an "advanced" feature of modern MCUs, and novice users tend to consider it too complicated to use. Instead, the concepts underlying the DMA are basically simple, and once you understand them it will be easy to use it. Moreover, the good news is that the CubeHAL is designed to abstract the most of DMA configuration steps for a given peripheral, leaving to the user the responsibility to provide just few basic configurations.

This chapter will guide you through the basic concepts related to the DMA usage, and it will offer an overview of the DMA characteristics in all STM32 families. As usual, this chapter does not aim to be exhaustive and to substitute the official ST documentation[2], which is a good thing to have as reference during the reading of this chapter. However, once you master the fundamental concepts related to the DMA, you will be able to dive inside your MCU datasheets easily.

9.1 Introduction to DMA

Before we can analyze the features offered by the `HAL_DMA` module, it is important to understand some fundamental concepts behind the DMA controller. The next paragraphs try to summarize the most important aspects to keep in mind during the study of this peripheral. The DMA implementation in STM32 MCUs evolved over the years with the advent of more powerful and modern families. The very first and more cheap STM32 families provide the simplest DMA architecture, with a quite

[1] This is not exactly true, as we will see next. But it is ok to consider that sentence true here.

[2] ST provides a dedicated application note about the DMA for every STM32 family. For example, the AN2548 (https://bit.ly/3kHEfFH) talks about the DMA in STM32F0/F1/F3/Gx/Lx MCUs, while the AN4031 (https://bit.ly/3kImCpn) is all about the DMA in STM32F2/F4/F7 families.

reduced possibility to mix DMA sources.
With the advent of more powerful F2/F4/F7 families, ST introduced a more flexible DMA architecture, increasing the number of communication channels while boosting the overall performances thanks to dedicated "bus bridges" that directly connects the DMA to the some peripheral avoiding the "traffic" on the *Bus Matrix*.
Finally, in recent STM32L4+/L5/Gx/H7 families, the freedom of mixing sources and destinations reached a new dimensions: thanks to the DMAMUX multiplexer, the interconnection between source and destination peripherals is no longer fixed by design, and there is the possibility to interconnect peripherals in a chain where data is exchanged just through the DMA without no intervention of the CPU.

If you start dealing with these three STM32 DMA architectures without adequate preparation, chances are that you will be totally confused especially by the ST terminology. For reasons not clear to this author, ST decided to change the vocabulary between the three different architectures, messing up with terms and definitions. Unfortunately, this terminology change reflects also in the CubeHAL, giving to programmer the idea that the DMA architectures differ a lot between each other. Instead, apart from an increase degree of flexibility, the concepts (and the technicalities) behind the three architectures are almost the same.

Next paragraphs will try to minimize the level of details, while focusing on things that matter. It is strongly suggested to keep on hands the reference manual for the STM32 MCU you are using. But trust me: if you are new to this matter, try to understand the overall picture before diving into the very specific details.

9.1.1 The Need of a DMA and the Role of the Internal Buses

Why the DMA is a so important feature? Every peripheral in an STM32 microcontroller needs to exchange data with the internal Cortex-M core. Some of them translate this data in electrical I/O signals to exchange it to the outside world according to a given communication protocol (this is the case, for example, of UART or SPI interfaces). Others are just designed so that the access to their registers inside the peripheral memory mapped region (from `0x4000 0000` to `0x5FFF FFFF`) causes a changing to their state (for example, the `GPIOx->ODR` register drives the state of all I/Os connected to that port). However, keep in mind that from the CPU point of view this also implies a memory transfer between the MCU core and the peripheral.

The MCU - in theory - could be designed so that every peripheral would have its own storage area (dedicated memories), and it in turn could be tightly coupled with the MCU core to minimize the costs related to memory transfers[3]. This, however, complicates the MCU architecture, requiring a lot of more silicon and more "active components" that consume power. So, the approach used in all embedded microcontrollers is to use some portions of the internal SRAM memory as temporary area storage for different peripherals. It is up to the user to decide how much room to dedicate to these areas. For example, let us consider this code fragment:

[3]This is what happens in some vector processors equipping really expensive supercomputers, but this is not the case of 32 cents CPUs like the STM32.

```
uint8_t buf[20];
...
HAL_UART_Receive(&huart2, buf, 20, HAL_MAX_DELAY);
```

Here we are going to read twenty bytes from the UART2 interface, and hence we allocate an array (the temporary storage) of the same size inside the SRAM. The `HAL_UART_Receive()` function will access twenty times to the `huart2.Instance->DR` data register to transfer bytes from the peripheral to the internal memory, plus it will poll the UART `RXNE` flag to detect when the new data is ready to be transferred. The CPU will be involved during these operations (see **Figure 9.1**), even if its role is "limited" to move data from the peripheral to the SRAM[4].

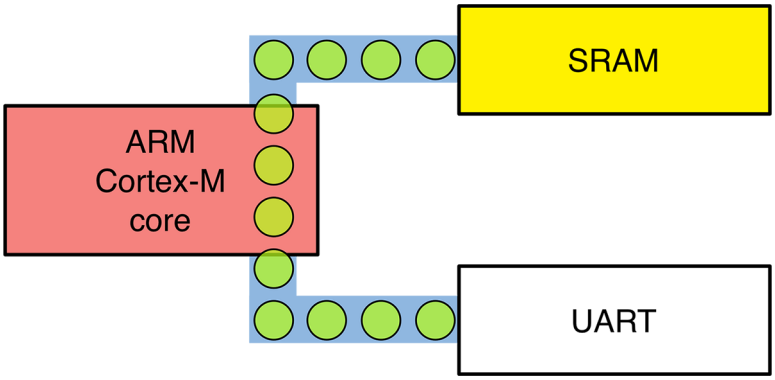


Figure 9.1: the flow of data during a transfer from peripheral to SRAM

While this approach simplifies the design of the hardware on the one hand, it introduces performance penalties on the other. The Cortex-M core is "responsible" to load data from peripheral memory to the SRAM, and this is a blocking operation, which not only prevents the CPU from doing other activities but it also requires the CPU to wait for "slower" units completing their job (some STM32 peripherals are connected to the MCU core by slower buses, as we will see in Chapter 10). This is the reason why high-performance microcontrollers provide hardware units dedicated to the transfer of data between peripherals and centralized buffer storage, that is the SRAM.

Before we go more in depth inside the DMA details, it is better to take an overview of all components involved in the transfer process of data from a peripheral to the SRAM memory and *vice versa*. We have already seen in Chapter 6 the bus architecture of the STM32F072 MCU, one of the simplest STM32 microcontrollers. The bus architecture is shown again in **Figure 9.2** for convenience. It differs a lot from other more performant STM32 families. We will analyze them later in this chapter, since it is best to keep it simple in this phase.

The figure says to us some important things:

- Both the *Cortex-M core* and the *DMA1 controller* interact with the other MCU peripherals through a series of buses. If it is still unclear, it is important to remark that also the FLASH

[4]Keep in mind that using the UART in interrupt mode does not change the story. Once the UART generates the interrupt to signal the MCU core that new data is arriving, it is always up to the CPU to "move" this data byte-by-byte from UART data register to the SRAM. That's the reason why from the performance point of view there is no difference between UART management in polling and interrupt mode.

and SRAM memories are components **outside** the MCU core, and so they need to interact each other through a bus interconnection.

- Both the *Cortex-M core* and the *DMA1 controller* are **masters**. This means they are the only units that can start a transaction on a bus. However, the access to the bus must be regulated so that they cannot access to the same **slave** peripheral at the same time.
- The *BusMatrix* manages the access arbitration between the Cortex-M core the DMA1 controller. The arbitration uses a *Round Robin* algorithm to rule the access to the bus. The BusMatrix is composed of two masters (CPU, DMA) and four slaves (FLASH interface, SRAM, AHB1 with AHB to *Advanced Peripheral Bus* (APB) bridge and AHB2). The BusMatrix also allows to automatically interconnect several peripherals between them.
- The *System bus* connects the Cortex-M core to the BusMatrix.
- The *DMA bus* connects the *Advanced High-performance Bus* (AHB) master interface of the DMA to the BusMatrix.
- The *AHB to APB bridge* provides full synchronous connections between the AHB and the APB bus, where the most of peripherals are connected.

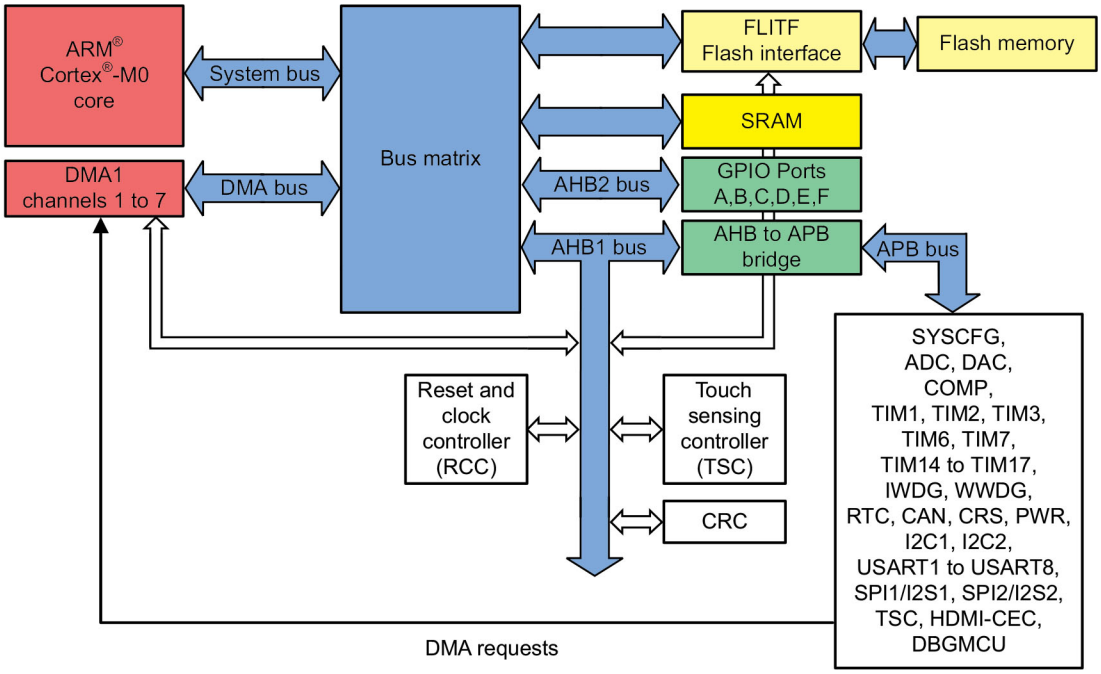


Figure 9.2: bus architecture of an STM32F072 microcontroller

We left off one other thing in **Figure 9.2**: the *DMA requests* arrow that goes from the peripherals block (white rectangle) to the DMA1 controller. What exactly is it for? In Chapter 7 we have seen that the NVIC controller notifies the Cortex-M core about asynchronous interrupt requests (IRQs) coming from peripherals. When a peripheral is ready to do something (e.g., the UART is ready to receive data or a timer overflows), it asserts a dedicated IRQ line. The MCU core executes after a given number of cycles the corresponding ISR, which contains the code necessary to handle the IRQ. Don't forget that the peripherals are **slave units**: they cannot access the bus independently. A

master is always needed to start a transaction. But, since peripherals are slave units[5], if we use the DMA to transfer data from peripherals to memory, we have a way to notify it that the peripherals are ready to exchange data. That is the reason why a dedicated number of *DMA requests lines* are available from peripherals to the DMA controller. We will see in the next paragraph how they are organized and how we can program them.

9.1.2 The DMA Controller

In every STM32 MCU, the DMA controller is a hardware unit that:

- has *two master ports*, named *peripheral* and *memory* port respectively, connected to the *Advanced High-performance Bus* (AHB), one able to interface a slave peripheral and the other one a memory controller (SRAM, flash, FSMC, etc.); in some DMA controllers a peripheral port is also able to interface a memory controller, allowing *memory-to-memory* transfers; in the majority of STM32 MCUs, a memory port is also able to interface a peripheral controller, allowing *peripheral-to-peripheral* transfers;
- has *one slave port*, connected to the AHB bus, used to program the DMA controller itself from the other master, that is the CPU;
- has a number of independent and programmable *channels*, each one connected by design or connectable to a given peripheral request line (`UART_TX`, `TIM_UP` etc.);
- defines different *priorities* to channels (on a programmable basis or by design), in order to arbitrate the access to the memory giving higher priority to faster and important peripherals;
- allows the data to flow in *both directions*, that is from *memory-to-peripheral* and from *peripheral-to-memory*:

Each STM32 MCU provides a variable number of DMAs and Channels according to its family and sales type. The **Table 9.1** reports their exact number for the STM32 MCUs equipping all Nucleo boards used in this book.

These characteristics are broadly common to all STM32 microcontrollers. However, as said at the beginning of this section, the DMA controller architectures changes between the very first STM32 families and the latest ones. This is the reason why we are going to treat them separately[6].

[5] Apart for some high-performance peripherals, like USB and Ethernet that needs to access to some memory buffers independently to avoid the loss of important data flowing to the communication medium.

[6] However, keep in mind that this book does not aim to be an exhaustive source of hardware details of each STM32 family. Always keep in your hands the reference manual for the MCU you are considering and look carefully to the chapter related to the DMA.

Nucleo P/N	Available DMAs	# Channels (DMA1 - DMA2)	MEM2MEM DMA
NUCLEO-F446RE	2	8 - 8	DMA2
NUCLEO-F401RE	2	8 - 8	DMA2
NUCLEO-G474RE	2	8 - 8	DMA1 + DMA2
NUCLEO-F303RE	2	7 - 5	DMA1 + DMA2
NUCLEO-F103RB	2	7 - 5	DMA1 + DMA2
NUCLEO-F072RB	2	5 - 7	DMA1 + DMA2
NUCLEO-L476RG	2	7 - 7	DMA1 + DMA2
NUCLEO-L152RE	2	7 - 5	DMA1 + DMA2
NUCLEO-L073RZ	1	7	DMA1

Table 9.1: The number of DMAs/Channels available in Nucleo boards used in this text

9.1.2.1 The DMA Implementation in F0/F1/F3/L0/L1/L4 MCUs

The **Figure 9.3** shows a representation of the DMA1 controller in F0/F1/F3/L0/L1/L4 MCUs. Some MCUs from these families provide a second DMA controller, DMA2. The DMA1 controller provides seven configurable *channels*, while DMA2 just five.
A channel is used to exchange data between two memory regions in the 4GB address space with a given degree of freedom. Each channel is bound to a given *request line*, which triggers the transfer between the two memory regions. Each request line has a variable number of peripheral request sources connected to it: a multiplexer is used to bound a given peripheral request source to a channel request line. However, a channel is bound during the chip design to a fixed set of peripherals and only one peripheral at once can be active in the same channel. For example, **Table 9.2**[7] shows how channels are bound to peripherals in an STM32F030 MCU. Every request line can be also triggered by "software". This ability is used to perform *memory-to-memory* transfers.

[7]The table is extracted from the ST RM0360 reference manual (https://bit.ly/1GfS3iC)

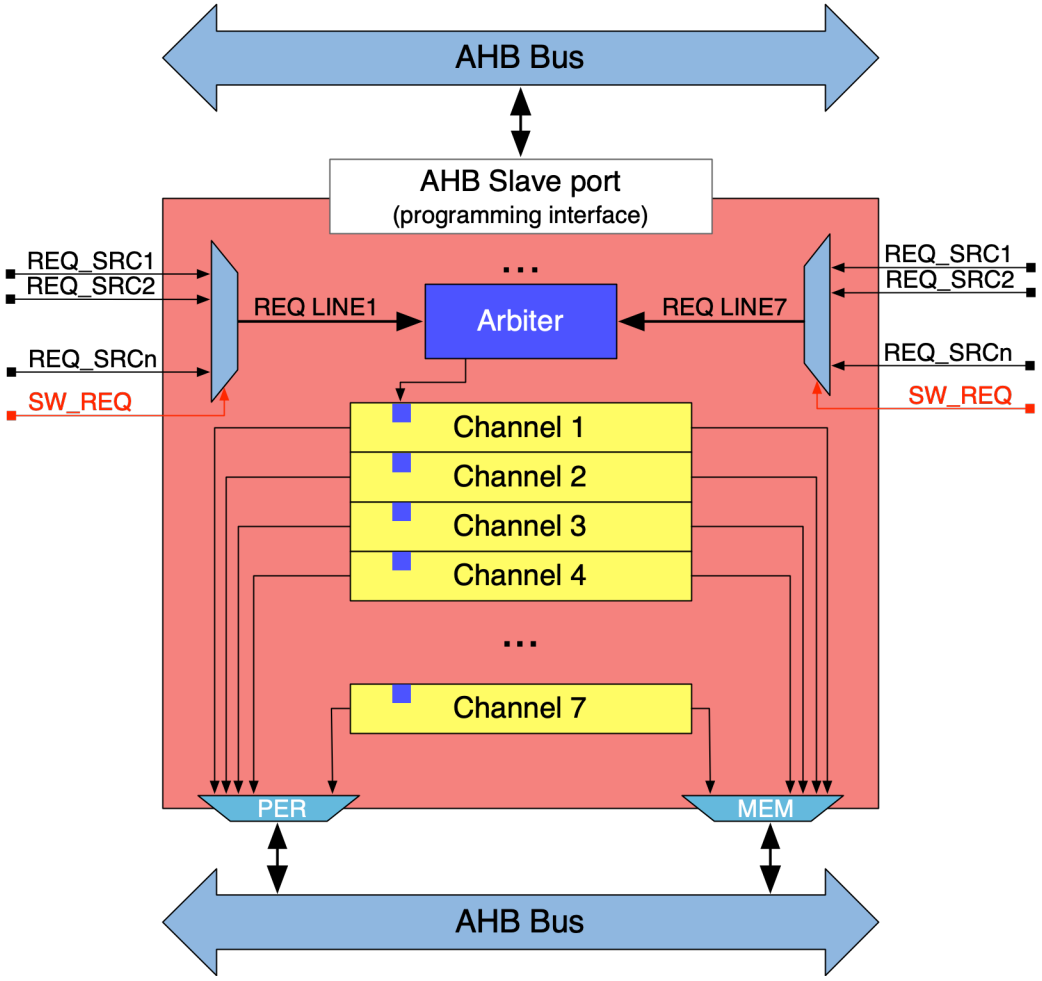


Figure 9.3: A representation of the DMA structure in F0/F1/F3/L0/L1/L4 MCUs

Each channel has a priority that allows to rule the access to the AHB bus. In oldest MCUs like STM32F1 ones, the priority is fixed: Channel 1 has the maximum priority, Channel 7 the lowest. In recent MCUs the priority can be configured using a four-level scale.
An internal arbiter rules the requests coming from the channels according to channel's priority. If two request lines activate a request and their channels have the same priority, the channel with the lower number wins the contention.

We have already seen in **Figure 9.2** the bus architecture of an STM32F072. For the sake of completeness, **Figure 9.4**[8] shows the bus architecture of more performant MCUs with the same DMA implementation (e.g., the STM32F1). As you can see, the two families have a quite different internal bus organization. You can see two additional buses named *ICode* and *DCode*. Why this difference?

[8]The figure is taken from the RM0008 reference manual from ST (https://bit.ly/1TNekGo)

Peripherals	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5
ADC	ADC	ADC	-	-	-
SPI	-	SPI1_RX	SPI1_TX	SPI2_RX	SPI2_TX
USART	-	USART1_TX USART3_TX	USART1_RX USART3_RX	USART1_TX USART2_TX	USART1_RX USART2_RX
I2C	-	I2C1_TX	I2C1_RX	I2C2_TX	I2C2_RX
TIM1	-	TIM1_CH1	TIM1_CH2	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_CH3 TIM1_UP
TIM3	-	TIM3_CH3	TIM3_CH4 TIM3_UP	TIM3_CH1 TIM3_TRIG	-
TIM6	-	-	TIM6_UP	-	-
TIM7	-	-	-	TIM7_UP	-
TIM15	-	-	-	-	TIM15_CH1 TIM15_UP TIM15_TRIG TIM15_COM
TIM16	-	-	TIM16_CH1 TIM16_UP	TIM16_CH1 TIM16_UP	-
TIM17	TIM17_CH1 TIM17_UP	TIM17_CH1 TIM17_UP	-	-	-

Table 9.2: How channels are bound to peripheral in an STM32F030 MCU

The most of STM32 MCUs share the same *computer architecture* except for STM32F0/G0/L0 that are based on the Cortex-M0/0+ cores. They, in fact, are the only Cortex-M cores based on the *von Neumann architecture*, compared to the other Cortex-M cores that are based on the *Harvard architecture*[9]. The fundamental distinction between the two architectures is that Cortex-M0/0+ cores access to flash memory, SRAM and peripherals using one common bus, while the other Cortex-M cores have two separated bus lines for the access to the flash (one for the fetch of instructions called *instruction bus*, or simply *I-Bus* or even *I-Code*, and one for the access to const data called *data bus*, or simply *D-Bus* or even *D-Code*) and one dedicated line for the access to SRAM and peripherals (also called *system bus*, or simply *S-Bus*). What advantages gives this to our applications?

[9]For the sake of completeness, we have to say that they are based on a *modified Harvard architecture* (https://en.wikipedia.org/wiki/Modified_Harvard_architecture), but let us leave the distinction to historians of computer science.

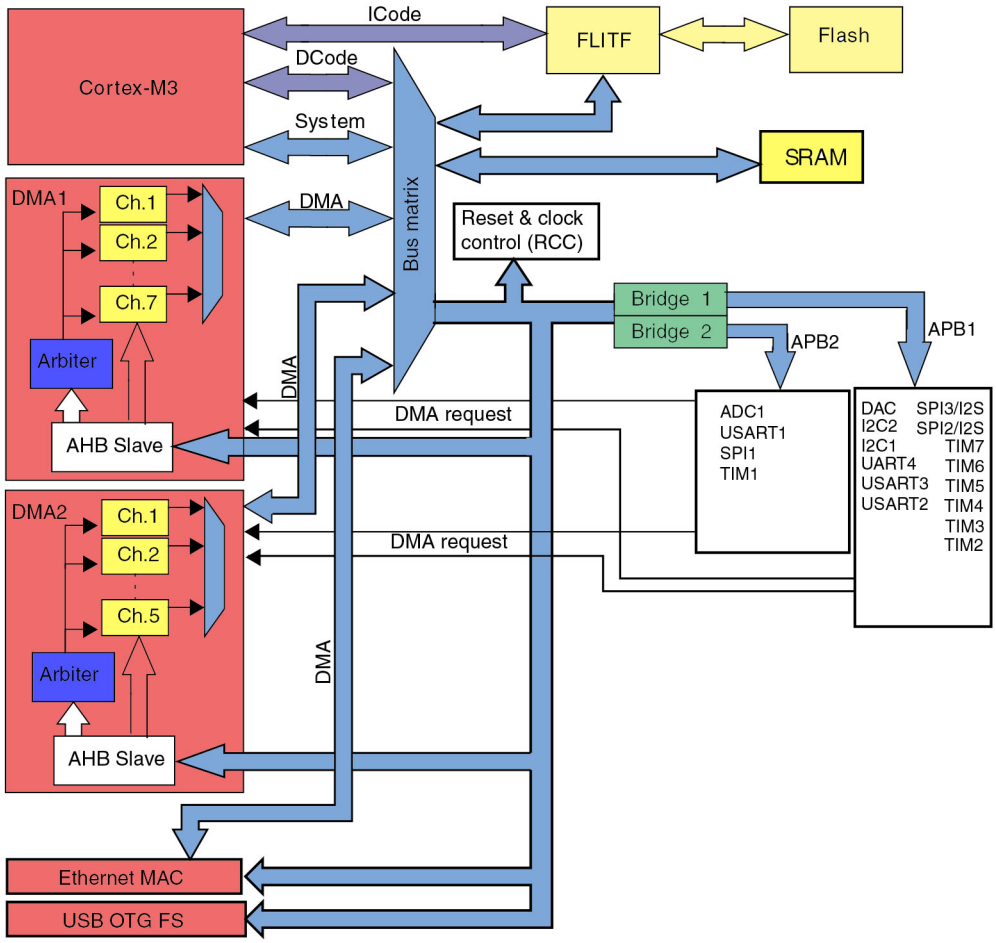


Figure 9.4: The bus architecture in an STM32F1 MCU from the *Connectivity Line*

In Cortex-M0/0+ cores the DMA and the Cortex core contend the access to memories and peripherals using the BusMatrix. Suppose that the CPU is performing math operations on data contained in its internal registers (R0-R14). If the DMA is transferring data to the SRAM, the BusMatrix arbitrates the access from the Cortex core to the flash memory to load the next instruction to execute. So, the MCU core is stalled waiting for its turn (more about this in a while). In the other Cortex-M cores, the CPU can access to the flash memory independently, boosting the overall performances. This is a fundamental difference that justifies the price of STM32F0 MCUs: they not only can have less SRAM and flash and run at lower frequencies, but they face a simpler and intrinsically less performant architecture.

However, it is important to remark that the BusMatrix implements scheduling policies to avoid that a given master (CPU and DMA in *Value Lines* MCUs, or CPU, DMA, Ethernet and USB in *Connectivity Lines* MCUs) stalls for too much time. Each DMA transfer is made up of four phases: a sample and arbitration phase, an address computation phase, bus access phase and a final acknowledgement phase (which is used to signal that the transfer has been completed). Each phase takes a single cycle, except for the bus access phase, which can last for a higher number of cycles. However, its maximum duration is fixed, and the BusMatrix guarantees that at the end of the acknowledgement

phase another master will be scheduled for the access to the bus. As we will see in the next paragraph, the STM32F2/F4/F7 families allow a more advanced parallelism while accessing to slave devices. The details of these aspects, however, are outside of the scope of this book. It is strongly suggested to have a look at the AN4031 (https://bit.ly/1n66sW7[10]) from ST to better understand them.

Finally, the DMA can also perform *peripheral-to-peripheral* transfers under particular conditions, as we will see next.

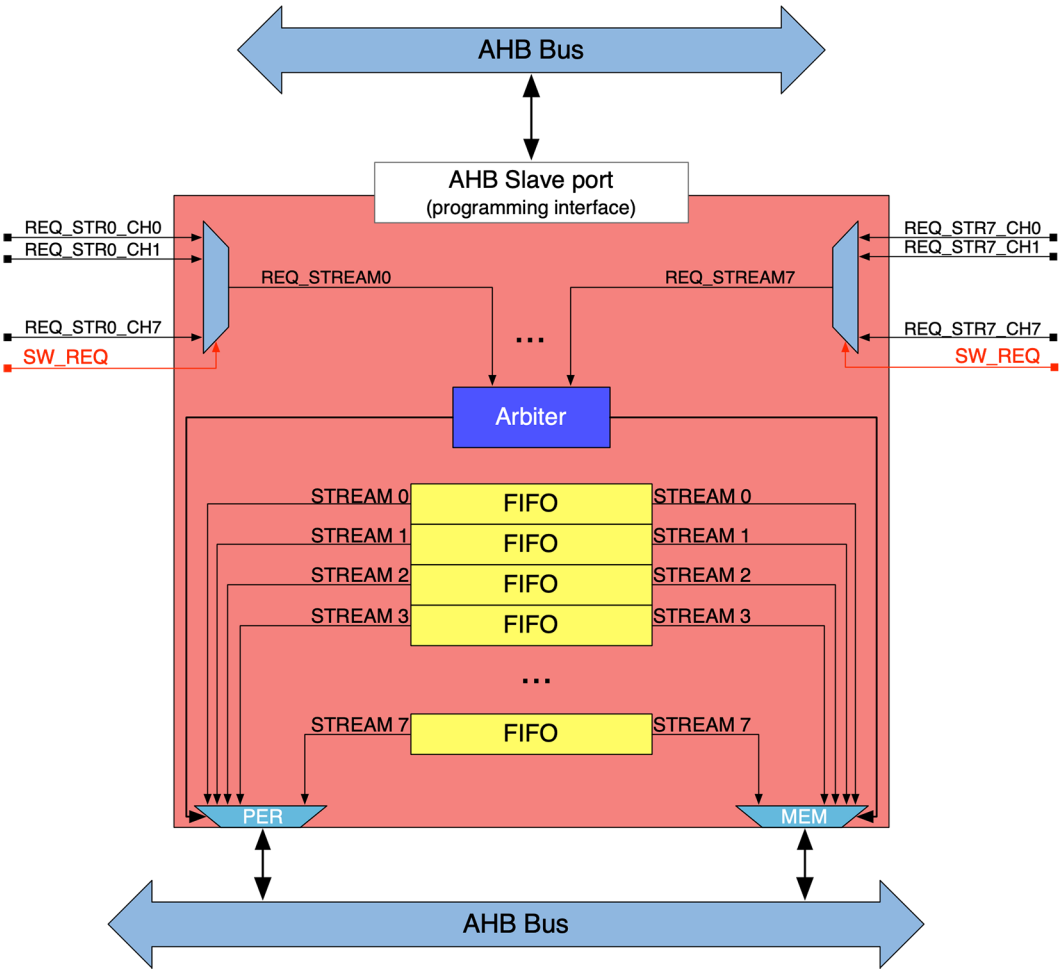


Figure 9.5: The DMA architecture in an STM32F2/F4/F7 MCU

9.1.2.2 The DMA Implementation in F2/F4/F7 MCUs

STMF2/F4/F7 MCUs implement a more advanced DMA controller, as shown in **Figure 9.5**. It offers a higher degree of flexibility compared to the DMA found in older STM32 MCUs. However, for reasons not clear to this author, ST guys decided to mess with the terminology related to DMA in these families, changing again mind with G0/G4/L4+/L5/H7 families. So, especially if you read the previous paragraph, it is important to clarify since the beginning that:

[10] https://bit.ly/1n66sW7

- with the term *streams* we are referring to the communication *channels* between a memory address and a peripheral address; so, *streams* is a synonymous for *channels*;
- with the term *channels* we are referring to *request lines* (also called *request streams* in ST official documentation).

In F2/F4/F7 families, every DMA implements 8 different *streams*. Each stream is dedicated to managing memory access requests from one or more peripherals. Each stream can have up to 8 *channels* (requests lines) in total (but keep in mind that only one channel/request can be active at the same time in a stream), and it has an arbiter for handling the priority between DMA requests, which is user configurable on a four-level basis. Every stream can be also triggered by "software". This ability is used to perform *memory-to-memory* transfers, but it is limited only to DMA2 as highlighted in **Table 9.1**.

A stream can optionally be configured to enable a four-word depth 32-bit First-In/First-Out (FIFO) memory buffer. The FIFO is used to temporarily store data coming from the source before transmitting it to the destination, especially the speed of the two source is different. When the data frame size of the two endpoints differs, the FIFO buffer can automatically perform data conversion while performing transfers. The supported operations are:

- 8-bit / 16-bit ⟶ 32-bit / 16-bit (data *packing*)
- 32-bit / 16-bit ⟶ 8-bit / 16-bit (data *unpacking*)

Every STM32F2/F4/F7 MCU provides two DMA controllers, for a total of 16 independent streams. Like in the other STM32 microcontrollers, a channel is bound to a fixed set of peripherals during the chip design. **Table 9.3** shows the DMA1 stream/channel request mapping in an STM32F401RE MCU. STM32F2/F4/F7 MCUs embed a multi-masters/multi-slaves architecture made of:

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX	I2C3_RX				I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4						USART2_RX	USART2_TX	
Channel 5			TIM3_CH4 TIM3_UP		TIM3_CH1 TIM3_TRIG	TIM3_CH2		TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	I2C3_TX	TIM5_UP	
Channel 7			I2C2_RX	I2C2_RX				I2C2_TX

Table 9.3: The DMA1 stream/channel request mapping in an STM32F401RE MCU

- Eight masters:
  - Cortex core I-bus
  - Cortex core D-bus
  - Cortex core S-bus
  - DMA1 memory bus
  - DMA2 memory bus
  - DMA2 peripheral bus
  - Ethernet DMA bus (if available)
  - USB high-speed DMA bus (if available)
- Eight slaves:
  - Internal flash memory I-Code bus
  - Internal flash memory D-Code bus
  - Main internal SRAM1
  - Auxiliary internal SRAM2 (if available)
  - Auxiliary internal SRAM3 (if available)
  - AHB1 peripherals including AHB-to-APB bridges and APB peripherals
  - AHB2 peripherals
  - AHB3 peripheral (FMC) (if available)

Masters and slaves are connected via a multi-layer BusMatrix ensuring concurrent access from separated masters and efficient operations, even when several high-speed peripherals work simultaneously. Moreover, a dedicated AHB-to-APB bridges allows direct access from masters (and so, DMAs too) to some peripheral, to avoid passing for the BusMatrix. This architecture is shown in **Figure 9.6**[11] for the case of STM32F405/415 and STM32F407/417 lines.

The multi-layer Bus Matrix allows different masters to perform data transfers concurrently as long as they are addressing different slave modules (but for a given DMA, only "one stream" at time can access to the bus). On top of the Cortex-M Harvard architecture and dual AHB port DMAs, this structure enhances data transfer parallelism, thus contributing to reduce the execution time, and optimizing the DMA efficiency and power consumption.

[11]The figure is taken from the AN4031 application note from ST (http://bit.ly/1n66sW7)

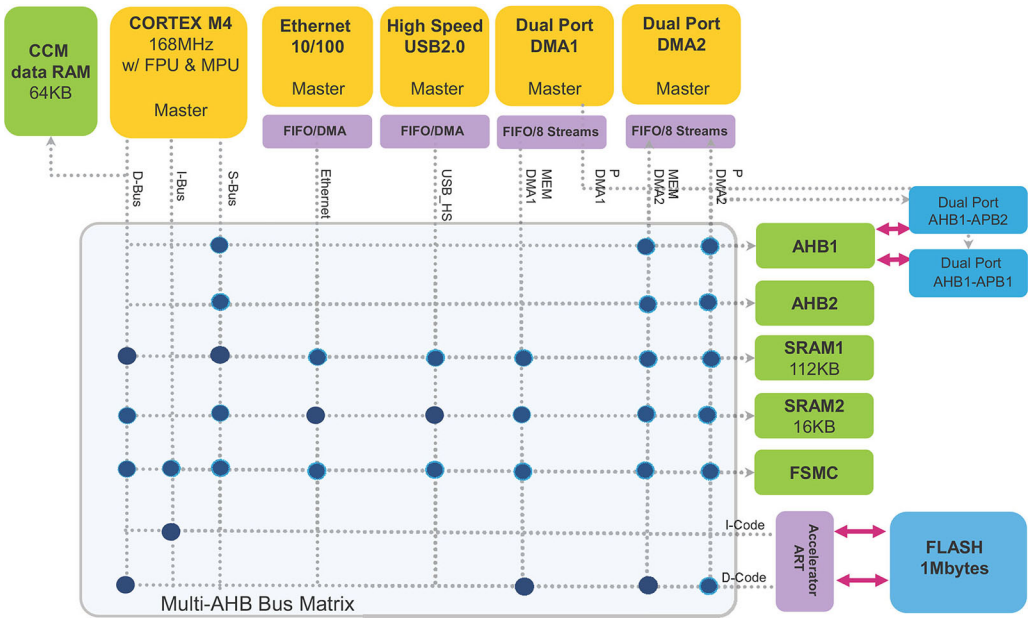


Figure 9.6: The multi-layer BusMatrix in an STM32F405 MCU

9.1.2.3 The DMA Implementation in G0/G4/L4+/L5/H7 MCUs

The DMA controller architecture in STM32G0/G4/L4+/L5/H7 MCUs is similar to the one in STM32F0/F1/F3/L0/L1/L4 families, but it is enhanced by a *DMA request Multiplexer* (DMAMUX) unit. DMAMUX adds a fully configurable routing of any DMA request from a given peripheral in DMA mode to any DMA channel of the two DMA controllers. This means that the *peripheral requests* bound to a given channel are not defined by design, giving to both programmers and hardware developers the maximum flexibility during design phase. DMAMUX does not add any clock cycle between the DMA request sent by the peripheral and the DMA request received by the configured DMA channel. It features synchronization of DMA requests using dedicated inputs. DMAMUX is also capable of generating requests from own trigger inputs or by software.

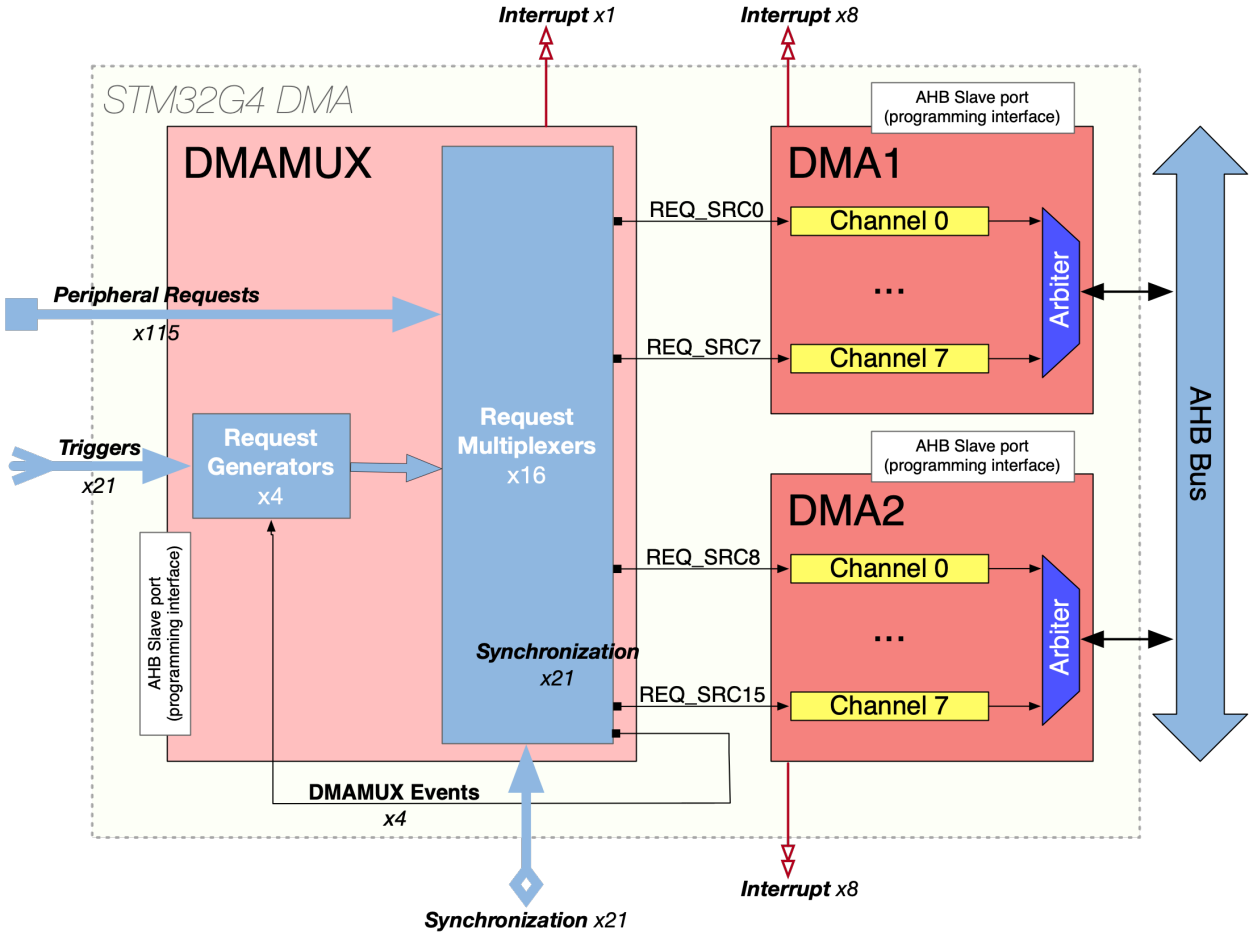


Figure 9.7: The overall DMA architecture in STM32G0/G4/L4+/L5/H7 families

Figure 9.7 shows the overall DMA architecture. It is a simplified diagram that we are going to better specify later. On the right side of the diagram, you can see the two DMAs (DMA1, DMA2). The architecture of the two DMA units[12] is the same of the one shown in **Figure 9.3**, so we will not detail it here. The main difference between F0/F1/F3/L0/L1/L4 families is that here the DMA request sources are not directly wired to the individual peripherals but come from the DMAMUX module. As suggested by the name, the DMAMUX is a module acting as a *multiplexer*. The output from DMAMUX are the request lines, which will fire the data transfer between the source and the destination addresses. The input to the DMAMUX is made of both the actual peripheral request lines and a set of requests not originating from the peripherals, plus a set of lines used to synchronize the data transfer, as we will see later.

[12]The picture shows just the architecture of STM32G4 DMA. However, the difference among the families is just related to the number of peripheral requests, triggers and synchronization lines.

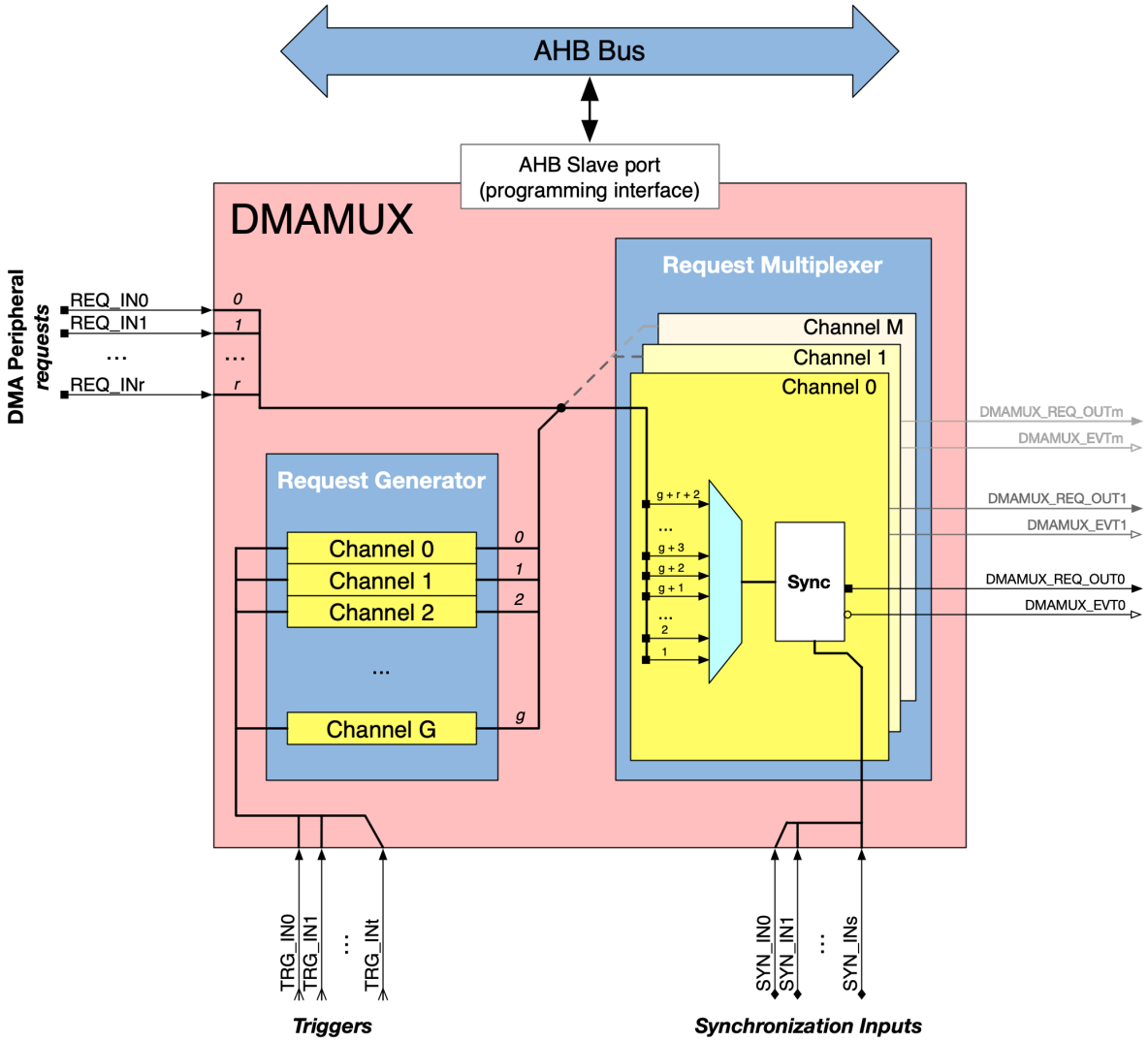


Figure 9.8: The DMAMUX module architecture

The DMAMUX architecture is a little more articulated and it is better detailed in **Figure 9.8**. As you can see, the **Request Multiplexer** module has dedicated sub-units for every channel in the two DMAs. Depending on the specific family, each DMA unit provides 7 or 8 channels. This means that the **Request Multiplexer** integrates 14 or 16 units dedicated to the individual channel.
Every channel unit has a set of request lines. These come both from the peripheral and from the **Request Generator** module we are going to detail in a while. Every channel can be bound to every peripheral request line, with the sole limitation that one request line can be bound to just one channel.

The **Request Generator** unit can be considered as an intermediary between a peripheral and the DMA controllers. It allows peripherals without DMA capability (such as RTC alarm or comparators) to generate a programmable number of DMA requests on an event. The trigger signal (mostly coming from EXTI_LINES and the LPTIM timer), the trigger polarity and the number of requests

define the behavior of the generator channel. Upon the trigger event reception, the corresponding generator channel starts generating DMA requests on its output. Each time the DMAMUX generated request is served by the connected DMA controller, a built-in DMA request counter (one counter per request generator channel) is decremented. When it underruns, the request generator channel stops generating DMA requests and the DMA request counter is automatically reloaded to its programmed value upon the next trigger event.

In order to perform peripheral-to-memory or memory-to-peripheral transfers, the DMA controller channel requires each time a peripheral DMA request line. Each time a request occurs, the DMA channel transfers data from/to peripheral. This mode is called *unconditional request forwarding*.
In addition to unconditional request forwarding, the synchronization unit in every channel sub-module allows the software to implement *conditional request forwarding*: the routing is effectively done only when a defined condition is detected. The DMA transfers can be synchronized with internal or external signals. For example, the user software can use the synchronization unit to initiate or adjust data transmission throughput. DMA request can be forwarded in one of the following ways:

- every time an edge is detected on a GPIO pin (EXTI);
- in response to a periodic event from a timer;
- in response to an asynchronous event from a peripheral;
- in response to an event from another request router (request chaining);

On top of DMA request conditioning, the synchronization unit allows the generation of events (`DMAMUX_EVTm` lines in **Figure 9.8**) that may be used by other DMAMUX sub-blocks (such as the request generator or another DMAMUX request multiplexer channel).

9.2 `HAL_DMA` Module

After a lot of talking, it is now the time to start writing code.

Strictly speaking, programming the DMA is fairly simple, especially if it is clear how the DMA works from a theoretical point of view. Moreover, the CubeHAL is designed to abstract the most of underlying hardware details.

All the HAL functions related to DMA manipulation are designed so that they accept as first parameter an instance of the C `struct DMA_HandleTypeDef`. This structure is slightly different from CubeF2/F4/F7 HALs and the other CubeHALs, due to the different DMA implementation as shown in the previous paragraphs. For this reason, we will show them separately.

9.2.1 `DMA_HandleTypeDef` in F0/F1/F3/L0/L1/L4 HALs

The `struct DMA_HandleTypeDef` is used to configure a given DMA channel and it is defined in the following way in CubeF0/F1/F3/L1 HALs[13]:

[13]Please, take note that not all struct fields are reported here, but just those ones the programmer needs to fill.

```
typedef struct {
  DMA_Channel_TypeDef       *Instance;           /* Register base address            */
  DMA_InitTypeDef           Init;                /* DMA communication parameters     */
  HAL_LockTypeDef           Lock;                /* DMA locking object               */
  __IO HAL_DMA_StateTypeDef State;               /* DMA transfer state               */
  void                      *Parent;             /* Parent object state              */
  void                      (* XferCpltCallback)( struct __DMA_HandleTypeDef * hdma);
  void                      (* XferHalfCpltCallback)( struct __DMA_HandleTypeDef * hdma);
  void                      (* XferErrorCallback)( struct __DMA_HandleTypeDef * hdma);
  __IO uint32_t             ErrorCode;           /* DMA Error code                   */
} DMA_HandleTypeDef;
```

Let us see more in depth the most important fields of this struct.

- `Instance`: is the pointer to the DMA/Channel pair descriptor we are going to use. For example, `DMA1_Channel5` indicates the fifth channel of DMA1. Remember that, in these STM32 families, channels are bound to peripherals during the MCU design, so consult the datasheet for your MCU to see the channel bound to the peripheral you want to use in DMA mode.
- `Init`: is an instance of the C `struct DMA_InitTypeDef`, which is used to configure the DMA/Channel pair. We will analyze it more in depth in a while.
- `Parent`: this pointer is used by the HAL to keep track of the peripheral handlers associated to the current DMA/Channel. For example, if we are using an UART in DMA mode, this field will point to an instance of `UART_HandleTypeDef`. We will see soon how peripheral handlers are "linked" to this field.
- `XferCpltCallback`, `XferHalfCpltCallback`, `XferErrorCallback`, `XferAbortCallback`: these are pointers to functions used as callbacks to signal the user code that a DMA transfer is *completed*, *half-completed*, *an error occurred* or *the transfer aborted*. They are automatically called by the HAL when a DMA interrupt is faired, by the function `HAL_DMA_IRQHandler()`, as we will see next.

All the DMA/Channel configuration activities are performed by using an instance of the C `struct DMA_InitTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t Direction;
  uint32_t PeriphInc;
  uint32_t MemInc;
  uint32_t PeriphDataAlignment;
  uint32_t MemDataAlignment;
  uint32_t Mode;
  uint32_t Priority;
} DMA_InitTypeDef;
```

- `Direction`: it defines the DMA transfer direction, and it can assume one of the values reported in **Table 9.4**.
- `PeriphInc`: as said in previous paragraphs, a DMA controller has one *peripheral port* used to specify the address of the peripheral register involved in the memory transfer (for example, for a UART interface the address of its *Data Register* (DR)). Since a DMA memory transfer usually involves several bytes, the DMA can be programmed to automatically increment the peripheral register for every byte transmitted. This is true both when a *memory-to-memory* transfer is performed and when the peripheral is byte, half-word and word addressable (like an external SRAM memory). In this case the field assume the value `DMA_PINC_ENABLE`, otherwise `DMA_PINC_DISABLE`.
- `MemInc`: this field has the same meaning of the `PeriphInc` field, but it involves the *memory port*. It can assume the value `DMA_MINC_ENABLE` to signal that the specified memory address has to be incremented after each byte transmitted, or the value `DMA_MINC_DISABLE` to leave it unchanged after each transfer.
- `PeriphDataAlignment`: transfer data sizes of the peripheral and memory are fully programmable through this field and the next one. It can assume a value from **Table 9.5**. The DMA controller is designed to automatically perform data alignment (packing/unpacking) when source and destination data sizes differ. Please, refer to the Reference Manual of your MCU for more information.
- `MemDataAlignment`: it specifies memory transfer data size, and it can assume a value from **Table 9.6**.
- `Mode`: the DMA controller in STM32 MCUs has two working modes: `DMA_NORMAL` and `DMA_CIRCULAR`. In *normal mode* the DMA sends the specified amount of data from source to destination port and stops the activities. It must be re-armed again to do another transfer. In *circular mode*, at the end of transmission, it automatically resets the transfer counter and starts transmitting again from the first byte of source buffer (that is, it treats the source buffer as a ring buffer). This mode is also called *continuous mode*, and it is the only way to achieve high transmission speed in some peripheral (e.g., high-speed SPI devices).
- `Priority`: one important feature of the DMA controller is the ability to assign priorities to each channel, to rule concurrent requests. This field can assume a value from **Table 9.7**. In case of concurrent requests from peripherals connected to channels with the same priority, the channel with lower number fires first.

Table 9.4: Available DMA transfer directions

DMA transfer direction	Description
`DMA_PERIPH_TO_MEMORY`	Peripheral to memory direction
`DMA_MEMORY_TO_PERIPH`	Memory to peripheral direction
`DMA_MEMORY_TO_MEMORY`	Memory to memory direction

Table 9.5: DMA Peripheral data size

Peripheral data size	Description
DMA_PDATAALIGN_BYTE	Peripheral data alignment : Byte
DMA_PDATAALIGN_HALFWORD	Peripheral data alignment : HalfWord
DMA_PDATAALIGN_WORD	Peripheral data alignment : Word

Table 9.6: DMA Memory data size

Peripheral data size	Description
DMA_MDATAALIGN_BYTE	Memory data alignment : Byte
DMA_MDATAALIGN_HALFWORD	Memory data alignment : HalfWord
DMA_MDATAALIGN_WORD	Memory data alignment : Word

Table 9.7: Available DMA channel priorities

DMA channel priority	Description
DMA_PRIORITY_LOW	Priority level : Low
DMA_PRIORITY_MEDIUM	Priority level : Medium
DMA_PRIORITY_HIGH	Priority level : High
DMA_PRIORITY_VERY_HIGH	Priority level : Very_High

To initialize the DMA according to the specified parameters in the `DMA_HandleTypeDef` and `DMA_InitTypeDef` structs, the following HAL function is used:

```
HAL_StatusTypeDef HAL_DMA_Init(DMA_HandleTypeDef *hdma);
```

9.2.2 DMA Configuration in G0/G4/L4+/L5/H7 HALs

So far, we have seen that in STM32G0/G4/L4+/L5/H7 MCUs the DMA is an evolution of the DMA in F0/F1/F3/L0/L1/L4 families: the DMAMUX gives the possibility to mix peripheral request sources with every DMA channel. For this reason, the `HAL_DMA` APIs for STM32G0/G4/L4+/L5/H7 MCUs are almost similar to what seen since now, apart for the DMAMUX configuration parts.

So, from the programmer point of view, the `DMA_HandleTypeDef` struct does not add relevant fields to take care of. If you look to its definition in the STM32G0/G4/L4+/L5/H7 HALs, you will notice additional fields that - however - are automatically populated by the `HAL_DMA_Init()` routines. This simplify a lot the porting of code between different STM32 families.
Instead, the configuration of DMAMUX needs to be carried on with dedicated structs and functions.

To bind a DMA channel to a signal coming from the **Request Generator** module, we have to perform two distinct configurations. First, we have to configure the DMA channel by using an instance of the `struct DMA_HandleTypeDef` seen before. Next, we have to configure the **Request Generator** module, by using the following `struct`:

```
typedef struct {
  uint32_t SignalID;      /*!< Specifies the ID of the signal used for DMAMUX
                               request generator */
  uint32_t Polarity;      /*!< Specifies the polarity of the signal on which the
                               request is generated */
  uint32_t RequestNumber; /*!< Specifies the number of DMA request that will be
                               generated after a signal event */
} HAL_DMA_MuxRequestGeneratorConfigTypeDef;
```

- `SignalID`: specifies the ID of the signal to use as input to the **Request Generator** module. It can assume a value from the **Table 9.8**.
- `Polarity`: specifies the polarity of the input signal and it can assume a value from the **Table 9.9**
- `RequestNumber`: specifies the number of requests will trigger after a signal event occurs. This can assume the values from 1 to 32.

Table 9.8: Request Generator Signal ID

Request Generator Signal ID	Description
`HAL_DMAMUX1_REQ_GEN_EXTI0`	Request generator Signal is EXTI0 IT
...	...
`HAL_DMAMUX1_REQ_GEN_EXTI15`	Request generator Signal is EXTI15 IT
`HAL_DMAMUX1_REQ_GEN_DMAMUX1_CH0_EVT`	Request generator Signal is DMAMUX1 Channel0 Event
...	...
`HAL_DMAMUX1_REQ_GEN_DMAMUX1_CH3_EVT`	Request generator Signal is DMAMUX1 Channel3 Event
`HAL_DMAMUX1_REQ_GEN_LPTIM1_OUT`	Request generator Signal is LPTIM1 OUT

Table 9.9: Request Generator Signal Polarity

Request Generator Signal Polarity	Description
`HAL_DMAMUX_REQ_GEN_NO_EVENT`	Stop request generator events
`HAL_DMAMUX_REQ_GEN_RISING`	Generate request on rising edge events
`HAL_DMAMUX_REQ_GEN_FALLING`	Generate request on falling edge events
`HAL_DMAMUX_REQ_GEN_RISING_FALLING`	Generate request on rising and falling edge events

To configure the Request Generator module with the above settings we need to use the function:

```
HAL_DMAEx_ConfigMuxRequestGenerator(DMA_HandleTypeDef *hdma,
                          HAL_DMA_MuxRequestGeneratorConfigTypeDef *pRequestGeneratorConfig);
```

which accepts the pointer to the `DMA_HandleTypeDef` instance corresponding to the DMA channel to bind to the Request Generator signal and the instance to the 'HAL_DMA_MuxRequestGenerator-ConfigTypeDef struct.

The DMAMUX allows the synchronization of a given channel by associating it to a sync signal. This is performed by using the `HAL_DMA_MuxSyncConfigTypeDef struct`, which is defined in this way:

```
typedef struct {
  uint32_t SyncSignalID;       /* Specifies the synchronization signal gating the DMA request \
in periodic mode.    */
  uint32_t SyncPolarity;       /* Specifies the polarity of the signal on which the DMA reques\
t is synchronized.   */
  FunctionalState SyncEnable;  /* Specifies if the synchronization shall be enabled or disable\
d.             */
  FunctionalState EventEnable; /* Specifies if an event shall be generated once the RequestNum\
ber is reached.*/
  uint32_t RequestNumber;      /* Specifies the number of DMA request that will be authorized \
after a sync event. */
} HAL_DMA_MuxSyncConfigTypeDef;
```

- `SyncSignalID`: specifies the synchronization signal used to start the DMA transaction and it can assume one value from the **Table 9.10**.
- `SyncPolarity`: specifies the polarity of the signal on which the DMA request is synchronized, and it can assume a value from the **Table 9.11**.
- `SyncEnable` and `EventEnable`: specifies if the synchronization shall be enabled or disabled and it can take the value `ENABLE` or `DISABLE`.
- `RequestNumber`: specifies the number of DMA request that will be authorized after a sync event.

Table 9.10: Request Generator Signal ID

DMAMUX SyncSignal ID	Description
`HAL_DMAMUX1_SYNC_EXTI0`	Synchronization Signal is EXTI0 IT
...	...
`HAL_DMAMUX1_SYNC_EXTI15`	Synchronization Signal is EXTI15 IT
`HAL_DMAMUX1_SYNC_DMAMUX1_CH0_EVT`	Synchronization Signal is DMAMUX1 Channel0 Event
...	...
`HAL_DMAMUX1_SYNC_DMAMUX1_CH3_EVT`	Synchronization Signal is DMAMUX1 Channel3 Event
`HAL_DMAMUX1_SYNC_LPTIM1_OUT`	Synchronization Signal is LPTIM1 OUT

Table 9.11: DMAMUX Sync Signal Polarity

Request Generator Signal Polarity	Description
`HAL_DMAMUX_SYNC_NO_EVENT`	Stop synchronization events
`HAL_DMAMUX_SYNC_RISING`	Synchronize with rising edge events
`HAL_DMAMUX_SYNC_FALLING`	Synchronize with falling edge events
`HAL_DMAMUX_SYNC_RISING_FALLING`	Synchronize with rising and falling edge events

To configure the channel synchronization, the following HAL function is used:

```
HAL_DMAEx_ConfigMuxSync(DMA_HandleTypeDef *hdma, HAL_DMA_MuxSyncConfigTypeDef *pSyncConfig);
```

which accepts the pointer to the `DMA_HandleTypeDef` instance corresponding to the DMA channel to bind to sync with the external source and the instance to the `HAL_DMA_MuxSyncConfigTypeDef` struct.

9.2.3 `DMA_HandleTypeDef` in F2/F4/F7 HALs

So far, we have seen that in STM32F2/F4/F7 MCUs the DMA adopts a slightly different terminology and organization. The CubeHAL so reflects these differences.
The `struct DMA_HandleTypeDef` is defined in the following way:

```
typedef struct {
  DMA_Stream_TypeDef         *Instance;          /* Register base address              */
  DMA_InitTypeDef            Init;              /* DMA communication parameters       */
  HAL_LockTypeDef            Lock;              /* DMA locking object                 */
  __IO HAL_DMA_StateTypeDef  State;             /* DMA transfer state                 */
  void                       *Parent;           /* Parent object state                */
  void                       (* XferCpltCallback)( struct __DMA_HandleTypeDef * hdma);
  void                       (* XferHalfCpltCallback)( struct __DMA_HandleTypeDef * hdma);
  void                       (* XferM1CpltCallback)( struct __DMA_HandleTypeDef * hdma);
  void                       (* XferErrorCallback)( struct __DMA_HandleTypeDef * hdma);
  __IO uint32_t              ErrorCode;         /* DMA Error code                     */
  uint32_t                   StreamBaseAddress; /* DMA Stream Base Address            */
  uint32_t                   StreamIndex;       /*!< DMA Stream Index                 */
} DMA_HandleTypeDef;
```

Let us see more in depth the most important fields of this struct.

- `Instance`: is the pointer to the stream descriptor we are going to use. For example, `DMA1_Stream6` indicates the seventh[14] stream of DMA1. Remember that a stream must be bound to a channel before it can be used. This is achieved through the `Init` field, as we will see in a while. Remember also that channels are bound to peripherals during the MCU design, so consult the datasheet for your MCU to see the channel bound to the peripheral you want to use in DMA mode.
- `Init`: is an instance of the C `struct DMA_InitTypeDef`, which is used to configure the DMA/Channel/Stream triple. We will study it more in depth in a while.
- `Parent`: this pointer is used by the HAL to keep track of the peripheral handlers associated to the current DMA/Channel. For example, if we are using an UART in DMA mode, this field will point to an instance of `UART_HandleTypeDef`. We will see soon how peripheral handlers are "linked" to this field.

[14] Stream count starts from zero.

- `XferCpltCallback`, `XferHalfCpltCallback`, `XferM1CpltCallback`, `XferErrorCallback`: these are pointers to functions used as callbacks to signal the user code that a DMA transfer is *completed*, *half-completed*, *the transmission of first buffer in a multi-buffer transfer is completed* or *an error occurred*. They are automatically called by the HAL when a DMA interrupt is faired, by the function `HAL_DMA_IRQHandler()`, as we will see next.

All the DMA/Channel configuration activities are performed by using an instance of the C `struct DMA_InitTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t Channel;
  uint32_t Direction;
  uint32_t PeriphInc;
  uint32_t MemInc;
  uint32_t PeriphDataAlignment;
  uint32_t MemDataAlignment;
  uint32_t Mode;
  uint32_t Priority;
  uint32_t FIFOMode;
  uint32_t FIFOThreshold;
  uint32_t MemBurst;
  uint32_t PeriphBurst;
} DMA_InitTypeDef;
```

- `Channel`: it specifies the DMA channel used for the given stream. It can assume the values `DMA_CHANNEL_0`, `DMA_CHANNEL_1` up to `DMA_CHANNEL_7`. Remember that peripherals are bound to streams and channels during the MCU design, so consult the datasheet for your MCU to see the stream bound to the peripheral you want to use in DMA mode.
- `Direction`: it defines the DMA transfer direction, and it can assume one of the values reported in **Table 9.4**.
- `PeriphInc`: as said in previous paragraphs, a DMA controller has one *peripheral port* used to specify the address of the peripheral register involved in the memory transfer (for example, for a UART interface the address of its *Data Register* (DR)). Since a DMA memory transfer usually involves several bytes, the DMA can be programmed to automatically increment the peripheral register for every byte transmitted. This is true both when a *memory-to-memory* transfer is performed and when the peripheral is byte, half-word and word addressable (like an external SRAM memory is). In this case the field assumes the value `DMA_PINC_ENABLE`, otherwise `DMA_PINC_DISABLE`.
- `MemInc`: this field has the same meaning of the `PeriphInc` field, but it involves the *memory port*. It can assume the value `DMA_MINC_ENABLE` to signal that the specified memory address has to be incremented after each byte transmitted, or the value `DMA_MINC_DISABLE` to leave it unchanged after each transfer.

- `PeriphDataAlignment`: transfer data sizes of the peripheral and memory are fully programmable through this field and the next one. It can assume a value from **Table 9.5**. The DMA controller is designed to automatically perform data alignment (packing/unpacking) when source and destination data sizes differ. This topic is outside the scope of this book. Please, refer to the Reference Manual of your MCU.
- `MemDataAlignment`: it specifies memory transfer data size, and it can assume a value from **Table 9.6**.
- `Mode`: the DMA controller in STM32 MCUs has two working modes: `DMA_NORMAL` and `DMA_CIRCULAR`. In *normal mode* the DMA sends the specified amount of data from source to destination port and stops the activities. It must be re-armed again to do another transfer. In *circular mode*, at the end of transmission, it automatically resets the transfer counter and starts transmitting again from the first byte of the source buffer (that is, it treats the source buffer as a ring buffer). This mode is also called *continuous mode*, and it is the only way to achieve high transmission speeds in some peripheral (e.g. fast SPI devices).
- `Priority`: one important feature of the DMA controller is the ability to assign priorities to each stream, in order to rule concurrent requests. This field can assume a value from **Table 9.7**. In case of concurrent requests from peripherals connected to streams with the same priority, the stream with lower number fires first.
- `FIFOMode`: it is used to enable/disable the DMA *FIFO mode* using `DMA_FIFOMODE_ENABLE/DMA_FIFOMODE_DISABLE` macros. In STM32F2/F4/F7 MCUs, each stream has an independent 4-word (4 * 32 bits) FIFO. The FIFO is used to temporarily store data coming from the source before transmitting it to the destination. When disabled, the *Direct mode* is used (this is the "normal" mode available in other STM32 MCUs).
  The FIFO mode introduces several advantages: it reduces SRAM access and so give more time for the other masters to access the Bus Matrix without additional concurrency; it allows software to do burst transactions which optimize the transfer bandwidth (more about this in a while); it allows packing/unpacking data to adapt source and destination data width with no extra DMA access.
  If DMA FIFO is enabled, data packing/unpacking and/or Burst mode can be used. The FIFO is automatically emptied according to a threshold level. This level is software-configurable between 1/4, 1/2, 3/4 or full size.
- `FIFOThreshold`: it specifies the FIFO threshold level, and it can assume a value from **Table 9.12**.
- `MemBurst`: a round robin scheduling policy rules the access of a DMA stream before it can transfer a sequence of bytes through the AHB bus. This "slows" down the transfer operations, and for some high-speed peripherals it can be a bottleneck. A burst transfer allows a DMA stream to transmit data repeatedly without going through all the steps required to transmit each piece of data in a separate transaction. The *burst mode* works in conjunction with FIFOs and it says nothing about the amount of bytes transferred. This is based on the settings of `MemDataAlignment` field (when we are doing a *memory-to-peripheral* transfer). `MemBurst` indicates the number of "shoots" performed by the stream, and it is made of bytes, half-word and word depending on the source configuration. The `MemBurst` field can assume one value from **Table 9.13**.

- `PeriphBurst`: This field has the same meaning of the previous one, but it is related to *peripheral-to-memory* transfers. It can assume a value from **Table 9.14**.

Table 9.12: Available FIFO threshold levels

DMA channel priority	Description
`DMA_FIFO_THRESHOLD_1QUARTERFULL`	FIFO threshold 1 quart full configuration
`DMA_FIFO_THRESHOLD_HALFFULL`	FIFO threshold half full configuration
`DMA_FIFO_THRESHOLD_3QUARTERSFULL`	FIFO threshold 3 quarts full configuration
`DMA_FIFO_THRESHOLD_FULL`	FIFO threshold full configuration

Table 9.13: Available DMA memory burst modes

DMA channel priority	Description
`DMA_MBURST_SINGLE`	Single burst
`DMA_MBURST_INC4`	Burst of 4 beats
`DMA_MBURST_INC8`	Burst of 8 beats
`DMA_MBURST_INC16`	Burst of 16 beats

Table 9.14: Available DMA peripheral burst modes

DMA channel priority	Description
`DMA_PBURST_SINGLE`	Single burst
`DMA_PBURST_INC4`	Burst of 4 beats
`DMA_PBURST_INC8`	Burst of 8 beats
`DMA_PBURST_INC16`	Burst of 16 beats

9.2.4 How to Perform DMA Transfers in Polling Mode

Once we have configured the DMA channel/stream, we have to do few other things:

- to setup the addresses on the memory and peripheral port;
- to specify the amount of data we are going to transfer;
- to arm the DMA;
- to enable the DMA mode on the corresponding peripheral.

The HAL abstracts the first three points by using the

```
HAL_StatusTypeDef HAL_DMA_Start(DMA_HandleTypeDef *hdma, uint32_t SrcAddress, uint32_t DstAddr\
ess, uint32_t DataLength);
```

while the fourth point is peripheral dependent, and we have to consult our specific MCU datasheet. However, as we will see later, the HAL also abstracts this point (for example, by using the corresponding `HAL_UART_Transmit_DMA()` function when configuring an UART in DMA mode).

Now we should have all the elements to see a fully working application. What we are going to do in the next example is just sending a string over the UART2 peripheral using DMA mode. The involved steps are:

- The UART2 is configured using the `HAL_UART` module, as we have seen in the previous chapter.
- The DMA1 channel (or the DMA1 channel/stream couple for STM32F4 based Nucleo boards) is configured to do a *memory-to-peripheral* transfer (see **Table 9.15**)
- The corresponding channel is armed to execute the transfer and UART is enabled in DMA mode.

Nucleo P/N	USART2_TX	UART2_RX
NUCLEO-G474RE	**DMA1/CH7**	DMA1/CH6
NUCLEO-F446RE	**DMA1/CH4/Stream6**	**DMA1/CH4/Stream5**
NUCLEO-F401RE		
NUCLEO-F303RE	**DMA1/CH7**	DMA1/CH6
NUCLEO-F103RB	**DMA1/CH7**	DMA1/CH6
NUCLEO-F072RB	**DMA1/CH4**	DMA1/CH5
NUCLEO-L476RG	**DMA1/CH7/Request2**	DMA1/CH6/Request2
NUCLEO-L152RE	DMA1/CH7	**DMA1/CH6**
NUCLEO-L073RZ	**DMA1/CH7/Request4**	DMA1/CH5/Request4

Table 9.15: How USART_TX/USART_RX DMA channels are mapped in the MCUs equipping Nucleo boards used in text

The following example, which is designed to work on a Nucleo-F072 (refer to book samples for the other Nucleo boards), shows how to do this easily.

Filename: `Core/Src/main-ex1.c`

```
31 UART_HandleTypeDef huart2;
32 DMA_HandleTypeDef hdma_usart2_tx;
33 char msg[] = "Hello STM32 Lovers! This message is transferred in DMA Mode.\r\n";
34
35 int main(void) {
36   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
37   HAL_Init();
38
39   /* Configure the system clock */
40   SystemClock_Config();
41
42   /* Initialize all configured peripherals */
43   MX_GPIO_Init();
44   MX_USART2_UART_Init();
45   MX_DMA_Init();
46
```

```
47    /* USART2_TX Init */
48    /* USART2 DMA Init */
49    hdma_usart2_tx.Instance = DMA1_Channel4;
50    hdma_usart2_tx.Init.Direction = DMA_MEMORY_TO_PERIPH;
51    hdma_usart2_tx.Init.PeriphInc = DMA_PINC_DISABLE;
52    hdma_usart2_tx.Init.MemInc = DMA_MINC_ENABLE;
53    hdma_usart2_tx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
54    hdma_usart2_tx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
55    hdma_usart2_tx.Init.Mode = DMA_NORMAL;
56    hdma_usart2_tx.Init.Priority = DMA_PRIORITY_LOW;
57    HAL_DMA_Init(&hdma_usart2_tx);
58
59    HAL_DMA_Start(&hdma_usart2_tx,  (uint32_t)msg,  (uint32_t)&huart2.Instance->TDR, strlen(msg)\
60  );
61    //Enable UART in DMA mode
62    huart2.Instance->CR3 |= USART_CR3_DMAT;
63    //Wait for transfer complete
64    HAL_DMA_PollForTransfer(&hdma_usart2_tx, HAL_DMA_FULL_TRANSFER, HAL_MAX_DELAY);
65    //Disable UART DMA mode
66    huart2.Instance->CR3 &= ~USART_CR3_DMAT;
67    //Turn LD2 ON
68    HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
69
70    /* Infinite loop */
71    while (1);
72  }
```

First, we configure *DMA1_Channel4* to do a *memory-to-peripheral* transfer. Since the USART peripheral has a *Transmit Data Register* (TDR) just one byte wide, we configure the DMA so that the peripheral address is not automatically incremented (`DMA_PINC_DISABLE`), while we want that the source memory address is automatically incremented at every byte sent (`DMA_MINC_ENABLE`). Once the configuration is completed, we call the `HAL_DMA_Init()` which performs the DMA interface configuration according to the information provided inside the `hdma_usart2_tx.Init` structure. Next, at line 59, we invoke the `HAL_DMA_Start()` routine, which configures the source memory address (that is the address of the `msg` array), the destination peripheral address (that is the address of `USART2->TDR` register) and the amount of data we are going to transmit. The DMA is now ready to shoot, and we start the transmission by setting the corresponding bit of USART2 peripheral, as shown in line 62. Finally, take note that the function `MX_DMA_Init()` (invoked at line 45) uses the macro `__HAL_RCC_DMA1_CLK_ENABLE()` to enable the DMA1 controller (remember that almost every STM32 internal module must be enabled by using the `__HAL_RCC_<PERIPHERAL>_CLK_ENABLE()` macro).

Since we do not know how long it takes to complete the transfer procedure, we use the function:

```
HAL_StatusTypeDef HAL_DMA_PollForTransfer(DMA_HandleTypeDef *hdma, uint32_t CompleteLevel, uin\
t32_t Timeout);
```

which automatically waits for full transfer completion. This way to send data in DMA mode is called "polling mode" in the official ST documentation. Once the transfer is completed, we disable the UART2 DMA mode and turn on the LD2 LED.

9.2.5 How to Perform DMA Transfers in Interrupt Mode

From the performance point of view, the DMA transfer in polling mode is meaningless, unless our code does not need to wait for transfer completion. If our goal is to improve the overall performances, there are no reasons to use the DMA controller and then to consume a lot of CPU cycles waiting for transfer completion. So, the best option is to arm the DMA and let it notify us when the transfer is completed.
The DMA is able to generate interrupts related to channel activities (for example, the DMA1 in an STM32F072 MCU has one IRQ for channel 1, one for channels 2 and 3, one for channels from 4 to 7). Moreover, three independent enable bits are available to enable IRQ on *half transfer*, *full transfer* and *transfer error*.

The DMA can be enabled in interrupt mode following these steps:

- define three functions acting as callback routines and pass them to function pointers `XferCpltCallback`, `XferHalfCpltCallback` and `XferErrorCallback` in a `DMA_HandleTypeDef` handler (**it is ok to define only the functions we are interested in, but set the corresponding pointer to NULL, otherwise strange faults may occur**);
- write down the ISR for the IRQ associated to the channel you are using and do a call to the `HAL_DMA_IRQHandler()` passing the reference to the `DMA_HandleTypeDef` handler;
- enable the corresponding IRQ in the NVIC controller;
- use the function `HAL_DMA_Start_IT()`, which automatically performs all the necessary setup steps for you, passing to it the same arguments of the `HAL_DMA_Start()`.



It is important to remark a thing about the `XferCpltCallback`, `XferHalfCpltCallback` and `XferErrorCallback` callbacks: we need to set them just when we are using DMA without the mediation of the CubeHAL. Let us clarify this concept.

Suppose that we are using the UART2 in DMA mode. If we are doing the DMA management ourselves, then it is ok to define those callback routines and to manage the necessary UART interrupt related configurations each time a transfer takes place. However, if we are using the `HAL_UART_Trasmit_DMA()`/`HAL_UART_Receive_DMA()` routines, then the HAL already correctly defines those callbacks and we do not have to change them. Instead, for example, to capture the DMA completion event for the UART, we need to define the function `HAL_UART_RxCpltCallback()`. Always consult the HAL documentation for the peripheral you are going to use in DMA mode.

The following example shows how to a DMA *memory-to-peripheral* transfer in interrupt mode.

Filename: `Core/Src/main-ex2.c`

```
50    hdma_usart2_tx.Instance = DMA1_Channel4;
51    hdma_usart2_tx.Init.Direction = DMA_MEMORY_TO_PERIPH;
52    hdma_usart2_tx.Init.PeriphInc = DMA_PINC_DISABLE;
53    hdma_usart2_tx.Init.MemInc = DMA_MINC_ENABLE;
54    hdma_usart2_tx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
55    hdma_usart2_tx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
56    hdma_usart2_tx.Init.Mode = DMA_NORMAL;
57    hdma_usart2_tx.Init.Priority = DMA_PRIORITY_LOW;
58    hdma_usart2_tx.XferCpltCallback = &DMATransferComplete;
59    HAL_DMA_Init(&hdma_usart2_tx);
60
61    /* DMA interrupt init */
62    HAL_NVIC_SetPriority(DMA1_Channel4_5_IRQn, 0, 0);
63    HAL_NVIC_EnableIRQ(DMA1_Channel4_5_IRQn);
64
65    HAL_DMA_Start_IT(&hdma_usart2_tx,  (uint32_t)msg,
66                     (uint32_t)&huart2.Instance->TDR, strlen(msg));
67
68    //Enable UART in DMA mode
69    huart2.Instance->CR3 |= USART_CR3_DMAT;
70
71    /* Infinite loop */
72    while (1);
73  }
74
75  void DMATransferComplete(DMA_HandleTypeDef *hdma) {
76    if(hdma->Instance == DMA1_Channel4) {
77      //Disable UART DMA mode
78      huart2.Instance->CR3 &= ~USART_CR3_DMAT;
79      //Turn LD2 ON
80      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
81    }
82  }
```

9.2.6 Using the HAL_UART Module with DMA Mode Transfers

In Chapter 8 we left out how to use the UART in DMA mode. We have already seen in the previous paragraphs how to do it. However, we had to play with some USART registers to enable the peripheral in DMA mode.

The HAL_UART module is designed to abstract from all underlying hardware details. The steps required to use it are the following:

- configure the DMA channel/stream hardwired to the UART you are going to use, as seen in this chapter;

- link the `UART_HandleTypeDef` to the `DMA_HandleTypeDef` using the `__HAL_LINKDMA()`;
- enable the DMA interrupt related to the channel/stream you are using and call the `HAL_DMA_IRQHandler()` routine from its ISR;
- enable the UART related interrupt and call the `HAL_UART_IRQHandler()` routine from its ISR (**this is really important, do not skip this step**[15]);
- Use the `HAL_UART_Transmit_DMA()` and `HAL_UART_Receive_DMA()` function to exchange data over the UART and be prepared to be notified of transfer completion implementing the `HAL_UART_RxCpltCallback()`.

The following code shows how to receive three bytes from UART2 in DMA mode in an STM32F072 MCU[16]:

```
uint8_t dataArrived = 0;
uint8_t data[3];

int main(void) {
  HAL_Init();
  Nucleo_BSP_Init(); //Configure the UART2

  //Configure the DMA1 Channel 5, which is wired to the UART2_RX request line
  hdma_usart2_rx.Instance = DMA1_Channel5;
  hdma_usart2_rx.Init.Direction = DMA_PERIPH_TO_MEMORY;
  hdma_usart2_rx.Init.PeriphInc = DMA_PINC_DISABLE;
  hdma_usart2_rx.Init.MemInc = DMA_MINC_ENABLE;
  hdma_usart2_rx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
  hdma_usart2_rx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
  hdma_usart2_rx.Init.Mode = DMA_NORMAL;
  hdma_usart2_rx.Init.Priority = DMA_PRIORITY_LOW;
  HAL_DMA_Init(&hdma_usart2_rx);

  //Link the DMA descriptor to the UART2 one
  __HAL_LINKDMA(&huart, hdmarx, hdma_usart2_rx);

  /* DMA interrupt init */
  HAL_NVIC_SetPriority(DMA1_Channel4_5_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel4_5_IRQn);

  /* Peripheral interrupt init */
  HAL_NVIC_SetPriority(USART2_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(USART2_IRQn);

  //Receive three bytes from UART2 in DMA mode
```

[15] It is important to enable the UART-related interrupt and to invoke the `HAL_UART_IRQHandler()` routine, because the HAL is designed so that UART-related errors (like parity error, overrun error, and so on) may be raised even when the UART is driven in DMA mode. By catching the error condition, the HAL suspends the DMA transfer and invokes the corresponding error callback to signal the error condition to the application layer.

[16] Arranging the DMA initialization code for other STM32 MCUs is left as exercise to the reader.

```
  HAL_UART_Receive_DMA(&huart2, &data, 3);

  while(!dataArrived); //Wait for the arrival of data from UART

  /* Infinite loop */
  while (1);
}

//This callback is automatically called by the HAL when the DMA transfer is completed
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
        dataArrived = 1;
}

void DMA1_Channel4_5_IRQHandler(void) {
  HAL_DMA_IRQHandler(&hdma_usart2_rx); //This will automatically call the HAL_UART_RxCpltCallb\
ack()
}
```

Where the `HAL_UART_RxCpltCallback()` is exactly called? In the previous paragraphs we have seen that the `DMA_HandleTypeDef` contains a pointer (named `XferCpltCallback`) to a function that is invoked by the `HAL_DMA_IRQHandler()` routine when the DMA transfer has been completed. However, when we use the HAL module for a given peripheral (`HAL_UART` in this case), we do not need to provide our own callbacks: they are defined internally by the HAL, which uses them to carry out its activities. The HAL offers us the ability to define our corresponding callback functions (`HAL_UART_RxCpltCallback()` for UART_RX transfers in DMA mode), which will be invoked automatically by the HAL, as shown in **Figure 9.7**. **This rule applies to all HAL modules**.

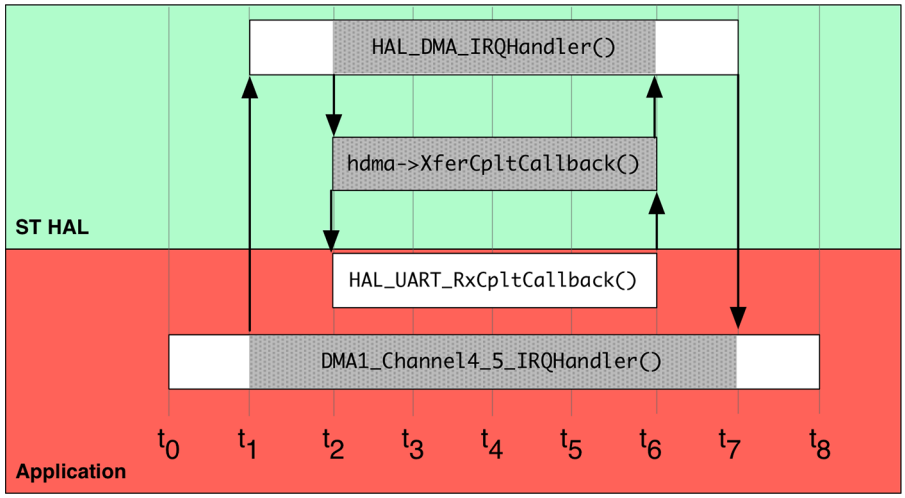


Figure 9.7: The call sequence generated by the `HAL_DMA_IRQHandler()`

As you can see, once mastered how the DMA controller works, is simple to use a peripheral using this transfer mode.

9.2.7 Programming the DMAMUX With the CubeHAL

One of the nine Nucleo boards used in this book is equipped with an STM32G4 MCU, which provides the DMAMUX peripheral. This gives us the opportunity to see both how to program the DMAMUX with the CubeHAL and how to synchronize a request transfer by using an external sync signal. The example we are going to analyze here is similar to the previous one: the USART2 is used in DMA mode to transfer a bunch of chars. This time, however, the transfer is synchronized to the EXTI13 lines, the interrupt line connected to the PC13 GPIO, which is wired to the USER button in all Nucleo-64 boards.

Filename: `Core/Src/main-ex3.c`

```
49    /* USART2_TX Init */
50    /* USART2 DMA Init */
51    hdma_usart2_tx.Instance = DMA1_Channel7;
52    hdma_usart2_tx.Init.Request = DMA_REQUEST_USART2_TX;
53    hdma_usart2_tx.Init.Direction = DMA_MEMORY_TO_PERIPH;
54    hdma_usart2_tx.Init.PeriphInc = DMA_PINC_DISABLE;
55    hdma_usart2_tx.Init.MemInc = DMA_MINC_ENABLE;
56    hdma_usart2_tx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
57    hdma_usart2_tx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
58    hdma_usart2_tx.Init.Mode = DMA_CIRCULAR;
59    hdma_usart2_tx.Init.Priority = DMA_PRIORITY_LOW;
60    HAL_DMA_Init(&hdma_usart2_tx);
61
62    pSyncConfig.SyncSignalID = HAL_DMAMUX1_SYNC_EXTI13;
63    pSyncConfig.SyncPolarity = HAL_DMAMUX_SYNC_FALLING;
64    pSyncConfig.SyncEnable = ENABLE;
65    pSyncConfig.EventEnable = ENABLE;
66    pSyncConfig.RequestNumber = strlen(msg);
67    HAL_DMAEx_ConfigMuxSync(&hdma_usart2_tx, &pSyncConfig);
68
69    __HAL_LINKDMA(&huart2,hdmatx,hdma_usart2_tx);
70
71    /* DMA interrupt init */
72    /* DMA1_Channel7_IRQn interrupt configuration */
73    HAL_NVIC_SetPriority(DMA1_Channel7_IRQn, 0, 0);
74    HAL_NVIC_EnableIRQ(DMA1_Channel7_IRQn);
75
76    HAL_UART_Transmit_DMA(&huart2, (uint8_t*)msg, strlen(msg));
77
78    /* Infinite loop */
79    while (1);
80  }
81
82  void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
83    if(huart->Instance == USART2) {
```

```
84      //Turn LD2 ON
85      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
86    }
87  }
```

The code should be easy to understand. The `USART2_TX` request source is bound to the `DMA1_Channel7`, but it could be connected to any DMA channel thanks to the DMAMUX. This time the DMA is configured to work in *circular mode* (line 58) for a reason that is going to be clear in a while. The channel is bound to the EXTI13 sync signal at lines 62:67: this means that the transfer will take place every time the GPIO13 level will go from up to down level. Finally, both the `EXTI15_10_IRQn` and `DMA1_Channel7_IRQn` IRQs are enabled and the DMA is armed by using the `HAL_UART_Transmit_DMA()`.

The above code will cause that the UART2_TX request signal will be asserted every time the USER button is pressed, since the DMA is configured in circular mode. The corresponding message will be printed on the serial console. Please, take note that that the DMA destination address is the one-byte `USART2->TDR` *Trasmit Data Register*. To allow the DMA to transfer all chars contained in the `msg` variable, the `pSyncConfig.RequestNumber` parameter is set to the length of the message. This will cause that the DMA will fire automatically `strlen(msg)` times. However, this implies that the string cannot exceed 32 characters.

Once the transfer of all chars is complete, the `DMA1_Channel7` interrupt will fire. This will cause that the `HAL_UART_TxCpltCallback()` at line 86 is called, inverting the state of the LD2 LED.

9.2.8 Miscellaneous Functions From `HAL_DMA` and `HAL_DMA_Ex` Modules

The `HAL_DMA` module provides other functions that help using the DMA controller. Let us see them briefly.

```
HAL_StatusTypeDef HAL_DMA_Abort(DMA_HandleTypeDef *hdma);
```

This function disables the DMA stream/channel. If a stream is disabled while a data transfer is ongoing, the current data will be transferred, and the stream will be effectively disabled only after the transfer of this single data is finished.

Some STM32 MCUs can perform multi-buffer DMA transfers, which allow to use two separated buffers during the transfer process: the DMA will automatically "jump" from the first buffer (named *memory0*) to the second one (named *memory1*) when the end of the first one is reached. This especially useful when DMA works in circular mode. The function:

```
HAL_StatusTypeDef HAL_DMAEx_MultiBufferStart(DMA_HandleTypeDef *hdma, uint32_t SrcAddress, uin\
t32_t DstAddress, uint32_t SecondMemAddress, uint32_t DataLength);
```

is used to setup multi-buffer DMA transfers. It is available only in F2/F4/F7 HALs. A corresponding `HAL_DMAEx_MultiBufferStart_IT()` is also available, which also takes care of enabling DMA interrupts.

The function:

```
HAL_StatusTypeDef HAL_DMAEx_ChangeMemory(DMA_HandleTypeDef *hdma, uint32_t Address, HAL_DMA_Me\
moryTypeDef memory);
```

changes the *memory0* or *memory1* address on the fly in a multi-buffer DMA transaction.

9.3 Using CubeMX to Configure DMA Requests

CubeMX can reduce to the minimum the effort required to setting up channel/stream requests. Once you have enabled a peripheral in the *Pinout* section, go inside the *System view* section and click on the **DMA** button. The *DMA Mode and Configuration* pane appears, as shown in **Figure 9.8**.

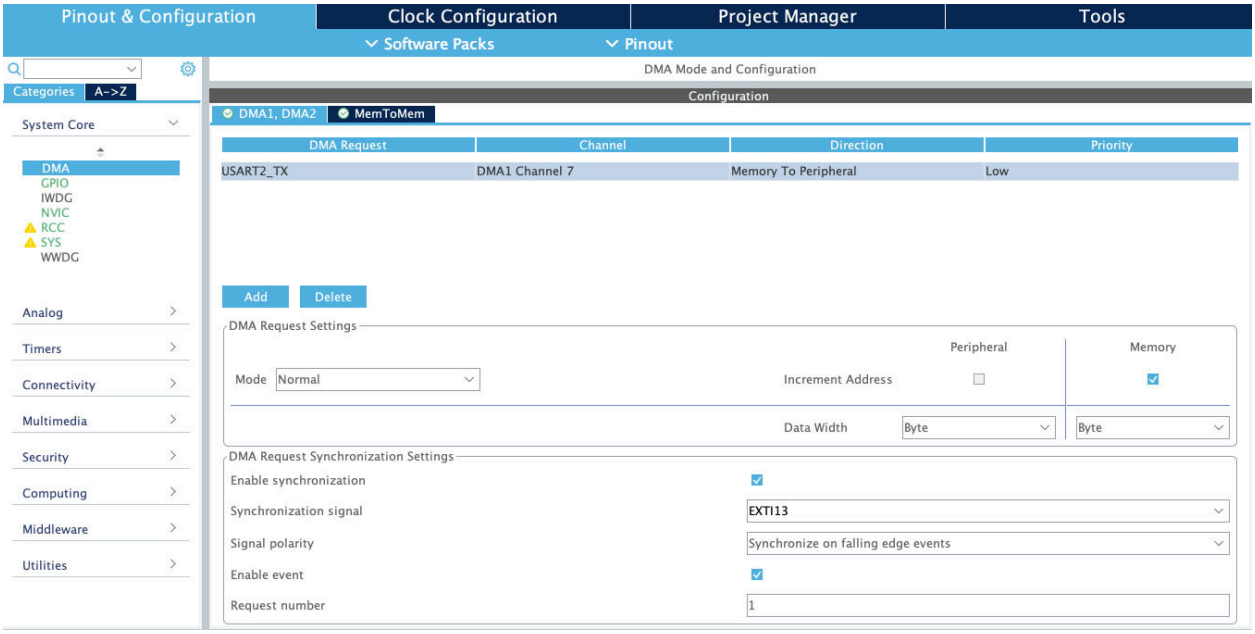


Figure 9.8: The DMA Configuration dialog in CubeMX

The dialog contains two tabs. The first one is related to peripheral requests. For example, if you want to enable a DMA request for USART2 in transmit mode (to do a *memory-to-peripheral* transfer), click on the **Add** button, and select the `USART2_TX` entry. CubeMX will automatically fill the remaining fields for you, selecting the right channel. You can then assign a priority to the request, and to set other things like the DMA mode, peripheral/memory increment, DMAMUX related settings, and so on. In the same way, it is possible to configure DMA channels/streams to do *memory-to-memory* transfers.

CubeMX will automatically generate the right initialization code for the used request/channels inside the **stm32XXxx_hal_msp.c** file.

9.4 Correct Memory Allocation of DMA Buffers

If you take a look at the source code of all examples presented in this chapter, you can see that DMA buffers (that is, both source and destination arrays used to perform *memory-to-peripheral* and *peripheral-to-memory* transfers) are always allocated at the global scope. Why are we doing that?

This is a common mistake that all novices sooner or later will do. When we declare a variable at the local scope (that is, on the stack frame of the called routine), that variable will "live" as long as that stack frame is active. When the called function exits, the stack area where the variable has been allocated is reassigned to other uses (to store the arguments or other local variables of the next called function). If we use a local variable as buffer for DMA transfers (that is, we pass to the DMA memory port the address of a memory location in the stack), then it will be very likely that DMA will access to a memory region containing other data, corrupting that memory area if we are doing a *peripheral-to-memory* transfer, unless we are sure that the function is never popped from the stack (this could be the case of a variable declared inside the `main()` function).

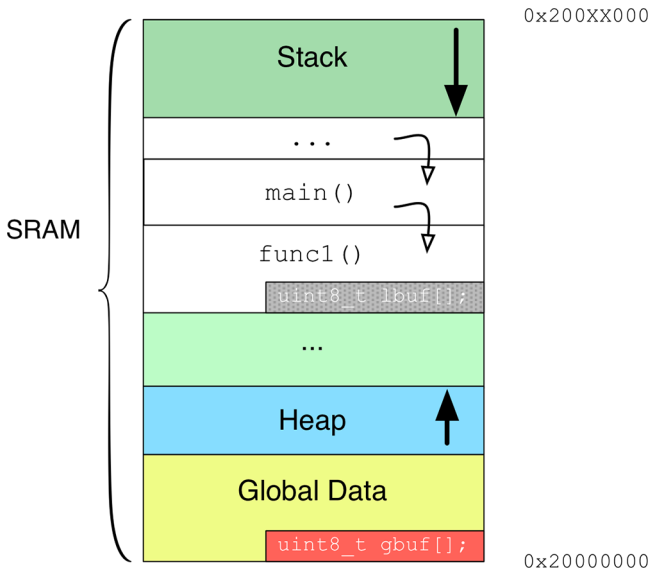


Figure 9.9: The difference between a variable allocated locally and globally

The **Figure 9.9** clearly shows the difference between a variable allocated locally (`lbuf`) and one allocated at the global scope (`gbuf`). `lbuf` will be active as long the `func1()` is on the stack.

If you want to avoid global variables in your application, another solution is represented by declaring it as `static`. As we will discover in Chapter 20, `static` variables are automatically allocated inside the `.data` region (*Global Data* region in **Figure 9.9**), even if their "visibility" is limited at the local scope.

9.5 A Case Study: The DMA *Memory-To-Memory* Transfer Performance Analysis

The DMA controller can be also used to do *memory-to-memory* transfers[17]. For example, it can be used to move a large array of data from flash memory to the SRAM, or to copy arrays in SRAM, or to zero a memory area. The C library usually provides a set of functions to accomplish this task. `memcpy()` and `memset()` are the most common ones. Surfing around in the web, you can find several tests that do a performance comparison between `memcpy()`/`memset()` routines and DMA transfers. The majority of these tests claim that usually the DMA is a lot slower than the Cortex-M core. Is this true? The answer is: it depends. So, why would you use the DMA when you have already those routines?

The story behind these tests is much more complicated, and it involves several factors like the memory align, the C library used and the right DMA settings. Let us consider the following test application (the code is designed to run on an STM32F4 MCU) divided in several stages:

```
12  DMA_HandleTypeDef hdma_memtomem_dma2_stream0;
13
14  const uint8_t flashData[] = {0xe7, 0x49, 0x9b, 0xdb, 0x30, 0x5a, ...};
15  uint8_t sramData[1000];
16
17  int main(void) {
18    HAL_Init();
19    Nucleo_BSP_Init();
20
21    hdma_memtomem_dma2_stream0.Instance = DMA2_Stream0;
22    hdma_memtomem_dma2_stream0.Init.Channel = DMA_CHANNEL_0;
23    hdma_memtomem_dma2_stream0.Init.Direction = DMA_MEMORY_TO_MEMORY;
24    hdma_memtomem_dma2_stream0.Init.PeriphInc = DMA_PINC_ENABLE;
25    hdma_memtomem_dma2_stream0.Init.MemInc = DMA_MINC_ENABLE;
26    hdma_memtomem_dma2_stream0.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
27    hdma_memtomem_dma2_stream0.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
28    hdma_memtomem_dma2_stream0.Init.Mode = DMA_NORMAL;
29    hdma_memtomem_dma2_stream0.Init.Priority = DMA_PRIORITY_LOW;
30    hdma_memtomem_dma2_stream0.Init.FIFOMode = DMA_FIFOMODE_ENABLE;
31    hdma_memtomem_dma2_stream0.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
32    hdma_memtomem_dma2_stream0.Init.MemBurst = DMA_MBURST_SINGLE;
33    hdma_memtomem_dma2_stream0.Init.PeriphBurst = DMA_MBURST_SINGLE;
34    HAL_DMA_Init(&hdma_memtomem_dma2_stream0);
35
36    GPIOC->ODR = 0x100;
37    HAL_DMA_Start(&hdma_memtomem_dma2_stream0,  (uint32_t)&flashData,  (uint32_t)sramData, 1000);
38    HAL_DMA_PollForTransfer(&hdma_memtomem_dma2_stream0, HAL_DMA_FULL_TRANSFER, HAL_MAX_DELAY);
39    GPIOC->ODR = 0x0;
```

[17]Remember that in STM32F2/F4/F7 MCUs only the DMA2 can be used for this kind of transfers.

```
40
41    while(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin));
42
43    hdma_memtomem_dma2_stream0.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
44    hdma_memtomem_dma2_stream0.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;
45
46    HAL_DMA_Init(&hdma_memtomem_dma2_stream0);
47
48    GPIOC->ODR = 0x100;
49    HAL_DMA_Start(&hdma_memtomem_dma2_stream0,  (uint32_t)&flashData,  (uint32_t)sramData, 250);
50    HAL_DMA_PollForTransfer(&hdma_memtomem_dma2_stream0, HAL_DMA_FULL_TRANSFER, HAL_MAX_DELAY);
51    GPIOC->ODR = 0x0;
52
53    HAL_Delay(1000); /* This is a really primitive form of debouncing */
54
55    while(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin));
56
57    GPIOC->ODR = 0x100;
58    memcpy(sramData, flashData, 1000);
59    GPIOC->ODR = 0x0;
```

Here we have two quite large arrays. One of these, `flashData`, is allocated inside the flash memory thanks to the `const` modifier[18]. We want to copy its content inside the `sramData` array, which is stored inside the SRAM as the name suggests, and we want to test how long it takes using DMA and `memcpy()` function.

First, we start testing the DMA. The `hdma_memtomem_dma2_stream0` handle is used to configure the DMA2 stream0/channel0 to execute a *memory-to-memory* transfer. In the first stage we configure the DMA stream to perform a byte-aligned memory transfer. Once the DMA configuration is completed, we start the transfer. Using an oscilloscope attached to Nucleo PC8 pin, we can measure how long the transfer takes. Pressing the Nucleo USER button (connected to PC13) causes the start of another test stage. This time we configure the DMA so that a word-aligned transfer is executed. Finally, at line 58 we test how long it takes to copy the array using `memcpy()`.

[18]The reason why this happens will be explained in Chapter 20.

Nucleo P/N	DMA M2M Byte-aligned	DMA M2M Word-aligned	DMA M2M Word-aligned FIFO DISABLED	memcpy() newlib	memcpy() newlib-nano	loop -O3 newlib
NUCLEO-G474RE	~17 µS	~6 µS	~5 µS	~6 µS	~32 µS	~6 µS
NUCLEO-F446RE	~19 µS	~7 µS	~6 µS	~7 µS	~40 µS	~7 µS
NUCLEO-F401RE	**~42 µS**	**~14 µS**	**~12 µS**	**~14 µS**	**~84 µS**	**~14 µS**
NUCLEO-F303RE	~128 µS	~36 µS	-	~36 µS	~194 µS	~36 µS
NUCLEO-F103RB	~160 µS	~42 µS	-	~34 µS	~248 µS	~34 µS
NUCLEO-F072RB	~134 µS	~38 µS	-	~40 µS	~254 µS	~40 µS
NUCLEO-L476RG	~88 µS	~20 µS	-	~20 µS	~92 µS	~20 µS
NUCLEO-L152RE	~252 µS	~64 µS	-	**~36 µS**	~380 µS	~36 µS
NUCLEO-L073RZ	~184 µS	~50 µS	-	~54 µS	~340 µS	~54 µS

Table 9.16: M2M transfer test results

The **Table 9.16** shows the results obtained for every Nucleo board. Let us focus on the the Nucleo-F401RE board. As you can see, the DMA M2M byte-aligned transfer takes ~**42µS**, while the DMA-M2M word-aligned transfer takes ~**14 µS**. This is a great speed-up, which proves that using the right DMA configuration can give us the best transfer performance, since we are moving 4 bytes at once for each DMA shoot. What about the *memcpy()*? As you can see from **Table 9.16**, it depends on the C library used. The GCC tool-chain we are using provides two C *run-time* libraries: one is named `newlib` and one `newlib-nano`. The first one is the most complete and speed-optimized of the two, while the second one is a reduced-size version. The `memcpy()` in the `newlib` library is designed to provide the fastest copy speed, at the expense of code size. It automatically detects word-aligned transfers, and it equals the DMA when doing word-aligned M2M transfers. So, it is much faster than DMA when doing byte-aligned M2M transfers and that is the reason why someone claims that `memcpy()` is always faster than the DMA. On the other hand, both Cortex-M core and the DMA need to access flash and SRAM memory using the same bus. So, there are no reasons why the MCU core should be faster than the DMA[19].
As you can see, the fastest transfer speed is achieved when the DMA stream/channel disables the internal FIFO buffer (~**12 µS**). It is important to remark that for STM32 MCUs with smaller flash memories the `newlib-nano` it is almost an unavoidable choice, unless the code can fit the flash space. But again, using the right DMA settings we can achieve the same performances of the speed-optimized version available in `newlib` library.

The last thing we have to analyze is the last column in **Table 9.16**. It shows how long it takes to do a memory transfer using a simple loop like the following one:

[19]Here I am clearly excluding some "privileged paths" between the Cortex-M core and SRAM. This is the role of the *Core-Coupled Memory* (CCM), a feature available in some STM32 MCUs and that we will explore better in Chapter 20.

```
...
GPIOC->ODR = 0x100;
for(int i = 0; i < 1000; i++)
  sramData[i] = flashData[i];
GPIOC->ODR = 0x0;
...
```

As you can see, with the maximum optimization level (-O3) it takes exactly the same time of `memcpy()`. Why does this happen?

```
...
  GPIOC->ODR = 0x100;
 8001968:       f44f 7380       mov.w   r3, #256        ; 0x100
 800196c:       6163            str     r3, [r4, #20]
 800196e:       4807            ldr     r0, [pc, #28]   ; (800198c <main+0x130>)
 8001970:       4907            ldr     r1, [pc, #28]   ; (8001990 <main+0x134>)
 8001972:       f44f 727a       mov.w   r2, #1000       ; 0x3e8
 8001976:       f000 f92d       bl      8001bd4 <memcpy>
  for(int i = 0; i < 1000; i++)
          sramData[i] = flashData[i];
  GPIOC->ODR = 0x0;
 800197a:       6165            str     r5, [r4, #20]
 ...
```

Looking at generated assembly code above you can see that the compiler automatically transforms the loop in a call to the `memcpy()` function. This clearly explains why they have the same performances.

Table 9.16 shows another interesting result. For an STM32F152RE MCU, the `memcpy()` in `newlib` is always twice faster than the DMA M2M. I do not know why this happens, but I have executed several tests and I can confirm the result.

Finally, other tests not reported here show that it is convenient to use DMA to do M2M transfers when the array has more than 30-50 elements, otherwise the DMA setup costs outweigh the benefits related to its usage. However, it is important to remark that the other advantage in using the DMA M2M transfer is that the CPU is free to accomplish other tasks while the DMA performs the transfer, even if its access to the bus slows the overall DMA performances.

How to switch to the `newlib` *run-time* library? This can be easily done in STM32CubeIDE, going in the project settings (**Project->Properties** menu), then going into **C/C++ Build->Settings** section and selecting the **MCU Settings** entry inside the **Tools Setting** section. To select the `newlib` library choose the entry **Standard C** in the **Runtime library** combo box. To select the `newlib-nano` choose the entry **Reduced C** (see **Figure 9.10**).

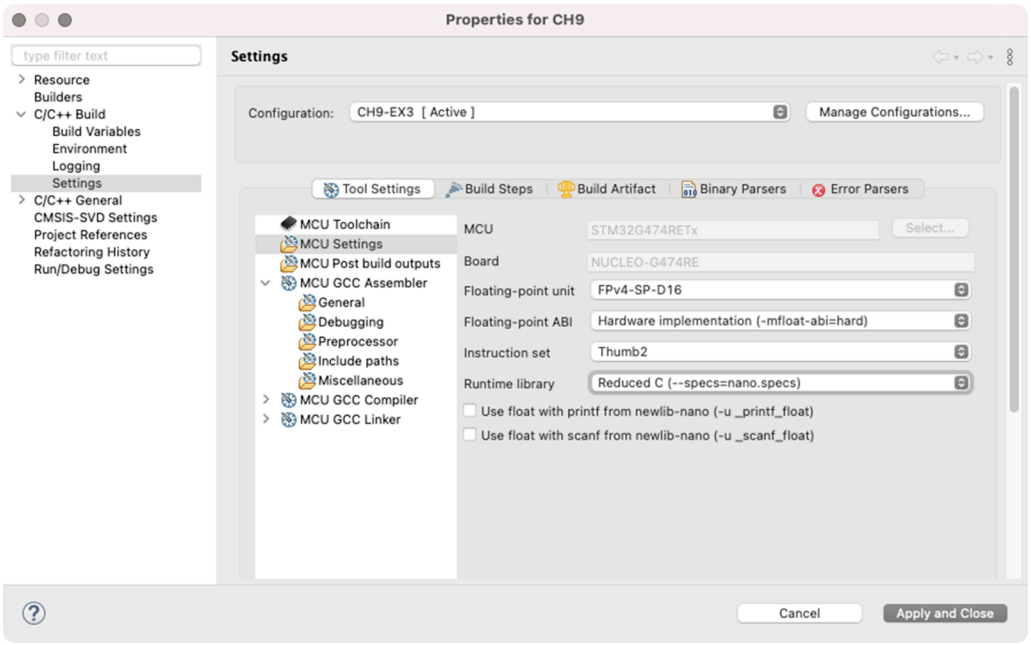


Figure 9.10: How to select `newlib`/`newlib-nano` *run-time* library

10. Clock Tree

Almost every digital circuit needs a way to synchronize its internal circuitry or to synchronize itself with other circuits. A clock is a device that generates periodic signals, and it is the most widespread form of *heartbeat* source in digital electronics.

The same clock signal, however, cannot be used to feed all components and peripherals provided by a modern microcontroller like STM32 ones. Moreover, power consumption is a critical aspect directly connected with the clock speed of a given peripheral. Having the ability to selectively disable or reduce the clock speed of some MCU parts allows to optimize the overall device power consumption. This requires that the clock is organized in a hierarchical structure, giving to the developer the possibility to choose different speeds and clock sources.

This chapter gives a brief introduction to the complex clock distribution network of an STM32 MCU. Its intent is to provide to the reader necessary tools to understand and manage the clock tree, showing the main functionalities of the `HAL_RCC` module. This chapter will be further completed with Chapter 19 dedicated to the power management.

10.1 Clock Distribution

The clock is a device that usually generates a square wave signal, with a 50% duty cycle, as the one shown in **Figure 10.1**[1].

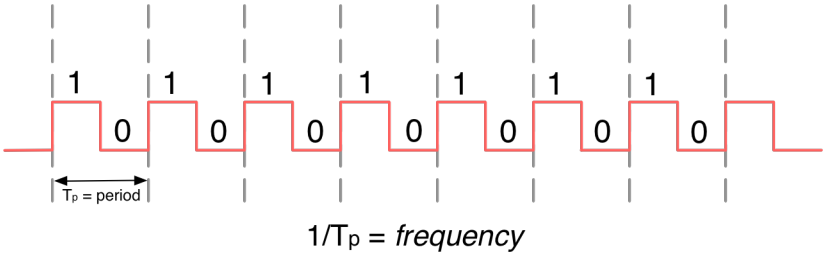


Figure 10.1: A typical clock signal with a 50% duty cycle

A clock signal oscillates between V_L and V_H voltage levels, which for STM32 microcontrollers are a fraction of the VDD supply voltage. The most fundamental parameter of a clock is the *frequency*, which indicates how many times it switches from V_L to V_H in a second. The frequency is expressed in Hertz.

[1] It is important to remark that the square wave represented in **Figure 10.1** is "ideal". The real square wave of a clock source has a trapezoidal form.

The majority of STM32 MCUs[2] can be *clocked* by two distinct clock sources alternatively: an internal RC oscillator[3] (named *High Speed Internal* (HSI)) or an external dedicated crystal oscillator[4] (named *High Speed External* (HSE)). There are several reasons to prefer an external crystal to the internal RC oscillator:

- An external crystal offers a higher precision compared to the internal RC network, which is rated of a 1% accuracy[5], especially when PCB operative temperatures are far from the ambient temperature of 25°C.
- Some peripherals, especially high-speed ones, can be clocked only by a dedicated external crystal running at a given frequency.

Together with the high-speed oscillator[6], another clock source can be used to bias the low-speed oscillator, which in turn can be clocked by an external crystal (named *Low Speed External* (LSE)) or the internal dedicated RC oscillator (named *Low Speed Internal* (LSI)). The low-speed oscillator is used to drive the *Real Time Clock* (RTC) and the *Independent Watchdog* (IWDT) peripheral.

The frequency of the high-speed oscillator does not establish the actual frequency neither of the Cortex-M core nor of the other peripherals. A complex distribution network, also called *clock tree*, is responsible for the propagation of the clock signal inside an STM32 MCU. Using several programmable *Phase-Locked Loops* (PLL) and prescalers, it is possible to increase/decrease the source frequency at needs (see **Figure 10.2**), depending on the performances we want to reach, the maximum speed for a given peripheral or bus and the overall global power consumption[7].

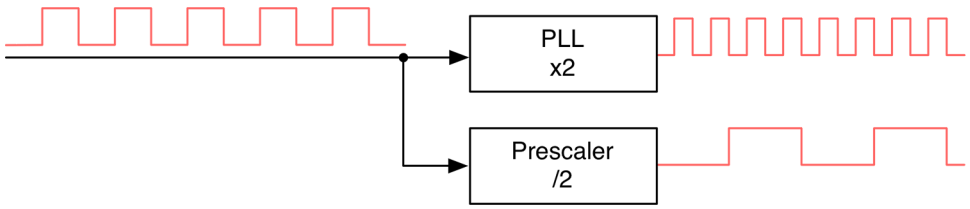


Figure 10.2: How the source clock signal frequency is increased/decreased using PLLs and prescalers

10.1.1 Overview of the STM32 Clock Tree

The clock tree of an STM32 MCU can have a really articulated structure. Even in "simpler" STM32F0 MCUs, the internal clock network can have up to four PLL/prescaler stages, and the *System Clock Multiplexer* (also known as *System Clock Switch* (SW)) can be fed by several alternate sources.

[2] There exist some STM32 MCUs, especially those ones with low pin count, that cannot be clocked by an external clock source.

[3] https://bit.ly/1TkDnUd

[4] https://bit.ly/20ymjJx

[5] A 1% accuracy may seem a good compromise, especially if you consider that you can save PCB space and the cost of a dedicated crystal, which is a device that has a non-negligible price. However, for time-constraint applications, 1% may be a huge shift. For example, a day is made of 86,400 seconds. An error equal to 1% means that in the worst case we can lose (or earn) up to 864 seconds, which is equal to 14,4 minutes! And things may worsen if temperature increases. This is the reason why it is mandatory to use an external low-speed crystal if you are going to use the RTC. However, a solution to increase this accuracy exists. More about this later.

[6] In this book we will refer to the *high-speed oscillator* as an "abstract" clock source, which has two mutually exclusive "concrete" sources: the HSE or the HSI oscillator. The same applies to the *low-speed oscillator*

[7] Remember that the power consumption of an MCU is about linear with its frequency. The higher is the frequency, the more power it consumes.

Nucleo P/N	High-speed oscillator	AHB bus speed	APB1 peripheral clocks	APB1 timer clocks	APB2 peripheral clocks	APB2 timer clocks
NUCLEO-G474RE	HSE/HSI	170MHz	170MHz	170MHz	170MHz	170MHz
NUCLEO-F446RE	HSE/HSI	180MHz	45MHz	90MHz	90MHz	180MHz
NUCLEO-F401RE	HSE/HSI	84MHz	42MHz	84MHz	84MHz	84MHz
NUCLEO-F303RE	HSE/HSI	72MHz	36MHz	72MHz	72MHz	72MHz
NUCLEO-F103RB	HSE	72MHz	36MHz	72MHz	72MHz	72MHz
	HSI	64MHz	32MHz	64MHz	64MHz	64MHz
NUCLEO-F072RB	HSE/HSI	48MHz	48MHz	48MHz	-	-
NUCLEO-L476RG	HSE/HSI	80MHz	80MHz	80MHz	80MHz	80MHz
NUCLEO-L152RE	HSE/HSI	32MHz	32MHz	32MHz	32MHz	32MHz
NUCLEO-L073RZ	HSE/HSI	32MHz	32MHz	32MHz	32MHz	32MHz

Table 10.1: The maximum clock speeds for AHB, APB1 and APB2 buses of the MCUs equipping Nucleo boards used in this text

Moreover, explaining in depth the clock tree of every STM32 family is a complex task, which also requires we focus our attention on a specific part number. This because the clock tree structure is affected mainly by the following key aspects:

- **The STM32 main family of the microcontroller**. For example, all STM32F0 MCUs provide just one peripheral bus (APB1), which can be clocked at the same Cortex-M core maximum frequency. Other STM32 microcontrollers usually provide two peripheral buses, and only one of these (APB2) can reach the maximum CPU clock speed. Instead, none of the peripheral buses available in an STM32F7 microcontroller can reach the maximum core frequency[8], while both APB1 and ABP2 in an STM32G4 MCU can reach the maximum core clock speed. **Table 10.1** reports the maximum clock speed for AHB, APB1 and APB2 buses (with related timers clock speed) of the MCUs equipping the Nucleo boards used as test board in this book: you can note that for the "old" STM32F103 it is possible to reach the maximum clock speed just using an external HSE oscillator.
- **The type and number of peripherals provided by the MCU**. The complexity of the clock tree increases with the number of available peripherals. Moreover, some peripherals require dedicated clock sources and speeds, which impact on the number of PLL stages.
- **The sales type and package of the MCU**, which determines the effective type and number of provided peripherals.

Even restricting our focus only on the nine MCUs equipping the Nucleo boards, this would require a long and tedious work, which involve a deep knowledge of all peripherals implemented by the given MCU. For these reasons, we will give a quick overview to the STM32 clock tree, leaving to the reader the responsibility to deepen the particular MCU he is considering. Moreover, as we will see in a while, thanks to CubeMX it is possible to abstract from the specific clock tree implementation,

[8]Except for timers on the APB2 bus.

unless we need to deal with specific PLL configurations for performance and power management reasons.

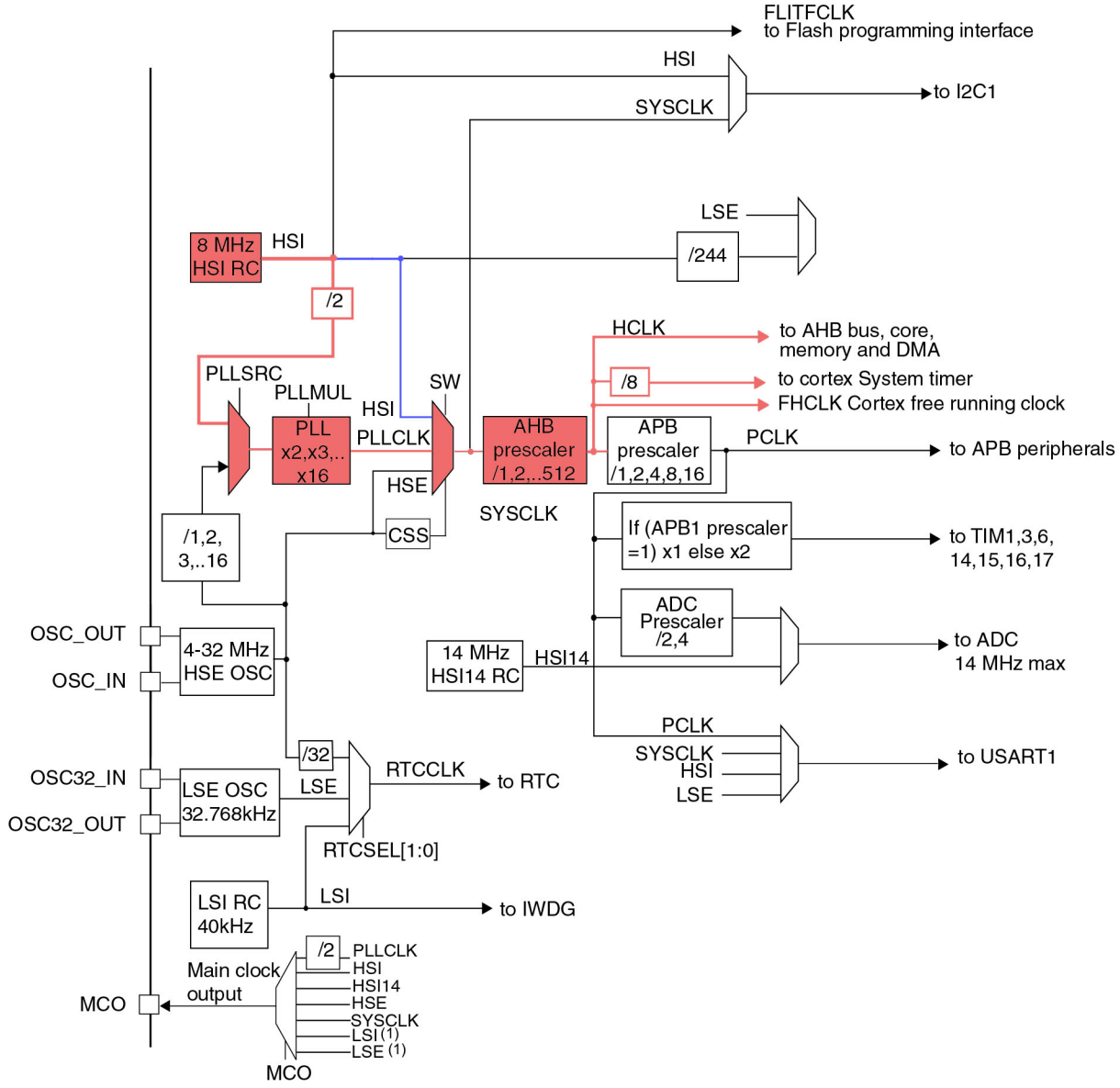


Figure 10.3: The clock tree of an STM32F030R8 MCU

Figure 10.3 shows the clock tree of one of the simplest STM32 microcontrollers: the STM32F030R8. It is extracted from the related reference manual[9] provided by ST. For a lot of novices of the STM32 platform that figure is completely meaningless and quite hard to decode, especially if they are also new to embedded microcontrollers. The most relevant path has been outlined in red: the one that goes from the HSI oscillator to the Cortex-M0 core, AHB bus and DMA. This is the path we have "used" since here silently, without dealing too much with its possible configurations. Let us introduce

[9] https://bit.ly/1GfS3iC

the most relevant parts of that path.

The path starts from the internal 8MHz oscillator. As said before, it is an RC oscillator factory-calibrated by ST for 1% accuracy at an ambient temperature of 25 °C. The HSI clock can then be used to feed the *System Clock Switch* (SW) as is (path highlighted in blue in **Figure 10.3**) or it can be used to feed the PLL multiplier after it has been divided by two thanks to an intermediate prescaler[10]. The main PLL so can multiply the 4MHz clock up to 12 times to obtain the maximum *System Clock Frequency* (SYSCLK) of 48MHz. The SYSCLK source can be used to feed the I2C1 peripheral (in alternative to the HSI) and another intermediate prescaler, the AHB prescaler, which can be used to lower the *High (speed) Clock* (HCLK), which in turn biases the AHB bus, the core and the *SysTimer*.



Why So Many Intermediate PLL/Prescaler Stages?

As said before, the clock speed determines the overall performances, but it also affects the total power consumption of the MCU. Having the capability to selectively turn ON/OFF - or reduce the clock speed - of some parts of the MCU gives the possibility to reduce the power consumption according to the effective computing power needed. As we will see in Chapter 19, L0/1/4/5 MCUs introduce even more PLL/prescaler stages to offer to developers more control on the overall MCU consumption. Together with a dedicated hardware design, this allows to create battery-powered devices that can be run even for years using the same battery.

The clock tree configuration is performed through a dedicated peripheral[11] named *Reset and Clock Control* (RCC), and it is a process essentially composed by three steps:

1. The high-speed oscillator source is selected (*HSI* or *HSE*) and properly configured, if the *HSE* is used.[12]
2. If we want to feed the *SYSCLK* with a frequency higher than the one provided by the high-speed oscillator, then we need to configure the *main PLL* (which provides the *PLLCLK* signal). Otherwise, we can skip this step.
3. The *System Clock Switch* (SW) is configured choosing the right clock source (*HSI*, *HSE*, or *PLLCLK*). Then we select the right AHB, APB1 and APB2 (if available) prescaler settings to reach the wanted frequency of the *High-speed clock* (HCLK - that is the one that feeds the core, DMAs and AHB bus), and the frequencies of *Advanced Peripheral Bus 1* (APB1) and APB2 (if available) buses.

Knowing the admissible values for PLLs and prescalers can be a nightmare, especially for more complex STM32 MCUs. Only some combinations are valid for a given STM32 microcontroller, and their improper configuration could potentially damage the MCU or at least cause malfunctions

[10] A prescaler is an "electronic counter" used to reduce high frequencies. In this case, the "/2" prescaler reduces the main 8MHz frequency to 4MHz.

[11] Sometimes, ST defines in its documents the RCC as "peripheral". Sometimes no. I am not sure that if it is properly a peripheral, but I will define it in the same way ST does. Sometimes.

[12] In STM32L0/1/4 MCUs, the *SYSCLK* can be also fed by another dedicated and low-power clock source, named MSI. We will talk about this clock source next.

(a wrong clock configuration could lead to abnormal behavior, strange and unpredictable resets and so on). Luckily for us, the STM32 engineers have provided a great tool to simplify the clock configuration: CubeMX.

10.1.1.1 The Multispeed Internal RC Oscillator in STM32L/U Families

The clock source and its distribution network have a non-negligible impact on the overall power consumption of the MCU. If we need a *SYSCLK* frequency higher or lower than the internal HSI clock source (which is 8MHz for the most of STM32 MCUs and 16MHz for some others), we have to increase/reduce it by using the *PLL Source Mux* and intermediate prescalers. Unfortunately, these components consume energy, and this can have a dramatic impact on battery-powered devices.

Clock Source	Frequency	Power consumpion	Accuracy	Settling time
MSI (default on Reset)	0.1-48MHz (4MHz default)	**0.6~155µA**	±1% @ 25°C ±3% @ 0-85°C	10~2.5µs
MSI (as clock source for PLL MUX)	0.1-48MHz		60ps (cycle to cycle jitter)	252.5µs
HSI	16MHz	155µA	±1% @ 25°C	3.8µs
HSE	4-48MHz	**~440µA** (8MHz, 10pF)	*(depending on external crystal)*	2ms
PLL MUX	2-80MHz	520µA (@344MHz VCO)	N/A	15µs (2MHz input)
LSI	32KHz	0.11µA	±10% @ 25°C	125µs
LSE	32.768kHz	~0.25µA	*(depending on external crystal)*	**~2s**

Table 10.2: A comparison between clock sources in an STM32L476 MCU

STM32L/U MCUs are explicitly designed for low-power applications, and they address this specific issue by supplying a dedicated internal clock source, named *MultiSpeed Internal* (MSI) RC oscillator. MSI is a low-power RC oscillator, with a ±1%@25°C factory pre-calibrated accuracy, which can increase up to ±3% in the 0-85°C range. The main characteristic of the MSI is that it supplies up to twelve different frequencies, without adding any external component. For example, the MSI in an STM32L476 provides an internal clock source ranging from 100kHz up to 48MHz. The MSI clock is used as *SYSCLK* after restart from Reset, wakeup from Standby and Shutdown low-power modes. After restart from Reset, the MSI frequency is set to its default (for example, the default MSI frequency in an STM32L476 is 4MHz). **Table 10.2** summarizes the most relevant characteristics of all possible clock sources in an STM32L476 MCU. As you can see, the best power consumption is achieved while the MCU is clocked by the MSI (without using the *PLL Multiplexer*). Moreover, this clock source guarantees the shortest startup time, if compared with the HSI. It is interesting to see that up to two seconds are required to stabilize the LSE clock: if startup speed is really important for your application, then using a separated thread[13] to start the LSE is an option to consider.

[13] This clearly implies the usage of an RTOS. We will study this matter in a later chapter.

In addition to the advantages related to low-power, when the MSI is used as source for the *PLL Source Mux* it provides a very accurate clock source which can be used by the USB OTG FS device without using an external dedicated crystal, while feeding the main PLL to run the system at the maximum speed.

10.1.2 Configuring Clock Tree Using CubeMX

We have already encountered in Chapter 4 the CubeMX *Clock Configuration* view. Now it is the right time to see how it works. The **Figure 10.4** shows the clock tree of the same F0 MCU seen so far. As you can see, thanks to the more room available on the screen, the distribution network looks less cumbersome.

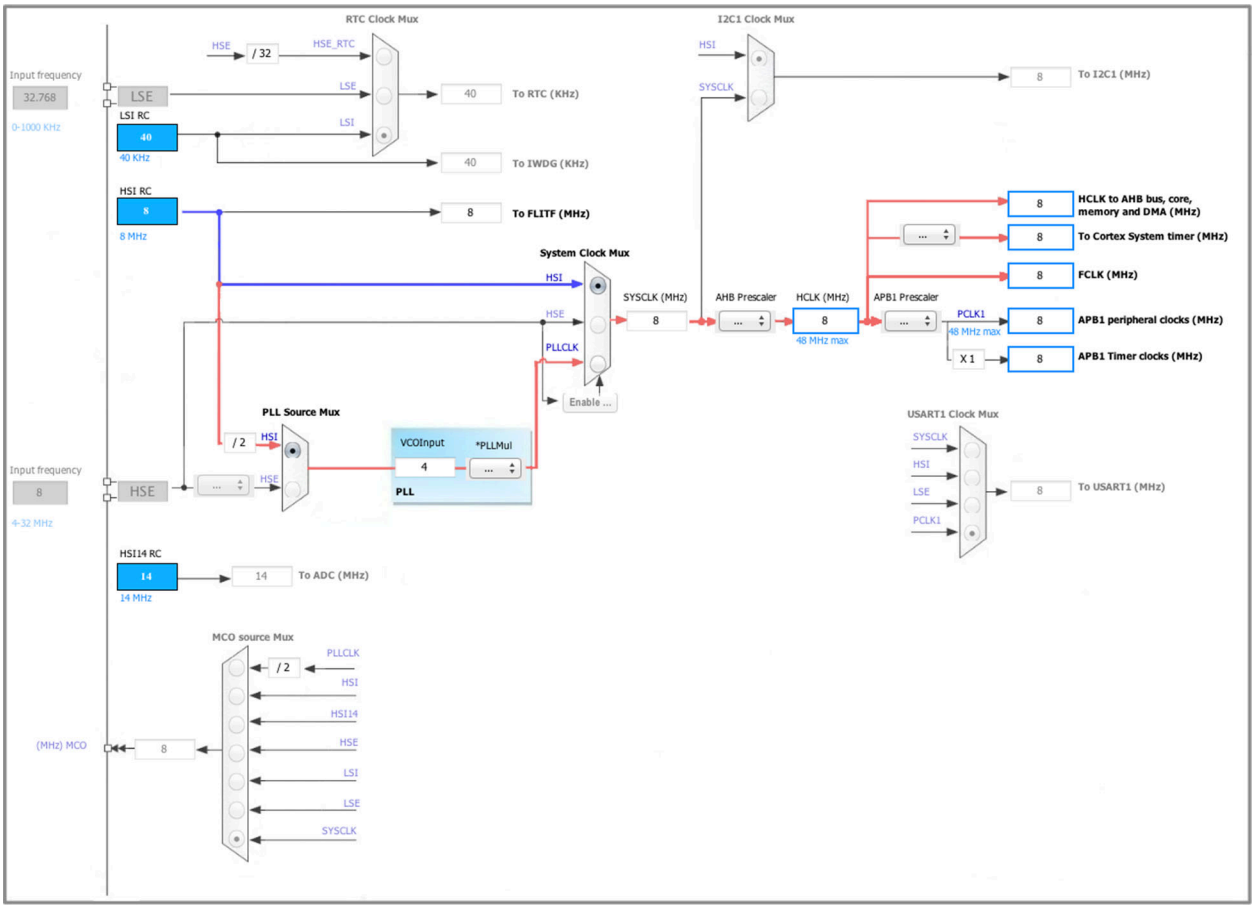


Figure 10.4: How the clock tree of an STM32F030R8 MCU is represented in CubeMX

Even in this case, the most relevant paths of the clock tree have been highlighted in red and blue. This should simplify the comparison with the **Figure 10.3**. When a new project is created, by default CubeMX chooses the HSI oscillator as default clock source. HSI is also chosen as default clock source for the *System Clock Mux*[14] (path in blue), as shown in **Figure 10.4**. This means that, for the MCU we are considering here, the Cortex-M core frequency will be equal to 8MHz.

[14]The *System Clock Mux* is the *System Clock Switch* (SW) seen in **Figure 10.3**

CubeMX also advises us about two things: the maximum frequency for the *High (speed) Clock* (HCLK) and the APB1 bus is equal to 48MHz in this MCU (labels in blue). To increase the CPU core frequency, we first need to select the PLLCLK as the source clock for the *System Clock Switch* and then choose the right PLL multiplier factor. However, CubeMX offers a quick way to do this: you can simply write "48" inside the HCLK field and hit the enter key. CubeMX will automatically arrange the settings, choosing the right clock tree path (the red one in **Figure 10.4**)

If your board relies on an external HSE/LSE crystal, you have to enable it in the RCC peripheral before you can use it as main clock source for the corresponding oscillator (we will see in a while how to do this step-by-step). Once the external oscillator is enabled, it is possible to specify its frequency (inside the blue box labeled "input frequency") and to configure the main PLL to achieve the desired SYSCLK speed (see **Figure 10.5**). Otherwise, the external oscillator input frequency can be used directly as source clock for the *System Clock Switch*.

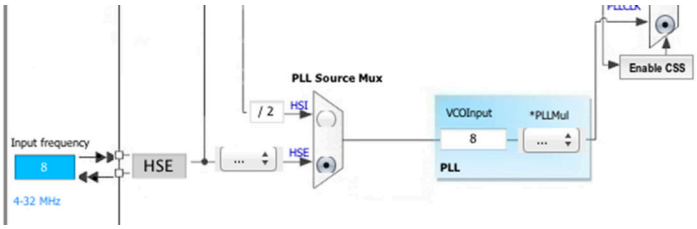


Figure 10.5: CubeMX allow to select the HSE oscillator once it is enabled using the RCC peripheral

We need to configure the RCC peripheral accordingly to enable an external clock source. This can be done from the *Pinout* view in CubeMX, as shown in **Figure 10.6**.

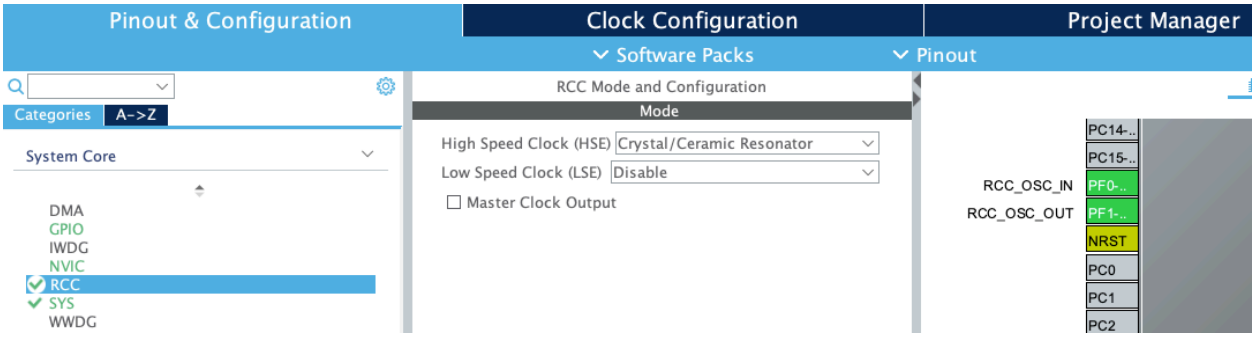


Figure 10.6: The configuration options provided by the RCC peripheral

For both HSE and LSE oscillators, CubeMX offers three configuration options:

- **Disable**: the external oscillator is not available/used, and the corresponding internal oscillator is used.
- **Crystal/Ceramic Resonator**: an external crystal/ceramic resonator is used, and the corresponding main frequency is derived from it. This implies that RCC_OSC_IN and RCC_OSC_OUT pins are used to interface the HSE, and the corresponding signal I/Os are unavailable for other usages (if we are using an external low-speed crystal, then the corresponding RCC_OSC32_IN and RCC_OSC32_OUT I/Os are used too).

- **BYPASS Clock Source**: an external clock source is used. The clock source is generated by another **active** device. This means that the RCC_OSC_OUT is leaved unused, and it is possible to use it as regular GPIO. In almost all development board from ST (included the Nucleo ones) the *Master Clock Output* (MCO) pin of the ST-LINK interface is used as external clock source for the target STM32 MCU. Enabling this option allows to use the ST-LINK MCO as HSE.

The RCC peripheral also allows to enable the *Master Clock Output* (MCO), which is a pin that can be used to clock another external device, allowing to save on the external crystal for this other IC. Once the MCO is enabled, it is possible to choose its clock source using the *Clock Configuration* view, as shown in **Figure 10.7**.

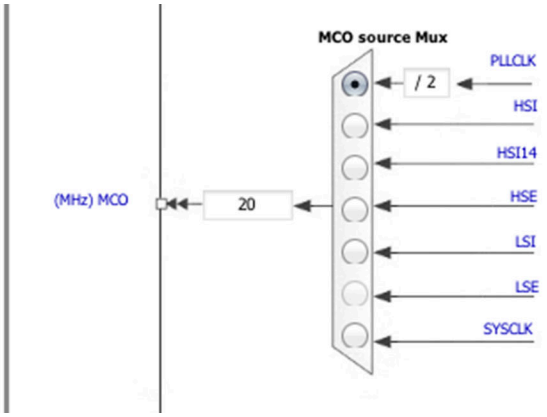


Figure 10.7: How to select the clock source for the MCO pin

10.1.3 Clock Source Options in Nucleo Boards

The Nucleo-64 development boards offer several alternatives for the clock sources. However, there is a strong distinction between the early Nucleo-64 boards and the more recent ones that are based on different board layout and the more recent ST-LINK v3 debugger.

To understand which Nucleo-64 release you have, it is important to check the board release number. The early releases were based on the MB1136 design. The newer ones are based on the MB1367 one. This information is usually reported on the PCB (by using a label on the back on older Nucleo), as shown in **Figure 10.8**.

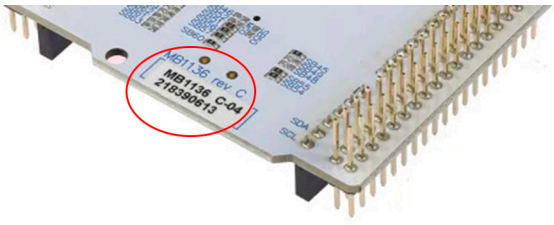


Figure 10.8: The Nucleo-64 PCB revision number

10.1.3.1 Clock Source in Nucleo-64 rev. MB1136 (older ones with ST-LINK V2.1)

10.1.3.1.1 OSC Clock Supply

There are four ways to configure the pins corresponding to external high-speed clock external high-speed clock (HSE):

- **MCO from ST-LINK**: MCO output of ST-LINK MCU is used as input clock. This frequency cannot be changed, it is fixed at 8 MHz and connected to PF0/PD0/PH0-OSC_IN of target STM32 MCU.
  The following configuration is needed:
  - SB55 OFF
  - SB16 and SB50 ON
  - R35 and R37 removed
- **HSE oscillator on-board from X3 crystal (not provided)**: for typical frequencies and its capacitors and resistors, refer to STM32 microcontroller datasheet. Please refer to the AN2867 for oscillator design guide for STM32 microcontrollers.
  The following configuration is needed:
  - SB54 and SB55 OFF
  - R35 and R37 soldered
  - C33 and C34 soldered
  - SB16 and SB50 OFF
- **Oscillator from external PF0/PD0/PH0**: from an external oscillator through pin 29 of the CN7 connector.
  The following configuration is needed:
  - SB55 ON
  - SB50 OFF
  - R35 and R37 removed
- **HSE not used**: PF0/PD0/PH1 and PF1/PD1/PH1 are used as GPIO instead of Clock
  The following configuration is needed:
  - SB54 and SB55 ON
  - SB16 and SB50 (MCO) OFF
  - R35 and R37 removed

There are two possible default configurations of the HSE pins depending on the version of NUCLEO board hardware. The board version MB1136 C-01/02/03 is mentioned on sticker placed on bottom side of the PCB.

- The board marking MB1136 C-01 corresponds to a board, configured for HSE not used.
- The board marking MB1136 C-02 (or higher) corresponds to a board, configured to use STLINK MCO as clock input.



Read Carefully

For Nucleo-L476RG the ST-LINK MCO output is not connected to OSCIN, to reduce power consumption in low power mode. Consequently, the HSE in a Nucleo-L476RG cannot be used unless an external crystal is mounted on X3 pad, as described before.

10.1.3.1.2 OSC 32kHz Clock Supply

There are three ways to configure the pins corresponding to low-speed clock (LSE):

- On-board oscillator: X2 crystal. Please refer to the AN2867[15] for oscillator design guide for STM32 microcontrollers.
- Oscillator from external PC14: from external oscillator through the pin 25 of CN7 connector. The following configuration is needed:
  - SB48 and SB49 ON
  - R34 and R36 removed
- LSE not used: PC14 and PC15 are used as GPIOs instead of low-speed Clock. The following configuration is needed:
  - SB48 and SB49 ON
  - R34 and R36 removed

There are two possible default configurations of the LSE depending on the version of NUCLEO board hardware. The board version MB1136 C-01/02/03 is mentioned on sticker placed on bottom side of the PCB.

- The board marking MB1136 C-01 corresponds to a board configured as LSE not used.
- The board marking MB1136 C-02 (or higher) corresponds to a board configured with onboard 32kHz oscillator.
- The board marking MB1136 C-03 (or higher) corresponds to a board using new LSE crystal (ABS25) and C26, C31 & C32 value update.



Read Carefully

All Nucleo boards with a release version equal to MB1136 C-02 have a severe issue with the values of the dumping resistor R34, R36 and with the capacitors C26, C31 & C32. This issue prevents the LSE to start correctly.

[15] https://bit.ly/2WbvHgS

10.1.3.2 Clock Source in Nucleo-64 rev. MB1367 (newer ones with ST-LINK v3)

10.1.3.2.1 OSC Clock Supply

There are four ways to configure the pins corresponding to external high-speed clock external high-speed clock (HSE):

- **MCO from ST-LINK**: MCO output of ST-LINK is used as input clock. This frequency cannot be changed, it is fixed at 8 MHz or 8.33MHz[16] and connected to the PF0-OSC_IN of the STM32 microcontroller. The configuration must be:
  - SB27 ON
  - SB25 and SB26 OFF
  - SB24 and SB28 OFF
- **HSE on-board oscillator from X3 crystal (default)**: for typical frequencies and its capacitors and resistors, refer to the STM32 microcontroller datasheet and to the Oscillator design guide for STM8S, STM8A and STM32 microcontrollers Application note (AN2867[17]) for the oscillator design guide. It is recommended to use a crystal with these characteristics: 24 MHz, 6 pF load capacitance, 20 ppm. The configuration must be:
  - SB25 and SB26 ON
  - SB24 and SB28 OFF
  - SB27 OFF
  - C56 and C59 soldered with 6.8 pF capacitors
- **Oscillator from external PF0**: from an external oscillator through the pin 29 of the CN7 connector. The configuration must be:
  - SB28 ON
  - SB24 OFF
  - SB25 and SB26 OFF
  - SB27 OFF
- **HSE not used**: PF0 and PF1 are used as GPIOs instead of as clock. The configuration must be:
  - SB24 and SB28 ON
  - SB27 OFF
  - SB25 and SB26 OFF

[16] By default, ST-LINK v3 is configured so that the MCO is clocked at HSI/2, which in an STM32F723IEK6 corresponds to 16MHz/2 = 8MHz. Starting from STLinkUpgrade 3.3.7 it is possible to configure ST-LINK v3 firmware so that MCO is clocked by HSE/3, which corresponds to 25MHz/3 = 8.33MHz. This solution provides a better clock accuracy instead of the internal HSI trimmed at 1%. In case of a switch to 8.33 MHz, the target application must be updated accordingly: a factor ×(24/25) must be applied in the PLL configuration in order to keep the clock tree unchanged behind (for instance, if PLLN = 400, set PLLN = 384). Moreover, the `HSE_VALUE` constant commonly defined into `stm32xxx_hal_conf.h` must be changed to value `8333333`. For more information refer to the RN0093.

[17] https://bit.ly/2WbvHgS

10.1.3.2.2 OSC 32kHz Clock Supply

There are three ways to configure the pins corresponding to low-speed clock (LSE):

- **On-board oscillator (default)**: X2 crystal. Refer to the Oscillator design guide for STM8S, STM8A and STM32 microcontrollers application note (AN2867[18]). It is recommended to use a crystal with these characteristics: 32.768 kHz, 6 pF load capacitance, 20 ppm. The configuration must be:
  - SB30 and SB31 ON
  - SB29 and SB32 OFF
- **Oscillator from external PC14**: from external oscillator through the pin 25 of CN7 connector. The configuration must be:
  - SB29 and SB32 ON
  - SB30 and SB31 OFF
- **LSE not used**: PC14 and PC15 are used as GPIOs instead of low-speed clock. The configuration must be:
  - SB29 and SB32 ON
  - SB30 and SB31 OFF

10.2 Overview of the `HAL_RCC` Module

So far we have seen that the *Reset and Clock Control* (RCC) peripheral is responsible of the configuration for the whole clock tree of an STM32 MCU. The `HAL_RCC` module contains the corresponding descriptors and routines of the CubeHAL to abstract from the specific RCC implementation. However, the actual implementation of this module inevitably reflects the peculiarities of the clock tree in a given STM32-series and part number. Deepening this module, as we have done for other HAL modules, is outside the scope of this book. It would require we keep track of too many differences among the several STM32 microcontrollers. So, we will now give a brief overview to its main features and to the steps involved during the configuration of the clock tree.

The most relevant C `struct` to configure the clock tree are `RCC_OscInitTypeDef` and `RCC_ClkInitTypeDef`. The first one is used to configure the RCC internal/external oscillator sources (HSE, HSI, LSE, LSI), plus some additional clock sources if provided by the MCU. For example, some STM32 MCUs from the F0 series (STM32F07x, STM32F0x2 and STM32F09x ones) provide USB 2.0 support, in addition to an internal dedicated and factory-calibrated high-speed oscillator running at 48MHz to bias the USB peripheral. If this is the case, the `RCC_OscInitTypeDef struct` is also used to configure those additional clock sources. The `RCC_OscInitTypeDef struct` also has a field that is instance of the `RCC_PLLInitTypeDef struct`, which configures the main PLL used to increase the speed of the source clock. It reflects the hardware structure of the main PLL and can be composed by several fields depending on the STM32 series (in STM32F2/4/7 MCUs it can have a quite complex structure).

[18] https://bit.ly/2WbvHgS

The `RCC_ClkInitTypeDef struct`, instead, is used to configure the source clock for the *System Clock Switch* (SWCLK), for the AHB bus and the APB1/2 buses.

CubeMX is designed to generate the right code initialization for the clock tree of our MCU. All the necessary code is packed inside the `SystemClock_Config()` routine, which we have encountered in the projects generated until now. For example, the following implementation of the `SystemClock_Config()` reflects the clock tree configuration for an STM32F030R8 MCU running at 48MHz:

```
1  void SystemClock_Config(void) {
2    RCC_OscInitTypeDef RCC_OscInitStruct;
3    RCC_ClkInitTypeDef RCC_ClkInitStruct;
4
5    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
6    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
7    RCC_OscInitStruct.HSICalibrationValue = 16;
8    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
9    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
10   RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL12;
11   RCC_OscInitStruct.PLL.PREDIV = RCC_PREDIV_DIV1;
12   HAL_RCC_OscConfig(&RCC_OscInitStruct);
13
14   RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_SYSCLK;
15   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
16   RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
17   RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
18   HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1);
19
20   HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);
21
22   HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
23
24   /* SysTick_IRQn interrupt configuration */
25   HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
26 }
```

Lines [5:12] select the HSI as source oscillator and enable the main PLL, setting the HSI as its clock source through the PLL multiplexer. The clock frequency is then increased by twelve times (settings the `PLLMUL` field). Lines [14:18] set the SYSCLK frequency. The PLLCLK is selected as clock source (line 15). In the same way, the SYSCLK frequency is selected as source for the AHB bus, and the same HCLK frequency (`RCC_HCLK_DIV1`) as source for the APB1 bus. The other lines of code set the *SysTick* timer, a special timer available in the Cortex-M core used to synchronize some internal HAL activities (or to drive the scheduler of an RTOS, as we will see in Chapter 23). The HAL is based on the convention that *SysTick* timer generates an interrupt ever 1ms. Since we are configuring the *SysTick* clock so that it runs at the maximum core frequency of 48MHz (which means that the SYSCLK performs 48.000.000 clock cycles every second), we can set the *SysTick* timer so that it

generates an interrupt every 48.000.000 cycles/1000ms = 48.000 clock cycles [19].

10.2.1 Compute the Clock Frequency at Run-Time

Sometimes it is important to know how fast is running the CPU core. If our firmware is designed so that it always runs at an established frequency, we can easily hardcode that value in the firmware using a symbolic constant. However, this is always a poor programming style, and it is totally inapplicable if we manage the CPU frequency dynamically. The CubeHAL provides a function that can be used to compute the SYSCLK frequency: the `HAL_RCC_GetSysClockFreq()`[20]. However, this function must be handled with special care. Let us see why.

The `HAL_RCC_GetSysClockFreq()` does not return the real SYSCLK frequency (it could never do this in a reliable way without having a known and precise external reference), but it bases the result on the following algorithm:

- if SYSCLK source is the HSI oscillator, then returns the value based on the `HSI_VALUE` macro;
- if SYSCLK source is the HSE oscillator, then returns the value based on the `HSE_VALUE` macro;
- if SYSCLK source is the PLLCLK, then returns a value based on HSI_VALUE/HSE_VALUE multiplied by the PLL factor, according to the specific STM32 MCU implementation.

`HSI_VALUE` and `HSE_VALUE` macros are defined inside the **stm32xxx_hal_conf.h** file, and they are **hardcoded** values. The `HSI_VALUE` is defined by ST during chip design, and we can trust the value of the corresponding macro (except for that 1% of accuracy). Instead, if we are using an external oscillator as HSE source, we must provide the actual value for the `HSE_VALUE` macro, otherwise the value returned by the `HAL_RCC_GetSysClockFreq()` function is wrong[21]. And this also affects the tick frequency (that is, how long it takes to generate the timer interrupt) of the *SysTick* timer.

We can also retrieve the core frequency by using the `SystemCoreClock` CMSIS global variable.



Read Carefully

If we decide to manipulate the clock tree configuration by hand without using CubeHAL routines, we have to remember that every time we change the SYSCLK frequency, we need to call the CMSIS function `SystemCoreClockUpdate()`, otherwise some CMSIS routines may give wrong results. This function is automatically called for us by the `HAL_RCC_ClockConfig()` routine.

[19] As we will see in the next chapter, a timer is *free counter* module, that is a device that counts from 0 to a given value at every clock cycle. Take note that, for the sake of completeness, the *SysTick* timer is a 24-bit *downcounter timer*, that is it counts from the configured maximum value (48.000 in our case) down to zero, and then automatically restarts again. The source clock of a timer establishes how fast this timer counts. Since here we are specifying that the clock source for the *SysTick* timer is the HCLK (line 22), then the counter will reach zero every 1ms.

[20] Pay attention that the Cortex-M core is not clocked by the SYSCLK frequency, but by the HCLK frequency, which could be lowered by the AHB prescaler. So, to recap, the core frequency is equal to `HAL_RCC_GetSysClockFreq()`/AHB-prescaler.

[21] The `HAL_RCC_GetSysClockFreq()` is defined to return an `uint32_t`. This means that it could return wrong results with fractional values for the HSE oscillator.

10.2.2 Enabling the *Master Clock Output*

As said before, depending on the IC package used, STM32 MCUs allow to route the clock signal to one or two output I/Os, called *Master Clock Output* (MCO). This is performed by using the function:

```
void HAL_RCC_MCOConfig(uint32_t RCC_MCOx, uint32_t RCC_MCOSource, uint32_t RCC_MCODiv);
```

For example, to route the PLLCLK to MCO1 pin in an STM32F401RE MCU (which corresponds to PA8 pin), we must invoke the above function in the following way:

```
HAL_RCC_MCOConfig(RCC_MCO1, RCC_MCO1SOURCE_PLLCLK, RCC_MCODIV_1);
```



Read Carefully

Please, take note that when configuring the MCO pin as output GPIO, its speed (that is, the *slew rate*) affects the quality of the output clock. Moreover, for higher clock frequencies the *compensation cell* must be enabled in the following way:

```
HAL_EnableCompensationCell();
```

Refer to the datasheet of your MCU for more about this.

10.2.3 Enabling the *Clock Security System*

The *Clock Security System* (CSS) is a feature of the RCC peripheral used to detect malfunctions of the external HSE. The CSS is an important feature in some critical applications, where a malfunction of the HSE could cause injuries to the user. Its importance is proven by the fact that the detection of a failure is noticed through the NMI exception, a Cortex-M exception that cannot be disabled.

When the failure of HSE is detected, the MCU automatically switch to the HSI clock, which is selected as source for the SYSCLK clock. So, if a higher core frequency is needed, we need to perform proper initializations inside the NMI exception handler.

To enable the CSS we use the `HAL_RCC_EnableCSS()` routine, and we need to define the handler for the NMI exception in the following way[22]:

```
void NMI_Handler(void) {
  HAL_RCC_NMI_IRQHandler();
}
```

The right way to catch the failure of the HSE clock is by defining the callback:

[22]There is no need to enable the NMI exception, because it is automatically enabled, and it cannot be disabled.

```
void HAL_RCC_CSSCallback(void) {
        //Catch the HSE failure and take proper actions
}
```

10.3 HSI Calibration

We have left uncommented one line of code in the `SystemClock_Config()` routine seen before: the instruction at line 7. It used to perform a fine-tune calibration of the HSI oscillator. But what exactly it does?

As said before, the frequency of the internal RC oscillators may vary from one chip to another due to manufacturing process variations. For this reason, HSI oscillators are factory-calibrated by ST to have a 1% accuracy at room temperature. After a reset, the factory calibration value is automatically loaded in the second byte (HSICAL) of the *RCC configuration register* (RCC_CR) (the **Figure 10.9** shows the implementation of this register in an STM32F401RE[23]).

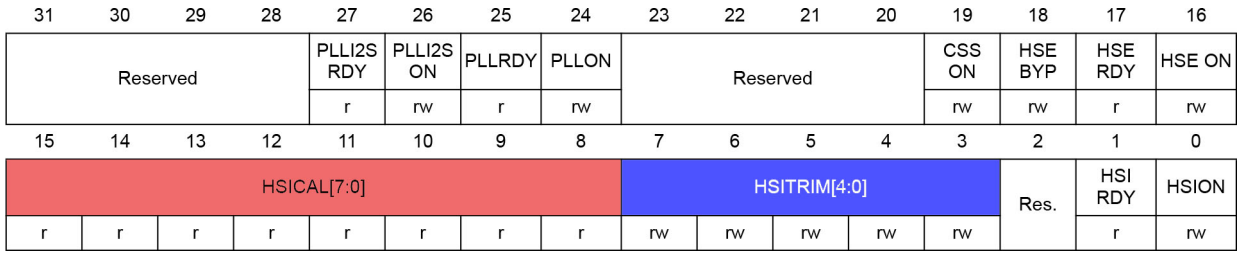


Figure 10.9: The RCC_CR register in an STM32F401RE MCU

The frequency of the internal RC oscillator can be fine-tuned to achieve better accuracy with wider temperature and supply voltage ranges. The trimming bits are used for this purpose. Five trimming bits RCC_CR->HSITRIM[4:0] are used for fine-tuning. The default trimming value is 16. An increase/decrease in this trimming value causes an increase/decrease in HSI frequency. The HSI oscillator is fine-tuned in steps of 0.5% of the HSI clock speed:

- Writing a trimming value in the range of 17 to 31 increases the HSI frequency.
- Writing a trimming value in the range of 0 to 15 decreases the HSI frequency.
- Writing a trimming value equal to 16 causes the HSI frequency to keep its default value.

The HSI can be calibrated using the following procedure:

1. set the internal high-speed RC oscillator system clock;
2. measure the internal RC oscillator frequency for each trimming value;
3. compute the frequency error for each trimming value (according to a known reference);

[23]The figure is taken from the RM0368 application note from ST (https://bit.ly/1Kq3SoE).

4. finally, set the trimming bits with the optimum value (corresponding to the lowest frequency error).

The internal oscillator frequency is not measured directly but it is computed from the number of clock pulses counted using a timer compared with the typical value. To do this, a very accurate reference frequency must be available such as the LSE frequency provided by the external 32.768 kHz crystal or the 50 Hz/60 Hz of the mains.

ST provides several application notes describing better this procedure (for example, the AN4067[24] is about the calibration procedure in the STM32F0 family). Please, refer to those documents for more information.

[24] https://bit.ly/1R8kEbf

11. Timers

Embedded devices perform some activities on a time basis. For really simple and inaccurate delays a busy loop could carry out the task, but using the CPU core to perform time-related activities is never a smart solution. For this reason, all microcontrollers provide dedicated hardware peripherals: the timers. Timers are not only timebase generators, but they also provide several additional features used to interact with the Cortex-M core and other peripherals, both internal and external to the MCU.

Depending on the family and package used, STM32 microcontrollers implement a variable number of timers, each one with specific characteristics. Some part numbers can provide up to 14 independent timers. Different from the other peripherals, timers have almost the same implementation in all STM32-series, and they are grouped inside nine distinct categories. The most relevant of these are: *basic*, *general purpose* and *advanced* timers.

STM32 timers are a powerful peripheral that offer a wide range of customizations. Moreover, some of their features are specific of the application domain. This would require a completely separated book to deepen the topic (you have to consider that usually more than 250 pages of a typical STM32 datasheet is dedicated to timers). This chapter, which is undoubtedly among the longest in the book, tries to shape the most relevant concepts regarding *basic* and *general purpose* timers in STM32 MCUs, looking to the related CubeHAL module used to program them.

11.1 Introduction to Timers

A timer is a *free-running* counter with a counting frequency that is a fraction of its source clock. The counting speed can be reduced using a dedicated prescaler for each timer[1]. Depending on the timer type, it can be clocked by the internal clock (which is derived from the bus where it is connected), by an external clock source or by another timer used as "master".

Usually, a timer counts from zero up to a given value, which cannot be higher than the maximum unsigned value for its resolution (for example, a 16-bit timer overflows when the counter reaches 65535), but it can also count on the contrary and in other ways we will see next.

The most advanced timers in an STM32 microcontroller have several features:

- They can be used as time base generator (which is the feature common to all STM32 timers).
- They can be used to measure the frequency of an external event (input capture mode).
- To control an output waveform, or to indicate when a period of time has elapsed (output compare mode).

[1] This is not entirely true, but it is ok to consider it true here.

  - One pulse mode (OPM) is a particular case of the input capture mode and the output compare mode. It allows the counter to be started in response to a stimulus and to generate a pulse with a programmable length after a programmable delay.
- To generate PWM signals in edge-aligned mode or center-aligned mode independently on each channel (PWM mode).
  - In some STM32 MCUs (notably from STM32F3 and recent STM32L4 series), some timers can generate a center-aligned PWM signals with a programmable delay and phase shift.

Depending on the timer type, a timer can generate interrupts or DMA requests when the following events occur:

- Update events
  - Counter overflow/underflow
  - Counter initialized
  - Others
- Trigger
  - Counter start/stop
  - Counter Initialize
  - Others
- Input capture/Output compare

11.1.1 Timer Categories in an STM32 MCU

STM32 timers can mainly grouped in nine categories. Let us give a brief look at each one of them.

- **Basic timers**: timers from this category are the simplest form of timers in STM32 MCUs. They are 16-bit timers used as time base generator, and they do not have output/input pins. *Basic timers* are also used to feed the DAC peripheral, since their *update event* can trigger DMA requests for the DAC (for this reason they are usually available in STM32 MCUs providing at least a DAC). *Basic timers* can be also used as "masters" for other timers.
- **General purpose timers**: they are 16/32-bit timers (depending on the STM32-series) providing the classical features that a timer of a modern embedded microcontroller is expected to implement. They are used in any application for output compare (timing and delay generation), One-Pulse Mode, input capture (for external signal frequency measurement), sensor interface (encoder, hall sensor), etc. Obviously, a *general purpose timer* can be used as time base generator, like a *basic timer*. Timers from this category provide four-programmable input/output channels.
  - **1-channel/2-channels**: they are two subgroups of *general purpose timers* providing only one/two input/output channel.

  - **1-channel/2-channels with one complimentary output**: same as previous type, but having a *dead time* generator on one channel. This allows having complementary signals with a time base independent from the advanced timers.
- **Advanced timers**: these timers are the most complete ones in an STM32 MCU. In addition to the features found in a *general purpose timer*, they include several features related to motor control and digital power conversion applications: three complementary signals with *dead time* insertion, emergency shut-down input.
- **High-resolution timer**: The high-resolution timer (HRTIM1) is a special timer provided by some microcontrollers from the STM32F3/G4 series (which are the series dedicated to motor control and power conversion) and the STM32H7 series. It allows generating digital signals with high-accuracy timings, such as PWM or phase-shifted pulses. It consists of 6 sub-timers, 1 master and 5 slaves, totaling 10 high-resolution outputs, which can be coupled by pairs for *dead time* insertion. It also features 5 fault inputs for protection purposes and 10 inputs to handle external events such as current limitation, zero voltage or zero current switching.
  HRTIM1 timer is made of a digital kernel clocked at core speed followed by delay lines. Delay lines with closed loop control guarantee a 217ps resolution whatever the voltage, temperature or chip-to-chip manufacturing process deviation. The high-resolution is available on the 10 outputs in all operating modes: variable duty cycle, variable frequency, and constant ON time. This book will not cover HRTIM1 timer. ST provides a well-written Application Note, the AN4539[2], which cover all aspects of HRTIM timers.
- **Low-power timers**: timers from this group are especially designed for low-power applications. Thanks to their diversity of clock sources, these timers are able to keep running in all power modes (except for *Standby* mode). Given this capability to run even with no internal clock source, *Low-power timers* can be used as a "pulse counter" which can be useful in some applications. They also have the capability to wake up the system from *low-power* modes.

[2] https://bit.ly/2YjCdmM

Timer Type	Counter resolution	Counter type	DMA	Channels	Complimentary channels	Synchronization	
						Master	Slave
Advanced	16-bit	up, down and center aligned	Yes	4	3	Yes	Yes
General purpose	16/32-bit	up, down and center aligned	Yes	4	0	Yes	Yes
Basic	16-bit	up	Yes	0	0	Yes	No
1-channel	16-bit	up	No	1	0	Yes (OC signal)	No
2-channels	16-bit	up	No	2	0	Yes	Yes
1-channel with one complementary output	16-bit	up	Yes	1	1	Yes (OC signal)	No
2-channel with one complementary output	16-bit	up	Yes	2	1	Yes	Yes
High-resolution	16-bit	up	Yes	10	10	Yes	Yes
Low-power	16-bit	up	No	1	0	No	No

Table 11.1: The most relevant feature of each timer category

Table 11.1[3] summarizes the most relevant features to keep on hand for each timer category.

11.1.2 Effective Availability of Timers in the STM32 Portfolio

Not all types of timers are available in all STM32 MCUs. It depends mainly on the STM32-series, the sales type and package used. The **Table 11.2** summarizes the distribution of the 22 timers in all STM32 families. An asterisk indicates that timer is not available in all MCUs in the series.

[3]The table is adapted from the one found in the AN4013(http://bit.ly/1WAewd6) from ST, an application note dedicated to STM32 timers that is best to keep in hand during this chapter.

Timer Type		STM32F0	STM32F1	STM32F3	STM32G0	STM32G4	STM32F2	STM32F4	STM32F7	STM32H7	STM32L0	STM32L1	STM32L4	STM32L4+	STM32L5	STM32U5
Advanced		TIM1	TIM1	TIM1*	TIM1	TIM1	TIM1	TIM1	TIM1	TIM1			TIM1	TIM1	TIM1	TIM1
			TIM8*	TIM8*		TIM8	TIM8	TIM8*	TIM8	TIM8			TIM8*	TIM8*	TIM8	TIM8
						TIM20*										
General purpose	16-bit	TIM3	TIM2	TIM3	TIM3	TIM3	TIM3	TIM3*	TIM3	TIM3	TIM2	TIM2	TIM3*	TIM3*	TIM3	TIM3
			TIM3	TIM4		TIM4	TIM4	TIM4*	TIM4	TIM4		TIM3	TIM4*	TIM4*	TIM4	TIM4
			TIM4*									TIM4				
			TIM5*	TIM19*												
	32-bit	TIM2		TIM2	TIM2	TIM2	TIM2	TIM2*	TIM2	TIM2			TIM2	TIM2	TIM2	TIM2
				TIM5*		TIM5	TIM5	TIM5	TIM5	TIM5		TIM5	TIM5*	TIM5*	TIM5	TIM5
Basic		TIM6*	TIM6*	TIM6	TIM6	TIM6	TIM6	TIM6*	TIM6	TIM6	TIM6	TIM6	TIM6	TIM6	TIM6	TIM6
		TIM7*	TIM7*	TIM7	TIM7	TIM7	TIM7		TIM7	TIM7	TIM7	TIM7	TIM7*		TIM7	TIM7
				TIM18*												
1-channel			TIM10*				TIM10	TIM10*	TIM10			TIM10				
			TIM11*				TIM11	TIM11	TIM11			TIM11				
			TIM13*	TIM13*			TIM13		TIM13	TIM13						
		TIM14	TIM14*	TIM14*	TIM14	TIM14	TIM14		TIM14	TIM14						
2-channels			TIM9				TIM9	TIM9	TIM9		TIM21	TIM9				
			TIM12	TIM12*			TIM12		TIM12	TIM12	TIM22*					
2-channel with one complementary output		TIM15*	TIM15*	TIM15	TIM15	TIM15				TIM15			TIM15	TIM15	TIM15	TIM15
1-channel with one complementary output		TIM16	TIM16*	TIM16	TIM16	TIM16				TIM16			TIM16	TIM16	TIM16	TIM16
		TIM17	*TIM17	TIM17	TIM17	TIM17				TIM17			TIM17*		TIM17	TIM17
High-resolution				HRTIM*		HRTIM1*										
Low-power					LPTIM1	LPTIM1		LPTIM1	LPTIM1	LPTIM1	LPTIM1		LPTIM1	LPTIM1	LPTIM1	LPTIM1
					LPTIM2	LPTIM2				LPTIM2			LPTIM2	LPTIM2	LPTIM2	LPTIM2
										LPTIM3					LPTIM3	LPTIM3
										LPTIM4						

Table 11.2: Which timers are implemented in each STM32-series

It is important to remark some things regarding **Table 11.2**:

- Given a specific timer (e.g., TIM1, TIM8, etc.), its implementation (features, number and types of registers, generated interrupts, DMA requests, peripheral interconnection[4], etc.) is the same[5] in all STM32 MCUs. This guarantees you that a firmware written to use a specific timer is portable to other MCUs or STM32-series having the same timer.
- The effective presence of a timer in an MCU belonging to the given family depends on sales type and the package used (packages with more pins may provide all timers implemented by

[4]With the term *peripheral interconnection*, we indicate the ability of some peripherals to "trigger" other peripherals, or to fire some of their DMA requests (for example, the TIM6 update event can trigger the DAC1 conversion). More about this topic in Chapter 13.

[5]As said at the beginning of this chapter, STM32 timers are the only peripherals that share the same implementation among all STM32 families. This is almost true, except for TIM2 and TIM5 timers, which have a 32-bit resolution in the majority of STM32 MCUs and 16-bit resolution in some early STM32 MCUs. Moreover, some specific features may have a slightly different implementation between some STM32 series (especially between "older" STM32F1 microcontrollers and more recent STM32 ones). Always consult the datasheet for your MCU before you plan to use dedicated features provided by some timers.

that family).
- The table was extracted, expanded and rearranged from the AN4013[6]. I checked carefully the values reported in that table and found some non-updated things. However, I am not totally sure that it faithfully adheres to actual implementation[7] of the whole STM32 portfolio (I should check more than 1500 microcontrollers to be sure of that values). For this reason, I left some cells empty, so you can eventually add values if you discover a mistake[8].

Table 11.3 reports the list of all timers implemented by the MCUs equipping the nine Nucleo boards we are considering in this book. It is important to underline some things reported in **Table 11.3**:

- STM32F401RE and STM32F103RB do not provide a *basic timer*.
- The "MAX clock speed" column reports the maximum clock speed for all timers in a given STM32 MCU. This means that the timer maximum clock speed depends on the bus where it is connected to. Always consult the datasheet to determine to which bus the timer is connected (see the *peripheral mapping section* of the datasheet) and use CubeMX *Clock configuration* view to determine the configured bus speed.
- The STM32G474RE MCU, which has been introduced on the market at the beginning of 2021, implements two features that are distinctive of the STM32L and STM32F3 series: a *low-power* timer and a *High-Resolution Timer*.

When dealing with timers, it is important to have a pragmatic approach. Otherwise, it is easy to get lost in their settings and in the corresponding HAL routines (the `HAL_TIM` and `HAL_TIM_EX` modules are among the most articulated in the CubeHAL).
For this reason, we will start studying how to use *basic timers*, whose functionalities are also common to more advanced STM32 timers.

[6] https://bit.ly/1WAewd6
[7] The table was arranged in September 2021. STM32 MCUs evolve almost day-by-day, so some things may be changed when you read this chapter.
[8] And eventually send me an email so that I can correct the table in next releases of the book :-)

Nucleo P/N	Basic Timers	General Purpose Timers	Advanced Timers	High-resolution Timers	Low-power Timers	MAX clock speed
NUCLEO-G474RE	TIM6-7	TIM2-5 TIM9-14 TIM15-17	TIM1 TIM8 TIM20	HRTIM1	LPTIM1	170MHz
NUCLEO-F446RE	TIM6-7	TIM2-5 TIM9-14	TIM1 TIM8	-	-	90/180MHz
NUCLEO-F401RE	-	TIM2-5 TIM9-11	TIM1	-	-	84MHz
NUCLEO-F303RE	TIM6-7	TIM2-4 TIM15-17	TIM1 TIM8 TIM20	-	-	72/144MHz
NUCLEO-F103RB	-	TIM3-4	TIM1	-	-	64/72MHz
NUCLEO-F072RB	TIM6-7	TIM2-3 TIM14-17	TIM1	-	-	48MHz
NUCLEO-L476RG	TIM6-7	TIM2-5 TIM15-17	TIM1 TIM8	-	LPTIM1 LPTIM2	80MHz
NUCLEO-L152RE	TIM6-7	TIM2-5 TIM9-11	-	-	-	32MHz
NUCLEO-L073RZ	TIM6-7	TIM2-3 TIM21-22	-	-	LPTIM1	32MHz

Table 11.3: Which timers are implemented in each STM32 MCU equipping nine Nucleo boards

11.2 Basic Timers

Basic timers TIM6, TIM7 and TIM18[9] are the simplest timers available in the STM32 portfolio. Even if they are not provided by all STM32 MCUs, it is important to underline that STM32 timers are designed so that more advanced timers implement the same features (in the same way) of less powerful ones, as shown in **Figure 11.1**. This means that it is perfectly possible to use a *general purpose timer* in the same way of a *basic timer*. The CubeHAL also reflects this hardware implementation: the base operations on all timers are performed by using the `HAL_TIM_Base_XXX` functions.

A single timer is referenced by using an instance of the C `struct TIM_HandleTypeDef`, which is defined in the following way:

[9]The TIM18 basic timer is only available in STM32F37x microcontrollers.

```
typedef struct {
  TIM_TypeDef                 *Instance;     /* Pointer to timer descriptor       */
  TIM_Base_InitTypeDef        Init;          /* TIM Time Base required parameters */
  HAL_TIM_ActiveChannel       Channel;       /* Active channel                    */
  DMA_HandleTypeDef           *hdma[7];      /* DMA Handlers array                */
  HAL_LockTypeDef             Lock;          /* Locking object                    */
  __IO HAL_TIM_StateTypeDef   State;         /* TIM operation state               */
} TIM_HandleTypeDef;
```

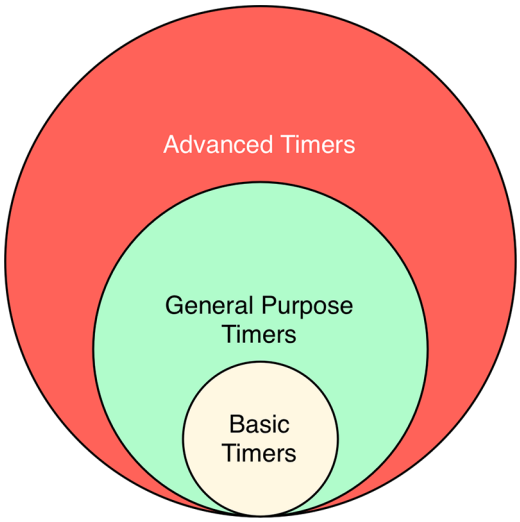


Figure 11.1: The relation between the three major categories of timers

Let us see more in depth the most important fields of this struct.

- `Instance`: is the pointer to the TIM descriptor we are going to use. For example, TIM6 is one of the basic timers available in the majority of STM32 microcontrollers.
- `Init`: is an instance of the C `struct TIM_Base_InitTypeDef`, which is used to configure the base timer functionalities. We will study it more in depth in a while.
- `Channel`: it indicates the number of active channels in those timers that provide one or more input/output channels (this is not the case of *basic timers*). It can assume one or more values from the `enum HAL_TIM_ActiveChannel`, and we will study its usage in a next paragraph.
- `*hdma[7]`: this is an array containing the pointers to `DMA_HandleTypeDef` descriptors for DMA requests associated to the timer. As we will see later, a timer can generate up to seven DMA requests.
- `State`: this is used internally by the HAL to keep track of the timer state.

All the timer configuration activities are performed by using an instance of the C `struct TIM_Base_InitTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t Prescaler;     /* Specifies the prescaler value used to divide the TIM clock. */
  uint32_t CounterMode;   /* Specifies the counter mode.                                  */
  uint32_t Period;        /* Specifies the period value to be loaded into the active
                             Auto-Reload Register at the next update event.               */
  uint32_t ClockDivision; /* Specifies the clock division.                                */
  uint32_t RepetitionCounter; /* Specifies the repetition counter value.                  */
} TIM_Base_InitTypeDef;
```

- `Prescaler`: it divides the timer clock by a factor ranging from 1 up to 65535 (this means that the prescaler register has a 16-bit resolution). For example, if the bus where the timer is connected runs at 48MHz, then a prescaler value equal to 48 lowers the counting frequency to 1MHz.
- `CounterMode`: it defines the counting direction of the timer, and it can assume one of the values from **Table 11.4**. Some counting modes are available only in *general purpose* and *advanced* timers. For *basic timers*, only the `TIM_COUNTERMODE_UP` is defined.
- `Period`: sets the maximum value for the timer counter before it restarts counting again. This can assume a value from `0x1` to `0xFFFF` (65535) for 16-bit timers, and from `0x1` to `0xFFFF FFFF` for TIM2 and TIM5 timers in those MCUs that implement them as 32-bit timers. **If `Period` is set to `0x0` the timer does not start**.
- `ClockDivision`: this bit-field indicates the division ratio between the internal timer clock frequency and sampling clock used by the digital filters on ETRx and TIx pins. Please, take note that it is not related to the frequency of the clock feeding the timer. This is a common confusion to STM32 novices. `ClockDivision` can assume one value from **Table 11.5**, and it is available only in *general purpose* and *advanced* timers. We will study digital filters on input pins of a timer later in this chapter. This field is also used by the *dead time* generator (a feature not described in this book).
- `RepetitionCounter`: every timer has a specific *update register* that keeps track of the timer overflow/underflow condition. This can also generate a specific IRQ, as we will see next. The `RepetitionCounter` says how many times the timer overflows/underflows before the *update register* is set, and the corresponding event is raised (if enabled). `RepetitionCounter` is only available in *advanced* timers.

Table 11.4: Available counter mode for a timer

Counter Mode	Description
`TIM_COUNTERMODE_UP`	The timer counts from zero up to the `Period` value (which cannot be higher than the timer resolution - 16/32-bit) and then generates an overflow event.
`TIM_COUNTERMODE_DOWN`	The timer counts down from the `Period` value to zero and then generates an underflow event.
`TIM_COUNTERMODE_CENTERALIGNED1`	In center-aligned mode, the counter counts from 0 to the `Period` value – 1, generates an overflow event, then counts from the `Period` value down to 1 and generates a counter underflow event. Then it restarts counting from 0. The Output compare interrupt flag of channels configured in output mode is set when the counter counts down.

Table 11.4: Available counter mode for a timer

Counter Mode	Description
`TIM_COUNTERMODE_CENTERALIGNED2`	Same as `TIM_COUNTERMODE_CENTERALIGNED1`, but the Output compare interrupt flag of channels configured in output mode is set when the counter counts up.
`TIM_COUNTERMODE_CENTERALIGNED3`	Same as `TIM_COUNTERMODE_CENTERALIGNED1`, but the Output compare interrupt flag of channels configured in output mode is set when the counter counts up and down.

Table 11.5: Available `ClockDivision` modes for *general purpose* and *advanced* timers

Clock division modes	Description
`TIM_CLOCKDIVISION_DIV1`	Computes 1 sample of the input signal on ETRx and TIx pins
`TIM_CLOCKDIVISION_DIV2`	Computes 2 sample of the input signal on ETRx and TIx pins
`TIM_CLOCKDIVISION_DIV4`	Computes 4 sample of the input signal on ETRx and TIx pins

11.2.1 Using Timers in *Interrupt Mode*

Before seeing a complete example, it is best to summarize what we have seen so far. A *basic timer*:

- is a *free-running* counter, which counts from 0 up to the value specified in the `Period`[10] field in the `TIM_Base_InitTypeDef` initialization structure, which can assume the maximum value of `0xFFFF` (`0xFFFF FFFF` for 32-bit timers);
- the counting frequency depends on the speed of the bus where the timer is connected, and it can be lowered up to 65536 times by setting the `Prescaler` register in the initialization structure;
- when the timer reaches the `Period` value it overflows, and the *Update Event* (UEV) flag is set[11]; the timer automatically restarts counting again from the initial value (which is always zero for *basic timers*)[12].

The `Period` and `Prescaler` registers determine the timer frequency, that is how long does it takes to overflow (or, if you prefer, how often an *Update Event* is generated), according to this simply formula:

$$UpdateEvent = \frac{Timer_{clock}}{(Prescaler + 1)(Period + 1)} \quad [1]$$

[10]The `Period` is used to fill the *Auto-reload register* (ARR) of the timer. I do not know why ST engineers have decided to name it in this way, since ARR is the register name used in all ST datasheets. This can lead to a lot of confusion, especially when you are new to the CubeHAL, but unfortunately there is nothing we can do.

[11]The *Update Event* (UEV) is latched to the prescaler clock, and it is automatically cleared on the next clock edge. Don't confuse the UEV with the *Update Interrupt Flag* (UIF), which must be cleared manually like every other IRQ. UIF is set only when the corresponding interrupt is enabled. As we will discover in Chapter 19, the UEV event, like all event flags set for other peripherals, allows to wake up the MCU when it enters a low-power mode using the `WFE` instruction.

[12]This is an important distinction with other microcontroller architectures (especially 8-bit ones) where timers need to be "rearmed" manually before they can start counting again.

For example, assume a timer connected to the APB1 bus in an STM32F072 MCU, with the HCLK set to 48MHz, a `Prescaler` value equal to 47999 and a `Period` value equal to 499. We have that timer will overflow at every:

$$UpdateEvent = \frac{48.000.000}{(47999+1)(499+1)} = 2Hz = \frac{1}{2}s = 0.5s$$

The following code, designed to run on a Nucleo-F072RB, shows a complete example using the TIM6[13]. The example is nothing more than the classical blinking LED, but this time we use a *basic timer* to compute delays.

Filename: `Core/Src/main-ex1.c`

```
5  TIM_HandleTypeDef htim6;
6
7  int main(void) {
8    HAL_Init();
9
10   Nucleo_BSP_Init();
11
12   htim6.Instance = TIM6;
13   htim6.Init.Prescaler = 47999; //48MHz/48000 = 1000Hz
14   htim6.Init.Period = 499;      //1000HZ / 500 = 2Hz = 0.5s
15
16   __HAL_RCC_TIM6_CLK_ENABLE();
17
18   HAL_NVIC_SetPriority(TIM6_DAC_IRQn, 0, 0);
19   HAL_NVIC_EnableIRQ(TIM6_DAC_IRQn);
20
21   HAL_TIM_Base_Init(&htim6);
22   HAL_TIM_Base_Start_IT(&htim6);
23
24   while (1);
25 }
```

Lines [12:14] configure TIM6 using the `Prescaler` and `Period` values computed before. The timer peripheral is then enabled by using the macro at line 19. The same applies to its IRQ. The timer is then configured at line 21 and started in interrupt mode using the `HAL_TIM_Base_Start_IT()` function[14]. The rest of the code is similar to what seen until now.

The `TIM6_DAC_IRQHandler()` ISR fires when the timer overflows, and the `HAL_TIM_IRQHandler()` is then called. The HAL will automatically handle for us all the necessary operations to properly manage the *update event*, and it will call the `HAL_TIM_PeriodElapsedCallback()` callback to signal us that the timer has been overflowed.

[13] Owners of the Nucleo boards equipping F401 and F103 STM32 MCUs will find a slightly different example using a *general purpose* timer. However, concepts remain the same.

[14] A really common mistake made by novices is to forget to start a timer, by using one of the `HAL_TIM_xxx_Start` function provided by the CubeHAL.



The Performance of the `HAL_TIM_IRQHandler()` Routine

For timers running really fast, the `HAL_TIM_IRQHandler()` has a non-negligible overhead. That function is designed to check up to nine different interrupt status flags, which requires several ARM assembly instructions to carry out the task. If you need to process the interrupts in the shortest time, probably it is best to handle the IRQ by yourself. Once again, the HAL is designed to abstract a lot of details to the user, but it introduces performance penalties that every embedded developer should know.



How to Choose the Values for `Prescaler` and `Period` Fields?

First of all, note that not all combinations of `Prescaler` and `Period` values lead to integer division of the timer clock frequency. For example, for a timer running at 48MHz, a `Period` equal to 65535 lowers the timer frequency to 732,4218 Hz. This author is used to divide the main frequency of the timer setting an integer divider for the `Prescaler` value (e.g., 47999 for a 48MHz timer - remember that, according to equation [1], the frequency is computed by adding 1 to both the `Prescaler` and `Period` values), and then playing with the `Period` value to achieve the wanted frequency. MikroElektronica provides a nice tool[15] to automatically compute those values, given a specific STM32 MCUs and the *HCLK* frequency. Unfortunately, the code it generates does not cover the CubeHAL at the time of writing this chapter.

11.2.1.1 Time Base Generation in *Advanced* Timers

So far we have seen that all base functionalities of a timer are configured through an instance of the `TIM_Base_InitTypeDef struct`. This `struct` contains a field named `RepetitionCounter` used to further increase the period between two consecutive *update events*: the timer will count a given number of times before setting the event and raising the corresponding interrupt. `RepetitionCounter` is only available in *advanced* timers, and this causes that the formula to compute the frequency of *update events* becomes:

$$UpdateEvent = \frac{Timer_{clock}}{(Prescaler + 1)(Period + 1)(RepetitionCounter + 1)}$$

Leaving the `RepetitionCounter` equal to zero (default behaviour), we obtain the same working mode of a *basic timer*.

11.2.2 Using Timers in *Polling Mode*

The CubeHAL provides three ways to use timers: *polling*, *interrupt* and *DMA mode*. For this reason, the HAL provides three distinct functions to start a timer: `HAL_TIM_Base_Start()`, `HAL_TIM_Base_Start_IT()` and `HAL_TIM_Base_Start_DMA()`. The idea behind the *polling mode* is that the timer

[15] http://www.mikroe.com/timer-calculator/

counter register (`TIMx->CNT`) is accessed continuously to check for a given value. But care must be taken when polling a timer. For example, it is quite common to find around in the web code like the following one:

```
...
while (1) {
    if(__HAL_TIM_GET_COUNTER(&tim) == value)
        ...
```

That way to poll for a timer is completely wrong, even if it apparently works in some examples. Why?

Timers run independently from the Cortex-M core. A timer can count fast, up to the same clock frequency of the CPU core. But checking a timer counter for equality (that is, to check if it is equal to a given value) requires several ARM assembly instructions, which in turn need several clock cycles. There is no guarantee that the CPU accesses to the counter register exactly at the same time it reaches the configured value (this happens only if the timer runs really slow). A better way is to check if the timer current counter value is equal or greater than the given value, or to check against the UIF flag status[16]: in the worst case we can have a shift in time measuring, but we will not lose the event at all (unless the timer runs really fast and we lose the subsequent events because the interrupt is masked - that is, UIF flag it still set before it is cleared manually by us or automatically by the HAL).

```
...
while (1) {
    if(__HAL_TIM_GET_FLAG(&tim) == TIM_FLAG_UPDATE) {
            //Clear the IRQ flag otherwise we lose other events
            __HAL_TIM_CLEAR_IT(htim, TIM_IT_UPDATE);
            ...
```

Having said that, timers are asynchronous peripherals, and the correct way to manage the overflow/underflow event is by using interrupts. There is no reason to not use a timer in interrupt mode unless the timer runs really fast and generating an interrupt after few microseconds (or even nanoseconds) would completely flood the MCU preventing it from processing other instructions[17].

11.2.3 Using Timers in *DMA Mode*

Timers are often programmed to work in *DMA mode*, especially when they are used to trigger other peripherals. This mode guarantees that the operations performed by the timer are deterministic and with the smallest possible latency, especially if they run fast. Moreover, the Cortex-M core is freed

[16] However this requires that the timer is enabled in interrupt mode, using the `HAL_TIM_Base_Start_IT()` function.

[17] Remember that even if the exception handling in a Cortex-M MCU has a deterministic latency (Cortex-M3/4/7/33 cores serve an interrupt in 12 CPU cycles, while Cortex-M0 does it in 15 cycles and Cortex-M0+ in 16 cycles) it has a non-negligible cost, which requires several nanoseconds in "low-speed" MCUs (for example, for an STM32F072 MCU running at 48MHz, an interrupt is serviced in about 300ns). This cost has to be added to the overhead introduced by the HAL during the interrupt management, as seen before.

from the timer management, which usually involves the handling of frequent ISRs that could congest the CPU. Finally, in some advanced modes, like the output PWM one, it is almost impossible to reach given switching frequencies without using the timer in *DMA Mode*.

For these reasons, timers offer up to seven DMA requests, which are listed in **Table 11.6**. *Basic timers* implement only the `TIM_DMA_UPDATE` request since they do not have input/output I/Os. However, it is useful to take advantage of the TIMx_UP request in those situations where we want to perform DMA transfers on a time-basis.

Table 11.6: DMA requests (the most of them are available only in *general purpose* and *advanced* timers

Timer DMA request	Description
`TIM_DMA_UPDATE`	Update request (it is generated on the UEV event)
`TIM_DMA_CC1`	Capture/Compare 1 DMA request
`TIM_DMA_CC2`	Capture/Compare 2 DMA request
`TIM_DMA_CC3`	Capture/Compare 3 DMA request
`TIM_DMA_CC4`	Capture/Compare 4 DMA request
`TIM_DMA_COM`	Commutation request
`TIM_DMA_TRIGGER`	Trigger request

The following example is another variation of the blinking LED application, but this time we use a timer in DMA mode to turn the LED ON/OFF. Here we are going to use the TIM6 timer programmed to overflow every 500ms: when this happens, the timer generates the TIM6_UP request (which in an STM32F072 MCU is bound to the third channel of DMA1) and the next element of a buffer is transferred to the GPIOA->ODR register in DMA circular mode, which causes that the LD2 blinks indefinitely.



Read Carefully

In STM32F2/F4/F7/L1/L4 families, only the DMA2 has full access to the Bus Matrix. This means that only timers whose requests are bound to this DMA controller can be used to perform transfers involving other peripheral (except for the internal and external volatile memories). For this reason, this example for Nucleo boards based on F2/F4/L1/L4 MCUs use TIM1 as base generator.

Filename: `Core/Src/main-ex2.c`

```
8  int main(void) {
9    uint8_t data[] = {0xFF, 0x0};
10
11   HAL_Init();
12
13   Nucleo_BSP_Init();
14
15   htim6.Instance = TIM6;
16   htim6.Init.Prescaler = 47999; //48MHz/48000 = 1000Hz
17   htim6.Init.Period = 499;      //1000HZ / 500 = 2Hz = 0.5s
```

```
18
19    __HAL_RCC_TIM6_CLK_ENABLE();
20
21    HAL_TIM_Base_Init(&htim6);
22
23    hdma_tim6_up.Instance = DMA1_Channel3;
24    hdma_tim6_up.Init.Direction = DMA_MEMORY_TO_PERIPH;
25    hdma_tim6_up.Init.PeriphInc = DMA_PINC_DISABLE;
26    hdma_tim6_up.Init.MemInc = DMA_MINC_ENABLE;
27    hdma_tim6_up.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
28    hdma_tim6_up.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
29    hdma_tim6_up.Init.Mode = DMA_CIRCULAR;
30    hdma_tim6_up.Init.Priority = DMA_PRIORITY_LOW;
31    HAL_DMA_Init(&hdma_tim6_up);
32
33    HAL_TIM_Base_Start(&htim6);
34    HAL_DMA_Start(&hdma_tim6_up, (uint32_t)data, (uint32_t)&GPIOA->ODR, 2);
35    __HAL_TIM_ENABLE_DMA(&htim6, TIM_DMA_UPDATE);
36
37    while (1);
38  }
```

Lines [26:33] configure the `DMA_HandleTypeDef` for the DMA1_Channel3 in circular mode. Then line 37 starts the DMA transfer so that the content of the `data` buffer is transferred inside the `GPIOA->ODR` register every time a TIM6_UP request is generated, that is the timer overflows. This causes that the LD2 LED blinks. Take note that we are not using the `HAL_TIM_Base_Start_DMA()` function here. Why not?

Looking to the implementation of the `HAL_TIM_Base_Start_DMA()` routine, you can see that ST engineers have defined it so that the DMA transfer is performed from the memory buffer to the `TIM6->ARR`, which corresponds to the `Period`.

```
HAL_TIM_Base_Start_DMA(TIM_HandleTypeDef *htim, uint32_t *pData, uint16_t Length) {
  ...
  /* Enable the DMA channel */
  HAL_DMA_Start_IT(htim->hdma[TIM_DMA_ID_UPDATE], (uint32_t)pData,
                   (uint32_t)&htim->Instance->ARR, Length);

  /* Enable the TIM Update DMA request */
  __HAL_TIM_ENABLE_DMA(htim, TIM_DMA_UPDATE);
  ...
```

Basically, we can use the `HAL_TIM_Base_Start_DMA()` only to change the timer `Period` every time it overflows. So, we need to configure the DMA by ourselves in order to perform this transfer.

In the next Chapter we will see a more useful application of how to use timers in DMA mode to perform ADC conversions regularly.

11.2.4 Stopping a Timer

The CubeHAL provides three functions to stop a running timer: `HAL_TIM_Base_Stop()`, `HAL_TIM_Base_Stop_IT()` and `HAL_TIM_Base_Stop_DMA()`. We pick one of these depending on the timer mode we are using (for example, if we have started a timer in *interrupt mode*, then we need to stop it using the `HAL_TIM_Base_Stop_IT()` routine). Each function is designed to properly disable IRQs and DMA configurations.

11.2.5 Using CubeMX to Configure a *Basic Timer*

CubeMX can reduce to the minimum the effort needed to configure a *basic timer*. Once the timer is enabled by checking the flag **Activated**, it can be configured from the *Configuration* view. The timer configuration view allows to setup the values for the `Prescaler` and `Period` registers, as shown in **Figure 11.2**. CubeMX will generate all the necessary initialization code inside the `MX_TIMx_Init()` function. Moreover, always in the same configuration dialog, it is possible to enable timer-related IRQs and DMA requests.

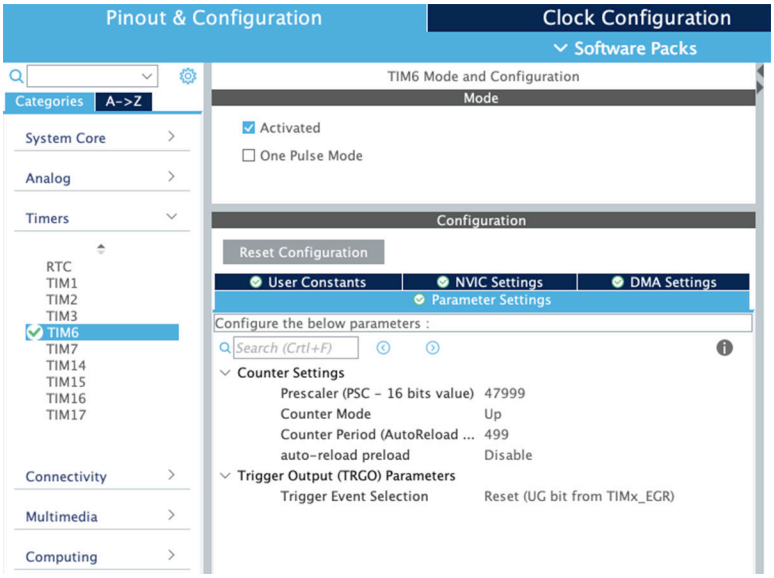


Figure 11.2: CubeMX allows to easily generate the necessary code to configure a timer

11.3 General Purpose Timers

The majority of STM32 timers are *general purpose* ones. Different from the *basic timers* seen before, they offer much more interaction capabilities, thanks to up to four independent channels that can be used to measure input signals, to output signals on a time basis, to generate *Pulse-Width Modulation* (PWM) signals. *General purpose* timers, however, offer much more functionalities that we will discover progressively in this part of the chapter.

11.3.1 Time Base Generator with External Clock Sources

The **Figure 11.3** shows the block diagram of a *general purpose* timer[18]. Some parts of the diagram have been masked: we will study them more in depth later. The path highlighted in red is used to feed the timer when the APB clock is selected as source: the internal clock **CK_INT** feeds the **Prescaler** (**PSC**), which in turn determines how fast the **Counter Register** (**CNT**) is increased/decreased. This one is compared with the content of the **auto-reload register** (which is filled with the value of the `TIM_Base_InitTypeDef.Period` field). When they match, the UEV event is generated, and the corresponding IRQ is fired, if enabled.

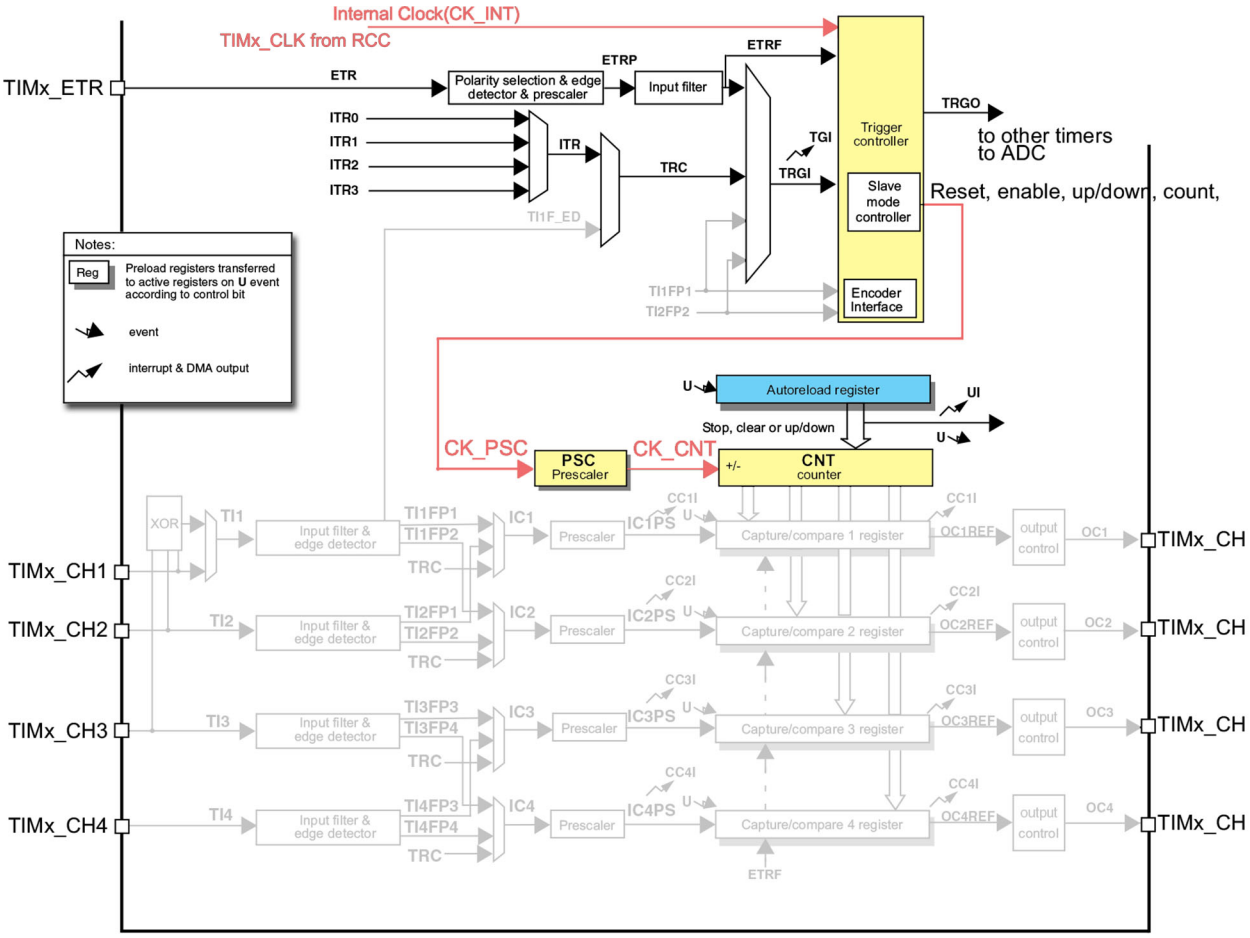


Figure 11.3: The structure of a *general purpose* timer

Looking at **Figure 11.3**, we can see that a timer can receive "stimuluses" from other sources. These can be divided in two main groups:

- *Clock sources*, which are used **to clock** the timer. They can come from external sources connected to the MCU pins or from other timers connected internally to the MCU. Keep in

[18]The figure is arranged from the one found in the RM0368(https://bit.ly/1Kq3SoE) reference manual from ST.

mind that a timer cannot work without a clock source, because this is used to increment the *counter register*.
- *Trigger sources*, which are used **to synchronize** the timer with external sources connected to the MCU pins or with other timers connected internally. For example, a timer can be configured to start counting when an external event *triggers* it. In this case the timer is clocked by another clock source (which can be both the APBx bus or an external clock source connected to the ETR2 pin), and it is controlled (that is, when it starts counting, etc.) by another device.

Depending on the timer type and its actual implementation, a timer can be clocked from:

- The internal TIMx_CLK provided by the RCC (shown in paragraph 11.2)
- Internal trigger input 0 to 3
  - ITR0, ITR1, ITR2 and ITR3 using another timer (master) as prescaler of this timer (slave) (shown in paragraph 11.3.1.2)
- External input channel pins (shown in paragraph 11.3.1.2)
  - Pin 1: TI1FP1 or TI1F_ED
  - Pin 2: TI2FP2
- External ETR pins:
  - ETR1 pin (shown in paragraph 11.3.1.2)
  - ETR2 pin (shown in paragraph 11.3.1.1)

A timer can, instead, be triggered from:

- Internal trigger inputs 0 to 3
  - ITR0, ITR1, ITR2 and ITR3 using another timer as master (shown in paragraph 11.3.2)
- External input channel pins (shown in paragraph 11.3.2)
  - Pin 1: TI1FP1 or TI1F_ED
  - Pin 2: TI2FP2
- External ETR1 pin

Let us study these ways to clock/trigger a timer from an external source by analyzing practical examples.

11.3.1.1 External Clock Mode 2

General purpose timers have the ability to be clocked from external sources, setting them in two distinct modes: *External Clock Source Mode 1* and *2*. The first one is available when the timer is configured in *slave* mode. We will study this mode in the next paragraph.

The second mode is, instead, activated simply by using an external clock source. This allows to use more accurate and dedicated sources, and eventually to further reduce the counting frequency. In fact, when the *External Clock Source Mode 2* is selected, the formula to compute the frequency of *update events* becomes:

$$UpdateEvent = \frac{EXT_{clock}}{(EXT_{clock}Prescaler)(Prescaler + 1)(Period + 1)(RepetitionCounter + 1)} \quad [2]$$

where EXT_{clock} is the frequency of the external source and $EXT_{clock}Prescaler$ is a source frequency divider that can assume the values 1, 2, 4 and 8.

The clock source of a *general purpose* timer can be selected by using the function `HAL_TIM_ConfigClockSource()` and an instance of the `struct TIM_ClockConfigTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t ClockSource;     /* TIM clock sources   */
  uint32_t ClockPolarity;   /* TIM clock polarity  */
  uint32_t ClockPrescaler;  /* TIM clock prescaler */
  uint32_t ClockFilter;     /* TIM clock filter */
} TIM_ClockConfigTypeDef;
```

- `ClockSource`: specifies the source of the clock signal used to bias the timer. It can assume a value from the **Table 11.7**. By default, the `TIM_CLOCKSOURCE_INTERNAL` mode is selected.
- `ClockPolarity`: indicates the polarity of the clock signal used to bias the timer. It can assume a value from the **Table 11.8**. By default, the `TIM_CLOCKPOLARITY_RISING` mode is selected.
- `ClockPrescaler`: specifies the prescaler for the external clock source. It can assume a value from the **Table 11.9**. By default, the `TIM_CLOCKPRESCALER_DIV1` value is selected.
- `ClockFilter`: this 4-bit field defines the frequency used to sample the external clock signal and the length of the digital filter applied to it. The digital filter is made of an event counter in which N consecutive events are needed to validate a transition on the output. Refer to the datasheet of your MCU about how the f_{DTS} (*Dead-Time Signal*) is computed. By default, the filter is disabled.

Table 11.7: Available clock source modes for *general purpose* and *advanced* timers

Clock source mode	Description
TIM_CLOCKSOURCE_INTERNAL	The timer is clocked by the APBx bus where the timer is connected to.
TIM_CLOCKSOURCE_ETRMODE1	This mode is called *External Clock Mode 1*[19] and it is available when the timer is configured in *slave mode*. The timer can be clocked by an internal/external source connected to ITR0, ITR1, ITR2, ITR3, TI1FP1, TI2FP2 or ETR1 pin.
TIM_CLOCKSOURCE_ETRMODE2	This mode is called *External Clock Mode 2*. The timer can be clocked by an external source connected to ETR2 pin.

Table 11.8: Available external clock polarity modes for *general purpose* and *advanced* timers

External clock polarity mode	Description
TIM_CLOCKPOLARITY_RISING	The timer is synchronized on the rising edge of the external clock source.
TIM_CLOCKPOLARITY_FALLING	The timer is synchronized on the falling edge of the external clock source.
TIM_CLOCKPOLARITY_BOTHEDGE	The timer is synchronized on rising and falling edges of the external clock source (this will increase the sampled frequency).

Table 11.9: Available external clock prescaler modes for *general purpose* and *advanced* timers

External clock prescaler mode	Description
TIM_CLOCKPRESCALER_DIV1	No prescaler used
TIM_CLOCKPRESCALER_DIV2	Capture performed once every 2 events
TIM_CLOCKPRESCALER_DIV4	Capture performed once every 4 events
TIM_CLOCKPRESCALER_DIV8	Capture performed once every 8 events

Let us build an example that shows how to use an external clock source for the TIM3 timer. The example consists in routing the *Master Clock Output* (MCO) pin to the TIM3_ETR2 pin, which corresponds to PD2 pin for all Nucleo boards providing this timer. This can be easily done by using the Morpho connectors, as shown in **Figure 11.4** for the Nucleo-F072RB (for your Nucleo, use CubeMX tool to identify the MCO pin and the corresponding pinout diagram from Appendix C).

[19] In the ST documentation these modes are also called *External Trigger mode 1* and *2* (ETR1 and ETR2).

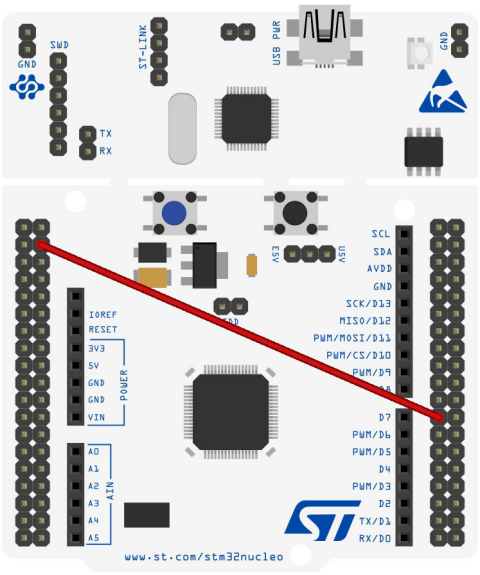


Figure 11.4: How to route the `MCO` pin to the `TIM3_ETR` pin in a Nucleo-F072RB board

The MCO pin is enabled and connected to the HSI clock source. The following code shows the most relevant parts of the example.

Filename: `Core/Src/main-ex3.c`

```
23 void MX_TIM3_Init(void) {
24   TIM_ClockConfigTypeDef sClockSourceConfig;
25
26   htim3.Instance = TIM3;
27   htim3.Init.Prescaler = 999;
28   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
29   htim3.Init.Period = 3999;
30   htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
31   htim3.Init.RepetitionCounter = 0;
32   HAL_TIM_Base_Init(&htim3);
33
34   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_ETRMODE2;
35   sClockSourceConfig.ClockPolarity = TIM_CLOCKPOLARITY_NONINVERTED;
36   sClockSourceConfig.ClockPrescaler = TIM_CLOCKPRESCALER_DIV1;
37   sClockSourceConfig.ClockFilter = 0;
38   HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig);
39
40   HAL_NVIC_SetPriority(TIM3_IRQn, 0, 0);
41   HAL_NVIC_EnableIRQ(TIM3_IRQn);
42 }
43
44 void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base) {
45   GPIO_InitTypeDef GPIO_InitStruct;
46   if(htim_base->Instance==TIM3)  {
47     /* Peripheral clock enable */
```

```
48      __HAL_RCC_TIM3_CLK_ENABLE();
49      __HAL_RCC_GPIOD_CLK_ENABLE();
50
51      /**TIM3 GPIO Configuration
52      PD2     ------> TIM3_ETR
53      */
54      GPIO_InitStruct.Pin = GPIO_PIN_2;
55      GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
56      GPIO_InitStruct.Pull = GPIO_NOPULL;
57      GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
58      HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
59   }
60  }
```

Lines [27:33] configure the TIM3 timer, setting its prescaler to 999 and period to 3999. Lines [34:38] configure the external clock source for TIM3. Since the HSI oscillator runs at 8MHz[20], using the equation [2] we can compute the UEV frequency, which is equal to:

$$UpdateEvent = \frac{8.000.000}{(1)(999+1)(3999+1)(0+1)} = 2Hz = 0.5s$$

Finally, lines [48:58] enable the TIM3 and configure the PD2 pin (which corresponds to the TIM3_-ETR2 pin) as input source.



Read Carefully

It is important to remark that the GPIO port D must be enabled, before we can use it as clock source for TIM3, by using the `__GPIOD_CLK_ENABLE()` macro. The same applies even to TIM3, which is enabled by using the `__TIM3_CLK_ENABLE()`: **this is required because the external clocks are not directly feeding the prescaler, but they are first synchronized with the APBx clock through dedicated logical blocks**.

11.3.1.2 External Clock Mode 1

STM32 *general purpose* and *advanced* timers can be configured to work in *master* or *slave* mode[21]. When configured to act as a *slave*, a timer can be fed by internal ITR0, ITR1, ITR2 and ITR3 lines, an external clock connected to the ETR1 pin or from other clock sources connected to TI1FP1 and TI2FP2 sources, which correspond to Channel 1 and 2 input pins. This working mode is called *External Clock Mode 1*.

[20] This HSI frequency is related to an STM32F072RB. In other STM32 MCUs the HSI speed is 16MHz. Please, consult the datasheet of your MCU and arrange the math accordingly.

[21] As we will see next, a timer can be configured to work in *master* and *slave* mode at the same time.



The *External Clock Mode 1* and *2* are rather confusing for all novices of the STM32 platform. Both modes are a way to clock a timer using an external clock source, but the first one is achieved by configuring the timer in *slave* mode (it is indeed a form of "triggering"), while the second one is obtained by simply selecting a different clock source. I do not know the origin of this nomenclature, and what are the practical effects of this distinction. However, it is important to remark here that the ways to configure a timer in ETR1 or ETR2 mode are completely different, as we will see in the next example.



Looking to **Figure 11.16** we can see that the TI1FP1 and TI2FP2 inputs are nothing more than the TI1 and TI2 input channels of a timer after the input filter has been applied.

To configure a timer in *slave* mode we use the function `HAL_TIM_SlaveConfigSynchro()` and an instance of the `struct TIM_SlaveConfigTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t  SlaveMode;         /* Slave mode selection */
  uint32_t  InputTrigger;      /* Input Trigger source */
  uint32_t  TriggerPolarity;   /* Input Trigger polarity */
  uint32_t  TriggerPrescaler; /* Input trigger prescaler */
  uint32_t  TriggerFilter;     /* Input trigger filter */
} TIM_SlaveConfigTypeDef;
```

- `SlaveMode`: when a timer is configured in *slave* mode, it can be clocked/triggered by several sources. This field can assume a value from **Table 11.10**. This paragraph is about the `TIM_SLAVEMODE_EXTERNAL1` mode.
- `InputTrigger`: defines the source that triggers/clocks the timer configured in *slave* mode. It can assume a value from **Table 11.11**
- `TriggerPolarity`: indicates the polarity of the trigger/clock source. It can assume a value from the **Table 11.12**.
- `TriggerPrescaler`: specifies the prescaler for the external clock source. It can assume a value from the **Table 11.13**. By default, the `TIM_TRIGGERPRESCALER_DIV1` value is selected.
- `TriggerFilter`: this 4-bit field defines the frequency used to sample the external clock/trigger signal connected to input pin and the length of the digital filter applied to it. The digital filter is made of an event counter in which N consecutive events are needed to validate a transition on the output. Refer to the datasheet of your MCU about how the f_{DTS} (*Dead-Time Signal*) is computed. By default, the filter is disabled.

Table 11.10: Available *slave modes* for *general purpose* and *advanced* timers

Slave modes	Working	Description
TIM_SLAVEMODE_DISABLE	**Disabled**	The *slave* mode is disabled (default value)
TIM_SLAVEMODE_RESET	**Trigger**	Rising edge of the selected trigger input (TRGI) reinitializes the counter and generates an update of the registers
TIM_SLAVEMODE_GATED	**Trigger**	The counter clock is enabled when the trigger input (TRGI) is high. The counter stops (but is not reset) as soon as the trigger becomes low. Both start and stop of the counter are controlled
TIM_SLAVEMODE_TRIGGER	**Trigger**	The counter starts at a rising edge of TRGI (but it is not reset). Only the start of the counter is controlled
TIM_SLAVEMODE_EXTERNAL1	**Clock**	Rising edges of the selected TRGI clock the counter
TIM_SLAVEMODE_COMBINED_RESETTRIGGER[22]	**Trigger**	Rising edge of the selected trigger input (TRGI) reinitializes the counter, generates an update of the registers and starts the counter

Table 11.11: Available trigger/clock sources for a timer working in *slave* mode

Trigger/clock source	Description
TIM_TS_ITR0	Trigger/clock source is the ITR0 line (which is internally connected to a master timer)
TIM_TS_ITR1	Trigger/clock source is the ITR1 line (which is internally connected to a master timer)
TIM_TS_ITR2	Trigger/clock source is the ITR2 line (which is internally connected to a master timer)
TIM_TS_ITR3	Trigger/clock source is the ITR3 line (which is internally connected to a master timer)
TIM_TS_TI1F_ED	Trigger/clock source is the TIM_TS_TI1F_ED line
TIM_TS_TI1FP1	Trigger/clock source is the TIM_TS_TI1FP1 line that corresponds to the Channel 1
TIM_TS_TI2FP2	Trigger/clock source is the TIM_TS_TI2FP2 line that corresponds to the Channel 2
TIM_TS_ETRF	Trigger/clock source is the ETR1 pin
TIM_TS_NONE	No external clock/trigger source

Table 11.12: Available trigger/clock polarity modes for a timer working in *slave* mode

Trigger/clock polarity mode	Description
TIM_TRIGGERPOLARITY_INVERTED	This is used when the external clock source is ETR1. ETR1 is noninverted, active at high level or rising edge
TIM_TRIGGERPOLARITY_NONINVERTED	This is used when the external clock source is ETR1. ETR1 is inverted, active at low level or falling edge
TIM_TRIGGERPOLARITY_RISING	Polarity for TIxFPx or TI1_ED trigger sources. The timer is synchronized on the rising edge of the external trigger source

[22]This mode is available only in some STM32F3 MCUs.

Table 11.12: Available trigger/clock polarity modes for a timer working in *slave* mode

Trigger/clock polarity mode	Description
`TIM_TRIGGERPOLARITY_FALLING`	Polarity for TIxFPx or TI1_ED trigger sources. The timer is synchronized on the falling edge of the external trigger source
`TIM_TRIGGERPOLARITY_BOTHEDGE`	Polarity for TIxFPx or TI1_ED trigger sources. The timer is synchronized on rising and falling edges of the external trigger source (this will increase the sampled frequency)

Table 11.13: Available trigger/clock prescaler modes for a timer working in *slave* mode

External clock prescaler mode	Description
`TIM_TRIGGERPRESCALER_DIV1`	No prescaler used
`TIM_TRIGGERPRESCALER_DIV2`	Capture performed once every 2 events
`TIM_TRIGGERPRESCALER_DIV4`	Capture performed once every 4 events
`TIM_TRIGGERPRESCALER_DIV8`	Capture performed once every 8 events

When the *External Clock Source Mode 1* is selected, the formula to compute the frequency of *update events* becomes:

$$UpdateEvent = \frac{TRGI_{clock}}{(Prescaler + 1)(Period + 1)(RepetitionCounter + 1)} \quad [3]$$

where $TRGI_{clock}$ is the frequency of the clock source connected to the ETR1 pin, the frequency of the internal/external trigger clock source connected to internal lines ITR0..ITR3 or the frequency of signal connected to external channels TI1FP1..T2FP2.

So, let us recap what seen until now:

- a timer can be clocked by an external source when working *only in master mode*[23] by connecting this source to the ETR2 pin;
- if the timer is working in *slave mode*, then it can be clocked by a signal connected to the ETR1 pin, by any trigger source connected to the internal lines ITR0…ITR2 (hence, the clock source can be only another timer) or by an input signal connected to the timer channels TI1 and TI2, which becomes TI1FP1 and TI2FP2 if the input filtering stage is activated.

Let us build another example that shows how to use an external clock source for the TIM3 timer. The example consists in routing the *Master Clock Output* (MCO) pin to the TI2FP2 pin (that is, the second channel of TIM3 timer), which in a Nucleo-F072RB corresponds to PA7 pin. This can be easily done by using the Morpho connectors, as shown in **Figure 11.5** (for your Nucleo, use CubeMX tool to identify both MCO and TI2FP2 pins).

[23] As we will discover later, the master/slave mode of a timer is not exclusively: a timer can be configured to work as a master and slave at the same time.

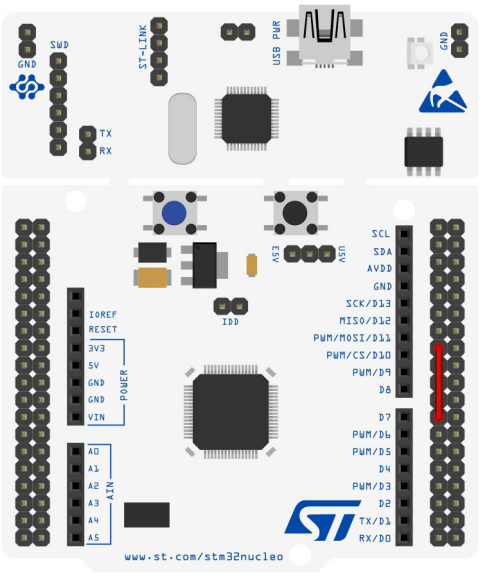


Figure 11.5: How to route the `MCO` pin to the TI2FP2 pin in a Nucleo-F072RB board

The MCO pin is enabled and connected to the HSI clock source, as seen in the previous example. The following code shows the most relevant parts of the example.

Filename: `Core/Src/main-ex4.c`

```
24 void MX_TIM3_Init(void) {
25   TIM_SlaveConfigTypeDef sSlaveConfig;
26
27   htim3.Instance = TIM3;
28   htim3.Init.Prescaler = 999;
29   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
30   htim3.Init.Period = 3999;
31   htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
32   HAL_TIM_Base_Init(&htim3);
33
34   sSlaveConfig.SlaveMode = TIM_SLAVEMODE_EXTERNAL1;
35   sSlaveConfig.InputTrigger = TIM_TS_TI2FP2;
36   sSlaveConfig.TriggerPolarity = TIM_TRIGGERPOLARITY_RISING;
37   sSlaveConfig.TriggerFilter = 0;
38   HAL_TIM_SlaveConfigSynchro(&htim3, &sSlaveConfig);
39
40   HAL_NVIC_SetPriority(TIM3_IRQn, 0, 0);
41   HAL_NVIC_EnableIRQ(TIM3_IRQn);
42 }
43
44 void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base) {
45   GPIO_InitTypeDef GPIO_InitStruct;
46   if(htim_base->Instance==TIM3)  {
47     /* Peripheral clock enable */
48     __HAL_RCC_TIM3_CLK_ENABLE();
```

```
49      __HAL_RCC_GPIOA_CLK_ENABLE();
50
51      /**TIM3 GPIO Configuration
52      PA7     ------> TIM3_CH2
53      */
54      GPIO_InitStruct.Pin = GPIO_PIN_7;
55      GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
56      GPIO_InitStruct.Pull = GPIO_NOPULL;
57      GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
58      GPIO_InitStruct.Alternate = GPIO_AF1_TIM3;
59      HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
60  }
```

Lines [34:38] configure TIM3 in *slave* mode. The input trigger source is set to TI2FP2, and the timer is synchronized to the rising edge of the input signal. Finally, lines [54:59] configure the PA7 as input pin for the second channel of TIM3.

11.3.1.3 Using CubeMX to Configure the Source Clock of a *General Purpose* Timer

Configuring the clock source of a *general purpose* timer can be a nightmare, especially for novices of the STM32 platform. CubeMX can simplify this process, even if a good understanding of master/slave modes and ETR1 and ETR2 modes is required.

To configure the timer in *External Clock Mode 2* it is sufficient to select ETR2 as clock source from the *Configuration* pane, as shown in **Figure 11.6**.

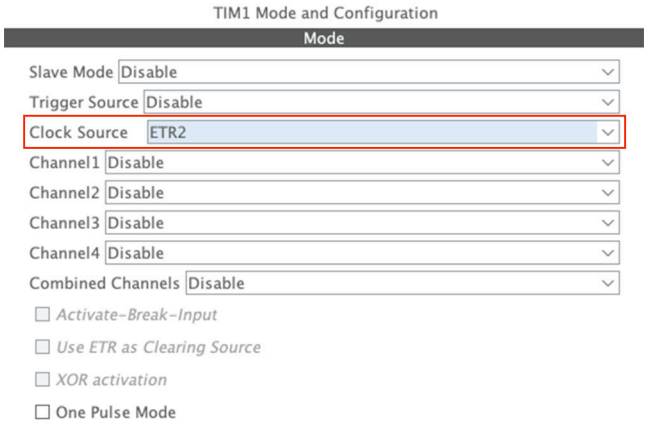


Figure 11.6: How to select the ETR2 mode from the IP pane

Once the clock source is selected, it is possible to set the external clock filter, polarity and prescaler from the configuration, as shown in **Figure 11.7**.

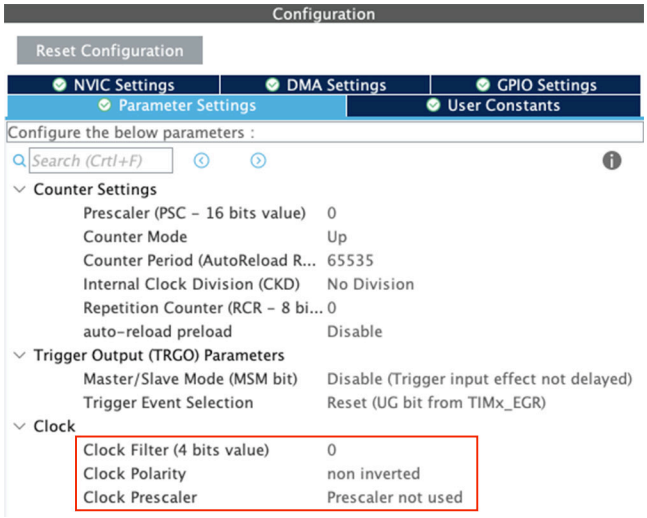


Figure 11.7: How to configure a timer working in ETR2 mode

To configure the timer in *External Clock Mode 1*, we have to select this mode from the **Slave** entry and then select the **Trigger Source** (which in this case is the clock source for the timer), as shown in **Figure 11.8**.



Figure 11.8: How to select the ETR1 mode from the *IP tree pane*

Once the clock source is selected, it is possible to set the other configuration parameters from the timer configuration dialog (not shown here).

11.3.2 Master/Slave Synchronization Modes

Once a timer operates in *master* mode it can feed another timer configured in *slave* mode through a dedicated output line, called *Trigger Output* (TRGO)[24], connected to the internal dedicated lines called ITR0, ITR1, ITR2 and ITR3. The *master* timer can both provide the clock source (and hence act as a first order prescaler - this is what we have studied in the previous paragraph) or trigger the *slave* timer.

These *Internal Trigger (ITR)* lines (ITR0, ITR1, ITR2 and ITR3) are precisely internal to the chip, and each line is hardwired between two defined timers. For example, in an STM32F072 MCU the TIM1

[24]Some STM32 microcontrollers, notably STM32F3 ones, provide two independent trigger lines, named TRGO1 and TRGO2. This case is not shown in this book.

TRGO line is connected to the ITR0 line of TIM2 timer, as shown in **Figure 11.9**.

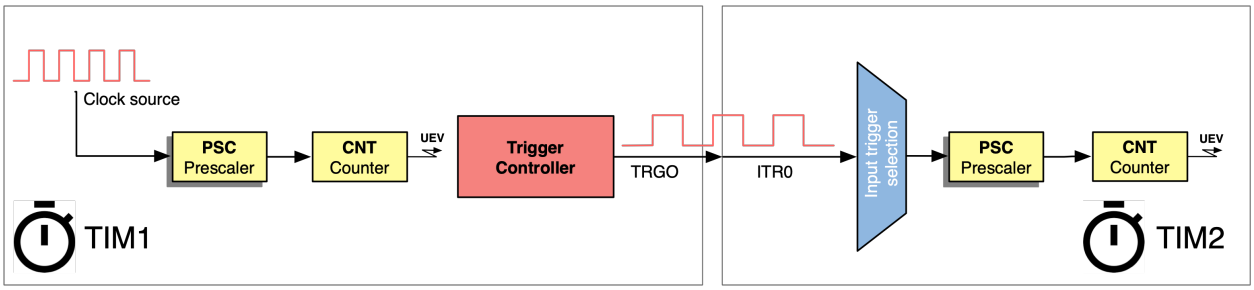


Figure 11.9: The TIM1 can fed the TIM2 timer through the ITR0 line

A timer configured as *slave* can also simultaneously act as *master* for another timer, allowing to create complex networks of timers. For example, the **Figure 11.10** shows how timers can be connected in cascade, while **Figure 11.11** shows how timers can form hierarchical structures using combinations of master/slave modes. Note that TIM1, TIM2 and TIM3 are internally interconnected through the same ITR0 line. This allows to synchronize several timers upon the same event (reset, enable, update, etc.).

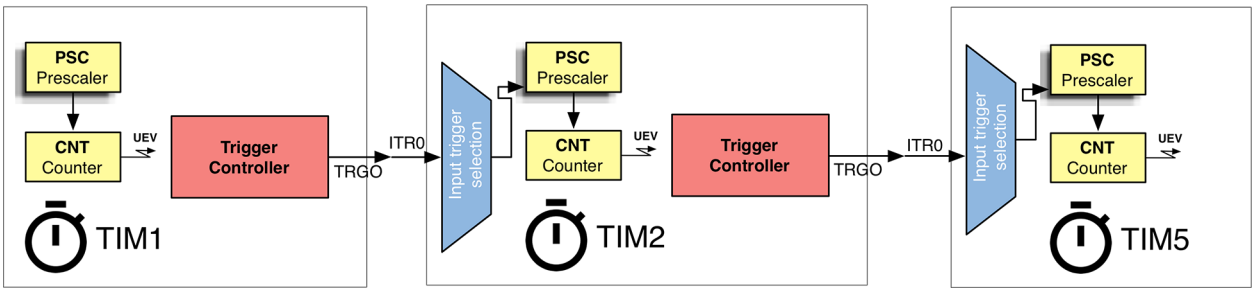


Figure 11.10: The combination of master/slave modes allows to configure timers in cascade

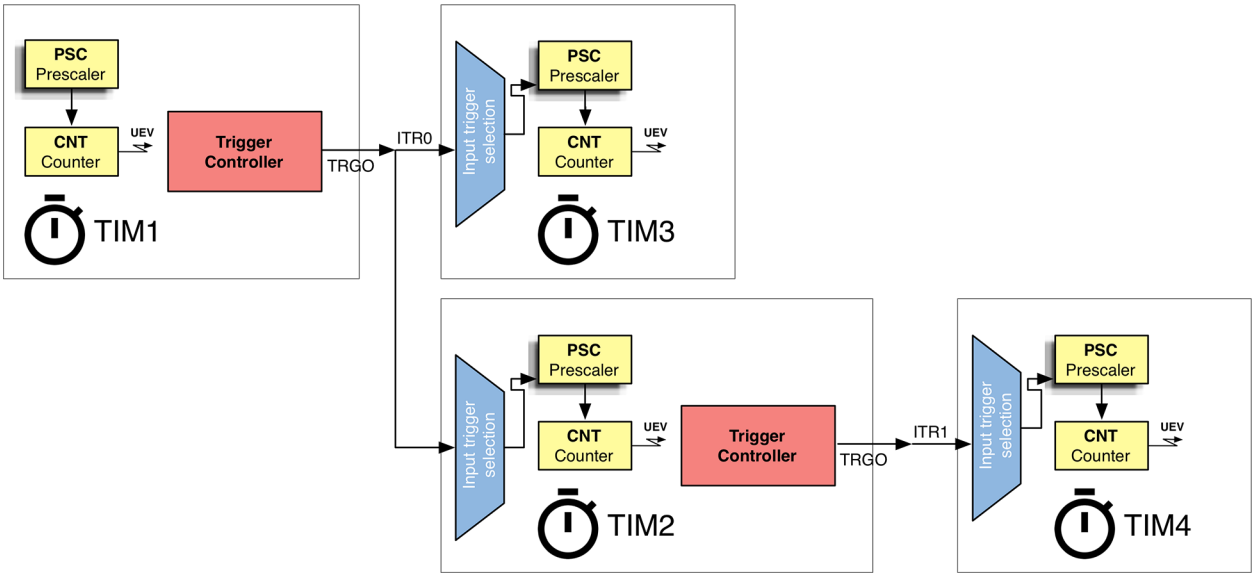


Figure 11.11: The combination of master/slave modes allows to configure timers in a hierarchical structure

To configure a timer in *master* mode we use the function `HAL_TIMEx_MasterConfigSynchronization()` and an instance of the `struct TIM_MasterConfigTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t  MasterOutputTrigger;   /* Trigger output (TRGO) selection */
  uint32_t  MasterSlaveMode;       /* Master/slave mode selection */
} TIM_MasterConfigTypeDef;
```

- `MasterOutputTrigger`: specifies the behaviour of the TRGO output and it can assume a value from **Table 11.14**.
- `MasterSlaveMode`: it is used to enable/disable the master/slave mode of a timer. It can assume the values `TIM_MASTERSLAVEMODE_ENABLE` or `TIM_MASTERSLAVEMODE_DISABLE`.

Table 11.14: Available trigger/clock sources for a timer working in *slave* mode

Timer *master* mode selection	Description
`TIM_TRGO_RESET`	The TRGO signal is generated when the UG bit of the TIMx->EGR register is set. More about this in paragraph 11.3.3
`TIM_TRGO_ENABLE`	The TRGO signal is generated when master timer is enabled. It is useful to start several timers at the same time or to control a window in which a slave timer is enabled
`TIM_TRGO_UPDATE`	The update event is selected as trigger output (TRGO). For instance a master timer can then be used as a prescaler for a slave timer (we have studied this mode in paragraph 11.3.1.2)
`TIM_TRGO_OC1`	The trigger output send a positive pulse as soon as a capture or a compare match occurred

Table 11.14: Available trigger/clock sources for a timer working in *slave* mode

Timer *master* mode selection	Description
TIM_TRGO_OC1REF	The trigger output send a positive pulse as soon as a capture or a compare match occurred on Channel 1
TIM_TRGO_OC2REF	The trigger output send a positive pulse as soon as a capture or a compare match occurred on Channel 2
TIM_TRGO_OC3REF	The trigger output send a positive pulse as soon as a capture or a compare match occurred on Channel 3
TIM_TRGO_OC4REF	The trigger output send a positive pulse as soon as a capture or a compare match occurred on Channel 4

Let us see an example that shows how to configure TIM1 and TIM3 in cascade mode, with TIM1 as master for TIM3 timer. TIM1 is used as clock source for TIM3 through the ITR0 line. Moreover, the TIM1 is configured so that it starts counting upon an external event on its TI1FP1 line, which in a Nucleo-F072 corresponds to PA8 pin: TIM1 starts counting when the PA8 pin goes high, and then it feeds the TIM3 timer through the ITR0 line.

Filename: `Core/Src/main-ex5.c`

```
12 int main(void) {
13   HAL_Init();
14
15   Nucleo_BSP_Init();
16   MX_TIM1_Init();
17   MX_TIM3_Init();
18
19   HAL_TIM_Base_Start_IT(&htim3);
20
21   while (1);
22 }
23
24 void MX_TIM1_Init(void) {
25   TIM_ClockConfigTypeDef sClockSourceConfig;
26   TIM_MasterConfigTypeDef sMasterConfig;
27   TIM_SlaveConfigTypeDef sSlaveConfig;
28
29   htim1.Instance = TIM1;
30   htim1.Init.Prescaler = 47999;
31   htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
32   htim1.Init.Period = 249;
33   htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
34   htim1.Init.RepetitionCounter = 0;
35   HAL_TIM_Base_Init(&htim1);
36
37   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
38   HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig);
39
```

```
  sSlaveConfig.SlaveMode = TIM_SLAVEMODE_TRIGGER;
  sSlaveConfig.InputTrigger = TIM_TS_TI1FP1;
  sSlaveConfig.TriggerPolarity = TIM_TRIGGERPOLARITY_RISING;
  sSlaveConfig.TriggerFilter = 15;
  HAL_TIM_SlaveConfigSynchron(&htim1, &sSlaveConfig);

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_ENABLE;
  HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig);
}

void MX_TIM3_Init(void) {
  TIM_SlaveConfigTypeDef sSlaveConfig;

  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 0;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 1;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  HAL_TIM_Base_Init(&htim3);

  sSlaveConfig.SlaveMode = TIM_SLAVEMODE_EXTERNAL1;
  sSlaveConfig.InputTrigger = TIM_TS_ITR0;
  HAL_TIM_SlaveConfigSynchro(&htim3, &sSlaveConfig);

  HAL_NVIC_SetPriority(TIM3_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(TIM3_IRQn);
}

void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base) {
  GPIO_InitTypeDef GPIO_InitStruct;
  if(htim_base->Instance==TIM3) {
    __HAL_RCC_TIM3_CLK_ENABLE();
  }

  if(htim_base->Instance==TIM1) {
    __HAL_RCC_TIM1_CLK_ENABLE();

    GPIO_InitStruct.Pin = GPIO_PIN_8;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF2_TIM1;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
  }
}
```

Lines [29:38] configure TIM1 to be clocked from the internal APB1 bus. Lines [40:44] configure TIM1 in *slave* mode, so that it starts counting when the TI1FP1 line goes high (that is, it is triggered). PA8 GPIO is configured accordingly in lines [74:79] (it is configured as `GPIO_AF2_TIM1`). Take note that the internal pull-down resistor is activated in line 76: this prevents that a floating input could accidentally trigger the timer. For the same reason, the `TriggerFilter` is set to the maximum level at line 43 (if you try to set it to zero, you will notice that it is easy to trigger accidentally the timer, even by simply touching the wire connected to PA8 pin).

Lines [46:48] configure TIM1 to work also in *master* mode. The timer will trigger its internal line (which is connected to the ITR0 line of TIM3) every time the *update event* is generated. Finally, lines [61:63] configure the TIM3 in *External Clock Mode 1*, selecting the ITR0 line as source clock.



Note that the, in order to have LD2 LED blinking every 500ms (2Hz), the TIM1 period is set to 249[25], which causes that the update frequency of TIM1 is 4Hz. This is required because, applying the equation [3], we have that:

$$UpdateEvent = \frac{4Hz}{(0+1)(1+1)(0+1)} = 2Hz = 0.5s$$

Remember that the `Period` field cannot be set to zero.

To trigger TIM1 you have to connect the PA8 pin to a +3V3 source. **Figure 11.12** shows how to connect it in a Nucleo-F072.

Finally, note that we do not call the `HAL_TIM_Base_Start()` function for the TIM1 timer (see the `main()` routine), because the timer is started upon the trigger event generated on Channel 1 (that is, when we connect the PA8 pin to the +3V3 source).

[25]Clearly, that prescaler value is referred to an STM32F072RB MCU running at 48MHz. For your Nucleo, check the book examples for the right prescaler setting.

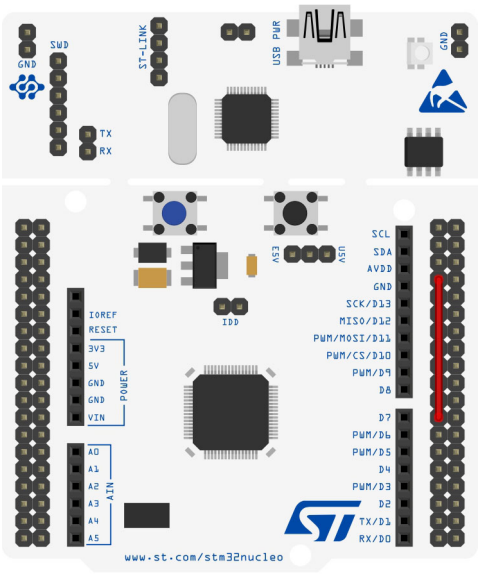


Figure 11.12: How to connect the TI2FP2 pin to AVDD pin in a Nucleo-F072R8 board

11.3.2.1 Enable Trigger-Related Interrupts

When a timer works in *slave* mode, the timer IRQ is raised, if enabled, every time the specified trigger event occurs. For example, when the *master* clock triggers due to an update event, the IRQ of the *slave* timer is faired and we can be notified of this by defining the callback:

```
void HAL_TIM_TriggerCallback(TIM_HandleTypeDef *htim) {
  ...
}
```

By default, the `HAL_TIM_Base_Start_IT()` does not enable this type of interrupt. We have to use the function `HAL_TIM_SlaveConfigSynchron_IT()`, instead of the function `HAL_TIM_SlaveConfigSynchron()`. Obviously, the corresponding timer's ISR must be defined, and the function `HAL_TIM_IRQHandler()` has to be called from it.

11.3.2.2 Using CubeMX to Configure the Master/Slave Synchronization

To configure a timer in *slave* mode from CubeMX, it is sufficient to select the desired trigger mode (**Reset Mode**, **Gated Mode**, **Trigger Mode**) from the *IP Pane tree* (**Slave mode** combo-box), and then select the **Trigger Source**, as shown in **Figure 11.13**. Remember that a timer configured in *slave* mode, and not working in *External Clock Mode 1*, must be clocked from the internal clock or by the ETR2 clock source.

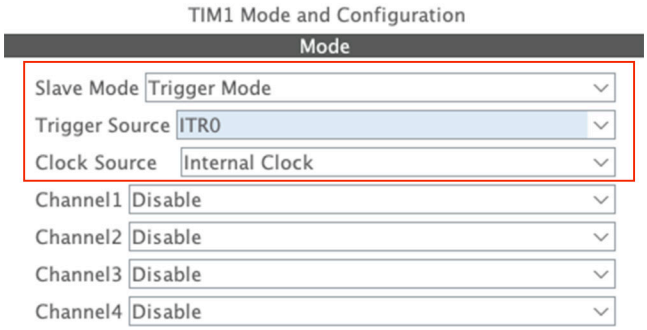


Figure 11.13: How to configure a timer in *slave* mode

Instead, to enable the *master* mode, we have to select this mode from the timer configuration view, as shown in **Figure 11.14**. Once the *master* mode is selected, it is possible to select the TRGO source event.

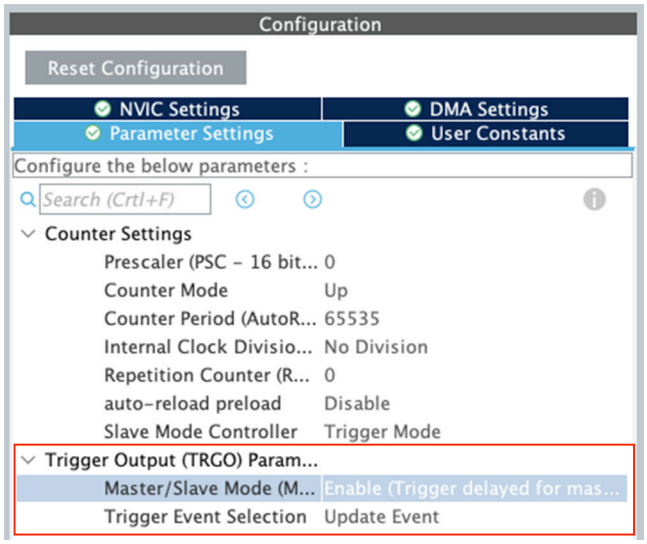


Figure 11.14: How to configure a timer in *master* mode

11.3.3 Generate Timer-Related Events by Software

Timers usually generate events when a given condition is met. For example, they generate the *Update Event* (UEV) when the counter register (CNT) matches the `Period` value. However, we can force a timer to generate a particular event by software. Every timer provides a dedicated register, named *Event Generator* (`EGR`). Some bits of this register are used to fire a timer-related event. For example, the first bit, named *Update Generator* (`UG`), allows to generate a UEV event when set. This bit is automatically cleared once the event is generated.

To generate events by software, the HAL provides the following function:

```
HAL_StatusTypeDef HAL_TIM_GenerateEvent(TIM_HandleTypeDef *htim, uint32_t EventSource);
```

which accepts the pointer to the timer handle and the event to generate. The `EventSource` parameter can assume one value from **Table 11.15**.

Table 11.15: Software-triggerable events

Event soure	Description
`TIM_EVENTSOURCE_UPDATE`	Timer update Event source
`TIM_EVENTSOURCE_CC1`	Timer Capture Compare 1 Event source
`TIM_EVENTSOURCE_CC2`	Timer Capture Compare 2 Event source
`TIM_EVENTSOURCE_CC3`	Timer Capture Compare 3 Event source
`TIM_EVENTSOURCE_CC4`	Timer Capture Compare 4 Event source
`TIM_EVENTSOURCE_COM`	Timer COM event source
`TIM_EVENTSOURCE_TRIGGER`	Timer Trigger Event source
`TIM_EVENTSOURCE_BREAK`	Timer Break event source

The `TIM_EVENTSOURCE_UPDATE` plays two important roles. The first one is related to the way the `Period` register (that is the TIMx->ARR register) is updated when the timer is running. By default, the content of the ARR register is transferred to the internal *shadow* register when the `TIM_EVENTSOURCE_UPDATE` event is generated, unless the timer is differently configured. More about this later.

The `TIM_EVENTSOURCE_UPDATE` event is also useful when the TRGO output of a timer configured as *master* is set in `TIM_TRGO_RESET` mode: in this case, the *slave* timer will be triggered only if the `TIMx->EGR` register is used to generate the `TIM_EVENTSOURCE_UPDATE` event (that is, the `UG` bit is set).

The following code shows how to software event generation works (the example is based on an STM32F401RE MCU). TIM3 and TIM4 are two timers configured in *master* and *slave* mode respectively. TIM4 is configured to work in ETR1 mode (that is, it is clocked by the *master* timer). TIM3 is configured to trigger the TRGO output (which is internally connected to the ITR2 line) when the `UG` bit of the `TIM3->EGR` register is set. Finally, we generate the UEV event manually every 200ms from the `main()` routine.

```
int main(void) {
  ...
  while (1) {
    HAL_TIM_GenerateEvent(&htim3, TIM_EVENTSOURCE_UPDATE);
    HAL_Delay(200);
  }
  ...
}

void MX_TIM3_Init(void){
  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;
```

```
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 65535;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 120;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  HAL_TIM_Base_Init(&htim3);

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig);

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_ENABLE;
  HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig);
}

void MX_TIM4_Init(void) {
  TIM_SlaveConfigTypeDef sSlaveConfig;

  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 0;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 1;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  HAL_TIM_Base_Init(&htim4);

  sSlaveConfig.SlaveMode = TIM_SLAVEMODE_EXTERNAL1;
  sSlaveConfig.InputTrigger = TIM_TS_ITR2;
  HAL_TIM_SlaveConfigSynchro_IT(&htim4, &sSlaveConfig);
}
```

11.3.4 Counting Modes

At the beginning of this chapter we have seen that a basic timer counts from zero to a given `Period` value. *General purpose* and *advanced* timers can count in other different ways, as reported in **Table 11.4**. The **Figure 11.15** shows the three main counting modes.

When a timer counts in `TIM_COUNTERMODE_DOWN` mode, it starts from the `Period` value and counts down to zero: when the counter reaches the end, the timer IRQ is raised and the UIF flag is set (that is, the *update event* is generated and the `HAL_TIM_PeriodElapsedCallback()` is called by the HAL).

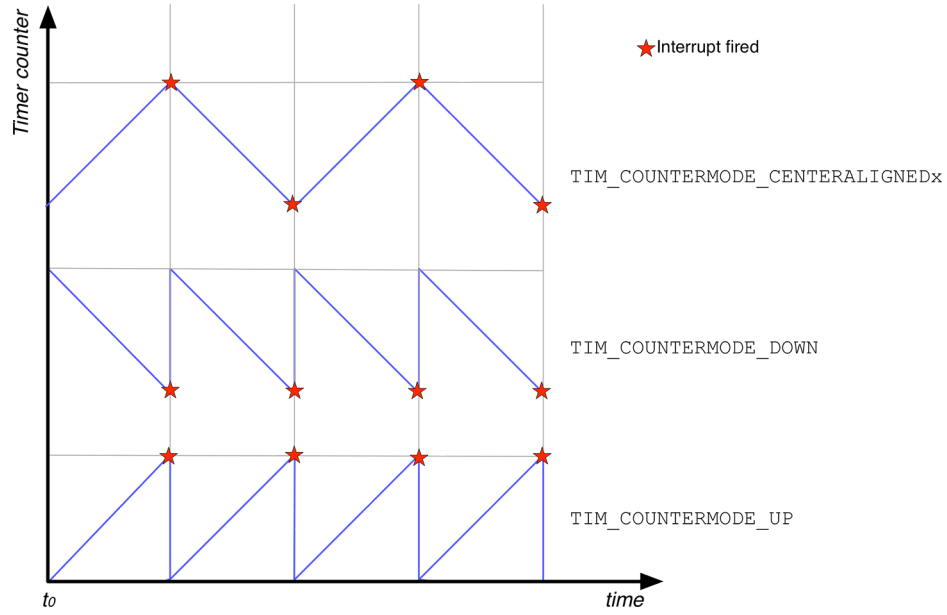


Figure 11.15: The three major counting modes of a *general purpose* timer

Instead, when a timer counts in `TIM_COUNTERMODE_CENTERALIGNED` mode, it starts counting from zero up to `Period` value: this causes that the timer IRQ is raised and the UIF flag is set (that is, the *update event* is generated and the `HAL_TIM_PeriodElapsedCallback()` is called by the HAL). Then the timer starts counting down to zero and another *update event* is generated (as well as the corresponding IRQ).

11.3.5 Input Capture Mode

General purpose timers have not been designed to be used as timebase generators. Even if it is perfectly possible to use them to accomplish this job, other timers like *basic* ones and the *SysTick* timer can be used to carry out this task. *General purpose* timers offer much more advanced capabilities, which can be used to drive other important time-related activities.

The **Figure 11.16** shows the structure of the input channels in a *general purpose* timer[26]. As you can see, each input is connected to an edge detector, which is also equipped with a filter used to "debounce" the input signal. The output of the edge detector goes into a source multiplexer (IC1, IC2, etc.). This allows to "remap" the input channels if a given I/O is allocated to another peripheral. Finally, a dedicated prescaler allows to "slow down" the frequency of the input signal, in order to match the timer running frequency if this cannot be lowered, as we will see in a while.

[26]Some *general purpose* timers (for example, TIM14) have less input channels and hence a simplified input stage structure. Refer to the reference manual for your MCU to know the exact structure of the timer you are going to use.

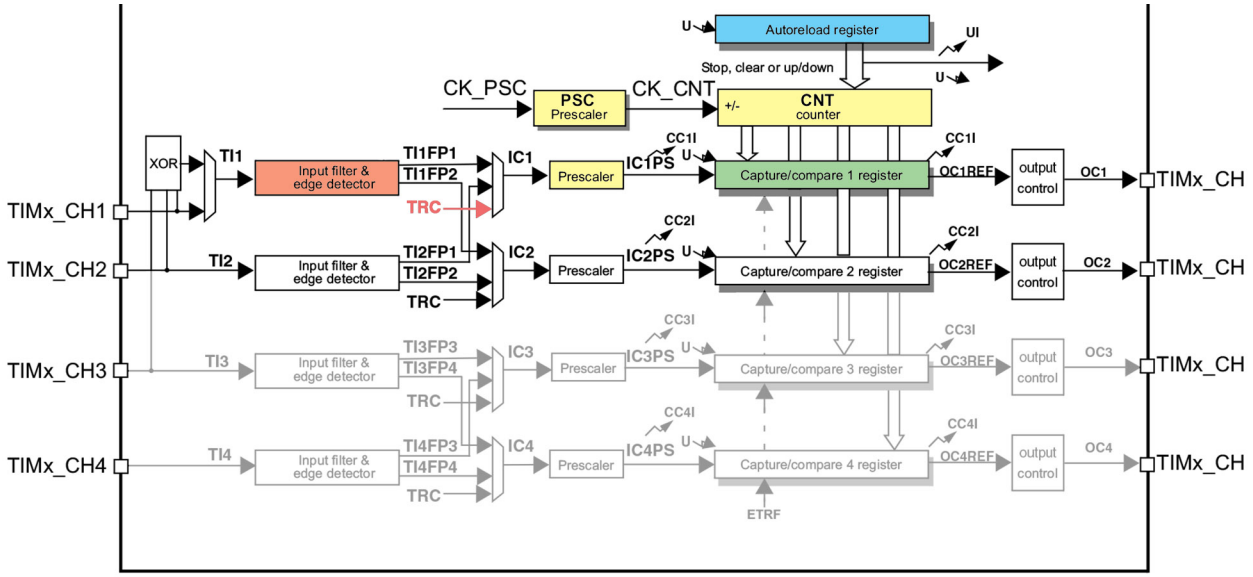


Figure 11.16: The structure of the input channel in a *general purpose* timer

The *input capture* mode offered by *general purpose* and *advanced* timers allows to compute the frequency of external signals applied to each one of the 4 channels that these timers provide. And the capture is performed independently for each channel.

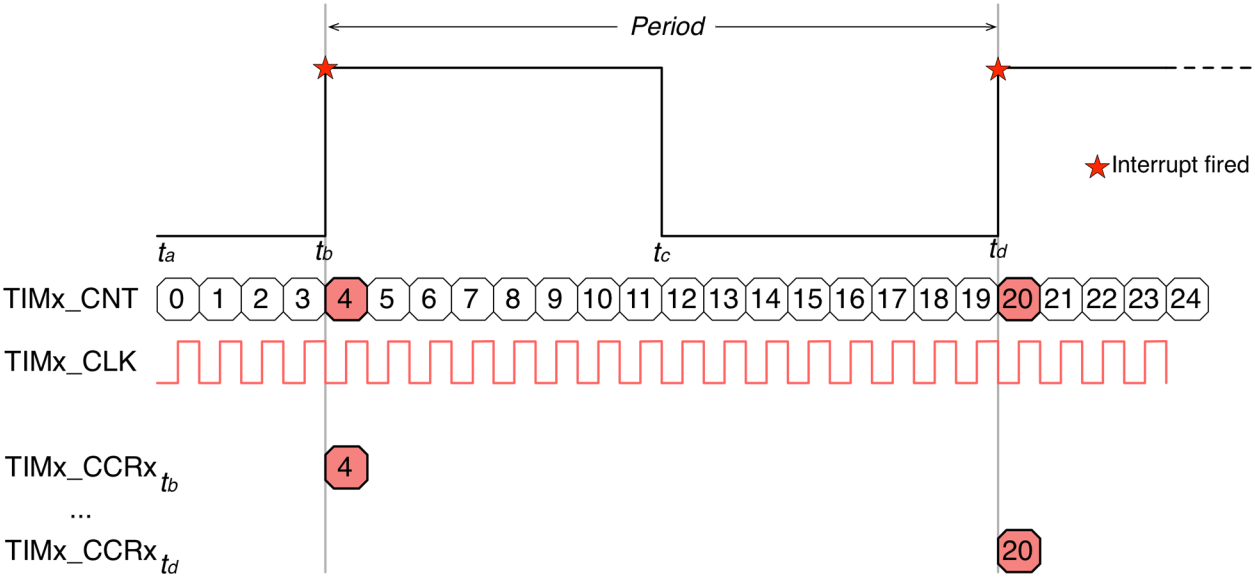


Figure 11.17: The capture process of an external signal feeding one of the timer channels

The **Figure 11.17** shows how the capture process works. **TIMx** is a timer, configured to work at a given **TIMx_CLK** clock frequency[27]. This means that it increments the **TIMx_CNT** register up to the `Period` value every $\frac{1}{TIMx_CLK}$ seconds. Supposing that we apply a square wave signal to one of the timer channels, and supposing that we configure the timer to trigger at every rising edge

[27]The timer clock frequency is independent from the way the timer works (in this case, input capture mode). As seen in the previous paragraphs, the timer clock depends on the bus frequency or the external clock source and on the related prescaler settings.

of the input signal, we have that the **TIMx_CCRx**[28] register will be updated with the content of the **TIMx_CNT** register at every detected transition. When this happens, the timer will generate a corresponding interrupt or a DMA request, allowing to keep track of the counter value.

To get the external signal period, two consecutive captures are needed. The period is calculated by subtracting these two values, CNT_0(the value *4* in **Figure 11.17**) and CNT_1(the value *20* in **Figure 11.17**), and using the following formula:

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

where:

$Capture = CNT_1 - CNT_0 \;\; if \; CNT_0 < CNT_1$
$Capture = (TIMx_Period - CNT_0) + CNT_1 \;\; if \; CNT_0 > CNT_1$

$CH_{Prescaler}$ is a further prescaler that can be applied to the input channel and $Polarity_{Index}$ is equal to 1 if the channel is configured to trigger on rising or falling edge of the input signal, or it is equal to 2 if both the edges are sampled.

Another relevant condition is that the UEV frequency should be lower than the sampled signal frequency. The reason why this is important is evident: if the timer runs faster that the sampled signal, then it will overflow (that is, it runs out the `Period` counter) before it can sample the signal edges (see **Figure 11.18**). For this reason, it usually convenient to set the `Period` value to the maximum, and increase the `Prescaler` factor to lower the counting frequency.

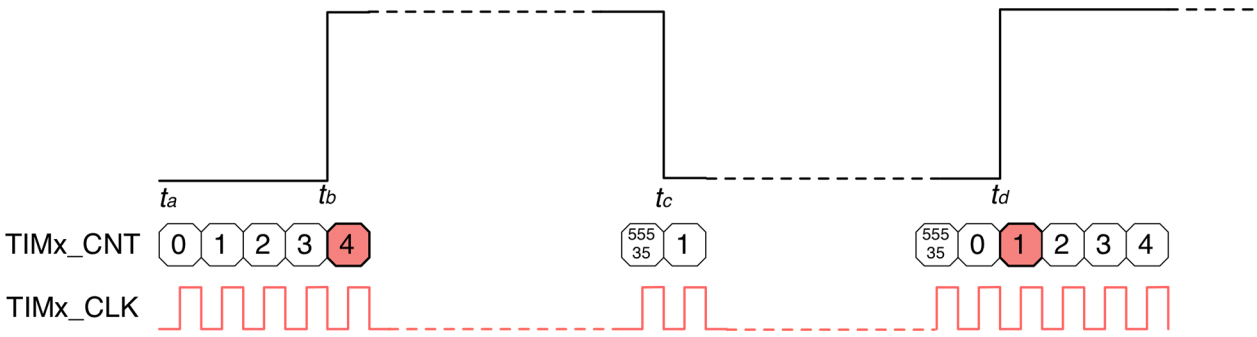


Figure 11.18: If the timer runs faster than the sample signal, then it overflow before the two rising edges are dected

To configure the input channels we use the function `HAL_TIM_IC_ConfigChannel()` and an instance of the C `struct TIM_IC_InitTypeDef`, which is defined in the following way:

[28] CCR is acronym for *Capture Compare Register* and the *x* is the channel number.

```
typedef struct {
  uint32_t ICPolarity;    /* Specifies the active edge of the input signal. */
  uint32_t ICSelection;   /* Specifies the input. */
  uint32_t ICPrescaler;   /* Specifies the Input Capture Prescaler. */
  uint32_t ICFilter;      /* Specifies the input capture filter. */
} TIM_IC_InitTypeDef;
```

- `ICPolarity`: specifies the polarity of the input signal, and it can assume a value from **Table 11.16**.
- `ICSelection`: specifies the used input of the timer. It can assume a value from **Table 11.17**. It is possible to selectively remap input channels to different input sources, that is (IC1,IC2) are mapped to (TI2,TI1) and (IC3,IC4) are mapped to (TI4,TI3). Usually this is used to differentiate rising-edge from falling-edge captures for signals where the T_{on} is different from T_{off}. It is also possible to capture from the same internal channel, named TRC, connected to ITR0..ITR3 sources.
- `ICPrescaler`: configures the prescaler stage of a given input. It can assume a value from **Table 11.18**.
- `ICFilter`: this 4-bit field defines the frequency used to sample the external clock signal connected to TIMx_CHx pin and the length of the digital filter applied to it. It is useful to debounce the input signal. Refer to the datasheet of your MCU for more information.

Table 11.16: Available input capture polarity

Input capture polarity mode	Description
`TIM_ICPOLARITY_RISING`	The rising edge of the external signal is captured
`TIM_ICPOLARITY_FALLING`	The falling edge of the external signal is captured
`TIM_ICPOLARITY_BOTHEDGE`	The rising and falling edges of the external signal determine the capture period (this will increase the frequency of the sampled signal)

Table 11.17: Available input capture selection modes

Input capture selection mode	Description
`TIM_ICSELECTION_DIRECTTI`	TIM Input 1, 2, 3 or 4 is selected to be connected to IC1, IC2, IC3 or IC4, respectively
`TIM_ICSELECTION_INDIRECTTI`	TIM Input 1, 2, 3 or 4 is selected to be connected to IC2, IC1, IC4 or IC3, respectively.
`TIM_ICSELECTION_TRC`	TIM Input 1, 2, 3 or 4 is selected to be connected to TRC (Trigger line in **Figure 11.3** - TRC input highlighted in red in **Figure 11.16**)

Table 11.18: Available input prescaler modes

Input capture prescaler mode	Description
TIM_ICPSC_DIV1	No prescaler used
TIM_ICPSC_DIV2	Capture performed once every 2 events
TIM_ICPSC_DIV4	Capture performed once every 4 events
TIM_ICPSC_DIV8	Capture performed once every 8 events

Now it is the right time to see a practical example. We are going to rearrange the Example 2 of this chapter so that we sample the switching frequency of PA5 pin (the one connected to LD2 LED) through the Channel 1 of TIM3 timer (in an STM32F072 MCU this pin coincides with PA6 pin). We so configure the Channel 1 as input capture pin, and we configure it in DMA mode so that it triggers the `TIM_DMA_ID_CC1` request to automatically fill a temporary buffer that stores the value of the TIM3_CNT register when the rising edge of input signal is detected.

Before we analyze the `main()` function, it is best to give a look at the TIM3 initialization routines.

Filename: `Core/Src/main-ex6.c`

```
59  /* TIM3 init function */
60  void MX_TIM3_Init(void) {
61    TIM_IC_InitTypeDef sConfigIC;
62
63    htim3.Instance = TIM3;
64    htim3.Init.Prescaler = 999;
65    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
66    htim3.Init.Period = 65535;
67    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
68    HAL_TIM_IC_Init(&htim3);
69
70    sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
71    sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
72    sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
73    sConfigIC.ICFilter = 0;
74    HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_1);
75  }
76
77  void HAL_TIM_IC_MspInit(TIM_HandleTypeDef* htim_ic) {
78    GPIO_InitTypeDef GPIO_InitStruct;
79    if (htim_ic->Instance == TIM3) {
80      /* Peripheral clock enable */
81      __HAL_RCC_TIM3_CLK_ENABLE();
82
83      /**TIM3 GPIO Configuration
84       PA6     ------> TIM3_CH1
85       */
86      GPIO_InitStruct.Pin = GPIO_PIN_6;
87      GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
```

```
88      GPIO_InitStruct.Pull = GPIO_NOPULL;
89      GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
90      GPIO_InitStruct.Alternate = GPIO_AF1_TIM3;
91      HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
92
93      /* Peripheral DMA init*/
94      hdma_tim3_ch1_trig.Instance = DMA1_Channel4;
95      hdma_tim3_ch1_trig.Init.Direction = DMA_PERIPH_TO_MEMORY;
96      hdma_tim3_ch1_trig.Init.PeriphInc = DMA_PINC_DISABLE;
97      hdma_tim3_ch1_trig.Init.MemInc = DMA_MINC_ENABLE;
98      hdma_tim3_ch1_trig.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
99      hdma_tim3_ch1_trig.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
100     hdma_tim3_ch1_trig.Init.Mode = DMA_NORMAL;
101     hdma_tim3_ch1_trig.Init.Priority = DMA_PRIORITY_LOW;
102     HAL_DMA_Init(&hdma_tim3_ch1_trig);
103
104     /* Several peripheral DMA handle pointers point to the same DMA handle.
105      Be aware that there is only one channel to perform all the requested DMAs. */
106     __HAL_LINKDMA(htim_ic, hdma[TIM_DMA_ID_CC1], hdma_tim3_ch1_trig);
107   }
108 }
```

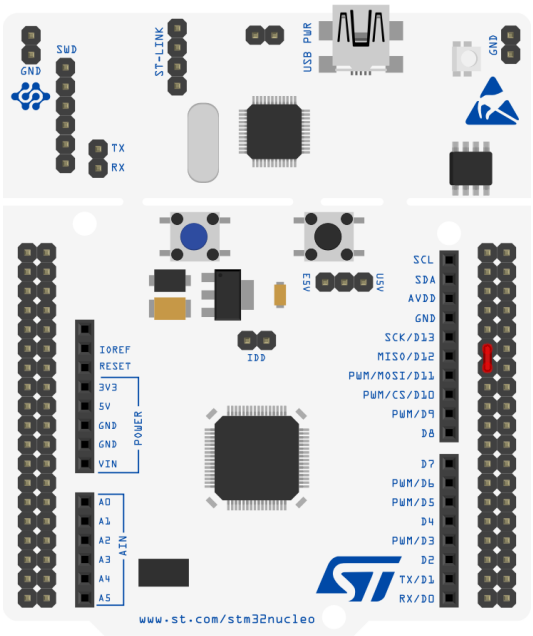


Figure 11.19: How to wire PA5 and PA6 pins in a Nucleo-F072RB

The `MX_TIM3_Init()` configures the TIM3 timer so that it runs at a frequency equal to ~0,732Hz. The first channel is then configured to trigger the capture event (TIM_DMA_ID_CC1) at every rising edge of the input signal. The `HAL_TIM_IC_MspInit()` then configures the hardware part (the PA6 pin connected to the TIM3 Channel 1) and the DMA descriptor used to configure the `TIM_DMA_ID_CC1`

request.



Here we have two things to note. First of all, the DMA is configured so that both the peripheral and memory data align are set to perform a 16-bit transfer, since the timer counter register is 16-bit wide. In those MCU where TIM2 and TIM5 timers have a counter register 32-bit wide, you need to setup the DMA to perform a word-aligned transfer. Next, since we are using the `HAL_TIM_IC_Init()` at line 69, the HAL is designed to call the function `HAL_TIM_IC_MspInit()` to perform low-level initializations, instead of the `HAL_TIM_Base_MspInit` one.

Filename: `Core/Src/main-ex6.c`

```
20  uint8_t  odrVals[] = { 0x0, 0xFF };
21  uint16_t captures[2];
22  volatile uint8_t  captureDone = 0;
23
24  int main(void) {
25    uint16_t diffCapture = 0;
26    char msg[30];
27
28    HAL_Init();
29
30    Nucleo_BSP_Init();
31    MX_DMA_Init();
32
33    MX_TIM3_Init();
34    MX_TIM6_Init();
35
36    HAL_DMA_Start(&hdma_tim6_up, (uint32_t) odrVals, (uint32_t) &GPIOA->ODR, 2);
37    __HAL_TIM_ENABLE_DMA(&htim6, TIM_DMA_UPDATE);
38    HAL_TIM_Base_Start(&htim6);
39
40    HAL_TIM_IC_Start_DMA(&htim3, TIM_CHANNEL_1, (uint32_t*) captures, 2);
41
42    while (1) {
43      if (captureDone != 0) {
44        if (captures[1] >= captures[0])
45          diffCapture = captures[1] - captures[0];
46        else
47          diffCapture = (htim3.Instance->ARR - captures[0]) + captures[1];
48
49        frequency = HAL_RCC_GetPCLK1Freq() / (htim3.Instance->PSC + 1);
50        frequency = (float) frequency / diffCapture;
51
52        sprintf(msg, "Input frequency: %.3f\r\n", frequency);
53        HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
54        while (1);
```

```
55        }
56      }
57  }
```

The most relevant part of the application is the `main()` function. We first initialize TIM6 timer (which is configured to run at 40Hz - this means that the PA5 pin is set HIGH every 50ms = 20Hz) using the `MX_TIM6_Init()` function and then we start it in DMA mode, as described so far in this chapter. Then we start TIM3 and we enable the DMA mode on the first channel, by using the `HAL_TIM_IC_Start_DMA()` function (line 40). The `captures` array is used to store the two consecutive captures acquired on the channel.

Lines [42:53] are the part where we compute the frequency of the external signal. When the two captures are performed, the global variable `captureDone` is set to 1 by the `HAL_TIM_IC_CaptureCallback()` callback function (not shown here), which is invoked at the end of the capture process. When this happens, we compute the frequency of the sample signal using the equation [4].

In order to make the example working properly, you need to wire PA5 and PA6 pins, as shown in **Figure 11.19**. Please, always check if in your Nucleo those pins are routed to that position on the Morpho connector.

11.3.5.1 Using CubeMX to Configure the Input Capture Mode

Thanks to CubeMX, it is easy to configure the input channels of a *general purpose* timer in the input capture mode. To bound one channel to the corresponding input (that is, IC1 to TI1), you have to select the **Input capture direct mode** for the desired channel, as shown in **Figure 11.20**.

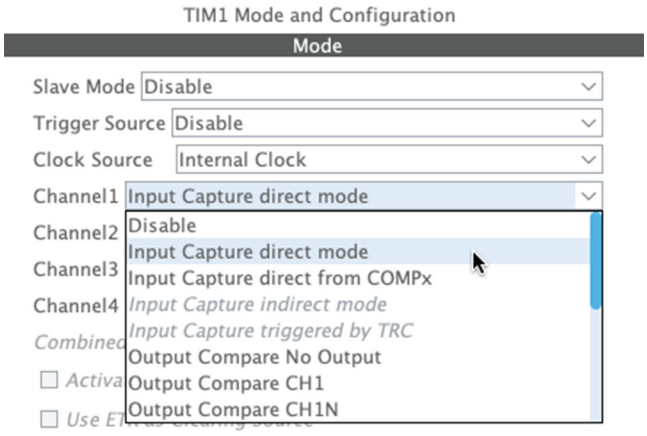


Figure 11.20: How to enable a channel in input capture mode

Instead, to map the other channel of the couple (IC1,IC2) or (IC3,IC4) to the same input (that is TI1 or TI2 for (IC1,IC2)), it is possible to enable the other channel in the couple in **Input capture indirect mode**, as shown in **Figure 11.21**. Finally, from the TIMx configuration view (not shown here), it is possible to configure the other input capture parameters (channel polarity, its filter, and so on).

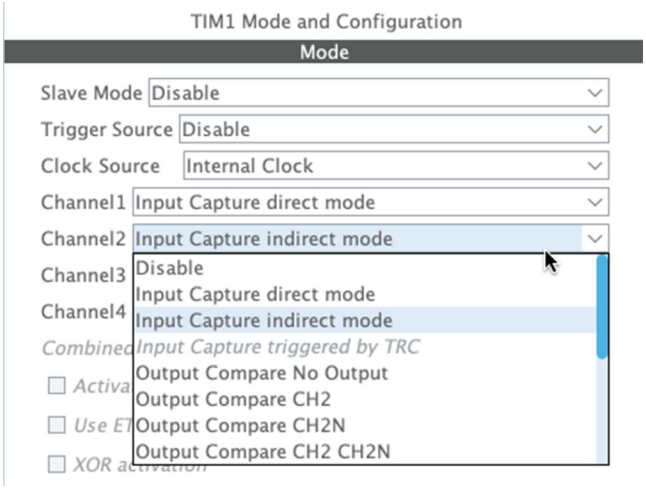


Figure 11.21: How to enable a channel in input capture indirect mode

11.3.6 Output Compare Mode

So far we have used a couple of techniques to control the GPIO level, one using interrupts and one the DMA. Both of them use the generation of UEV event to toggle a GPIO configured as output pin. The *output compare* is a mode offered by *general purpose* and *advanced* timers that allows to control the status of output channels when the channel compare register (TIMx_CCRx) matches with the timer counter register (TIMx_CNT).

There are six[29] output compare modes available to programmers:

- *Output compare timing*[30]: the comparison between the output compare register (CCRx) and the counter (CNT) has no effect on the output. This mode is used to generate a timing base.
- *Output compare active*: set the channel output to active level on match. The channel output is forced high when the counter (CNT) matches the capture/compare register (CCRx).
- *Output compare inactive*: set channel to inactive level on match. The channel output is forced low when the counter (CNT) matches the capture/compare register (CCRx).
- *Output compare toggle*: the channel output toggles when the counter (CNT) matches the capture/compare register (CCRx).
- *Output compare forced active/inactive*: the channel output is forced high (active mode) or low (inactive mode) independently from counter value.

Each channel of the timer is configured in output compare mode by using the function `HAL_TIM_OC_ConfigChannel()` and an instance of the C `struct TIM_OC_InitTypeDef`, which is defined in the following way:

[29]The output compare modes are actually eight, but two of them are related to PWM output, and they will be analyzed in the next paragraph.
[30]This mode in CubeMX is called *Frozen mode*.

```
typedef struct {
  uint32_t OCMode;       /* Specifies the TIM mode. */
  uint32_t Pulse;        /* Specifies the pulse value to be loaded
                            into the Capture Compare Register. */
  uint32_t OCPolarity;   /* Specifies the output polarity. */
  uint32_t OCNPolarity;  /* Specifies the complementary output polarity.*/
  uint32_t OCFastMode;   /* Specifies the Fast mode state. */
  uint32_t OCIdleState;  /* Specifies the TIM Output Compare pin state during Idle state.*/
  uint32_t OCNIdleState; /* Specifies the complementary  TIM Output Compare pin
                            state during Idle state. */
} TIM_OC_InitTypeDef;
```

- `OCMode`: specifies the output compare mode and it can assume a value from **Table 11.19**.
- `Pulse`: the content of this field will be stored inside the CCRx register and it establishes when to trigger the output. Please, ensure that the timer period is set to a multiple of the `Pulse` field, otherwise be prepared to handle the reminder of integer division accordingly.
- `OCPolarity`: defines the output channel polarity when the CCRx registers matches with the CNT one. It can assume a value from **Table 11.20**.
- `OCNPolarity`: defines the complimentary output polarity. It is a mode available only in TIM1 and TIM8 *advanced* timers, which allow to generate, on additional dedicated channels, complimentary signals (that is, when the CH1 is HIGH the CH1*n* is LOW and *vice versa*). This feature is especially designed for motor control applications, and it is not described in this book. It can assume a value from **Table 11.21**.
- `OCFastMode`: specifies the fast mode state. This parameter is valid only in PWM1 and PWM2 mode and it can assume the values `TIM_OCFAST_DISABLE` and `TIM_OCFAST_ENABLE`.
- `OCIdleState`: specifies the channel output compare pin state during the timer idle state. It can assume the values `TIM_OCIDLESTATE_SET` and `TIM_OCIDLESTATE_RESET`. This parameter is available only in TIM1 and TIM8 *advanced* timers.
- `OCNIdleState`: specifies the complementary channel output compare pin state during the timer idle state. It can assume the values `TIM_OCNIDLESTATE_SET` and `TIM_OCNIDLESTATE_RESET`. This parameter is available only in TIM1 and TIM8 *advanced* timers.

Table 11.19: Available output compare modes

Output compare mode	Description
`TIM_OCMODE_TIMING`	The comparison between the output compare register (CCRx) and the counter (CNT) has no effect on the output (aka, *frozen mode*)
`TIM_OCMODE_ACTIVE`	Set the channel output to active level on match
`TIM_OCMODE_INACTIVE`	Set channel to inactive level on match
`TIM_OCMODE_TOGGLE`	The channel output toggles when the counter (CNT) matches the capture/compare register (CCRx)
`TIM_OCMODE_PWM1`	PWM Mode 1 - see next paragraph
`TIM_OCMODE_PWM2`	PWM Mode 2 - see next paragraph
`TIM_OCMODE_FORCED_ACTIVE`	The channel output is forced high independently from the counter value
`TIM_OCMODE_FORCED_INACTIVE`	The channel output is forced low independently from the counter value

Table 11.20: Available output compare polarity modes

Output compare polarity mode	Description
`TIM_OCPOLARITY_HIGH`	When the CCRx and CNT registers match, the output channel is set high
`TIM_OCPOLARITY_LOW`	When the CCRx and CNT registers match, the output channel is set low

Table 11.21: Available complementary output compare polarity modes

Complementary output compare polarity mode	Description
`TIM_OCNPOLARITY_HIGH`	When the CCRx and CNT registers match, the complementary output channel is set high
`TIM_OCNPOLARITY_LOW`	When the CCRx and CNT registers match, the complementary output channel is set low

When the CCRx registers matches with the timer CNT counter, and the channel is configured to work in output compare mode, a specific interrupt is generated (if enabled). This allows to control the switching frequency of each channel independently, and eventually perform phase shift between channels. The channel frequency can be computed using the following formula:

$$CHx_Update = \frac{TIMx_CLK}{CCRx} \quad [5]$$

where:

$TIMx_CLK$ is the running frequency of the timer and $CCRx$ is the `Pulse` value of the `TIM_OnePulse_InitTypeDef struct` used to configure the channel. This means that we can compute the `Pulse` value, given a channel frequency, in the following way:

$$\texttt{Pulse} = \frac{TIMx_CLK}{CHx_Update} \quad [6]$$

Clearly, it is important to underline that the timer frequency must be set so that the `Pulse` value computed with [6] is lower than the timer `Period` value (the $CCRx$ value cannot be higher than the `TIM->ARR` value, which corresponds to the timer's `Period`).

The following example shows how to generate two output square wave signals, one running at 25kHz and one at 50kHz. It uses the Channel 1 and 2 (bound to OC1 and OC2) of TIM3 timer and it is designed to run on a Nucleo-F072RB.

Filename: `Core/Src/main-ex7.c`

```
17 volatile uint16_t CH1_FREQ = 0;
18 volatile uint16_t CH2_FREQ = 0;
19
20 int main(void) {
21   HAL_Init();
22
23   Nucleo_BSP_Init();
24   MX_TIM3_Init();
25
26   HAL_TIM_OC_Start_IT(&htim3, TIM_CHANNEL_1);
27   HAL_TIM_OC_Start_IT(&htim3, TIM_CHANNEL_2);
28
29   while (1);
30 }
31
32 /* TIM3 init function */
33 void MX_TIM3_Init(void) {
34   TIM_OC_InitTypeDef sConfigOC;
35
36   htim3.Instance = TIM3;
37   htim3.Init.Prescaler = 2;
38   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
39   htim3.Init.Period = 63999;
40   htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
41   HAL_TIM_OC_Init(&htim3);
42
43   CH1_FREQ = computePulse(&htim3, 25000); /* 25kHZ switching frequency */
44   CH2_FREQ = computePulse(&htim3, 50000); /* 50kHZ switching frequency */
45
46   sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
47   sConfigOC.Pulse = 0;
48   sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
49   sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
50   HAL_TIM_OC_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1);
51
52   sConfigOC.Pulse = 0;
53   HAL_TIM_OC_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_2);
54 }
```

Lines [48:59] configure Channel 1 and 2 to work as output compare channels. Both are configured in toggle mode (that is, they invert the state of the GPIO every time the CCRx register matches with the CNT timer register). The TIM3 is configured to run at 16MHz, and hence the function `computePulse()`, which uses the equation [6], will return the values `640` and `320` to have a channel switching frequency equal to 50kHz and 100kHz respectively. However, the above code is still not sufficient to drive the GPIO at that frequency. Here we are configuring the channels so that they

will toggle their output every time the timer CNT register is equal to 640 for Channel 1 and to 320 for Channel 2. But this means that the switching frequency is equal to:

$$\frac{16.000.000}{65535+1} = 244Hz$$

and we only have a shift of 10µs between the two channels, as shown by **Figure 11.22**. That 65535 value corresponds to the timer `Period` value, that is the maximum value reached by the timer CNT register.

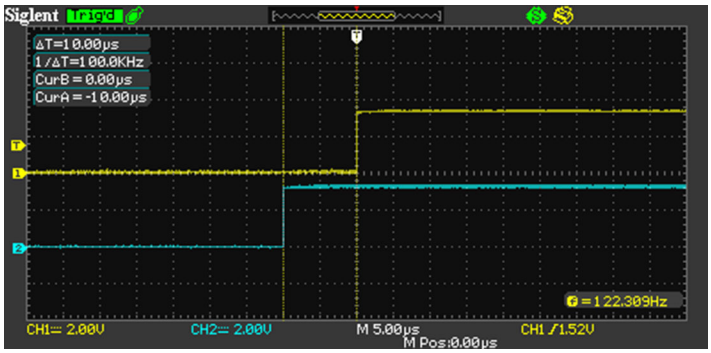


Figure 11.22: The toggling shift between channels 1 and 2

To reach the desired switching frequency[31], we need to toggle the output every 640 and 320 ticks of the TIM3 CNT register. To do so, we can define the following callback routine:

Filename: `Core/Src/main-ex7.c`

```
56 void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim) {
57   uint32_t pulse;
58   uint16_t arr = __HAL_TIM_GET_AUTORELOAD(htim);
59
60   /* TIMx_CH1 toggling with frequency = 50KHz */
61   if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1) {
62     pulse = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
63     /* Set the Capture Compare Register value */
64     if((pulse + CH1_FREQ) < arr)
65       __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_1, (pulse + CH1_FREQ));
66     else
67       __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_1, (pulse + CH1_FREQ) - arr);
68   }
69
70   /* TIMx_CH2 toggling with frequency = 100KHz */
71   if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2) {
72   pulse = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
73   /* Set the Capture Compare Register value */
74   if((pulse + CH2_FREQ) < arr)
```

[31]Please, ensure that the GPIO speed is configured so that the allowed maximum switching frequency is in the same range of the desired timer switching frequency.

```
75      __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_2, (pulse + CH2_FREQ));
76    else
77      __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_2, (pulse + CH2_FREQ) - arr);
78    }
79  }
```

The `HAL_TIM_OC_DelayElapsedCallback()` is automatically called by the HAL every time the Channel CCRx register matches the timer counter. We can so increase the `Pulse` (that is, the CCRx register) by `CH1_FREQ` for Channel 1 and by `CH2_FREQ` for Channel 2. This causes that the corresponding channel will switch at the wanted frequency, as shown in **Figure 11.23**.

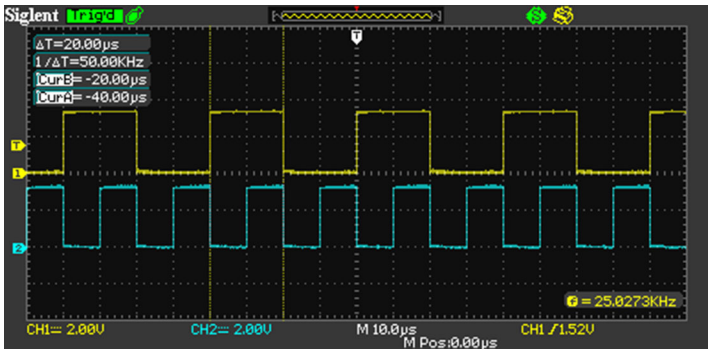


Figure 11.23: Channel 2 is configured to switch twice as fast as channel 1

The same result may be obtained using the DMA mode and a pre-initialized vector, eventually stored in the flash memory by using the `const` modifier:

```
const uint16_t ch1IV[] = {320, 640, 960, ...};
...
HAL_TIM_OC_Start_DMA(&htim3, TIM_CHANNEL_1, (uint32_t)ch1IV, sizeof(ch1IV));
```

11.3.6.1 Using CubeMX to Configure the Output Compare Mode

The configuration process of the output compare mode in CubeMX is identical to the one for the input capture mode. The first step is to select the **Output compare CHx** mode for the desired channel, as shown in **Figure 11.20**. Next, from the TIMx configuration view (not shown here), it is possible to configure the other output compare parameters (the output mode, channel polarity, and so on).

11.3.7 Pulse-Width Generation

The square waves generated until now have all one common characteristic: they have a T_{ON} period equal to the T_{OFF} one. For this reason, they are also said to have a 50% duty cycle. A *duty cycle* is the percentage of one period (for example, 1s) in which a signal is active. As a formula, a duty cycle is expressed as:

$$D = \frac{T_{ON}}{Period} \times 100\% \quad [8]$$

where D is the duty cycle, T_{ON} is the time the signal is active. Thus, a 50% duty cycle means the signal is on 50% of the time but off 50% of the time. The duty cycle says nothing about how long it lasts. The "on time" for a 50% duty cycle could be a fraction of a second, a day, or even a week, depending on the length of the period. The *pulse width* is the duration of the T_{ON}, given the actual *period*. For example, assuming a period of 1s, a duty cycle of 20% generates a pulse width of 200ms.

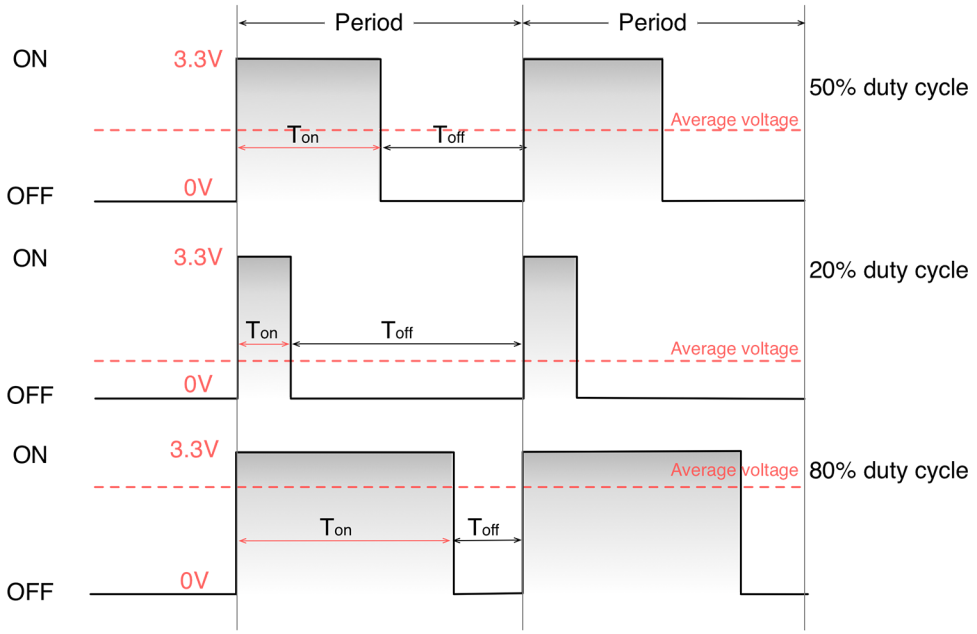


Figure 11.24: Three different duty cycles - 50%, 20% and 80%

The **Figure 11.24** shows three different duty cycles: 50%, 20% and 80%.

Pulse-width modulation (PWM) is a technique used to generate several pulses with different duty cycles in a given period of time or, if you prefer, at a given frequency. PWM has many applications in digital electronics, but all of them can be grouped in two main categories:

- control the output voltage (and hence the current);
- encoding (that is, modulate) a message (that is, a series of bytes in digital electronics[32]) on a carrier wave (which runs at a given frequency).

Those two categories can be expanded in several practical usages of the PWM technique. Focusing our attention on the control of the output voltage, we can find several applications:

- generation of an output voltage ranging from 0V up to VDD (that is, the maximum allowed voltage for an I/O, which in an STM32 is 3.3V);

[32]However, keep in mind that the PWM as modulation technique is not limited to digital electronics, but it originates in the "analog era" when it was used to modulate an audio wave on a carrier frequency.

  - dimming of LEDs;
  - motor control;
  - power conversion;
- generation of an output wave running at a given frequency (sine wave, triangle, square, and so on);
- sound output;

With adequate output filtering, which usually involves the usage of a *low-pass* filter, the PWM can replicate the behavior of a DAC, even if the MCU does not provide one. By varying the duty cycle of the output pin, it is possible to regulate the output voltage proportionally. An amplifier can increase/decrease the voltage range at a need, and it is also possible to control high currents and loads using power transistors.

A timer channel is configured in PWM mode by using the function `HAL_TIM_PWM_ConfigChannel()` and an instance of the C `struct TIM_OC_InitTypeDef` seen in the previous paragraph. The `TIM_OC_InitTypeDef.Pulse` field defines the duty cycle, and it ranges from `0` up to the timer `Period` field. The longer is the `Period` the wider is the tuning range. This means that we can fine-tune the output voltage.



The choice of the period, which determines the frequency of the output signal together with the timer clock (internal, external and so on), is not a detail to be left to chance. It depends on the specific application field, and it can have a severe impact on the overall EMI emissions. Moreover, some devices controlled with PWM technique may emit audible noise at given frequencies. This is the case of electric motors, which could emit unwanted buzzing noise when controlled at frequencies in the hearing range. Another example, not too much related here but with a similar genesis, is the noise emitted by power inductors in switching power supplies, which use the concept underlying the PWM to regulate their output voltage, and therefore the current. Sometimes, the output noise is unavoidable, and it is required to use varnishing products to reduce the problem. Other times, the right frequency come from "natural limitations": dimming a LED at a frequency close to 100Hz is usually sufficient to avoid visible flickering of the light.

There are two PWM modes available: *PWM mode 1* and *2*. Both are configurable through the field `TIM_OC_InitTypeDef.OCMode`, using the values `TIM_OCMODE_PWM1` and `TIM_OCMODE_PWM2`. Let us see the differences.

- **PWM mode 1**: in upcounting, the channel is active as long as `Period < Pulse`, else inactive. In downcounting, the channel is inactive as long as `Period > Pulse`, else active.
- **PWM mode 2**: in upcounting, channel 1 is inactive as long as `Period < Pulse`, else active. In downcounting, channel 1 is active as long as `Period > Pule`, else inactive.

The following example shows a typical application of the PWM technique: LED dimming. The example is designed to run on a Nucleo-F401RE and it fades ON/OFF the LD2 LED[33].

[33]Unfortunately, not all Nucleo boards have the LD2 LED connected to a timer channel (this depends on the fact that the pinout of LQFP-64 STM32 microcontrollers is not perfectly compatible). Only seven of them have this feature. Owners of other Nucleo boards have to rearrange the example using an external LED.

Filename: `Core/Src/main-ex8.c`

```
11  int main(void) {
12    HAL_Init();
13
14    Nucleo_BSP_Init();
15    MX_TIM2_Init();
16
17    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
18
19    uint16_t dutyCycle = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_1);
20
21    while(1) {
22      while(dutyCycle < __HAL_TIM_GET_AUTORELOAD(&htim2)) {
23        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, ++dutyCycle);
24        HAL_Delay(1);
25      }
26
27      while(dutyCycle > 0) {
28        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, --dutyCycle);
29        HAL_Delay(1);
30      }
31    }
32  }
33
34  /* TIM3 init function */
35  void MX_TIM2_Init(void) {
36    TIM_OC_InitTypeDef sConfigOC;
37
38    htim2.Instance = TIM2;
39    htim2.Init.Prescaler = 499;
40    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
41    htim2.Init.Period = 999;
42    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
43    HAL_TIM_PWM_Init(&htim2);
44
45    sConfigOC.OCMode = TIM_OCMODE_PWM1;
46    sConfigOC.Pulse = 0;
47    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
48    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
49    HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1);
50  }
```

Lines [45:49] configure the first channel of timer TIM2 to work in *PWM Mode 1*. The duty cycle will be range from 0 up to 999, which corresponds to the `Period` value. This means that we can regulate the output voltage with steps of ~0,0033V if the output is well filtered (and the PCB has a good layout). This is close to the performances of a 10bit DAC.

Lines [21:32] is where the fading effect takes place. The first loop increments the value of the `Pulse` (which corresponds to the *Capture Compare Register 1* (CCR1)) up to the `Period` value (which corresponds to the *Auto Reload Register* (ARR)) every 1ms. This means that in less than 1s the LED becomes full bright. The second loop, in the same way, decrements the `Pulse` field unless it reaches zero.



The update frequency of the timer is set to 84MHz[34]/(499+1)(999+1)=168Hz. The same frequency can be obtained by setting the `Prescaler` to 249 and the `Period` to 1999. But the fading effect changes. Why that happens? If you cannot explain the difference, I strongly suggest taking a break before going on, and doing experiments by yourself.

11.3.7.1 Generating a Sinusoidal Wave Using PWM

An output square wave generated with the PWM technique can be filtered to generate a smoothed signal, that is an analog signal that has a reduced *peak-to-peak* voltage (V_{pp}). A *Resistor-Capacitor* (RC) *low-pass* filter (see **Figure 11.25**) can cut-off all those AC signals having a frequency higher than a given threshold. The general rule of thumb of RC low-pass filters is that the lower is the cut-off frequency the lower is the V_{pp}[35]. An RC low-pass filter uses an important characteristic of capacitors: the ability to block DC currents while allowing the passing of AC ones: given the R/C time constant formed by the resistor-capacitor network, the filter will short to ground those AC signal with a frequency higher than the RC constant, allowing to pass DC component of the signal and lower frequency AC voltages.

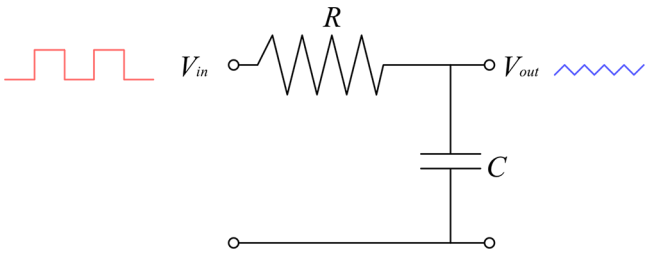


Figure 11.25: A typical low pass filter implemented with a resistor and a capacitor

While this circuit is very simple, choosing the appropriate values for R (the resistance) and C (the capacitance) encompass some design decisions: how much ripple we can tolerate and how fast the filter needs to respond. These two parameters are mutually exclusive. In most filters, we would like to have the perfect filter – one that passes all frequencies below the cut-off frequency, with no voltage ripple. Unfortunately, this ideal filter does not exist: to reduce the ripple to zero we have to choose a very large filter, which causes that it will take a lot of time to the output to become stable. While this could be acceptable for a continuous and fixed voltage, this has severe impact on the quality of the output signal if we are trying to generate a complex waveform from the PWM signal.

[34]The maximum frequency of timers in an STM32F401RE MCU, when clocked from the APB1 bus, is 84MHz.

[35]When dealing with filters to smooth an output wave it is more convenient to consider the effects on the output voltage than the response in frequency of the filter. However, the math under the *transfer function* of a filter is outside the scope of this book. If interested, this on-line calculator(https://bit.ly/22breq2) allows to evaluate the V_{pp} output given a V_{IN}, the PWM frequency and the R and C values.

The cut-off frequency (f_c) of a first order RC low-pass filter is expressed by the formula:

$$f_c = \frac{1}{2\pi RC} \quad [9]$$

Figure 11.26 shows the effect of a low-pass filter on a PWM signal with a frequency of 100Hz. Here we have chosen a 1K resistor and a 10µF capacitor. This means that the cut-off frequency is equal to:

$$f_c = \frac{1}{2\pi 10^3 \times 10^{-5}} \approx 15.9Hz$$

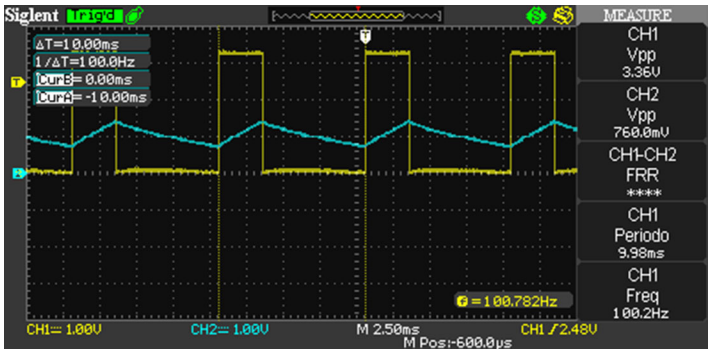


Figure 11.26: The effect of a low-pass filter with cut-off frequency equal to 15.9Hz

Figure 11.27 shows the effect of the low-pass filter with a 4300K resistor and a 10µF capacitor. This means that the cut-off frequency is equal to:

$$f_c = \frac{1}{2\pi(4.3 \times 10^3) \times 10^{-5}} \approx 3.7Hz$$

As you can see, the second filter allows to have a (V_{pp}) equal to about 160mV, which is a voltage difference passable for a lot of applications.

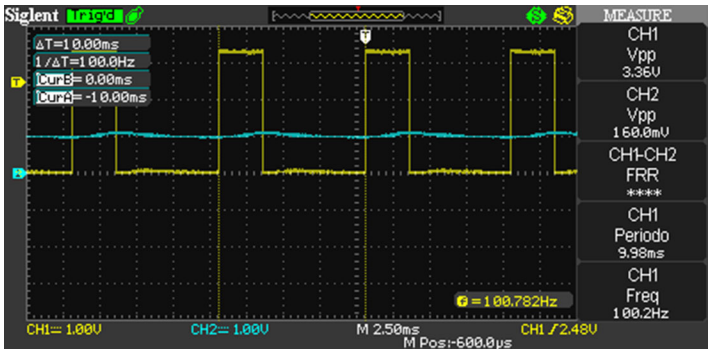


Figure 11.27: The effect of a low-pass filter with cut-off frequency equal to 3.7Hz

By varying the output voltage (which implies that we vary the duty cycle) we can generate an arbitrary output waveform, whose frequency is a fraction of the PWM period. The basic idea here

is to divide the waveform we want, for example a sine wave, into '*x*' number of divisions. For each division we have a single PWM cycle. The T_{ON} time (that is, the duty cycle) directly corresponds to the amplitude of the waveform in that division, which is calculated using *sin()* function.

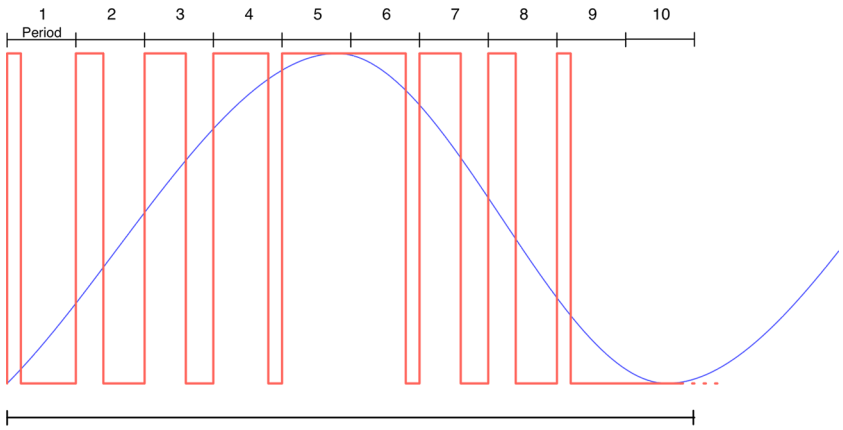


Figure 11.28: How a sine wave can be approximated with multiple PWM signals

Consider the diagram shown in **Figure 11.28**. Here the sine wave has been divided in 10 steps. So here we will require 10 different PWM pulses increasing/decreasing in sinusoidal manner. A PWM pulse with 0% duty cycle will represent the min amplitude (0V), the one with 100% duty cycle will represent max amplitude(3.3V). Since out PWM pulse has voltage swing between 0V to 3.3V, our sine wave will swing between 0V to 3.3V too.

It takes 360 degrees for a sine wave to complete one cycle. Hence for 10 divisions we will need to increase the angle in steps of 36 degrees. This is called the *Angle Step Rate* or *Angle Resolution*. We can increase the number of divisions to get more accurate waveform. But as divisions increase, we also need to increase the resolution, which implies that we have to increase the frequency of the timer used to generate the PWM signal (the faster runs the timer the smaller is the period).

Usually, 200 divisions are a good approximation for an output wave. This means that if we want to generate a 50Hz sine wave, we need to run the timer at a 50Hz*200 = 10kHz. The pulse period will be equal to 200 (the number of steps - this means that we vary the output voltage by 3.3V/200=0.016V), and so the `Prescaler` value will be (assuming an STM32F072 MCU running at 48MHz):

$$Prescaler = \frac{48MHz}{50Hz \times 200_{divisions} \times 200_{\texttt{Pulse}}} = 24$$

The following example shows how to generate a 50Hz pure sine wave in an STM32F072MCU running at 48MHz.

Filename: `Core/Src/main-ex9.c`

```
14  #define PI    3.14159
15  #define ASR   1.8 //360 / 200 = 1.8
16
17  int main(void) {
18    uint16_t IV[200];
19    float angle;
20
21    HAL_Init();
22
23    Nucleo_BSP_Init();
24    MX_TIM3_Init();
25
26    for (uint8_t i = 0; i < 200; i++) {
27      angle = ASR*(float)i;
28      IV[i] = (uint16_t) rint(100 + 99*sinf(angle*(PI/180)));
29    }
30
31    HAL_TIM_PWM_Start_DMA(&htim3, TIM_CHANNEL_1, (uint32_t *)IV, 200);
32
33    while (1);
34  }
35
36  /* TIM3 init function */
37  void MX_TIM3_Init(void) {
38    TIM_OC_InitTypeDef sConfigOC;
39
40    htim3.Instance = TIM3;
41    htim3.Init.Prescaler = 23;
42    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
43    htim3.Init.Period = 199;
44    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV4;
45    HAL_TIM_PWM_Init(&htim3);
46
47    sConfigOC.OCMode = TIM_OCMODE_PWM1;
48    sConfigOC.Pulse = 0;
49    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
50    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
51    HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1);
52
53    hdma_tim3_ch1_trig.Instance = DMA1_Channel4;
54    hdma_tim3_ch1_trig.Init.Direction = DMA_MEMORY_TO_PERIPH;
55    hdma_tim3_ch1_trig.Init.PeriphInc = DMA_PINC_DISABLE;
56    hdma_tim3_ch1_trig.Init.MemInc = DMA_MINC_ENABLE;
57    hdma_tim3_ch1_trig.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
58    hdma_tim3_ch1_trig.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
59    hdma_tim3_ch1_trig.Init.Mode = DMA_CIRCULAR;
```

```
60    hdma_tim3_ch1_trig.Init.Priority = DMA_PRIORITY_LOW;
61    HAL_DMA_Init(&hdma_tim3_ch1_trig);
62
63    /* Several peripheral DMA handle pointers point to the same DMA handle.
64     Be aware that there is only one channel to perform all the requested DMAs. */
65    __HAL_LINKDMA(&htim3, hdma[TIM_DMA_ID_CC1], hdma_tim3_ch1_trig);
66    __HAL_LINKDMA(&htim3, hdma[TIM_DMA_ID_TRIGGER], hdma_tim3_ch1_trig);
67  }
```

The most relevant part is represented by lines [26:29]. That lines of code are used to generate the *Initialization Vector* (IV), that is the vector containing the `Pulse` values used to generate the sine wave (which corresponds to the output voltage levels). The C `sinf()` returns the sine of the given angle expressed in *radians*. So, we need to convert the angular expresses in degrees to radians using the formula:

$$Radians = \frac{\pi}{180^{\circ}} \times Degrees$$

However, in our case we have divided the sine wave cycle in 200 steps (that is, we have divided the circumference in 200 steps), so we need to compute the value in radians of each step. But since sine gives negative values for angle between 180° and 360° (see **Figure 11.29**) we need to scale it, since PWM output values cannot be negative.

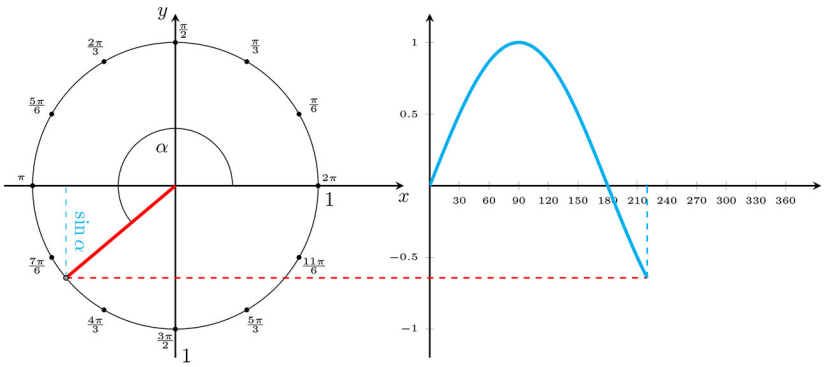


Figure 11.29: The values assumed by sine function between 180° and 360°

Once the `IV` vector is generated, we can start PWM in DMA mode. The DMA1_Channel4 is configured to work in circular mode, so that it automatically sets the value of the TIMx_CCRx register according to the `Pulse` values contained in `IV`. Using a timer in DMA mode is the best way to generate arbitrary function without introducing latency and affecting the Cortex-M core. However, often IVs are hardcoded inside the program, using `const` arrays automatically stored in the flash memory. You can find several on-line tools to do this, like the one provided here[36].

[36] https://bit.ly/1QPfm4k

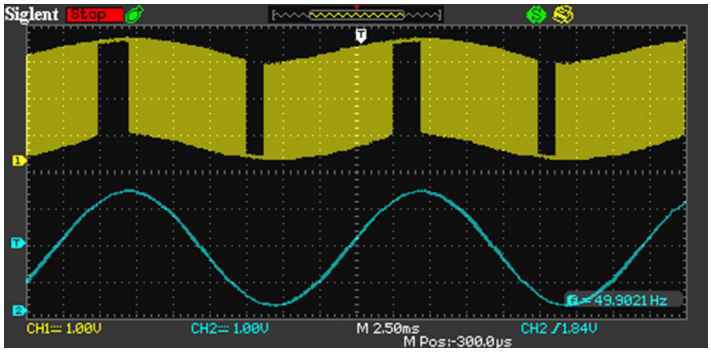


Figure 11.30: How timers allow to approximate a 50Hz sine wave using PWM

Figure 11.30 shows the output from TIM3 Channel 1: as you can see, using an adequate filtering stage[37], it is easy to generate a pure 50Hz sine wave.

11.3.7.2 Using CubeMX to Configure the PWM Mode

The configuration process of the PWM mode in CubeMX is straightforward once the fundamental concepts of PWM generation have been mastered. The first step is to select the **PWM Generation CHx** mode for the desired channel, as shown in **Figure 11.20**. Next, from the TIMx configuration view (not shown here), it is possible to configure the other PWM settings (*PWM mode 1* or *2*, channel polarity, and so on).

11.3.8 One Pulse Mode

One Pulse Mode (OPM) is a mix of the input capture and the output compare modes offered by *general purpose* and *advanced* timers. It allows the counter to be started in response to a stimulus and to generate a pulse with a programmable duration (PWM) after a programmable delay.

OPM is a mode designed to work exclusively with Channel 1 and 2 of a timer. We can decide which of the two channels is the output and which is the input by using the function:

```
HAL_TIM_OnePulse_ConfigChannel(TIM_HandleTypeDef *htim, TIM_OnePulse_InitTypeDef* sConfig,
                               uint32_t OutputChannel,  uint32_t InputChannel);
```

Both the channels are configured with an instance of the C `struct TIM_OnePulse_InitTypeDef`, which is defined in the following way:

[37] Here, I have used a 100ohm resistor an a 10µF capacitor, which give a cut-off frequency of ∼159Hz and a V_{pp} equal to 0.08V.

```
typedef struct {
  uint32_t Pulse;      /* Specifies the pulse value to be loaded into the CCRx register.*/
  /* Output channel configuration */
  uint32_t OCMode;       /* Specifies the TIM mode. */
  uint32_t OCPolarity;   /* Specifies the output polarity. */
  uint32_t OCNPolarity;  /* Specifies the complementary output polarity. */
  uint32_t OCIdleState;  /* Specifies the TIM Output Compare pin state during Idle state.*/
  uint32_t OCNIdleState; /* Specifies the TIM Output Compare pin state during Idle state.*/
  /* Input channel configuration */
  uint32_t ICPolarity;   /* Specifies the active edge of the input signal. */
  uint32_t ICSelection;  /* Specifies the input. */
  uint32_t ICFilter;     /* Specifies the input capture filter. */
} TIM_OnePulse_InitTypeDef;
```

The `struct` is logically divided in two parts: one related to the configuration of the input channel, and one to the output. We will not go into the details of the `struct` fields, because they are similar to what seen so far when we have talked about input capture and output compare modes.

An important aspect to understand is the way the timer computes delay and pulse durations. The delay is computed according to the following formula:

$$Delay = \frac{\texttt{Pulse}}{(\frac{TIMx_CLK}{\texttt{Prescaler+1}})} \quad [10]$$

while the duration (that is, the duty cycle) of the pulse is computed with this one:

$$Duration = \frac{\texttt{Period - Pulse}}{(\frac{TIMx_CLK}{\texttt{Prescaler+1}})} \quad [11]$$

This means that, once the input channel detects the trigger event, the timer starts counting and when the CNT register reaches the CCRx register (`Pulse`) it generates the output signal, which lasts until the CNT register reaches the ARR register (`Period`), that is `Period - Pulse`.

The OPM can be set as single shoot or in repetitive mode. This is performed by using the

```
HAL_TIM_OnePulse_Init(TIM_HandleTypeDef *htim, uint32_t OnePulseMode);
```

which accepts the pointer to the timer handler and the symbolic constant `TIM_OPMODE_SINGLE` to configure OPM in single shoot or `TIM_OPMODE_REPETITIVE` to enable repetitive mode.

The following example shows how to configure TIM3 in OPM mode in an STM32F072 MCU.

Filename: `Core/Src/main-ex10.c`

```
12  int main(void) {
13    HAL_Init();
14
15    Nucleo_BSP_Init();
16    MX_TIM3_Init();
17
18    HAL_TIM_OnePulse_Start(&htim3, TIM_CHANNEL_1);
19
20    while (1);
21  }
22
23  /* TIM3 init function */
24  void MX_TIM3_Init(void) {
25    TIM_OnePulse_InitTypeDef sConfig;
26
27    htim3.Instance = TIM3;
28    htim3.Init.Prescaler = 47;
29    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
30    htim3.Init.Period = 65535;
31    HAL_TIM_OnePulse_Init(&htim3, TIM_OPMODE_SINGLE);
32
33    /* Configure the Channel 1 */
34    sConfig.OCMode = TIM_OCMODE_PWM1;
35    sConfig.OCPolarity = TIM_OCPOLARITY_LOW;
36    sConfig.Pulse = 19999;
37
38    /* Configure the Channel 2 */
39    sConfig.ICPolarity = TIM_ICPOLARITY_RISING;
40    sConfig.ICSelection = TIM_ICSELECTION_DIRECTTI;
41    sConfig.ICFilter = 0;
42
43    HAL_TIM_OnePulse_ConfigChannel(&htim3, &sConfig, TIM_CHANNEL_1, TIM_CHANNEL_2);
44  }
```

Lines [34:36] configure the output channel in *PWM Mode 1*, while lines [39:41] configure the input channel. The `HAL_TIM_OnePulse_ConfigChannel()`, at line 43, configures the two channels, setting the Channel 1 as the output and the Channel 2 as the input. Finally the `HAL_TIM_OnePulse_Start()` (called at line 18) starts the timer in OPM mode. By biasing the PA7 pin in a Nucleo-F072RB, the timer will start after a delay of 20ms, and it will generate a PWM of about 45ms, as shown in **Figure 11.31**.

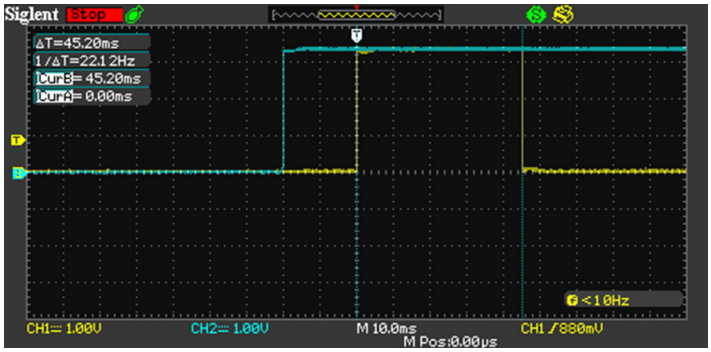


Figure 11.31: How the *One Pulse mode* works

The output channel of a timer running in One Pulse can be configured even in other modes different from the PWM one.

11.3.8.1 Using CubeMX to Configure the OPM Mode

To enable the OPM mode using CubeMX, the first step is to configure the two Channel 1 and 2 independently, and then to select the **One Pulse Mode** checkbox, as shown in **Figure 11.32**. Next, from the TIMx configuration view (not shown here), it is possible to configure the other channels settings.

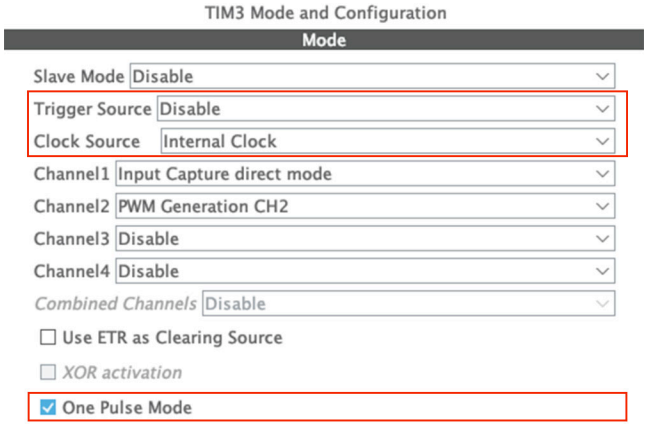


Figure 11.32: How to enable the *One Pulse mode* in a timer

11.3.9 Encoder Mode

Rotary encoders are devices that have a wide range of applications. They are used to measure the speed as well as the angular position of rotating objects. They can be used to measure RPM and direction of a motor, to control servomotors as well step motors, and so on. There are several types of rotary encoders: optical, mechanical, magnetic.

Incremental encoders are a type of rotary encoders that provide cyclic output when they detect movement. The mechanical type requires debouncing and is typically used as “digital potentiometer”.

Most modern home and car stereos use mechanical rotary encoders for volume control. The incremental rotary encoder is the most widely used of all rotary encoders due to its low cost and ability to provide signals that can be easily interpreted to provide motion related information such as velocity.

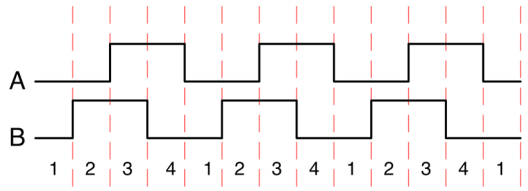


Figure 11.33: The square waves emitted by a quadrature encoder on A and B channels

They employ two outputs called *A* and *B*, which are called quadrature outputs, as they are 90 degrees out of phase, as shown in **Figure 11.33**. The direction of the motor depends on if phase *A* leads phase *B*, or phase *B* leads phase *A*. An optional third channel, *index pulse*, occurs once per revolution and it is used as a reference to measure an absolute position. There are several ways to detect direction and position of a rotary encoder. By connecting the *A* and *B* pins to two MCU I/Os it is possible to detect when the signal goes HIGH and LOW. This can be performed both manually (using interrupts to capture when the channel changes status) or by using a timer: its channels can be configured in input capture mode and the capture values are compared to compute the direction and speed of the encoder.

STM32 *general purpose* timers provide a convenient way to read rotary encoders: this mode is indeed called *encoder mode* and it simplifies a lot the capture process. When a timer is configured in encoder mode, the timer counter register (TIMx_CNT) is incremented/decremented on the edge of input channels.

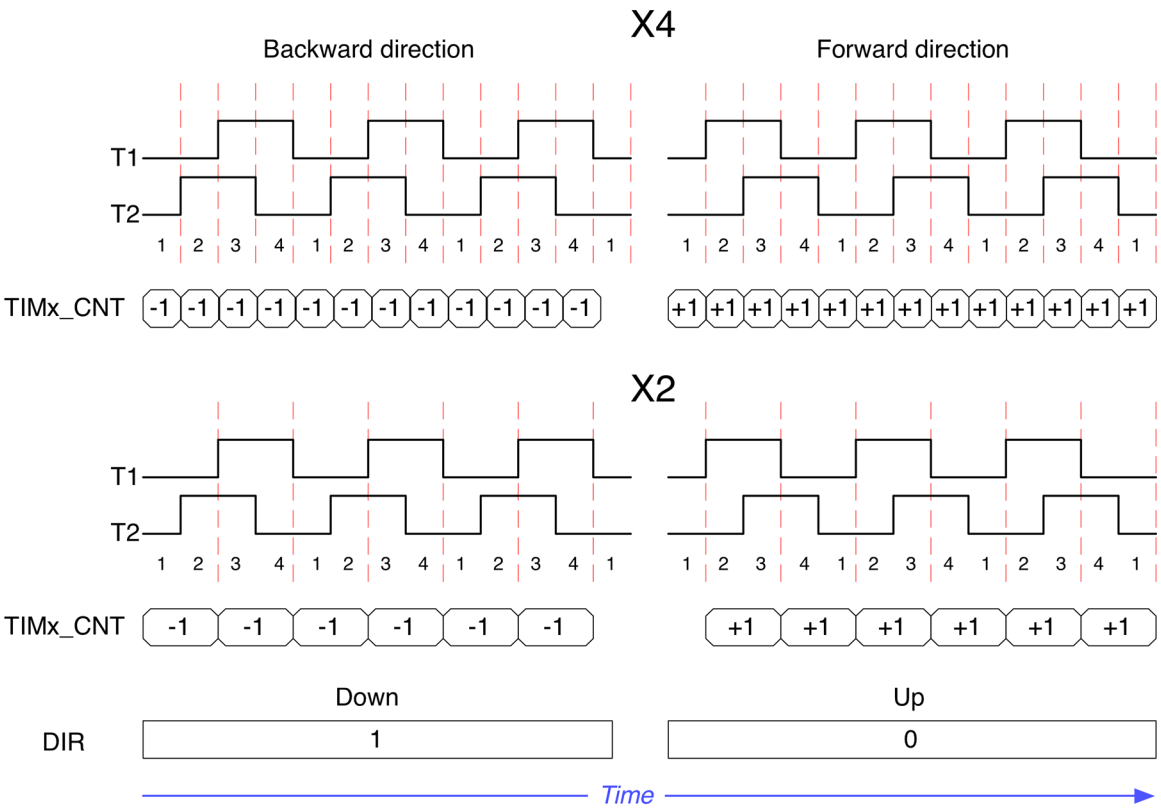


Figure 11.34: How encoder speed and direction are computed by a timer in *encoder mode*

There are two capturing modes available: *X2* and *X4*. In *X2* mode the CNT register is incremented/decremented on every edge of only one channel (either T1 or T2). In *X4* mode the CNT register is updated on every edge of both the channels: this doubles the capture frequency. The direction of the movement is automatically derived and made available to the programmer in the TIMx_DIR register, as shown in **Figure 11.34**. By comparing the value of the counter register on a regular basis, it is possible to derive the number of RPM, given the number of pulses the encoder emits per revolution.

Incremental mechanical encoders usually need to be debounced, due to noisy output. A comparator is usually used as filtering stage of these devices, especially if they are used to interface motors and other noisy devices. Under certain conditions, the input filter stage of an STM32 timer can be used to filter the *A* and *B* channels, reducing the number of BOM components.

The encoder mode is available only on TI1 and TI2 channels, and it is activated by using the function `HAL_TIM_Encoder_Init()` and an instance of the C `struct TIM_Encoder_InitTypeDef`, which is defined in the following way.

```
typedef struct {
  /* T1 channel */
  uint32_t EncoderMode;   /* Specifies the active edge of the input signal. */
  uint32_t IC1Polarity;   /* Specifies the active edge of the input signal. */
  uint32_t IC1Selection;  /* Specifies the input. */
  uint32_t IC1Prescaler;  /* Specifies the Input capture prescaler. */
  uint32_t IC1Filter;     /* Specifies the input capture filter. */
  /* T2 channel */
  uint32_t IC2Polarity;   /* Specifies the active edge of the input signal. */
  uint32_t IC2Selection;  /* Specifies the input. */
  uint32_t IC2Prescaler;  /* Specifies the Input capture prescaler. */
  uint32_t IC2Filter;     /* Specifies the input capture filter. */
} TIM_Encoder_InitTypeDef;
```

We have encountered the majority of the `TIM_Encoder_InitTypeDef` fields in the previous paragraphs. The only remarkable one is the `EncoderMode`, which can assume the values `TIM_ENCODERMODE_TI1` or `TIM_ENCODERMODE_TI2` to set the *X2* encoder mode on one of the two channels, and the value `TIM_ENCODERMODE_TI12` to set the *X4* mode so that the TIMx_CNT register is updated on every edge of TI1 and TI2 channels.

The following example, designed to run on a Nucleo-F072RB, simulates an incremental encoder by using the TIM1 in output compare mode. TIM1 OC1 and OC2 (PA8, PA9) channels are routed to TIM3 TI1 and TI2 channels (PA6, PA7) using the *morpho connector*, and they are configured so that they generate two square wave signals having the same period but shifted in phase. The TIM3 is then configured in encoder mode. The *SysTick* timer is used to generate the timebase: every 1s, the number of pulses is computed, together with the encoder direction. The number of RPMs is then derived, assuming an encoder that generates 4 pulses for every revolution. Finally, by pressing the USER button it is possible to change the phase shift between phase *A* and *B*: this will invert the encoder revolution.

Filename: `Core/Src/main-ex11.c`

```
22 #define PULSES_PER_REVOLUTION 4
23
24 int main(void) {
25   HAL_Init();
26
27   Nucleo_BSP_Init();
28   MX_TIM1_Init();
29   MX_TIM3_Init();
30
31   HAL_TIM_Encoder_Start(&htim3, TIM_CHANNEL_ALL);
32   HAL_TIM_OC_Start(&htim1, TIM_CHANNEL_1);
33   HAL_TIM_OC_Start(&htim1, TIM_CHANNEL_2);
34
35   cnt1 = __HAL_TIM_GET_COUNTER(&htim3);
36   tick = HAL_GetTick();
```

```
37
38    while (1) {
39      if (HAL_GetTick() - tick > 1000L) {
40        cnt2 = __HAL_TIM_GET_COUNTER(&htim3);
41        if (__HAL_TIM_IS_TIM_COUNTING_DOWN(&htim3)) {
42          if (cnt2 < cnt1) /* Check for counter underflow */
43            diff = cnt1 - cnt2;
44          else
45            diff = (65535 - cnt2) + cnt1;
46        } else {
47          if (cnt2 > cnt1) /* Check for counter overflow */
48            diff = cnt2 - cnt1;
49          else
50            diff = (65535 - cnt1) + cnt2;
51        }
52
53        sprintf(msg, "Difference: %d\r\n", diff);
54        HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
55
56        speed = ((diff / PULSES_PER_REVOLUTION) / 60);
57
58        /* If the first three bits of SMCR register are set to 0x3
59         * then the timer is set in X4 mode (TIM_ENCODERMODE_TI12)
60         * and we need to divide the pulses counter by two, because
61         * they include the pulses for both the channels */
62        if ((TIM3->SMCR & 0x3) == 0x3)
63          speed /= 2;
64
65        sprintf(msg, "Speed: %d RPM\r\n", speed);
66        HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
67
68        dir = __HAL_TIM_IS_TIM_COUNTING_DOWN(&htim3);
69        sprintf(msg, "Direction: %d\r\n", dir);
70        HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
71
72        tick = HAL_GetTick();
73        cnt1 = __HAL_TIM_GET_COUNTER(&htim3);
74      }
75
76      if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET) {
77        /* Invert rotation by swapping CH1 and CH2 CCR value */
78        tim1_ch1_pulse = __HAL_TIM_GET_COMPARE(&htim1, TIM_CHANNEL_1);
79        tim1_ch2_pulse = __HAL_TIM_GET_COMPARE(&htim1, TIM_CHANNEL_2);
80
81        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, tim1_ch2_pulse);
82        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2, tim1_ch1_pulse);
83      }
```

```
84    }
85  }
86
87  /* TIM1 init function */
88  void MX_TIM1_Init(void) {
89    TIM_OC_InitTypeDef sConfigOC;
90
91    htim1.Instance = TIM1;
92    htim1.Init.Prescaler = 9;
93    htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
94    htim1.Init.Period = 999;
95    HAL_TIM_Base_Init(&htim1);
96
97    sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
98    sConfigOC.Pulse = 499;
99    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
100   sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
101   sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
102   sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
103   sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
104   HAL_TIM_OC_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1);
105
106   sConfigOC.Pulse = 999; /* Phase B is shifted by 90° */
107   HAL_TIM_OC_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2);
108 }
109
110 /* TIM3 init function */
111 void MX_TIM3_Init(void) {
112   TIM_Encoder_InitTypeDef sEncoderConfig;
113
114   htim3.Instance = TIM3;
115   htim3.Init.Prescaler = 0;
116   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
117   htim3.Init.Period = 65535;
118
119   sEncoderConfig.EncoderMode = TIM_ENCODERMODE_TI12;
120
121   sEncoderConfig.IC1Polarity = TIM_ICPOLARITY_RISING;
122   sEncoderConfig.IC1Selection = TIM_ICSELECTION_DIRECTTI;
123   sEncoderConfig.IC1Prescaler = TIM_ICPSC_DIV1;
124   sEncoderConfig.IC1Filter = 0;
125
126   sEncoderConfig.IC2Polarity = TIM_ICPOLARITY_RISING;
127   sEncoderConfig.IC2Selection = TIM_ICSELECTION_DIRECTTI;
128   sEncoderConfig.IC2Prescaler = TIM_ICPSC_DIV1;
129   sEncoderConfig.IC2Filter = 0;
130
```

```
131   HAL_TIM_Encoder_Init(&htim3, &sEncoderConfig);
132 }
```

Function `MX_TIM1_Init()` configures the TIM1 timer so that its OC1 and OC2 channels work in output compare mode, triggering their output every ~20µs. The two outputs are shifted in phase by setting two different `Pulse` values (lines 84 and 92). The `MX_TIM3_Init()` function configures the TIM3 in encoder *X4* mode (`TIM_ENCODERMODE_TI12`).

The `main()` function is designed so that every 1000 ticks of the *SysTimer* (which is configured to generate a tick every 1ms) the current content of the counter register (`cnt2`) is compared with a saved value (`cnt1`): according to the encoder direction (up or down), the difference is computed, and the speed is calculated. The code needs also to detect an eventual overflow/underflow of the counter and compute the difference accordingly. Take also note that, since we are performing a comparison every one second, TIM1 must be configured so that the sum of pulses generated by channels *A* and *B* should be less than 65535 per second. For this reason, we slow down TIM1 setting a `Prescaler` equal to 9. Finally, lines [76:83] invert the phase shift between *A* and *B* (that is, OC1 and OC2 channels of TIM1 timer) when the Nucleo user button is pressed.

11.3.9.1 Using CubeMX to Configure the *Encoder Mode*

To enable the *encoder mode* using CubeMX, the first step is to enable this mode from the **Combined Channels** combo box, as shown in **Figure 11.35**. Next, from the TIMx configuration view (not shown here), it is possible to configure the other channels settings.

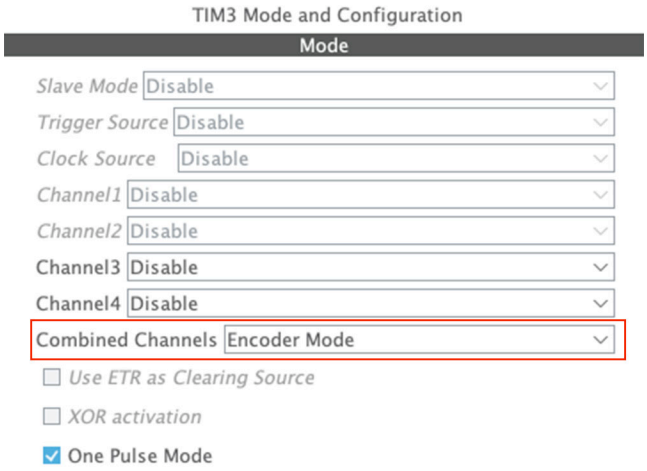


Figure 11.35: How to enable the *encoder mode* in a timer

11.3.10 Other Features Available in *General Purpose* and *Advanced* Timers

The features seen so far represent the most common usages of a timer. However, STM32 *general purpose* and *advanced* timers provide other important functionalities, really useful in some specific

application domains. We will now give a quick overview to these additional capabilities. Since these functionalities share common concepts found in other application shown in previous paragraphs, we will not go too much into details of these topics (especially because it is not so easy to arrange examples without dedicated hardware).

11.3.10.1 *Hall Sensor* Mode

In a brushed DC motor, brushes control the commutation by physically connecting the coils at the correct moment. In *Brush-Less DC* (BLDC) motors the commutation is controlled by electronics, using PWM. The electronics can either have position sensor inputs, which provide information about when to commutate, or use the *Back Electromotive Force* (B_{EF}) generated in the coils. Position sensors are most often used in applications where the starting torque varies greatly or where a high initial torque is required. Position sensors are also often used in applications where the motor is used for positioning.

Hall-effect sensors, or simply Hall sensors, are mainly used to compute the position of three-phases BLDC motors (one sensor for each phase). STM32 *general purpose* timers can be programmed to work in *Hall sensor mode*. By setting the first three input in XOR mode, it is possible to automatically detect the position of the rotor.

This is done using the advanced-control timers (TIM1) to generate PWM signals to drive the motor and another timer (e.g., TIM3) referred to as "interfacing timer". This interfacing timer captures the three timer input pins (CC1, CC2, CC3) connected through a XOR to the TI1 input channel (see **Figure 11.16**). TIM3 is in *slave mode*, configured in reset mode; the slave input is TI1F_ED[38]. Thus, each time one of the three inputs toggles, the counter restarts counting from 0. This creates a time base triggered by any change on the Hall inputs.

On the "interfacing timer" (TIM3), capture/compare channel 1 is configured in capture mode, capture signal is TRC (See **Figure 11.16** - TRC is highlighted in red). The captured value, which corresponds to the time elapsed between 2 changes on the inputs, gives information about motor speed. The "interfacing timer" can be used in output mode to generate a pulse which changes the configuration of the channels of the advanced-control timer (TIM1) (by triggering a COM event). The TIM1 timer is used to generate PWM signals to drive the motor. To do this, the interfacing timer channel must be programmed so that a positive pulse is generated after a programmed delay (in output compare or PWM mode). This pulse is sent to the *advanced* timer (TIM1) through the TRGO output.

11.3.10.2 Combined Three-Phase PWM Mode and Other Motor-Control Related Features

The ST32F3 family is the one dedicated to advanced power conversion and motor control. Some STM32F3 MCUs, notably STM32F30x and STM32F3x8, provide the ability to generate one to three center-aligned PWM signals with a single programmable signal ANDed in the middle of the pulses. Moreover, they can generate up to three complementary outputs with insertion of *dead time*. These

[38] ED is acronyms for *Edge Detector* and it is an internal filtered timer input enabled when only one of the three inputs in XOR is HIGH.

features, in addition to the *Hall sensor mode* seen before, allow to build electronic devices suitable for the motor control. For more information about this, refer to the AN4013[39] from ST.

11.3.10.3 Break Input and Locking of Timer Registers

The break input is an emergency input in the motor control application. The break function protects power switches driven by PWM signals generated with the advanced timers. The break input is usually connected to fault outputs of power stages and 3-phase inverters. When activated, the break circuitry shuts down the TIM outputs and forces them to a predefined safe state.

Moreover, *advanced* timers offer a gradual protection of their registers, programming the LOCK bits in the BDTR register. There are three locking levels available, which selectively lock up to all timer register. For more information refer to the reference manual for your MCU.

11.3.10.4 Preloading of Auto-Reload Register

We have left uncommented one thing from **Figure 11.16**. The ARR register is graphically represented with a shadow. This happens because it is preloaded, that is writing to or reading from the ARR register accesses the *preload* register. The content of the *preload* register is transferred to the *shadow* register (that is, the register internal to the timer that effectively contains the counter value to match) **permanently** or at each UEV event if and only if the *auto-reload preload bit* (APRE) is enabled in the TIMx->CR1 register. If so, a UEV event can be generated setting the corresponding bit in the TIMx->EGR register: this will cause that the content of the *preload* register is transferred in the *shadow* one and the new value will be taken in account by the timer. Obviously, if you stop the timer, you can change the content of the ARR register freely.

This is an important aspect to clarify. When a timer is stopped, we can configure the ARR register using the `TIM_Base_InitTypeDef.Period structure`: the content of the `Period` field is transferred in the TIMx->ARR register by the `HAL_TIM_Base_Init()` function. This will cause that the UEV event is generated and, if enabled, the corresponding IRQ will be raised. It is important to remark that this happens even when the timer is configured for the first time since the peripheral was reset. Let us consider this code:

```
htim6.Instance = TIM6;
htim6.Init.Prescaler = 47999; //48MHz/48000 = 1kHz
htim6.Init.Period = 4999;     //1kHz / 5000 = 5s
htim6.Init.CounterMode = TIM_COUNTERMODE_UP;

__HAL_RCC_TIM6_CLK_ENABLE();

HAL_NVIC_SetPriority(TIM6_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(TIM6_IRQn);

HAL_TIM_Base_Init(&htim6);
HAL_TIM_Base_Start_IT(&htim6);
```

[39] https://bit.ly/1WAewd6

The above code configures the TIM6 timer so that it expires after 5 seconds. However, if you rearrange that code in a complete example, you can see that the IRQ fires almost immediately after the `HAL_TIM_Base_Start_IT()` function is called. This is due to the fact that the `HAL_TIM_Base_Init()` routine generates an UEV events to transfer the content of the `TIM6->ARR` register inside the internal *shadow* register. This causes that the UIF flag is set and the IRQ fires when the `HAL_TIM_Base_Start_IT()` enables it.

We can bypass this behaviour by setting the `URS` bit inside the TIMx->CR1 register: this will cause that the UEV event is generated only when the counter reaches the overflow/underflow.

It is possible to configure the timer so that the ARR register is buffered, by setting the `TIM_CR1_ARPE` bit in the TIMx->CR1 control register. This will cause that the content of the *shadow* register is updated automatically. Unfortunately, the HAL does not seem to offer an explicit macro to do that, and we need to access to the timer register at low-level:

```
TIM3->CR1 |= TIM_CR1_ARPE;  //Enable preloading
TIM3->CR1 &= ~TIM_CR1_ARPE; //Disable preloading
```

Preloading is especially useful when we use a timer in output compare mode with multiple output channels enabled and each one with its own capture value, and we have to be sure that any change to the CCRx register takes place at the same time. This is especially true if we use a timer for motor control or power conversion. Enabling the preload feature guarantees us that the new setting from the CCRx register will take place on the next overflow/underflow of timer counter.

11.3.11 Debugging and Timers

During a debug session, when the execution is suspended due to a hardware or software breakpoint, by default timers are not stopped. Sometimes is, instead, useful to stop a timer during debug, especially if it is used to drive an external device.

STM32 timers can be selectively configured to stop when the core is halted due to a breakpoint. The HAL macro `__HAL_DBGMCU_FREEZE_TIMx()` (where the `x` corresponds to timer number) enables this working mode of a timer. Additionally, the outputs of the timers having complementary outputs are disabled and forced to an inactive state. This feature is extremely useful for applications where the timers are controlling power switches or electrical motors. It prevents the power stages from being damaged by excessive current, or the motors from being left in an uncontrolled state when hitting a breakpoint.

The macro `__HAL_DBGMCU_UNFREEZE_TIMx()` restores the default behaviour (that is, the timer does not stop during a breakpoint).



Please, take note that, before invoking the `__HAL_DBGMCU_FREEZE_TIMx()` macro, the *MCU debug component* (DBGMCU) must be enabled by calling the `__HAL_RCC_DBGMCU_CLK_ENABLE()` macro.

11.4 SysTick Timer

SysTick is a special timer, internal to the Cortex-M core, provided by all STM32 microcontrollers. It is mainly used as timebase generator for the CubeHAL and the RTOS (if used). The most important thing about *SysTick* timer is that, if used as timebase generator for the HAL, it must be configured to generate an exception every 1ms: the exception handler will increment the *system tick counter* (a global, 32-bit wide and static variable), which can be accessed by calling the `HAL_GetTick()` routine.

The *SysTick* is a 24-bit downcounter, clocked by the AHB bus (that is, it has the same frequency of the *High (speed) Clock* - HCLK). Its clock speed can be eventually divided by 8 using the function:

```
void HAL_SYSTICK_CLKSourceConfig(uint32_t CLKSource);
```

which accepts the parameters `SYSTICK_CLKSOURCE_HCLK` and `SYSTICK_CLKSOURCE_HCLK_DIV8`.

The *SysTick* update frequency is determined by the starting value of the *SysTick* counter, which is configured using the function:

```
uint32_t HAL_SYSTICK_Config(uint32_t TicksNumb);
```

To configure the *SysTick* timer so that it generates an update event every 1ms, and assuming that it is clocked at the same speed of the AHB bus, it is sufficient to invoke the `HAL_SYSTICK_Config()` in the following way:

```
HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);
```

The `HAL_SYSTICK_Config()` routine is also responsible of enabling the timer and its `SysTick_IRQn` exception[40]. The priority of the exception can be configured at compile time setting the `TICK_INT_PRIORITY` symbolic constant in the **include/stm32XXxx_hal_conf.h** file, or by calling the `HAL_NVIC_SetPriority()` on the `SysTick_IRQn` exception, as seen in Chapter 7.

When the *SysTick* timer reaches zero, the `SysTick_IRQn` exception is raised, and the corresponding handler is called. CubeMX already provides for us the right function body, which is defined in the following way:

```
void SysTick_Handler(void) {
  HAL_IncTick();
  HAL_SYSTICK_IRQHandler();
}
```

[40]Remember that the `SysTick_IRQn` is an exception and not an interrupt, even if it is common to refer to it as interrupt. This means that we cannot use the `HAL_NVIC_EnableIRQ()` function to enable it.

The `HAL_IncTick()` automatically increments the global *SysTick* counter, while the `HAL_SYSTICK_IRQHandler()` contains nothing more than a call to the `HAL_SYSTICK_Callback()` routine, which is a callback that we can optionally implement to be notified when the timer underflows.



Read Carefully

Avoid to use slow code inside the `HAL_SYSTICK_Callback()` routine, otherwise the timebase generation could be affected. This may lead to unpredictable behaviour of some HAL modules, which rely on the exact **1ms** timebase generation.

Moreover, care must be taken when using `HAL_Delay()`. This function provides accurate delay (in milliseconds) based on *SysTick* counter. This implies that if `HAL_Delay()` is called from a peripheral ISR process, then the *SysTick* interrupt must have higher priority (numerically lower) than the peripheral interrupt. Otherwise the caller ISR process will be blocked (because the global tick counter is never incremented).

To suspend the system timebase generation, it is possible to use `HAL_SuspendTick()` routine, while to resume it the `HAL_ResumeTick()` one.

11.4.1 Use Another Timer as System Timebase Source

SysTick timer has just one relevant application: as timebase generator for the HAL or an optional RTOS. Since the *SysTick* clock cannot be easily prescaled to more flexible counting frequencies, it is not suitable to be used as a conventional timer. However, it has a relevant limitation that we will better analyze in Chapter 23: it is not suitable to be used with *tickless* modes offered by some RTOS for low-power applications. For this reason, sometimes it is important to use another timer (maybe a LPTIM) as system timebase generator. Finally, as we will discover in Chapter 23, when using an RTOS it is convenient to separate the timebase source for the HAL and for the RTOS.

CubeMX allows to easily use another timer instead of *SysTick*. To perform this, go in the *Pinout* view, then open the **RCC** entry from the *Categories* pane and select the **Timebase source**, as shown in **Figure 11.36**.

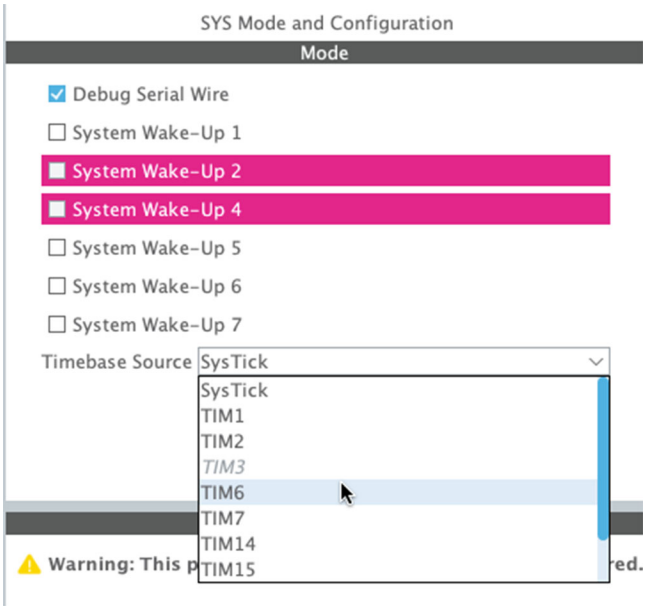


Figure 11.36: How to select another timer as system timebase source

CubeMX will generate an additional file named **stm32XXxx_hal_timebase_TIM.c** containing the definition of `HAL_InitTick()` (which contains all the necessary code to initialize the timer so that it overflows every 1ms), `HAL_SuspendTick()` and `HAL_ResumeTick()`, plus the definition of the `HAL_TIM_PeriodElapsedCallback()`, which contains the call to the `HAL_IncTick()` routine. This "overriding" of the HAL routines is possible thanks to the fact that those function are defined `__weak` inside the HAL source files.

11.5 A Case Study: How to Precisely Measure Microseconds with STM32 MCUs

Sometimes, especially when dealing with communication protocols not implemented in hardware by a peripheral, we need to precisely measure delays ranging from 1 up to a fistful of microseconds. This leads to another more general question: how to measure microseconds precisely in STM32 MCUs?

There are several ways to do this, but some methods are more accurate and other ones are more versatile among different MCUs and clock configurations.

Let us consider one member of the STM32F4 family: STM32F401RE. This micro is able to run up to 84MHz using internal RC clock. This means that ever 1µs, the clock cycles 84 times. So, we need a way to count 84 clock cycles to assert that 1µs is elapsed (I am assuming that you can tolerate the internal RC clock 1% accuracy).

Sometimes, it is common to find around delay routines like the following one:

```
void delay1US() {
    #define CLOCK_CYCLES_PER_INSTRUCTION    X
    #define CLOCK_FREQ                      Y  //IN MHZ (e.g., 16 for 16 MHZ)

    volatile int cycleCount = CLOCK_FREQ / CLOCK_CYCLE_PER_INSTRUCTION;

    while (cycleCount--);
}
```

But how to establish how many clock cycles are required to compute one step of the `while(cycleCount--)` instruction? Unfortunately, it is not simple to give an answer. Let us assume that `cycleCount` is equal to 1. Doing some tests (I will explain later how I have done them), with compiler optimizations disabled (option -O0 to GCC), we can see that in this case the whole C instruction requires 24 cycles to execute. How is it possible that? You have to figure out that our C statement is unrolled in several assembly instructions, as we can see if we disassemble the firmware binary file:

```
...
 while(counter--);
 800183e:       f89d 3003       ldrb.w  r3, [sp, #3]
 8001842:       b2db            uxtb    r3, r3
 8001844:       1e5a            subs    r2, r3, #1
 8001846:       b2d2            uxtb    r2, r2
 8001848:       f88d 2003       strb.w  r2, [sp, #3]
 800184c:       2b00            cmp     r3, #0
 800184e:       d1f6            bne.n   800183e <delay1US+0x3e>
```

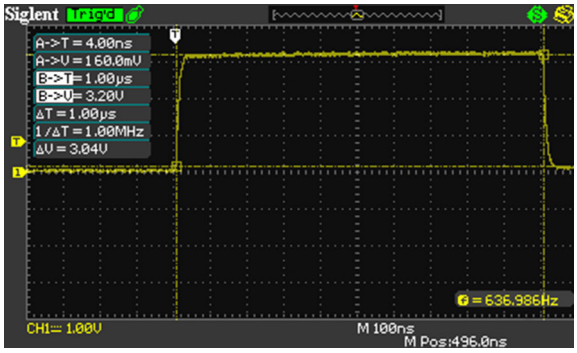
Moreover, another source of latency is related to the fetch of instructions from internal MCU flash (which differs a lot from "low-cost" STM32 MCUs and more powerful ones, like the STM32F4 and STM32F7 with the ART accelerator, which is designed to zero the flash access latency). So that instruction has a "basic cost" of 24 cycles. How many cycles are required if `cycleCount` is equal to 2? In this case the MCU requires 33 cycles, that is 9 additional cycles. This means that if we want to spin for 84 cycles, `cycleCount` has to be equal to (84-24)/9, which is about 7. So, we can write our delay function in a more general way:

```
void delayUS(uint32_t us) {
        volatile uint32_t counter = 7*us;
        while(counter--);
}
```

Testing this function with this code:

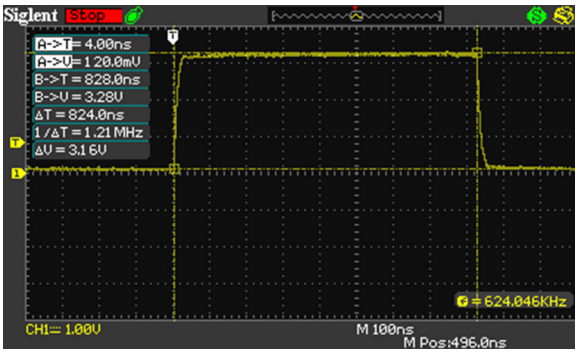
```
while(1) {
        delayUS(1);
        GPIOA->ODR = 0x0;
        delayUS(1);
        GPIOA->ODR = 0x20;
}
```

we can check, using an oscilloscope attached to PA5 pin, that we obtain the delay we are looking for:



Is this way to delay 1µs consistent? Unfortunately, the answer is no. First of all, it works well only when this specific MCU (STM32F401RE) works at full speed (84MHz). If we decide to use a different clock speed, we need to rearrange it doing tests. Second, it is subject to compiler optimizations, as we are going to see soon, and to CPU internal caches on *D-Bus* and *I-Bus* available in some STM32 microcontrollers (these caches can be eventually disabled by setting the (`PREFETCH_ENABLE`, `INSTRUCTION_CACHE_ENABLE`, `DATA_CACHE_ENABLE` in the **include/stm32XXxx_hal_conf.h** file).

Let us enable GCC optimizations for "size" (-Os). What results do we obtain? In this case we have that the `delayUS()` function costs only 72 CPU cycles, that is ~850ns. The oscilloscope confirms this:



And what happens if we enable the maximum optimization for speed (-O3)? In this case we have only 64 CPU cycles, that is our `delayUS()` lasts only ~750ns. However, this issue can be addressed using specific GCC pragma directives:

```
#pragma GCC push_options
#pragma GCC optimize ("O0")
void delayUS(uint32_t us) {
        volatile uint32_t counter = 7*us;
        while(counter--);
}
#pragma GCC pop_options
```

However, if we want to use a lower CPU frequency or we want to port our code to a different STM32 MCU, we still need to redo tests again and derive the number of cycles empirically.



However, take in account that the lower the CPU frequency is the more difficult is to delay for 1µs precisely, because the number of cycles is fixed for a given instruction, but there is less amount of cycles in the same unit of time.

So, how can we obtain a precise 1µs delay without doing tests if we change hardware setup?

One answer may be represented by setting a timer that overflows every 1µs (just setting its `Period` to the peripheral bus speed in MHz - for example, for an STM32F401RE we need to set the `Period` to (84 - 1)), and we may increment a global variable that keeps track of elapsed microseconds. This is the same way *SysTick* timer is used for the timebase generation of the HAL.

However, this approach is impractical, especially for low-speed STM32 MCUs. Generating an interrupt every 1µs (which in an STM32F0 MCU running at full speed would mean every 48 CPU cycles) would congest the MCU, reducing the overall multiprogramming degree. Moreover, the interrupt management has a non-negligible cost (from 12 up to 16 cycles), which would affect the 1µs timebase generation.
In the same way, polling the timer for the value of its counter is also impractical: a lot of time would be spent checking the counter against a starting value, and the handling of the timer overflow/underflow would impact on the timebase generation.

A more robust solution comes from the previous tests. How have I measured CPU cycles? Cortex-M3/4/7 processors can have an optional debug unit, named *Data Watchpoint and Tracing* (DWT), that provides watchpoints, data tracing, and system profiling for the processor. One register of this unit is `CYCCNT`, which counts the number of cycles performed by CPU. So, we can use this special unit available to count the number of cycles performed by the MCU during instruction execution.

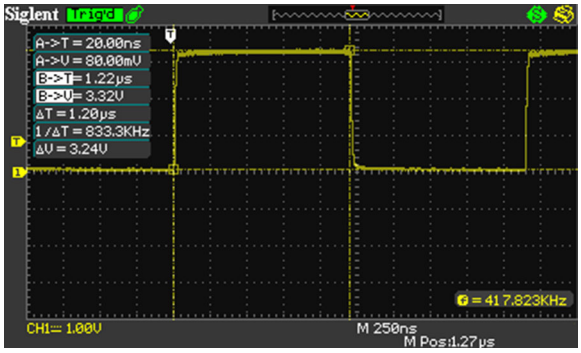
```
uint32_t cycles = 0;

/* DWT struct is defined inside the core_cm4.h file */
DWT->CTRL |= 1 ; // enable the counter
DWT->CYCCNT = 0; // reset the counter
delayUS(1);
cycles = DWT->CYCCNT;
cycles--; /* We subtract the cycle used to transfer
             CYCCNT content to cycles variable */
```

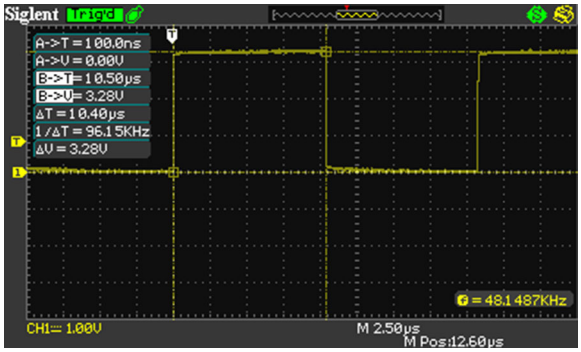
Using DWT we can build a more generic `delayUS()` routine in this way:

```
#pragma GCC push_options
#pragma GCC optimize ("O3")
void delayUS_DWT(uint32_t us) {
        volatile uint32_t cycles = (SystemCoreClock/1000000L)*us;
        volatile uint32_t start = DWT->CYCCNT;
        do  {
        } while(DWT->CYCCNT - start < cycles);
}
#pragma GCC pop_options
```

How much precise this function is? If you are interested to the best resolution at 1µs, this function will not help you, as shown by the scope.



The best performance is achieved when the higher compiler optimization level is set. As you can see, for a wanted delay of 1µs, the function gives about 1.22µs delay (22% slower). However, if we need to spin for 10µs, we obtain a real delay of 10.5µs (5% slower), which is closer to what we want.



Starting from a delay of 100µs the error is completely negligible.

Why this function is not so precise? To understand why this function is less precise from the other one, you have to figure out that we are using a series of instructions to check how many cycles are expired since the function is started (the while condition). These instructions cost CPU cycles both to update the internal CPU registers with the content of `CYCCNT` register and to do comparison and branching. However, the advantage of this function is that it automatically detects CPU speed, and it works out of the box especially if we are working on faster processors.

If you want full control over compiler optimizations, the best 1µs delay can be reached using this macro fully written in assembler:

```
#define delayUS_ASM(us) do {                          \
  asm volatile ("MOV R0,%[loops]\n                    \
                 1: \n                                \
                 SUB R0, #1\n                         \
                 CMP R0, #0\n                         \
                 BNE 1b \t"                           \
                : : [loops] "r" (16*us) : "memory"    \
                );                                    \
} while(0)
```

This is the most optimized way to write the `while(counter--)` function. Doing tests with the scope, I found that 1µs delay can be obtained when the MCU execute this loop 16 times at 84MHZ. However, this macro has to be rearranged if your processor speed is lower, and keep in mind that being a macro, it is “expanded” every time you use it, causing the increase of firmware size.

12. Analog-To-Digital Conversion

It is quite common to interface analog peripherals to a microcontroller. In the digital era, there are still a lot of devices that produce analog signals: sensors, potentiometers, transducers and audio peripherals are just few examples of analog devices that generate a variable voltage, which usually ranges in a fixed interval. By reading this voltage, we can convert it in a numerical entity useful to be processed by our firmware. For example, the TMP36 is a quite-popular temperature sensor, which produces a variable voltage proportional to the circuit operating voltage (it is said to give a *ratiometric output*) and the ambient temperature.

All STM32 microcontrollers provide at least one *Analog-to-Digital Converter* (ADC), a peripheral able to acquire several input voltages through dedicated I/O, and to convert them to a number. The input voltage is compared against a well know and fixed voltage, also known as *reference voltage*. This reference voltage can be either derived from the VDDA domain or, in MCUs with high pin count, supplied by an external and fixed reference voltage generator (those MCUs provide a dedicated pin named VREF+). The majority of STM32 MCUs provide a 12-bit ADC. Some of them from the STM32F3 and STM32H7 portfolio even a 16-bit ADC.

Differently from other STM32 peripherals seen so far, ADCs can diverge a lot between the various STM32-series and even inside a given family. For this reason, will give only an introduction to this useful peripheral, leaving to the reader the responsibility to analyze in depth the ADC in the specific MCU he is considering.

Before we analyze the features offered by the ADC in an STM32 microcontroller, and the related CubeHAL, it is best to give a quick introduction to the way this peripheral works.

12.1 Introduction to SAR ADC

In almost all STM32 microcontrollers, the ADC is implemented as a 12-bit *Successive Approximation Register* ADC[1]. Depending on the sales type and packaged used, it can have a variable number of multiplexed input channels (usually more than ten channels in the most of STM32 MCUs with high pin count), allowing to measure signals from external sources. Moreover, some internal channels are also available: a channel for internal temperature sensor (V_{SENSE}), one for internal reference voltage (V_{REFINT}), one for monitoring external V_{BAT} power supply and a channel for monitoring LCD voltage in those MCUs providing a native monochrome passive LCD controller (for example, the STM32L053 is one of these). ADCs implemented in more recent STM32 families (STM32F3/L4/L4+/L5/G0/G5/H7) are also capable of converting fully differential inputs. **Table 12.1**

[1]At the time of writing this chapter, the ADC provided by STM32F37/38xx and STM32H7 series is the only notably exception to this rule, since they provide a more accurate 16-bit ADC with Sigma-Delta(Σ-Δ) modulator. This type of ADC will not be covered in this book. ST provides the AN4207 to cover this topic. However, the HAL routines to use it have the same organization.

lists the exact ADC peripherals number and their related input sources for all STM32 MCUs equipping the nine Nucleo boards we are considering in this book.

Nucleo P/N	ADC Peripherals	Total External Inputs	Differential Inputs	Availabe Resolutions
NUCLEO-G474RE	5	Up to 16	Up to 15	6/8/10/12 bits
NUCLEO-F446RE	3	Up to 16	-	
NUCLEO-F401RE	1	16	-	
NUCLEO-F303RE	4	Up to 14	Up to 9	
NUCLEO-F103RB	2	16	-	12 bits
NUCLEO-F072RB	1	16	-	6/8/10/12 bits
NUCLEO-L476RG	3	Up to 16	Up to 15	
NUCLEO-L152RE	1	23	-	
NUCLEO-L073RZ	1	16	-	

Table 12.1: The availability of ADC peripheral in STM32 MCUs equipping Nucleo boards used in this text

A/D conversion of the various channels can be performed in single, continuous, scan or discontinuous mode. The result of the ADC is stored in a left- or right-aligned 16-bit data register. Moreover, the ADC also implements the analog watchdog feature, which allows the application to detect if the input voltage goes outside the user-defined higher or lower thresholds: if this happens, a dedicated IRQ fires.

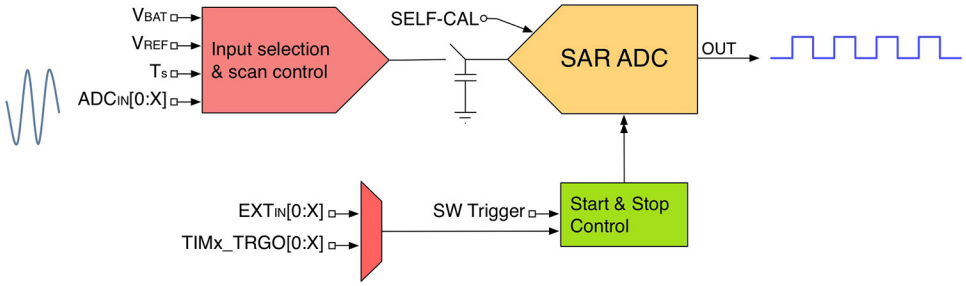


Figure 12.1: The simplified structure of an ADC

Figure 12.1 schematizes the structure of the ADC[2]. An *input selection and scan control unit* performs the selection of the input source to the ADC. Depending on the conversion mode (single, scan or continuous mode), this unit automatically switches among the input channels, so that everyone can be sampled periodically. The output from this unit feeds the ADC.

Figure 12.1 also shows another important part of the ADC: the *start and stop control* unit. Its role is to control the A/D conversion process, and it can be triggered by software or by a variable number of input sources. Moreover, it is internally connected to the TRGO line of some timers so that time-driven conversions can be automatically performed in DMA mode. We will analyze this important

[2]**Figure 12.1** is a simplified representation of the ADC. Since the ADC implementation can differ a lot among the several STM32 families, here we are going to consider a simplified view that clearly describes how the ADC unit is designed.

mode of the ADC peripheral later.

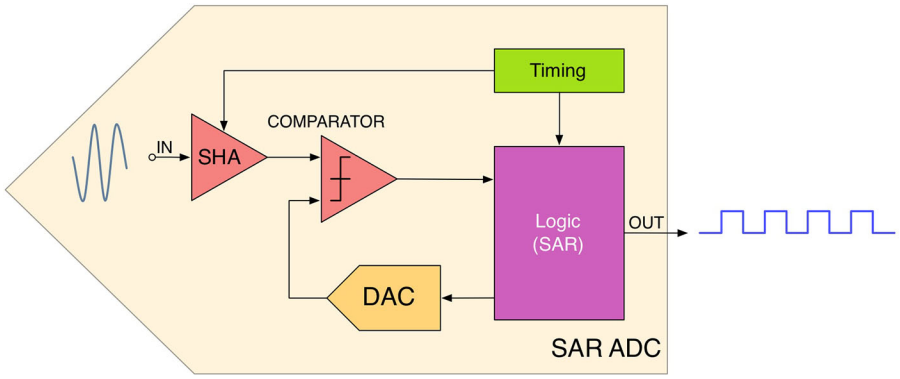


Figure 12.2: The internal structure of a SAR ADC

Figure 12.2 shows the main blocks forming the SAR ADC unit shown in **Figure 12.1**. The input signal goes through the SHA unit. As you can see in **Figure 12.1**, a switch and a capacitor are in series with the ADC input. That part represents the *Sample-and-Hold* (SHA) unit shown in **Figure 12.2**, which is a feature available in all ADCs. This unit plays the important role to keep the input signal constant during the conversion cycle. Thanks to an internal timing unit, which is regulated by a configurable clock as we will see later, the SAR constantly connects/disconnects the input source by closing/opening the "switch" in **Figure 12.1**. To keep the voltage level of the input constant, the SHA is implemented with a network of capacitors: this ensure that source signal is kept at a certain level during the A/D conversion, which is a procedure that requires a given amount of time, depending on the conversion frequency chosen.

The output from the SHA module feeds a comparator that compares it with another signal generated by an internal DAC. The result of comparison is sent to the *logic* unit, which computes the numerical representation of the input signal according to a well-characterized algorithm. This algorithm is what distinguishes SAR ADC from other A/D converters.

The *Successive Approximation* algorithm computes the voltage of the input signal by comparing it with the one generated by the internal DAC, which is a fraction of the V_{REF} voltage: if the input signal is higher than this internal reference voltage, then this is further increased until the input signal is lower. The final result will correspond to a number ranging from zero to the maximum 12-bit unsigned integer, that is $2^{12} - 1 = 4095$. Assuming $V_{REF} = 3300mV$, we have that 3300mV is represented with 4095. This means that $1_{10} = \frac{3300}{4095} \approx 0.8mV$. For example, an input voltage equal to 2.5V will be converted to:

$$x = \frac{4095}{3300mV} \times 2500mV = 3102$$

The SAR algorithm works in the following way:

1. The output *data register* is zeroed and the MSB bit is set to 1. This will correspond to a well-defined voltage level generated by the internal DAC.

2. The output of the DAC is compared with the input signal V_{IN}:
    1. if V_{IN} is higher, than the bit is left to 1;
    2. if V_{IN} is lower, than the bit is set back to 0;
3. The algorithm proceeds to the next MSB bit in the *data register* until all bits are either set to 1 or 0.

Figure 12.3 represents the conversion process made by the SAR logic unit inside a 4-bit ADC. Let us consider the path highlighted in red and let us suppose that $V_{IN} = 2700mV$ and $V_{REF} = 3300mV$. The algorithm start by setting the MSB to 1, which corresponds to $1000_2 = 8_{10}$. This means that:

$$x = \frac{3300mV}{2^4 - 1} \times 8_{10} = 1760mV$$

Being V_{IN} higher than 1760mV the 4th bit is left equal to 1 and the algorithm passes to the next MSB bit. The *data register* is now equal to $1100_2 = 12_{10}$, and the DAC generates an output equal to 2640mV. Being V_{IN} still higher than this value the 3rd bit is left again equal to 1. The register is so set to $1110_2 = 14_{10}$, which leads to an internal voltage equal to 3080mV. This time V_{IN} is lower, and the second bit is reset to zero. The algorithm now sets the 1st bit to 1, which leads to an internal voltage equal to 2860mV. This value is still higher than V_{IN} and the algorithm resets the last bit to zero. The ADC so detects that the input voltage is something close to 2640mV. Clearly, the more resolution the ADC provides, the more close to V_{IN} the converted value will be.

As you can see, the SAR algorithm essentially performs a search in a binary tree. The great advantage of this algorithm is that the conversion is performed in N-cycles, where N corresponds to the ADC resolution. So, a 12-bit ADC requires twelve cycles to perform a conversion. But how long a cycle can last? The number of cycles per seconds, that is the ADC frequency, is a performance evaluation parameter of the ADC. SAR ADCs can be really fast, especially if the ADC resolution is decreased (less sampled bit corresponds to less cycles per conversion). However, the impedance of the analog signal source, or series resistance (R_{IN}), between the source and the MCU pin causes a voltage drop across it because of the current flowing into the pin.

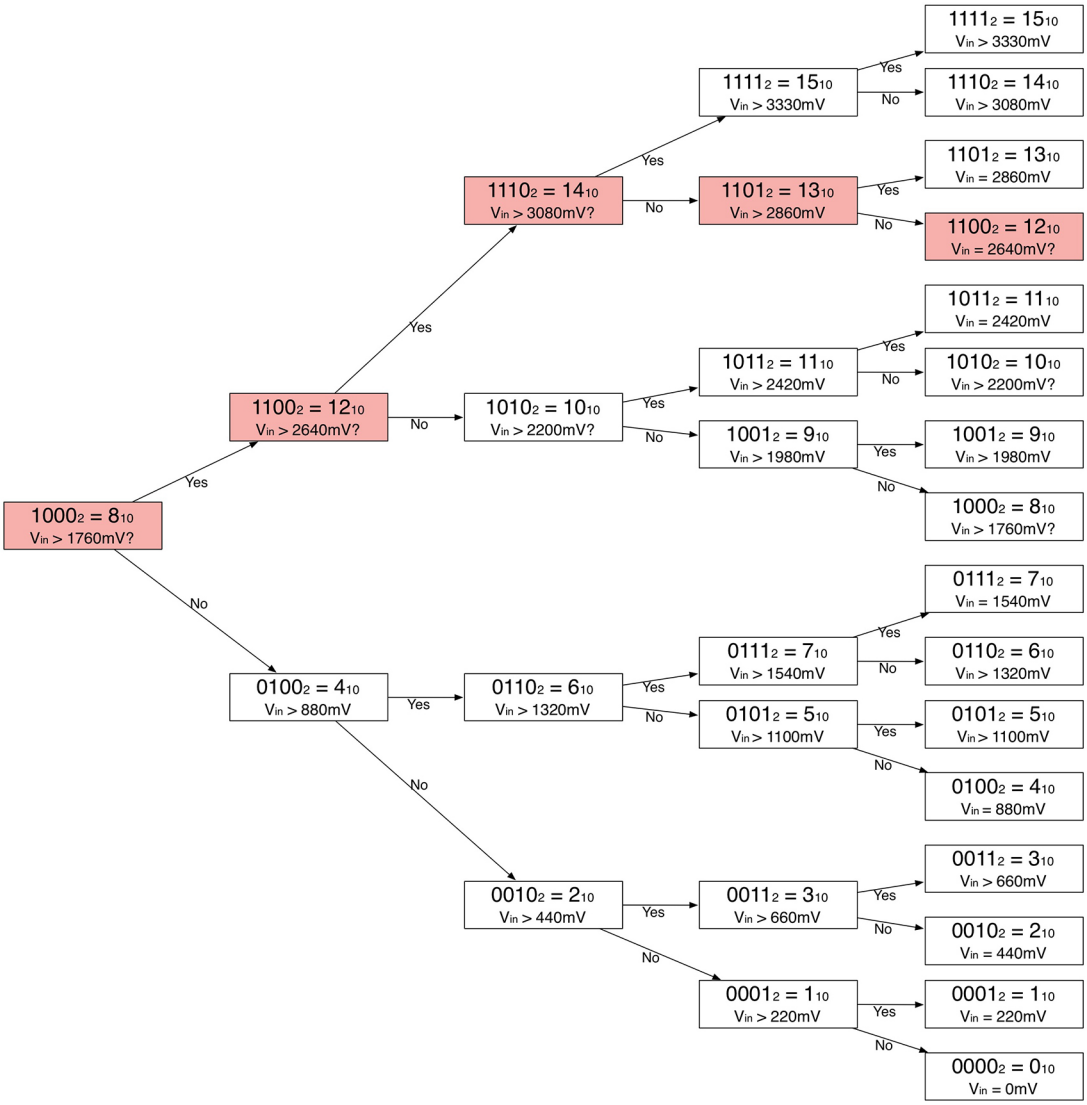


Figure 12.3: The conversion process made by a SAR ADC

The charging of the internal capacitor network (that we indicate with C_{ADC}) is controlled by the switch in **Figure 12.1** having a resistance equal to R_{ADC}. With the addition of source resistance (that is, $R_{TOT} = R_{ADC} + R_{IN}$), the time required to fully charge the hold capacitor increases. **Figure 12.4** shows the analog signal source resistance effect. The effective charging of C_{ADC} is governed by R_{TOT}, so the charging time constant becomes $t_C = (R_{ADC} + R_{IN}) \times C_{ADC}$. If the sampling time is less than the time required to fully charge the C_{ADC} through R_{TOT} ($t_S < t_C$), the digital value converted by the ADC is less than the actual value. In general, it is necessary to wait a multiple of t_C to achieve a reasonable accuracy.

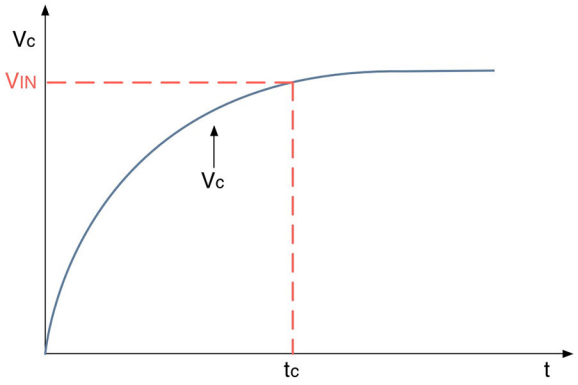


Figure 12.4: The effect of the ADC resistance on the analog signal source

For high speed A/D conversions, it is important to take in account the effect of the PCB layout and proper decoupling during board design. ST provides a well-written application note, the AN2834[3][4], which offers several and important tips to take the best from ADC integrated in STM32 MCUs.

12.2 HAL_ADC Module

After a brief introduction to the most important features offered by the ADC peripheral in STM32 microcontrollers, it is the right time to dive into the related CubeHAL APIs.

To manipulate the ADC peripheral, the HAL defines the C `struct ADC_HandleTypeDef`, which is defined in the following way:

```
typedef struct {
  ADC_TypeDef       *Instance;                  /* Pointer to ADC descriptor */
  ADC_InitTypeDef   Init;                      /* ADC initialization parameters */
  __IO uint32_t     NbrOfCurrentConversionRank; /* ADC number of current conversion rank */
  DMA_HandleTypeDef *DMA_Handle;               /* Pointer to the DMA Handler */
  HAL_LockTypeDef   Lock;                      /* ADC locking object */
  __IO uint32_t     State;                     /* ADC communication state */
  __IO uint32_t     ErrorCode;                 /* Error code */
} ADC_HandleTypeDef;
```

Let us analyze the most important fields of this `struct`.

- `Instance`: is the pointer to the ADC descriptor we are going to use. For example, `ADC1` is the descriptor of the first ADC peripheral.
- `Init`: is an instance of the C `struct ADC_InitTypeDef`, which is used to configure the ADC. We will study it more in depth in a while.
- `NbrOfCurrentConversionRank`: corresponds to the current *i-th* channel (*rank*) in a regular conversion group. We will describe it better soon.

[3] https://bit.ly/1rHj9ZN
[4] The equivalent application note for the STM32F37x/38x series is the AN4207(https://bit.ly/3AXmbxj).

- `DMA_Handle`: this is the pointer to the DMA handler configured to perform A/D conversion in DMA mode. It is automatically configured by the `__HAL_LINKDMA()` macro.

ADC configuration is performed by using an instance of the C `struct ADC_InitTypeDef`, which is defined in the following way[5]:

```
typedef struct {
  uint32_t ClockPrescaler;        /* Selects the ADC clock frequency */
  uint32_t Resolution;            /* Configures the ADC resolution mode */
  uint32_t ScanConvMode;          /* The scan sequence direction. */
  uint32_t ContinuousConvMode;    /* Specifies whether the conversion is performed in
                                     Continuous or Single mode */
  uint32_t DataAlign;             /* Specifies whether the ADC data alignment
                                     is left or right */
  uint32_t NbrOfConversion;       /* Specifies the number of ranks that will be converted
                                     within the regular group sequencer */
  uint32_t NbrOfDiscConversion;   /* Specifies the number of discontinuous conversions in
                                     which the  main sequence of regular group */
  uint32_t DiscontinuousConvMode; /* Specifies whether the conversion sequence of regular
                                     group is performed in Complete-sequence/Discontinuous
                                     sequence */
  uint32_t ExternalTrigConv;      /* Select the external event used to trigger the start
                                     of conversion */
  uint32_t ExternalTrigConvEdge;  /* Select the external trigger edge and enable it */
  uint32_t DMAContinuousRequests; /* Specifies whether the DMA requests are performed in
                                     one shot or in continuous mode */
  uint32_t EOCSelection;          /* Specifies what EOC (End Of Conversion) flag is used
                                     for conversion polling and interruption */
} ADC_InitTypeDef;
```

Let us analyze the most relevant field of this `struct`.

- `ClockPrescaler`: defines the speed of the clock (ADCCLK) for the analog circuitry part of ADC. In the previous paragraph we have seen that the ADC has an internal timing unit that controls the switching frequency of the input switch (see **Figure 12.2**). The ADCCLK establishes the speed of this timing unit, and it impacts on the number of samples per seconds, because it defines the amount of time used by each conversion cycle. This clock is generated from the peripheral clock divided by a programmable prescaler that allows the ADC to work at f_{PCLK}/2,/4,/6 or /8 (refer to the datasheets of the specific MCU for the maximum values of ADCCLK and its prescaler). In some STM32 MCUs the ADCCLK can also be derived from the HSI oscillator. The value of this field affects the ADCCLK speed of all ADCs implemented in the MCU.

[5]The `ADC_InitTypeDef struct` slightly differs from the one defined in CubeF0 and CubeL0 HALs. This because the ADC in those families does not provide the ability to define custom input sampling sequences (by assigning *rank* values). Moreover, the ADC in those families provide the ability to perform oversampling of the input signal, and in CubeL0 HAL it is possible to enable dedicated low-power features offered by the ADC in those MCUs. For more information, refer to the CubeHAL source code.

- `Resolution`: apart from STM32F1 MCUs, whose ADC does not allow to select the resolution of samples (see **Table 12.1**), using this field it is possible to define the A/D conversion resolution. It can assume a value from **Table 12.2**. The higher is the resolution the fewer conversions are possible in a second. If speed is not relevant for your application, it is strongly suggested to set the bit resolution to the maximum and the conversion speed to the minimum.
- `ScanConvMode`: this field can assume the value `ENABLE` or `DISABLE` and it is used to enable/disable the scan conversion mode. More about this later.
- `ContinuousConvMode`: specifies if the conversion is performed in single or continuous mode, and it can assume the value `ENABLE` or `DISABLE`. More about this later.
- `NbrOfConversion`: specifies the number of channels of the regular group that will be converted in scan mode. This number shall be equal to the number of configured ranks.
- `DataAlign`: specifies the data align of the converted result. ADC *data register* is implemented as half-word register. Since only 12-bits are used to store the conversion, this parameter establishes how these bits are aligned inside the register. It can assume the value `ADC_DATAALIGN_LEFT` or `ADC_DATAALIGN_RIGHT`.
- `ExternalTrigConvEdge`: select the external trigger source to drive conversion using a timer.
- `EOCSelection`: depending on the conversion mode (single or continuous conversions) the ADC sets the *End Of Conversion* (EOC) flag accordingly. This field is used by the ADC polling or interrupt API to determine when a conversion is completed, and it can assume the values `ADC_EOC_SEQ_CONV` for continuous conversion, and `ADC_EOC_SINGLE_CONV` for single conversions.

Table 12.2: Available resolution options for the ADC

ADC resolution	Description
`ADC_RESOLUTION_12B`	ADC 12-bit resolution
`ADC_RESOLUTION_10B`	ADC 10-bit resolution
`ADC_RESOLUTION_8B`	ADC 8-bit resolution
`ADC_RESOLUTION_6B`	ADC 6-bit resolution

Before we can start doing a practical example, we have to analyze another two topics: how input channels are configured and how their input signals are sampled.

12.2.1 Conversion Modes

ADCs implemented in STM32 MCUs provide several conversion modes useful to deal with different application scenarios. Now we are going to brief introduce the most relevant of them: the AN3116[6] from ST describes all possible conversion modes provided by the ADC.

12.2.1.1 Single-Channel, Single Conversion Mode

This is the simplest ADC mode. In this mode, the ADC performs the single conversion (single sample) of a single channel, as shown in **Figure 12.5**, and stops when conversion is finished.

[6] https://bit.ly/39EFUpk

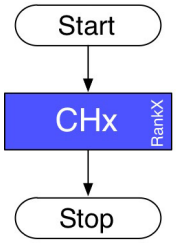


Figure 12.5: Single-channel, single conversion mode

12.2.1.2 Scan Single Conversion Mode

This mode, also called *multichannel single mode* in some ST documents, is used to convert some channels successively in independent mode. Using *ranks*, you can use this ADC mode to configure any sequence of up to 16 channels successively with different sampling times and in custom orders. You can, for example, carry out the sequence shown in **Figure 12.6**. In this way, you do not have to stop the ADC during the conversion process in order to reconfigure the next channel with a different sampling time. This mode saves additional CPU load and heavy software development. Scan conversions are carried out in DMA mode.

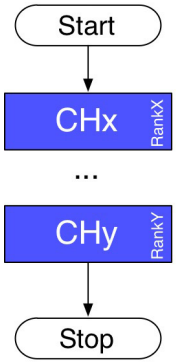


Figure 12.6: Scan single conversion mode

For example, this mode can be used when starting a system depends on some parameters like knowing the coordinates of the arm's tip in a manipulator arm system. In this case, you have to read the position of each articulation in the manipulator arm system at power-on to determine the coordinates of the arm's tip. This mode can also be used to make single measurements of multiple signal levels (voltage, pressure, temperature, etc.) to decide if the system can be started or not in order to protect the people and equipment.

12.2.1.3 Single-Channel, Continuous Conversion Mode

This mode converts a single channel continuously and indefinitely in regular channel conversion. The continuous mode feature allows the ADC to work in the background. The ADC converts the channels continuously without any intervention from the CPU. Additionally, the DMA can be used in circular mode, thus reducing the CPU load.

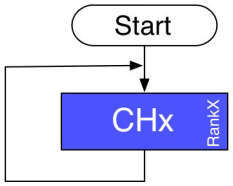


Figure 12.7: Single-channel, continuous conversion

For example, this ADC mode can be implemented to monitor a battery voltage, the measurement and regulation of an oven temperature using a PID, etc.

12.2.1.4 Scan Continuous Conversion Mode

This mode is also called *multichannel continuous mode* and it can be used to convert some channels successively with the ADC in independent mode. Using *ranks*, you can configure any sequence of up to 16 channels successively with different sampling times and different orders. This mode is similar to the *multichannel single conversion mode* except that it does not stop converting after the last channel of the sequence, but it restarts the conversion sequence from the first channel and continues indefinitely. Scan conversions are carried out in DMA mode.

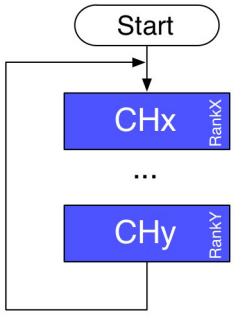


Figure 12.8: Scan continuous conversion mode

This mode may be used, for example, to monitor multiple voltages and temperatures in a multiple battery charger. The voltage and temperature of each battery are read during the charging process. When the voltage or the temperature reaches the maximum level, the corresponding battery should be disconnected from the charger.

12.2.1.5 Injected Conversion Mode

This mode is intended for use when conversion is triggered by an external event or by software. The injected group has priority over the regular channel group. It interrupts the conversion of the current channel in the regular channel group.

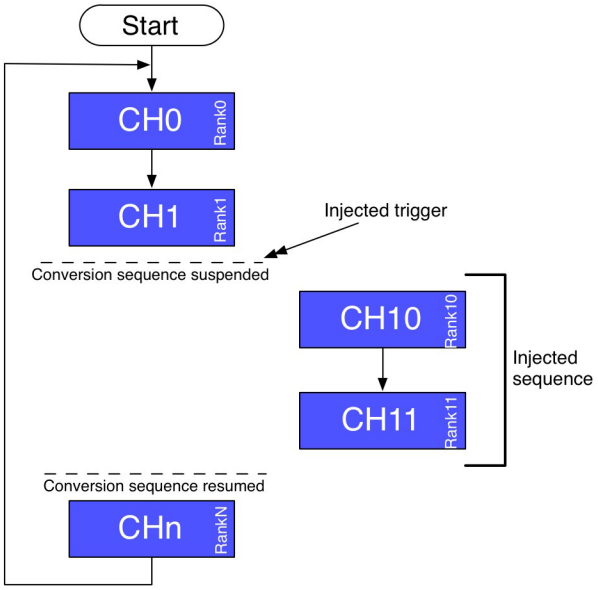


Figure 12.9: Injected conversion mode

For example, this mode can be used to synchronize the conversion of channels to an event. It is interesting in motor control applications where transistor switching generates noise that impacts ADC measurements and results in wrong conversions. Using a timer, the injected conversion mode can thus be implemented to delay the ADC measurements to after the transistor switching.

12.2.1.6 Dual Modes

Dual mode is available in STM32 microcontrollers that feature two ADCs: ADC1 master and ADC2 slave. ADC1 and ADC2 triggers are synchronized internally for regular and injected channel conversion. ADC1 and ADC2 work together. In some devices, there are up to 3 ADCs: ADC1, ADC2 and ADC3. In this case ADC3 always works independently and is not synchronized with the other ADCs.

Dual mode works so that when the conversion ends the result from ADC1 and ADC2 is simultaneously saved inside the ADC1 32-bit *data register*. By separating the two results, we can acquire the data coming from two separated channels at the same time.

For more information regarding dual mode, refer to AN3116[7] from ST.

12.2.2 Channel Selection

Depending on the STM32 family and package used, ADCs in STM32 MCUs can convert signals from a variable number of channels. In F0 and L0 families the allocation of channel is fixed: the first one is always IN0, the second IN1 and so on. User can decide only if a channel is enabled or not. This means that in scan mode the first sampled channel will be always IN0, the second IN1 and so on. Other

[7] https://bit.ly/39EFUpk

STM32 MCUs, instead, offer the notion of *group*. A group consists of a sequence of conversions that can be done on any channel and in any order. While input channels are fixed and bound to specific MCU pins (that is, IN0 is the first channel, IN1 the second and so on), they can be logically reordered to form custom sampling sequences. The reordering of channels is performed by assigning to them an index ranging from 1 to 16. This index is called *rank* in the CubeHAL.

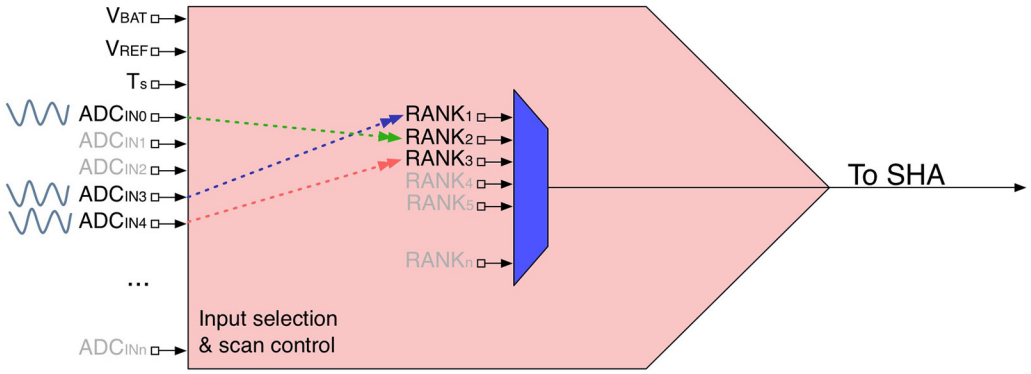


Figure 12.10: How input channels can be reordered using *ranks*

The **Figure 12.10** shows this concept. Although the IN4 channel is fixed (for example, it is connected to PA4 pin in an STM32F401RE MCU), it can be logically assigned to the *rank 1* so that it will be the first channel to be sampled. Those MCUs offering this possibility also allow to select the sampling speed of each channel individually, differently from F0/L0 MCUs where the configuration is ADC-wide.

The channel/rank configuration is performed by using an instance of the C `struct ADC_Channel-ConfTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t Channel;              /* Specifies the channel to configure into ADC rank */
  uint32_t Rank;                 /* Specifies the rank ID */
  uint32_t SamplingTime;         /* Sampling time value for the selected channel */
  uint32_t Offset;               /* Reserved for future use, can be set to 0 */
} ADC_ChannelConfTypeDef;
```

- `Channel`: specifies the channel ID. It can assume the value `ADC_CHANNEL_0`, `ADC_CHANNEL_-1`...`ADC_CHANNEL_N`, depending on the effective number of available channels.
- `Rank`: correspond to the *rank* associated to the channel. It can assume a value from 1 to 16, which is the maximum number of user-definable *ranks*.
- `SamplingTime`: specifies the sampling time value to be set for the selected channel, and it corresponds to the number or ADC cycles. This number cannot be arbitrary, but it is part of a selected list of values. As we will see later, CubeMX helps a lot offering the list of admissible values for the specific MCU you are considering.

There exist two groups for each ADC:

- A *regular group*, made of up to 16 channels, which corresponds to the sequence of sampled channels during a scan conversion.
- An *injected group*, made of up to 4 channels, which corresponds to the sequence of injected channel if an injected conversion is performed.



The CubeHAL and its non-linear evolution

When working with the CubeHAL, especially if you are new to it and you are following this book's samples to start learning it, you have *to pay close attention* while porting some code made for a given STM32 family to another one. For example, consider the `ADC_ChannelConfTypeDef.Rank` field that in many STM32 MCUs can assume the values `1..16`. For those MCUs, it is ok to indicate the rank with the decimal number. But for more recent STM32 MCUs, that value is a result of a computed value that needs to follow a precise scheme. This happens, for example, for the G474RE MCU, where the rank must be specified with the CubeHAL_LL `LL_ADC_REG_RANK_1` macro, otherwise the configuration is totally broken.
ST should increase the effort in standardizing several things inside the CubeHAL. Unless totally sure, it is always a good thing to start generating the initialization code with CubeMX before re-using other code. This author spent a whole night and the day after before realizing this paramount thing while migrating this chapter's examples. Sad story.

12.2.3 ADC Resolution and Conversion Speed

It is possible to perform faster conversions by reducing the ADC resolution[8]. The sampling time, in fact, is defined by a fixed number of cycles (usually 3) plus a variable number of cycles depending on the A/D resolution. The minimum conversion time for each resolution is then as follows:

- 12 bits: 3 + ∼12 = 15 ADCCLK cycles
- 10 bits: 3 + ∼10 = 13 ADCCLK cycles
- 8 bits: 3 + ∼8 = 11 ADCCLK cycles
- 6 bits: 3 + ∼6 = 9 ADCCLK cycles

By reducing the resolution is so possible to increase the number of maximum samples per seconds, reaching even more then 15Msps in some STM32 MCUs. Remember that the ADCCLK is derived from the peripheral clock: this means that SYSCLK and PCLK speeds impact on the maximum number of samples per second.

[8]This is not possible in STM32F1 MCUs.

12.2.4 A/D Conversions in Polling Mode

Like the majority of STM32 peripherals, the ADC can be driven in three modes: *polling*, *interrupt* and *DMA* mode. As we will see later, a timer can eventually drive this last mode so that A/D conversions take place at regular interval. This is extremely useful when we need to sample signals at a given frequency, like in audio applications.

Once the ADC controller is configured by using an instance of the `ADC_InitTypeDef struct` passed to the `HAL_ADC_Init()` routine, we can start the peripheral using the `HAL_ADC_Start()` function. Depending on the conversion mode chosen, ADC will convert each selected input continuously or just once: in this case, to convert again the selected inputs we need to call the `HAL_ADC_Stop()` function before calling again the `HAL_ADC_Start()`.

In *polling mode*, we use the function

```
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef* hadc, uint32_t Timeout);
```

to determine when the A/D conversion is complete, and the result is available inside the ADC *data register*. The function accepts the pointer to the ADC handler descriptor and a `Timeout` value, which represents the maximum time expressed in milliseconds we are willing to wait. Alternatively, we can pass the `HAL_MAX_DELAY` to wait indefinitely.

To retrieve the result, we can use the function:

```
uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef* hadc);
```

We are now finally ready to analyze a complete example. We will start by seeing the APIs used to perform conversions in *polling mode*. As you will see, there is nothing new compared to what seen so far with other peripherals.

The example we are going to study does a simple thing: it uses the internal temperature sensor available in all STM32 MCUs as source for the ADC. The temperature sensor is connected to an internal ADC input. The exact input number depends on the specific MCU family and package. For example, in an STM32F401RE MCU the temperature sensor is connected to the IN18 of ADC1 peripheral. However, the HAL is designed to abstract this specific aspect. Before we analyze the real code, it is best to give a quick look at the electrical characteristics of the temperature sensor, which are reported in the datasheet of the MCU you are considering.

Symbol	Parameter	Min	Typ	Max	Unit
T_L(1)	V_{SENSE} linearity with temperature	-	±1	±2	°C
Avg_Slope(1)	Average slope	-	2.5	-	mV/°C
V_{25}(1)	Voltage at 25 °C	-	0.76	-	V
t_{START}(2)	Startup time	-	6	10	µs
T_{S_temp}(2)	ADC sampling time when reading the temperature (1 °C accuracy)	10	-	-	µs

Table 12.3: Electrical characteristics of the temperature sensor in an STM32F401RE MCU

Table 12.3 shows the characteristics of the temperature sensor in an STM32F401RE MCU. It has a typical accuracy of 1°C[9] and an average slope of 2.5mV/°C. Moreover, the temperature sensor junction works so that at 25°C the voltage drops is 760mV. This means that, to calculate the detected temperature we can use the formula:

$$Temp(^{\circ}C) = \frac{(V_{SENSE} - V_{25})}{Avg_Slope} + 25 \quad [1]$$

The following code shows how to perform an A/D conversion of the internal temperature sensor output in an STM32F401RE MCU.

Filename: `Core/Src/main-ex1.c`

```
31 /* Private variables ---------------------------------------------------------*/
32 ADC_HandleTypeDef hadc1;
33 UART_HandleTypeDef huart2;
34 char msg[30];
35 uint16_t rawValue;
36 float temp;
37
38 int main(void) {
39   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
40   HAL_Init();
41
42   /* Configure the system clock */
43   SystemClock_Config();
44
45   /* Initialize all configured peripherals */
46   MX_GPIO_Init();
47   MX_USART2_UART_Init();
48   MX_ADC1_Init();
```

[9] STM32 internal temperature sensors are factory calibrated during the IC production. Two temperatures are usually sampled at 30°C and 110°C. They are called `TS_CAL1` and `TS_CAL2` respectively. The detected temperatures are stored inside the non-volatile system memory. The exact memory address is reported in the specific datasheet. Using this data, it is possible to perform a linearization of the detected temperatures, so that the error is leaded back in the typical accuracy value of 1°C. ST provides an application note dedicated to this topic: the AN3964(https://bit.ly/1XfbuO6). However, keep in mind that the internal temperature sensor measures the temperature of the IC (and therefore of the PCB). According to the specific STM32 family, the MCU running frequency, operations performed, peripherals enabled, power section and so on, the detected temperature can be much higher than the effective ambient temperature. For example, this author has verified that an STM32F7 MCU running at 200MHz has a working temperature of about ~45°C, at a room temperature of 20°C.

```
49
50    /* Starts the ADC */
51    HAL_ADC_Start(&hadc1);
52
53    while (1) {
54      HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
55
56      rawValue = HAL_ADC_GetValue(&hadc1);
57      temp = ((float)rawValue) / 4095 * 3300;
58      temp = ((temp - 760.0) / 2.5) + 25;
59
60      sprintf(msg, "ADC rawValue: %hu\r\n", rawValue);
61      HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
62
63      sprintf(msg, "Temperature: %f\r\n", temp);
64      HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
65    }
66  }
67
68  static void MX_ADC1_Init(void) {
69    ADC_ChannelConfTypeDef sConfig = {0};
70
71    /** Configure the global features of the ADC (Clock, Resolution,
72     *  Data Alignment and number of conversion) */
73    hadc1.Instance = ADC1;
74    hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
75    hadc1.Init.Resolution = ADC_RESOLUTION_12B;
76    hadc1.Init.ScanConvMode = DISABLE;
77    hadc1.Init.ContinuousConvMode = ENABLE;
78    hadc1.Init.DiscontinuousConvMode = DISABLE;
79    hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
80    hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
81    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
82    hadc1.Init.NbrOfConversion = 1;
83    hadc1.Init.DMAContinuousRequests = DISABLE;
84    hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
85    HAL_ADC_Init(&hadc1);
86
87    /** Configure for the selected ADC regular channel its corresponding
88     *  rank in the sequencer and its sample time. */
89    sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
90    sConfig.Rank = 1;
91    sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
92    HAL_ADC_ConfigChannel(&hadc1, &sConfig);
93  }
```

The first part to analyze is the function `MX_ADC1_Init()`, which initializes the ADC1 peripheral. First

of all, at line 74 the ADC is configured so that the ADCCLK (that is, the clock of the analog part of the ADC) is the half of the PCLK2 frequency[10], which in an STM32F401RE running at its maximum speed is 84MHz. Next, the ADC resolution is configured to the maximum: 12-bit. The *scan conversion mode* is disabled (line 76), while the *continuous conversion mode* is enabled (line 77) so that we can repeatedly poll for a conversion without stopping and then restarting the ADC. Therefore, the EOC flag is set to `ADC_EOC_SEQ_CONV` at line 84. Take note that the parameter `NbrOfConversion` at line 82 is completely meaningless and redundant in this case, because the *single conversion mode* automatically assumes that the number of sampled channels is equal to 1.

Lines [89:92] configure the temperature sensor channel and assign it the *rank 1*: even if we are not performing a *scan conversion*, we need to specify the *rank* for the channel used. The sampling time is set to 480 cycles: this means that, given the clock speed of 84MHz, and considered that the ADCCLK is set to the half of the PCLK speed, we have that an A/D conversion is performed every 10µs[11].



Why we are choosing that conversion speed? The reason comes from the **Table 12.3**, which states that the ADC sampling time, T_{S_temp}, is equal to 10µs to have an accuracy of 1°C. For example, if you increase the speed to 3 cycles, by setting the `SamplingTime` field to `ADC_SAMPLETIME_3CYCLES` you will see that the converted result is often completely wrong.

Always in the same table you can find another interesting data: the temperature sensor start time (that is, the time needed to stabilize the output voltage when the sensor is enabled) ranges between 6 and 10µs. However, we do not need to take care of this aspect, since the `HAL_ADC_ConfigChannel()` routine is designed to handle the startup time correctly. This means that, the function will perform busy-wait for 10µs to allow the temperature sensor to settle.

We can now focus on the `main()` routine. Once the ADC1 peripheral is started (line 51), we start an infinite loop that cyclically polls the ADC for the A/D conversion. When completed, we can retrieve the converted value and apply equation [1] to compute the temperature in Celsius degrees. The result is finally printed on the UART2 interface.



Read Carefully

To work properly, this example requires the support to `float` datatypes in `newlib-nano`. Follow the instructions reported in Chapter 5 in the paragraph dedicated to I/O retargeting.

F1

[10]Please, take note that at the time of writing this chapter - September 2021 - CubeMX does not allow to set the maximum ADC frequency to the half of the PCLK2 frequency, claiming that this is not possible in F401RE P/N. This appears not true to this author, because it was perfectly possible in older CubeMX and, at the same time, the official ST datasheet clearly states that the ADC is able to run at the half of the clock speed of PCLK2.

[11]That number comes from the fact that the ADCCLK interface, running at 48MHz, performs 48 cycles every 1µs. So, 480 cycles divided for 48 cycles/µs gives 10µs.

The HAL_ADC module in the CubeF1 HAL slightly differs from the other HALs. To start a conversion driven by software it is required that the parameter hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START is specified during the ADC initialization. This completely differs from what other HALs do, and it is not clear why ST developers have adopted this different approach. Moreover, even CubeMX offers a different configuration to take in account this peculiarity when it generates the corresponding initialization code. Refer to book examples for the complete configuration procedure.

12.2.5 A/D Conversions in Interrupt Mode

Performing an A/D conversion in *interrupt* mode is not too much different from what seen so far. As usual, we have to define the ISR connected to the ADC global interrupt, to assign a wanted interrupt priority and to enable the corresponding IRQ. Like all other HAL peripherals, we have to call the HAL_ADC_IRQHandler() from the ADC ISR and to implement the callback routine HAL_ADC_ConvCpltCallback(), which is automatically called by the HAL when a conversion ends. Finally, all the ADC related interrupts are enabled by starting the ADC using the HAL_ADC_Start_IT() function.

The following example just shows how to perform a conversion in *interrupt* mode. The initialization code for the ADC is the same used in the previous example.

```
int main(void) {
  HAL_Init();
  Nucleo_BSP_Init();

  /* Initialize all configured peripherals */
  MX_ADC1_Init();

  HAL_NVIC_SetPriority(ADC_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(ADC_IRQn);

  HAL_ADC_Start_IT(&hadc1);

  while (1);
}

void ADC_IRQHandler(void) {
   HAL_ADC_IRQHandler(&hadc1);
}

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
  char msg[30];
  uint16_t rawValue;
  float temp;

  rawValue = HAL_ADC_GetValue(&hadc1);
```

```
  temp = ((float)rawValue) / 4095 * 3300;
  temp = ((temp - 760.0) / 2.5) + 25;

  sprintf(msg, "rawValue: %hu\r\n", rawValue);
  HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);

  sprintf(msg, "Temperature: %f\r\n", temp);
  HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
}
```

12.2.6 A/D Conversions in DMA Mode

The most interesting mode to drive the ADC peripheral is the *DMA* one. This mode allows to perform conversions without the intervention of the CPU and, using DMA in circular mode, we can easily setup ADC so that it performs continuous conversions. Moreover, as we will discover next, this mode is perfect to drive conversions using a timer, allowing to sample input signal at a fixed sampling rate. It is also mandatory to use the ADC peripheral in DMA mode when we want to perform conversions of multiple channels using *scan mode*.

To perform A/D conversions in DMA mode, as usual the steps involved in this process are the following ones:

- Setup the ADC peripheral according to the wanted conversion mode (scan single, scan continuous, etc).
- Setup the DMA channel/stream corresponding to the ADC controller used.
- Link the DMA handler descriptor to the ADC handler using the `__HAL_LINKDMA()` macro.
- Enable the DMA and the IRQ associated to the DMA stream used.
- Start the ADC in DMA mode using the `HAL_ADC_Start_DMA()` passing the reference to the array used to store acquired data from the ADC.
- Be prepared to capture EOC event by defining the `HAL_ADC_ConvCpltCallback()`[12] callback.

The following example, designed to run on an STM32F401RE MCU, shows how to perform a *single scan* conversion using DMA mode. The first part we are going to analyze is the one related to the setup of both ADC peripheral and DMA controller.

[12]The `HAL_ADC` module also provides the `HAL_ADC_ConvHalfCpltCallback()` callback called when half of the scan conversion sequence is completed.

Filename: `Core/Src/main-ex2.c`

```
80  static void MX_ADC1_Init(void) {
81    ADC_ChannelConfTypeDef sConfig = {0};
82
83    /** Configure the global features of the ADC (Clock, Resolution,
84     *  Data Alignment and number of conversion) */
85    hadc1.Instance = ADC1;
86    hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV8;
87    hadc1.Init.Resolution = ADC_RESOLUTION_12B;
88    hadc1.Init.ScanConvMode = ENABLE;
89    hadc1.Init.ContinuousConvMode = DISABLE;
90    hadc1.Init.DiscontinuousConvMode = DISABLE;
91    hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
92    hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
93    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
94    hadc1.Init.NbrOfConversion = 3;
95    hadc1.Init.DMAContinuousRequests = DISABLE;
96    hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
97    HAL_ADC_Init(&hadc1);
98
99    /** Configure for the selected ADC regular channel its corresponding
100    *  rank in the sequencer and its sample time. */
101   sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
102   sConfig.Rank = 1;
103   sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
104   HAL_ADC_ConfigChannel(&hadc1, &sConfig);
105
106   sConfig.Rank = 2;
107   HAL_ADC_ConfigChannel(&hadc1, &sConfig);
108
109   sConfig.Rank = 3;
110   HAL_ADC_ConfigChannel(&hadc1, &sConfig);
111
112   /* ADC1 DMA Init */
113   hdma_adc1.Instance = DMA2_Stream0;
114   hdma_adc1.Init.Channel = DMA_CHANNEL_0;
115   hdma_adc1.Init.Direction = DMA_PERIPH_TO_MEMORY;
116   hdma_adc1.Init.PeriphInc = DMA_PINC_DISABLE;
117   hdma_adc1.Init.MemInc = DMA_MINC_ENABLE;
118   hdma_adc1.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
119   hdma_adc1.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
120   hdma_adc1.Init.Mode = DMA_NORMAL;
121   hdma_adc1.Init.Priority = DMA_PRIORITY_LOW;
122   hdma_adc1.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
123   if (HAL_DMA_Init(&hdma_adc1) != HAL_OK);
124
125   __HAL_LINKDMA(&hadc1,DMA_Handle,hdma_adc1);
```

```
126
127 }
128
129 static void MX_DMA_Init(void) {
130   /* DMA controller clock enable */
131   __HAL_RCC_DMA2_CLK_ENABLE();
132
133   /* DMA interrupt init */
134   /* DMA2_Stream0_IRQn interrupt configuration */
135   HAL_NVIC_SetPriority(DMA2_Stream0_IRQn, 1, 0);
136   HAL_NVIC_EnableIRQ(DMA2_Stream0_IRQn);
137 }
138
139 void DMA2_Stream0_IRQHandler(void) {
140   HAL_DMA_IRQHandler(&hdma_adc1);
141 }
```

The `MX_ADC1_Init()` configures the ADC so that it perform a *single scan* of the three channels. The ADCCLK is set to the lowest one (line 86), and the *scan mode* is enabled (line 88). As you can see, we are configuring the ADC so that it always performs a conversion from the internal temperature sensor: this is not useful, but unfortunately Nucleo boards do not embed analog peripherals to play with.

The init code goes on configuring the DMA2 Stream0/Channel0 so that peripheral-to-memory transfers are performed when the ADC completes a conversion. Clearly, the conversion sequence is specified by the *rank* assigned to a channel. Since the ADC *data register* is 16-bit wide, we configure the DMA so that a half-word transfer is performed (lines [118:119]). Finally, the function `MX_DMA_Init()` enables the IRQ connected to the DMA2_Stream0 and its ISR will calls `HAL_DMA_IRQHandler()` routine. This will cause that the `HAL_ADC_ConvCpltCallback()` function is automatically called by the HAL when the scan conversion ends. The callback sets a global variable used to signal the end of conversion.

Filename: `Core/Src/main-ex2.c`

```
37 char msg[30];
38 uint16_t rawValues[3];
39 float temp;
40 volatile uint8_t convCompleted = 0;
41
42 int main(void) {
43   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
44   HAL_Init();
45
46   /* Configure the system clock */
47   SystemClock_Config();
48
49   /* Initialize all configured peripherals */
```

```
50    MX_DMA_Init();
51    MX_ADC1_Init();
52    MX_GPIO_Init();
53    MX_USART2_UART_Init();
54
55    /* Starts the ADC in DMA mode*/
56    HAL_ADC_Start_DMA(&hadc1, (uint32_t*)rawValues, 3);
57
58    while(!convCompleted);
59
60    HAL_ADC_Stop_DMA(&hadc1);
61
62    for(uint8_t i = 0; i < hadc1.Init.NbrOfConversion; i++) {
63      temp = ((float)rawValues[i]) / 4095 * 3300;
64      temp = ((temp - 760.0) / 2.5) + 25;
65
66      sprintf(msg, "rawValue %d: %hu\r\n", i, rawValues[i]);
67      HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
68
69      sprintf(msg, "Temperature %d: %f\r\n",i,  temp);
70      HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
71    }
72
73    while (1);
74  }
75
76  void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef*hadc) {
77    convCompleted = 1;
78  }
```

The above lines of code show the `main()` function. The ADC is started in DMA mode at line 56, by passing the pointer to the `rawValues` array and the number of conversions: this has to correspond to the `hadc1.Init.NbrOfConversion` field at line 94 (and so to the number of configured ranks). Finally, when the `convCompleted` variable is set to 1 by the `HAL_ADC_ConvCpltCallback()` routine (lines [76:78]), the content of the `rawValues` array is converted and the result is printed on the UART2 interface. Please take note that the `HAL_ADC_Stop_DMA()` is invoked at line 60: this operation is not performed to stop the conversion (which automatically stops after the three samples), but to allow successive usages of the ADC peripheral in DMA mode (otherwise the conversion will not start).

STM32L0

Owners of STM32L0 based boards will find a slightly different sample. In the STM32L0 MCUs the ranks are fixedly assigned to ADC channels and the sequence reflects the channel numbering scheme. Thus, the number of ranks in the sequence is defined by the number of channels enabled, rank of each channel is defined by channel number (channel 0 fixed on rank 0, channel 1 fixed on rank1, etc.). This implies that the temperature sensor channel cannot be assigned to more than 1 rank.

For this reason, this and the next example will cycle on just one channel for the STM32L073RZ examples.

12.2.6.1 Convert Multiple Times the Same Channel in DMA Mode

To perform a given number of conversions of the same channel (or the same channel sequence) in DMA mode, you need to do the following way:

- Set the `hadc.Init.ContinuousConvMode` field to `ENABLE`.
- Allocate a sufficient-sized buffer.
- Pass to the `HAL_ADC_Start_DMA()` the number of wanted acquisitions.

12.2.6.2 Multiple and not Continuous Conversions in DMA Mode

To perform multiple conversions in DMA mode, you need to do the following steps:

- Set the `hadc.Init.DMAContinuousRequests` field to `ENABLE`.
- Call the `HAL_ADC_Start_DMA()` to start conversions in DMA mode.

If, instead, the `hadc.Init.DMAContinuousRequests` field is set to `DISABLE`, then you need to call the `HAL_ADC_Stop_DMA()` at the end of every conversion sequence and before calling the `HAL_ADC_Start_DMA()` again. Otherwise, the conversion will not start.

12.2.6.3 Continuous Conversions in DMA Mode

To perform continuous conversions in DMA mode, you need to do the following steps:

- Set the `hadc.Init.ContinuousConvMode` field to `ENABLE`.
- Set the `hadc.Init.DMAContinuousRequests` field to `ENABLE`, otherwise the ADC does not retrigger the DMA once the first scan sequence completes.
- Configure the DMA Stream/Channel in `DMA_CIRCULAR` mode.

12.2.7 Errors Management

ADC peripheral has the ability to notify developers in case a conversion is lost. This error condition happens when a continuous or scan mode conversion is ongoing, and the ADC *data register* is overwritten by the successive transaction before it is read. When this happens a special bit in the ADC_SR register is set and the ADC interrupt is generated.

We can capture the *overrun* error by implementing the following callback:

```
void HAL_ADC_ErrorCallback(ADC_HandleTypeDef *hadc);
```

When the *overrun* error occurs, DMA transfers are disabled, and DMA requests are no longer accepted. In this case, if a DMA request is made, the regular conversion in progress is aborted and further regular triggers are ignored. It is then necessary to clear the OVR flag and the DMAEN bit of the used DMA stream, and to reinitialize both the DMA and the ADC to have the wanted converted channel data transferred to the right memory location (all these operations are automatically performed by the HAL when calling the `HAL_ADC_Start_DMA()` routine).

We can simulate an *overrun* error by enabling the continuous conversion mode in the previous example, and setting to `ENABLE` the `hadc.Init.DMAContinuousRequests` field[13]: if the ADC interrupt is enabled, and the `HAL_ADC_IRQHandler()` is invoked from it, then you will be able to catch the *overrun* error.



The *overrun* error is not only related to wrong configurations of the ADC interface. It can be generated even when the ADC works in DMA circular mode. For a custom design based on an STM32F4 MCU I made a while ago, where the DMA was heavily exploited by several peripherals, I experienced that the *overrun* error could occur when other concurrent transactions are performed by the DMA. Even if the bus arbitration should avoid race conditions, especially when priorities are properly set, I experienced this error in some non-reproducible situations. By correctly handling the *overrun* error I was able to restart conversions when this happened. Needless to say that, before I realized the source of unexpected stops in DMA conversion, I spent several days trying to debug the issue.

12.2.8 Timer-Driven Conversions

ADC peripheral can be configured to be driven from a timer through the TRGO trigger line. The timer used to perform this operation is hardwired during the chip design. For example, in an STM32F401RE MCU the ADC1 peripheral can be synchronized using the TIM2 timer. This feature is extremely useful to perform ADC conversions at a given frequency. For example, we can sample an audio wave generated by a microphone at 20kHz frequency. The result data can be then stored in a persistent memory.

The ADC conversions can be driven by timers both in *interrupt* and *DMA* mode. The former is useful when we sample just one channel at low frequencies. The latter is mandatory for *scan mode* conversions at high frequencies. To enable timer-driven conversions you can follow this procedure:

- Configure the timer connected to the ADC through the TRGO line according to the wanted sampling frequency.

[13] In some STM32 MCUs it is also required to explicitly enable the *overrun* detection by setting the `hadc.Init.Overrun` to `ADC_OVR_DATA_OVERWRITTEN`. Consult the HAL source code for the MCU family you are considering.

- Configure the timer's TRGO line so that it triggers every time the update event is generated (`TIM_TRGO_UPDATE`)[14].
- Configure the ADC so that the selected timer TRGO line triggers the conversions and be sure that *continuous conversion mode* is disabled (because it is the TRGO line that fires the conversion). Moreover, set the `hadc.Init.DMAContinuousRequests` field to `ENABLE` and the DMA in circular mode if you want to perform N conversion at time indefinitely, or set the `hadc.Init.DMAContinuousRequests` field to `DISABLE` if you want to stop after N conversions are performed.
- Be sure to set the `hadc.Init.ContinuousConvMode` field to `DISABLE`, otherwise the ADC performs conversions by its own without waiting the timer trigger.
- Start the timer.
- Start the ADC in *interrupt* or *DMA* mode.

The following example shows how to trigger a conversion every 1s in an STM32F401RE MCU using the TIM2 timer.

Filename: `Core/Src/main-ex3.c`

```
85   /** Configure the global features of the ADC (Clock, Resolution,
86    *  Data Alignment and number of conversion) */
87   hadc1.Instance = ADC1;
88   hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV8;
89   hadc1.Init.Resolution = ADC_RESOLUTION_12B;
90   hadc1.Init.ScanConvMode = DISABLE;
91   hadc1.Init.ContinuousConvMode = DISABLE;
92   hadc1.Init.DiscontinuousConvMode = DISABLE;
93   hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
94   hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIG2_T2_TRGO;
95   hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
96   hadc1.Init.NbrOfConversion = 3;
97   hadc1.Init.DMAContinuousRequests = ENABLE;
98   hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
99   HAL_ADC_Init(&hadc1);
100
101  /** Configure for the selected ADC regular channel its corresponding
102   *  rank in the sequencer and its sample time. */
103  sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
104  sConfig.Rank = 1;
105  sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
106  HAL_ADC_ConfigChannel(&hadc1, &sConfig);
107
108  sConfig.Rank = 2;
109  HAL_ADC_ConfigChannel(&hadc1, &sConfig);
110
```

[14]Please, take note that it is important to configure the timer's TRGO output mode by using the `HAL_TIMEx_MasterConfigSynchronization()` routine even if the timer does not work in *master* mode. This is a source of confusion for novice users, and I have to admit that that is a little bit counter-intuitive.

```
111   sConfig.Rank = 3;
112   HAL_ADC_ConfigChannel(&hadc1, &sConfig);
113 
114   /* ADC1 DMA Init */
115   hdma_adc1.Instance = DMA2_Stream0;
116   hdma_adc1.Init.Channel = DMA_CHANNEL_0;
117   hdma_adc1.Init.Direction = DMA_PERIPH_TO_MEMORY;
118   hdma_adc1.Init.PeriphInc = DMA_PINC_DISABLE;
119   hdma_adc1.Init.MemInc = DMA_MINC_ENABLE;
120   hdma_adc1.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
121   hdma_adc1.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
122   hdma_adc1.Init.Mode = DMA_CIRCULAR;
123   hdma_adc1.Init.Priority = DMA_PRIORITY_LOW;
124   hdma_adc1.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
125   HAL_DMA_Init(&hdma_adc1);
126 
127   __HAL_LINKDMA(&hadc1,DMA_Handle,hdma_adc1);
128 
129 }
130 
131 static void MX_TIM2_Init(void) {
132   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
133   TIM_MasterConfigTypeDef sMasterConfig = {0};
134 
135   htim2.Instance = TIM2;
136   htim2.Init.Prescaler = 47999;
137   htim2.Init.Period = 1999;
138   htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
139   htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
140   htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
141   HAL_TIM_Base_Init(&htim2);
142 
143   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
144   HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig);
145 
146   sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
147   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
148   HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig);
149 }
```

Let us start from the analysis of the `MX_TIM2_Init()` function. First, it configures the TIM2 timer so that it overflows ever 1s in a Nucleo-F401RE running at 84MHz (lines [136:137]). Next, the timer is configured so that the TRGO line is asserted when it overflows (line 146).
The ADC, instead, is configured to perform three conversions from the same channel (the channel connected to the temperature sensor). The ADC is also configured to be triggered from the TIM2 TRGO line (lines [93:94]) and so that DMA requests trigger the conversion continuously (line 97).

The DMA is so configured accordingly to work in circular mode (line 122).

Filename: `Core/Src/main-ex3.c`

```
85  int main(void) {
86    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
87    HAL_Init();
88
89    /* Configure the system clock */
90    SystemClock_Config();
91
92    /* Initialize all configured peripherals */
93    MX_DMA_Init();
94    MX_ADC1_Init();
95    MX_GPIO_Init();
96    MX_USART2_UART_Init();
97    MX_TIM2_Init();
98
99    HAL_TIM_Base_Start(&htim2);
100   HAL_ADC_Start_DMA(&hadc1, (uint32_t*)rawValues, 3);
101
102   while(1) {
103      while(!convCompleted);
104
105      for(uint8_t i = 0; i < hadc1.Init.NbrOfConversion; i++) {
106       temp = ((float)rawValues[i]) / 4095 * 3300;
107       temp = ((temp - 760.0) / 2.5) + 25;
108
109       sprintf(msg, "rawValue %d: %hu\r\n", i, rawValues[i]);
110       HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
111
112       sprintf(msg, "Temperature %d: %f\r\n",i,  temp);
113       HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
114     }
115     convCompleted = 0;
116   }
117 }
118
119 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef*hadc) {
120   convCompleted = 1;
121 }
```

The above code shows the content of the `main()` function, which should be easy to understand. The timer is started at line 56 and the ADC is started in DMA mode to perform three acquisitions from the ADC. If you run the example, you can see that ever three seconds the DMA completes the transfer and the `convCompleted` variable is set: this causes that the three conversions are printed on the UART2 interface every three seconds.

12.2.9 Conversions Driven by External Events

In some STM32 MCUs it is possible to configure an EXTI line to trigger A/D conversions. For example, in an STM32F401RE MCU the EXTI line 11 can be enabled for such uses. This means that any MCU pin connected on that line (PA11, PB11, etc.) is a valid source to trigger conversions. Take note that it is not possible to use an EXTI line and a timer as trigger source at the same time.

12.2.10 ADC Calibration

The ADC peripheral implemented by some STM32 families, like the STM32F1/F3/L0/L4/G4 ones, provide an automatic calibration procedure that drives all the calibration sequence including the power-on/off sequence of the ADC. During the procedure, the ADC calculates a calibration factor, which is 7-bit wide,f and which is applied internally to the ADC until the next ADC power-off. During the calibration procedure, the application must not use the ADC and must wait until calibration is complete. Calibration is preliminary to any ADC operation. It removes the offset error that may vary from chip to chip due to process or bandgap variation. The calibration factor to be applied for single-ended input conversions is different from the factor to be applied for differential input conversions.

The `HAL_ADC_Ex` module provides three functions useful to work with ADC calibration. The

```
HAL_ADCEx_Calibration_Start(ADC_HandleTypeDef* hadc, uint32_t SingleDiff);
```

automatically performs a calibration procedure. It must be called right after the `HAL_ADC_Init()`, and before any `HAL_ADC_Start_XXX()` routine is used. Passing the parameter `ADC_SINGLE_ENDED` a single-ended calibration is performed, while passing the `ADC_DIFFERENTIAL_ENDED` performs a differential input calibration.

The function

```
uint32_t HAL_ADCEx_Calibration_GetValue(ADC_HandleTypeDef* hadc, uint32_t SingleDiff);
```

is used to retrieve the computed calibration value, while the

```
HAL_StatusTypeDef HAL_ADCEx_Calibration_SetValue(ADC_HandleTypeDef* hadc,
                  uint32_t SingleDiff, uint32_t CalibrationFactor);
```

is used to set up a custom derived calibration value. For more information, please refer to the reference manual for the MCU you are considering.

12.3 Using CubeMX to Configure ADC Peripheral

CubeMX allows to easily configure the ADC peripheral in a few steps. The first one consists in enabling the wanted ADC channels in the *IP Tree* view, as shown in **Figure 12.11**.

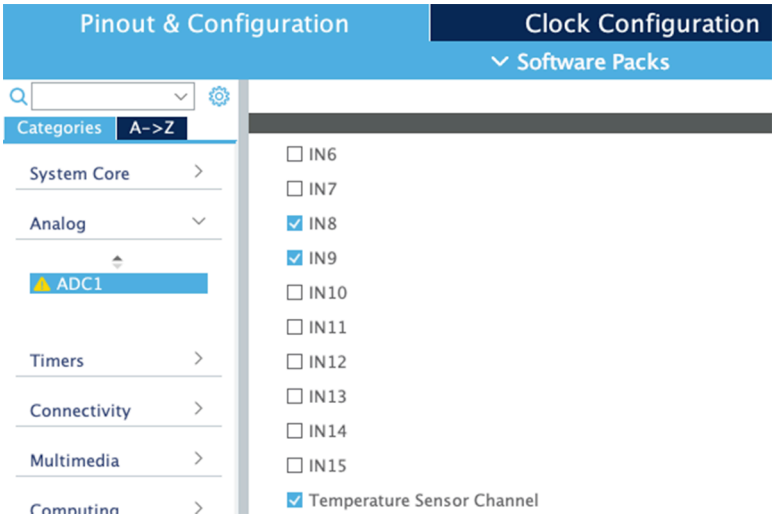


Figure 12.11: The *IP Tree* view pane allows to select input channels of the ADC

Once the inputs are enabled, we can configure the ADC peripheral from the *Configuration* view, as shown in **Figure 12.12**.

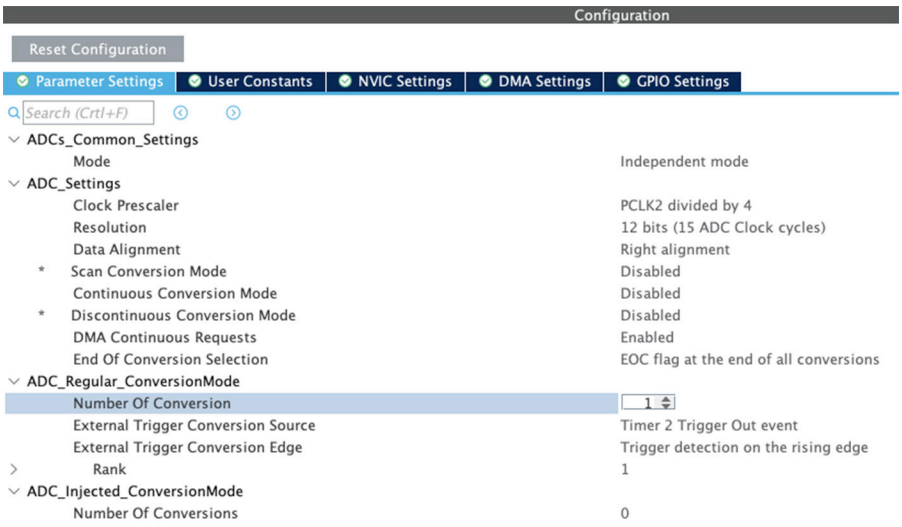


Figure 12.12: The ADC configuration view in CubeMX

The fields reflect the ADC configuration parameters seen so far. There is only one part that tends to confuse novice users: the way channels are configured. In fact, we first need to configure the number of channels used by setting the **Number of Conversion** field. Next, (this is really important) we need to click elsewhere in the configuration dialog so that the number of **Rank** fields increases according

to the specified number of channels. In those MCU providing the notion of *regular* and *injected* groups we can select the sampling speed for each channel independently. CubeMX will generate all the initialization code automatically.

F1

As stated before in this chapter, the `HAL_ADC` module in the CubeF1 HAL differs from the other HALs. To start a conversion driven by software it is required that the parameter `hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START` is specified during the ADC initialization. CubeMX reflects this different configuration, but it is tricky to understand how to configure the peripheral in the right way. So, to enable software-driven conversion, the **External Trigger Conversion Edge** parameter must be set to **Trigger detection on the rising edge**. This makes the field **External Trigger Conversion Source** available, and you have to select the entry **Software trigger**. Otherwise, you will not be able to perform conversions.

13. Digital-To-Analog Conversion

In the previous chapter we focused our attention on the ADC controller, showing the most relevant characteristics of this important peripheral that all STM32 microcontrollers provide. The reverse of this operation is demanded to the *Digital to Analog Converter* (DAC).

Depending on the family and package used, STM32 microcontrollers usually provide only a DAC with one or two dedicated outputs, apart from few part numbers from the STM32F3-series that implement two DACs, the first one with two outputs and the other one with just one output. Some more recent MCUs from STM32G4 series provide even up to 5 independent DAC modules, but just the first two have output I/Os: the other ones have just the possibility to feed internal peripherals like OPAMP, comparators and ADC if supported.

DAC channels can be configured to work in 8/12-bit mode, and the conversion of the two channels can be performed independently or simultaneously: this last mode is useful in those applications where two independent but synchronous signals must be generated (for example, in audio applications). Like the ADC peripheral, even the DAC can be triggered by a dedicated timer, to generate analog signals at a given frequency.

This chapter gives a quick introduction to the most relevant characteristics of this peripheral, leaving to the reader the responsibility to deepen the features of the DAC in the specific STM32 microcontroller he is considering. As usual, we are now going to give a brief explanation about how a DAC controller works.

13.1 Introduction to the DAC Peripheral

A DAC is a device that converts a number to an analog signal, which is proportional to a supplied reference voltage V_{REF} (see **Figure 13.1**). There are many categories of DACs. Some of these include *Pulse Width Modulators* (PWM), interpolating, sigma-delta DACs and high speed DACs. We have analyzed how to use an STM32 timer to generate PWM signals in Chapter 11, and we have used this capability to generate an output sine wave with the help of a RC low-pass filter.

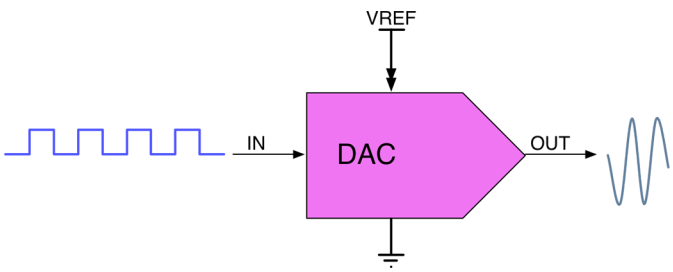


Figure 13.1: The general structure of a DAC

DAC peripherals available in STM32 microcontrollers are based on the common R-2R resistor ladder network. A *resistor ladder* is an electrical circuit made of repeating units of resistors, and it is an inexpensive and simple way to perform a digital-to-analog conversion using repetitive resistor networks, made with high-precise resistors. The network acts as a programmable voltage divider between the reference voltage and the ground.

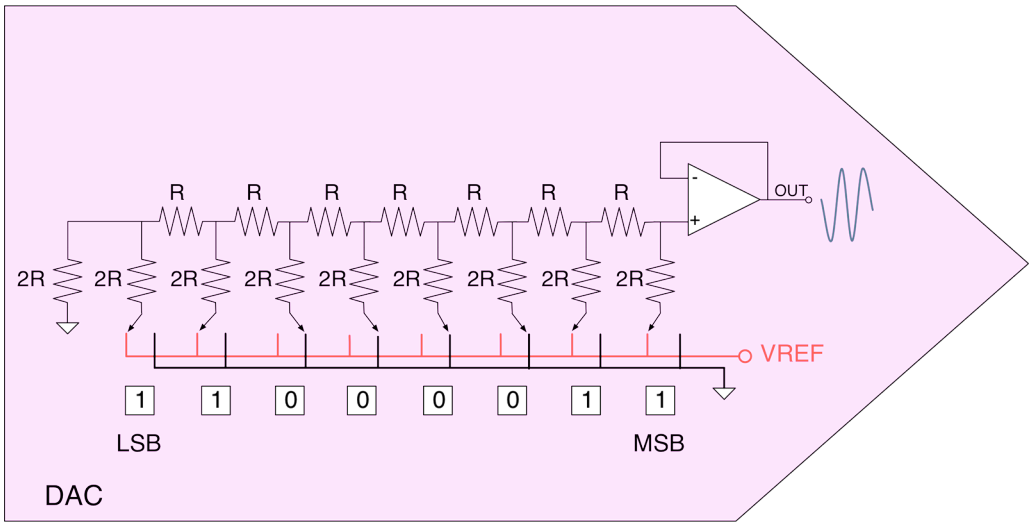


Figure 13.2: How a R-2R network can be used to convert a digital quantity to an analog signal

A 8-bit R–2R resistor ladder network is shown in **Figure 13.2**. Each bit of the DAC is driven by digital logic gates. Ideally, these gates switch the input bit between V = 0 (logic 0) and V = V_{REF} (logic 1). The R–2R network causes these digital bits to be weighted in their contribution to the output voltage V_{OUT}. Depending on which bits are set to 1 and which to 0, the output voltage will have a corresponding stepped value between 0 and V_{REF} minus the value of the minimal step, corresponding to bit 0.

For a given numeric value D, of a R–2R DAC with N bits and $0V/V_{REF}$ logic levels, the output voltage V_{OUT} is:

$$V_{OUT} = \frac{V_{REF} \times D}{2^N} \quad [1]$$

For example, if N = 12 (hence 2^N = 4096) and V_{REF} = 3.3 V (typical analog supply voltage in an STM32 MCU), then V_{OUT} will vary between 0V (VAL = 0 = 00000000_2) and the maximum (VAL = 4095 = 11111111_2):

$$V_{OUT} = 3.3 \times \frac{4095}{4096} \approx 3.29V$$

with steps (corresponding to VAL = 1):

$$\Delta V_{OUT} = 3.3 \times \frac{1}{4096} \approx 0.0002V$$

However, always keep in mind that the precision and stability of the DAC output is heavily affected by the quality of VDDA power domain and the layout of PCB.

In STM32 microcontrollers, the DAC module has an accuracy of 12-bit, but it can be configured to work in 8-bit too. In 12-bit mode, the data could be left- or right-aligned. Depending on the sales type and package used, the DAC has two output channels, each one with its own converter. In dual DAC channel mode, conversions could be done independently or simultaneously when both channels are grouped together for synchronous update operations. An input reference pin, VREF+ (shared with others analog peripherals) is available for better resolution. As it happens for the ADC peripheral, even the DAC may be used in conjunction with the DMA controller to generate variable output voltages at a given fixed frequency. This is extremely useful in audio applications, or when we want to generate analog signals working at a given carrier frequency. As we will see later in this chapter, the STM32 DACs have the ability to generate noise waves and triangular waves.

Finally, the DAC implemented in STM32 MCUs integrates an output buffer for each channel (see **Figure 13.2**), which can be used to reduce the output impedance and to drive external loads directly without having to add an external operational amplifier. Each DAC channel output buffer can be enabled and disabled.

Table 13.1 lists the exact number of DAC peripherals and their related output channels for all STM32 MCUs equipping the nine Nucleo boards we are considering in this book.

Nucleo P/N	DAC Peripherals	DAC External Channels
NUCLEO-G474RE	4	3
NUCLEO-F446RE	1	2
NUCLEO-F401RE	-	-
NUCLEO-F303RE	1	2
NUCLEO-F103RB	-	-
NUCLEO-F072RB	1	2
NUCLEO-L476RG	1	2
NUCLEO-L152RE	1	2
NUCLEO-L073RZ	1	2

Table 13.1: The availability of DAC peripheral in STM32 MCUs equipping Nucleo boards

13.2 HAL_DAC Module

After a brief introduction to the most important features offered by the DAC peripheral in STM32 microcontrollers, it is the right time to dive into the related CubeHAL APIs.

To manipulate the DAC peripheral, the HAL defines the C `struct DAC_HandleTypeDef`, which is defined in the following way:

```
typedef struct {
  DAC_TypeDef                 *Instance;     /* Pointer to DAC descriptor         */
  __IO HAL_DAC_StateTypeDef   State;         /* DAC communication state           */
  HAL_LockTypeDef             Lock;          /* DAC locking object                */
  DMA_HandleTypeDef           *DMA_Handle1;  /* Pointer DMA handler for channel 1 */
  DMA_HandleTypeDef           *DMA_Handle2;  /* Pointer DMA handler for channel 2 */
  __IO uint32_t               ErrorCode;     /* DAC Error code                    */
} DAC_HandleTypeDef;
```

Let us analyze the most important fields of this `struct`.

- `Instance`: is the pointer to the DAC descriptor we are going to use. For example, `DAC1` is the descriptor of the first DAC peripheral.
- `DMA_Handle{1,2}`: this is the pointer to the DMA handler configured to perform D/A conversions in DMA mode. In DACs with two output channels, there exist two independent DMA handlers used to perform conversions for each channel.

As you can see, the `DAC_HandleTypeDef struct` differs from the other handler descriptors used so far. In fact, it does not provide a dedicated `Init` parameter, used by the `HAL_DAC_Init()` function to configure the DAC. This because the effective configuration of the DAC is performed at channel level, and it is demanded to the `struct DAC_ChannelConfTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t DAC_Trigger;     /* Specifies the external trigger for the selected
                               DAC channel */
  uint32_t DAC_OutputBuffer;/* Specifies whether the DAC channel output buffer
                               is enabled or disabled */
} DAC_ChannelConfTypeDef;
```

- `DAC_Trigger`: specifies the source used to trigger the DAC conversion. It can assume the value `DAC_TRIGGER_NONE` when the DAC is driven manually using the `HAL_DAC_SetValue()` function; the value `DAC_TRIGGER_SOFTWARE` when the DAC is driven in DMA mode without a timer to "clock" the conversions; the value `DAC_TRIGGER_Tx_TRGO` to indicate a conversion driven by adedicated timer.
- `DAC_OutputBuffer`: enables the dedicated output buffer.

To configure a DAC channel, we use the function:

```
HAL_StatusTypeDef HAL_DAC_ConfigChannel(DAC_HandleTypeDef* hdac,
                                        DAC_ChannelConfTypeDef* sConfig, uint32_t Channel);
```

which accepts the pointer to an instance of the `DAC_HandleTypeDef struct`, the pointer to an instance of the `DAC_ChannelConfTypeDef struct` seen before and the macro `DAC_CHANNEL_1` to configure the first channel and `DAC_CHANNEL_2` for the second one.

In some more recent STM32 microcontrollers, like the STM32L476 or the STM32G474, the DAC also provides additional low-power features. For example, it is possible to enable the *sample-and-hold* circuitry that allows to keep the output voltage stable even if the DAC is powered off. This is extremely useful in battery-powered applications. In these MCUs the structure of the `DAC_ChannelConfTypeDef struct` differs, to allow the configuration of these additional features. Refer to the HAL source code for the MCU you are considering.

13.2.1 Driving the DAC Manually

The DAC peripheral can be driven manually or using the DMA and a trigger source (e.g., a dedicated timer). We are now going to analyze the first method, which is used when we do not need conversions at high frequencies.

The first step consists in starting the peripheral by calling the function

```
HAL_StatusTypeDef HAL_DAC_Start(DAC_HandleTypeDef* hdac, uint32_t Channel);
```

The function accepts the pointer to an instance of the `DAC_HandleTypeDef struct`, and the channel to activate (`DAC_CHANNEL_1` or `DAC_CHANNEL_2`).

Once the DAC channel is enabled, we can perform a conversion by calling the function:

```
HAL_StatusTypeDef HAL_DAC_SetValue(DAC_HandleTypeDef* hdac, uint32_t Channel,
                                   uint32_t Alignment, uint32_t Data);
```

where the `Alignment` parameter can assume the value `DAC_ALIGN_8B_R` to drive the DAC in 8-bit mode, `DAC_ALIGN_12B_L` or `DAC_ALIGN_12B_R` to drive the DAC in 12-bit mode passing the output value left- or right-aligned respectively.

The following example, designed to run on a Nucleo-F072RB, shows how to drive the DAC peripheral manually. The example is based on the fact that in the majority of Nucleo boards providing the DAC peripheral one of the output channels corresponds to the PA5 pin, which is connected to LD2 LED. This allows us to fade ON/OFF the LD2 using the DAC.

Filename: `Core/Src/main-ex1.c`

```
8  DAC_HandleTypeDef hdac;
9
10 /* Private function prototypes -----------------------------------------------*/
11 static void MX_DAC_Init(void);
12
13 int main(void) {
14   HAL_Init();
15   Nucleo_BSP_Init();
16
17   /* Initialize all configured peripherals */
18   MX_DAC_Init();
19
20   HAL_DAC_Init(&hdac);
21   HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
22
23   while(1) {
24     int i = 2000;
25     while(i < 4000) {
26       HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, i);
27       HAL_Delay(1);
28       i+=4;
29     }
30
31     while(i > 2000) {
32       HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, i);
33       HAL_Delay(1);
34       i-=4;
35     }
36   }
37 }
38
39 /* DAC init function */
40 void MX_DAC_Init(void) {
41   DAC_ChannelConfTypeDef sConfig;
42   GPIO_InitTypeDef GPIO_InitStruct;
43
44   __HAL_RCC_DAC1_CLK_ENABLE();
45
46   /* DAC Initialization  */
47   hdac.Instance = DAC;
48   HAL_DAC_Init(&hdac);
49
50   /**DAC channel OUT2 config */
51   sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
52   sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
53   HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_2);
```

```
54
55    /* DAC GPIO Configuration
56       PA5     ------> DAC_OUT2
57    */
58    GPIO_InitStruct.Pin = GPIO_PIN_5;
59    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
60    GPIO_InitStruct.Pull = GPIO_NOPULL;
61    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
62  }
```

The code is straightforward. Lines [40:62] configure the DAC so that the Channel 2 is used as output channel. For this reason, the PA5 is configured as analog output (lines [58:61]). Please, take note that since we are going to drive the DAC conversions manually, the channel trigger source is set to `DAC_TRIGGER_NONE` (line 51). Finally, the `main()` is nothing more than an infinite loop that increases/decreases the output voltage so that the LD2 fades ON/OFF.

13.2.2 Driving the DAC in DMA Mode Using a Timer

The most common usage of the DAC peripheral is to generate an analog waveform with a given frequency (e.g. in audio applications). If this the case, then the best way to drive the DAC is by using the DMA and a timer to trigger the conversions.

To start the DAC and perform a transfer in DMA mode we need to configure the corresponding DMA channel/stream pair and use the function:

```
HAL_StatusTypeDef HAL_DAC_Start_DMA(DAC_HandleTypeDef* hdac, uint32_t Channel,
                                    uint32_t* pData, uint32_t Length, uint32_t Alignment);
```

which accepts the pointer to an instance of the `DAC_HandleTypeDef struct`, the channel to activate (`DAC_CHANNEL_1` or `DAC_CHANNEL_2`), the pointer to the array of values to transfer in DMA mode, its length, and the alignment of output values in memory, which can assume the value `DAC_ALIGN_8B_R` to drive the DAC in 8-bit mode, `DAC_ALIGN_12B_L` or `DAC_ALIGN_12B_R` to drive the DAC in 12-bit mode passing the output value left- or right-aligned respectively.

For example, we can easily generate a sinusoidal wave using the DAC. In Chapter 11 we have analyzed how to use the PWM mode of a timer to generate sine waves. If our MCU provides a DAC, then the same operation can be carried out more easily. Moreover, depending on the specific application, by enabling the output buffer we can avoid external passives at all.

To generate a sinusoidal wave running at a given frequency, we have to divide the complete period in a number of steps. Usually more than 200 steps are a good approximation for an output wave. This means that if we want to generate a 50Hz sine wave, then we need to perform a conversion every:

$$f_{sinewave} = 50Hz * 200 = 10kHz \quad [2]$$

Since the STM32 DAC has a resolution of 12-bit, we have to divide the value 4095, which corresponds to the maximum output voltage, by 200 steps using the following formula:

$$DAC_{Output} = \left(sin\left(x \cdot \frac{2\pi}{n_s}\right) + 1\right)\left(\frac{4096}{2}\right) \quad [3]$$

where n_s is the number of samples, that is 200 in our example.

Using the above formula, we can generate an initialization vector to feed the DAC in DMA mode. Like for the ADC peripheral, we can use a timer configured to trigger the TRGO line at the frequency given by [2]. The following example shows how to generate a 50Hz sine wave using the DAC in an STM32F072 MCU.

Filename: `Core/Src/main-ex2.c`

```
#define PI    3.14159
#define SAMPLES 200

/* Private variables ---------------------------------------------------------*/
DAC_HandleTypeDef hdac;
TIM_HandleTypeDef htim6;
DMA_HandleTypeDef hdma_dac_ch1;

/* Private function prototypes -----------------------------------------------*/
static void MX_DAC_Init(void);
static void MX_TIM6_Init(void);

int main(void) {
  uint16_t IV[SAMPLES], value;

  HAL_Init();
  Nucleo_BSP_Init();

  /* Initialize all configured peripherals */
  MX_TIM6_Init();
  MX_DAC_Init();

  for (uint16_t i = 0; i < SAMPLES; i++) {
    value = (uint16_t) rint((sinf(((2*PI)/SAMPLES)*i)+1)*2048);
    IV[i] = value < 4096 ? value : 4095;
  }

  HAL_DAC_Init(&hdac);
  HAL_TIM_Base_Start(&htim6);
  HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, (uint32_t*)IV, SAMPLES, DAC_ALIGN_12B_R);

  while(1);
}
```

```
40
41  /* DAC init function */
42  void MX_DAC_Init(void) {
43    DAC_ChannelConfTypeDef sConfig;
44    GPIO_InitTypeDef GPIO_InitStruct;
45
46    __HAL_RCC_DAC1_CLK_ENABLE();
47
48    /**DAC Initialization  */
49    hdac.Instance = DAC;
50    HAL_DAC_Init(&hdac);
51
52    /**DAC channel OUT1 config */
53    sConfig.DAC_Trigger = DAC_TRIGGER_T6_TRGO;
54    sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
55    HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_1);
56
57    /**DAC GPIO Configuration
58    PA4     ------> DAC_OUT1
59    */
60    GPIO_InitStruct.Pin = GPIO_PIN_4;
61    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
62    GPIO_InitStruct.Pull = GPIO_NOPULL;
63    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
64
65    /* Peripheral DMA init*/
66    hdma_dac_ch1.Instance = DMA1_Channel3;
67    hdma_dac_ch1.Init.Direction = DMA_MEMORY_TO_PERIPH;
68    hdma_dac_ch1.Init.PeriphInc = DMA_PINC_DISABLE;
69    hdma_dac_ch1.Init.MemInc = DMA_MINC_ENABLE;
70    hdma_dac_ch1.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
71    hdma_dac_ch1.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
72    hdma_dac_ch1.Init.Mode = DMA_CIRCULAR;
73    hdma_dac_ch1.Init.Priority = DMA_PRIORITY_LOW;
74    HAL_DMA_Init(&hdma_dac_ch1);
75
76    __HAL_LINKDMA(&hdac,DMA_Handle1,hdma_dac_ch1);
77  }
78
79
80  /* TIM6 init function */
81  void MX_TIM6_Init(void) {
82    TIM_MasterConfigTypeDef sMasterConfig;
83
84    __HAL_RCC_TIM6_CLK_ENABLE();
85
86    htim6.Instance = TIM6;
```

```
87    htim6.Init.Prescaler = 0;
88    htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
89    htim6.Init.Period = 4799;
90    HAL_TIM_Base_Init(&htim6);
91
92    sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
93    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
94    HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig);
95  }
```

The function `MX_DAC_Init()` configures the DAC so that the first channel performs a conversion when the TIM6 TRGO line is generated. Moreover, the DMA is configured accordingly, setting it in circular mode so that it transfers the content of the initialization vector in the DAC data register continuously. The `MX_TIM6_Init()` function sets the TIM6 so that it overflows with a frequency equal to 10kHz, triggering the TRGO line that is internally connected to the DAC. Finally, lines [29:32] generate the initialization vector according to the equation [3]. Its content is then used to feed the DAC, which is started in DMA mode after the TIM6 is enabled.

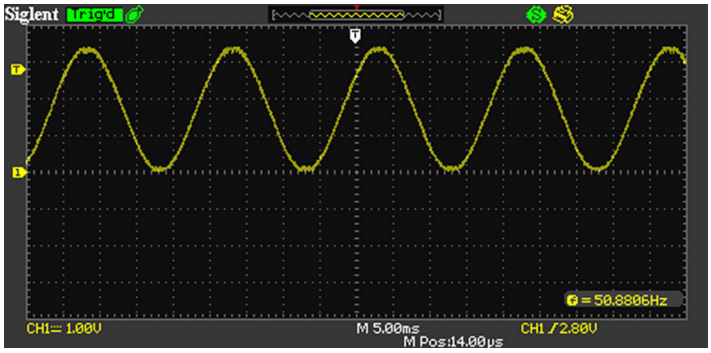


Figure 13.3: The output sine wave generated with the DAC peripheral

By connecting an oscilloscope probe to the PA4 pin of our Nucleo board we can see the output sine wave generate by the DAC (see **Figure 13.3**).

If we are interested in knowing when a DAC conversion in DMA mode has been completed, we can implement the callback function:

```
void HAL_DACEx_ConvCpltCallbackChX(DAC_HandleTypeDef* hdac);
```

which is automatically called by the `HAL_DMA_IRQHandler()` routine invoked from the ISR of the DMA channel associated to the DAC peripheral. The final `X` in the function name must be replaced with `1` or `2` depending on the channel used.



Read Carefully

Please, take note that in STM32G4 series, the DAC peripheral registers have to be accessed by words (32-bit). So, owners of Nucleo-G474RE board will find that the `IV` array and `value` variable are defined as `unit32_t`.

13.2.3 Triangular Wave Generation

In several audio applications it is useful to generate triangular waves. While it is perfectly possible to generate a triangular wave using the DMA technique seen before, STM32 DACs allow to generate waves with a triangular shape in hardware.

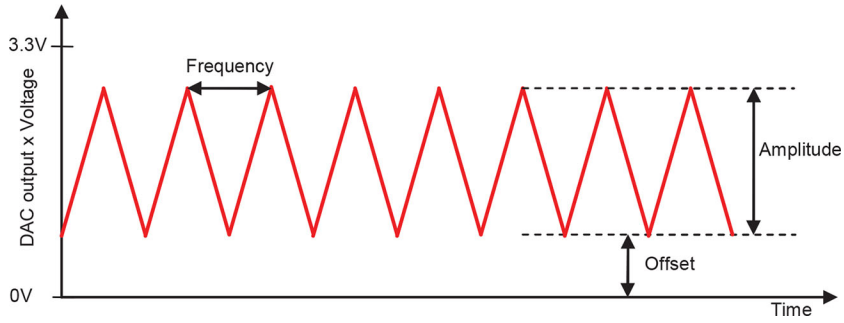


Figure 13.4: A triangular wave generated with the DAC

The **Figure 13.4** shows the three parameters that define the shape of the triangular wave. Let us analyze them.

- **Amplitude**: it is a value ranging from 0 to 0xFFF and it determines the maximum height of the wave. It is directly connected to the **offset** value, as we will see next. **Amplitude** cannot be an arbitrary value, but it is part of a list of of fixed values. Consult the HAL source code for the complete list of admissible values.
- **Offset**: it is the minimum output value, and it represents the lowest point of the wave. The sum of the **offset** and **amplitude** cannot exceed the maximum value of 0xFFF. This means that the maximum **amplitude** of the wave will be given by the difference **amplitude - offset**.
- **Frequency**: is the frequency of the wave and it is determined by the update frequency of the timer connected to the DAC. The update frequency of the timer is determined by the equation [4] below. This means that if we want to generate a 50Hz triangular wave with an amplitude equal to 2047, the prescaler of a timer running at 48MHz needs to be configured to 234.

$$f_{UEV} = 2 \cdot amplitude \cdot f_{wave} \quad [4]$$

To generate a triangular waveform we use the function

```
HAL_StatusTypeDef HAL_DACEx_TriangleWaveGenerate(DAC_HandleTypeDef* hdac, uint32_t Channel,
                                                 uint32_t Amplitude);
```

which accepts the DAC channel to use and the wanted amplitude. The wave offset, instead, is configured using the `HAL_DAC_SetValue()` routine. The full procedure to generate a triangular wave is the following one:

- Configure the DAC channel used to generate the wave.
- Configure the timer associated to the DAC, and configure its prescaler according to equation [4].
- Start the DAC using the `HAL_DAC_Start()` function.
- Configure the wanted offset value using the `HAL_DAC_SetValue()` routine.
- Start triangular wave generation by calling the `HAL_DACEx_TriangleWaveGenerate()` function.

13.2.4 Noise Wave Generation

STM32 DACs are also able to generate noise waves (see **Figure 13.5**), using a pseudo-random generator. This is useful in some application domains, like audio applications and RF systems. Moreover, it can be also used to increase the accuracy of ADC peripheral[1].

To generate a variable-amplitude pseudo-noise, an LFSR (linear feedback shift register) is available in the DAC. This register is preloaded with the value 0xAAA, which may be masked partially or totally. This value is then added up to the DAC data register contents without overflow and this value is then used as output value.

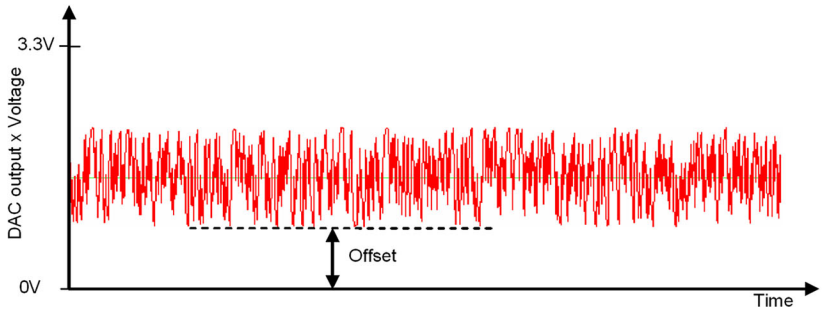


Figure 13.5: a noise wave generated with the DAC

To generate the noise wave, we can use the HAL routine

```
HAL_StatusTypeDef HAL_DACEx_NoiseWaveGenerate(DAC_HandleTypeDef* hdac, uint32_t Channel,
                                              uint32_t Amplitude);
```

which accepts the channel used to generate the wave and the amplitude value, which is added to the LFSR content to generate the pseudo-random wave. Like for the triangular wave generation, a timer can be used to trigger conversion: this means that the frequency of the wave is determined by the overflow frequency of the timer.

[1]ST provides the AN2668(https://bit.ly/25lJoqx) dedicated to this topic.

14. I²C

Nowadays even the simplest PCB contains two or more digital *integrated circuits* (IC), in addition to the main MCU, designated to specific tasks. ADCs and DACs, EEPROM memories, sensors, logic I/O ports, RTC clocks, RF circuits and dedicated LCD controllers are just a small list of possible ICs specialized in doing just a single task. Modern digital electronics design is all about the right selection (and programming) of powerful, specific and, most of the times, cheap ICs to mix on the final PCB.

Depending on the characteristics of these ICs, they are often designed to exchange messages and data with a programmable device (which usually is, but not limited to, a microcontroller) according to a well-defined communication protocol. Two of the most used protocols for *intra-board* communications are the I²C and the SPI, both date back to early '80 but still widespread in the electronics industry, especially when communication speed is not a strict requirement and it is limited to the PCB boundaries[1].

Almost all STM32 microcontrollers provide dedicated hardware peripherals able to communicate using I²C and SPI protocols. This chapter is the first of two dedicated to this topic, and it briefly introduces the I²C protocol and the related CubeHAL APIs to program this peripheral. If interested in deepening the I²C protocol, the UM10204 by NXP[2] provides the complete and the most updated specification.

14.1 Introduction to the I²C specification

The *Inter-Integrated Circuit* (aka I²C - pronounced *I-squared-C* or very rarely *I-two-C*) is a hardware specification and protocol developed by the semiconductor division of Philips (now *NXP Semiconductors*) back in 1982. It is a *multi-slave*[3], half-duplex, single-ended 8-bit oriented serial bus specification, which uses only two wires to interconnect a given number of slave devices to a master. Until October 2006, the development of I²C-based devices was subject to the payment of royalty fees to Philips, but this limitation has been superseded[4].

[1] Although there exist applications where I²C and SPI protocols are used to exchange messages over external wires (usually with a length around the meter), these specifications were not designed to guarantee the robustness of communication over potentially noisy mediums. For this reason, their application is limited to the single PCB.

[2] https://bit.ly/3E18iPF

[3] The I²C can be also a *multi-master* protocol, meaning that two or more masters can exist on the same bus, but only one master at a time can take the bus control and it is up to masters to arbitrate the access to the bus. In practice, it is rare to use the I²C in multi-master mode in embedded systems. This book does not cover the multi-master mode.

[4] You still have to pay royalties to NXP if you want to receive an official and licensed I²C address pool for your devices, but I think that this not the case of readers of this book.

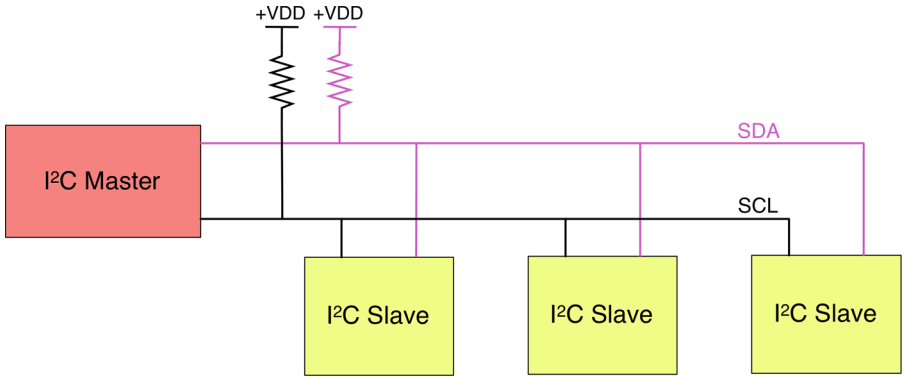


Figure 14.1: A graphical representation of the I²C bus

The two wires forming an I²C bus are bidirectional *open-drain lines*, named *Serial Data Line* (SDA) and *Serial Clock Line* (SCL) respectively (see **Figure 14.1**). The I²C protocol specifies that these two lines need to be pulled up with resistors. The sizing of these resistors is directly connected with the bus capacitance and the transmission speed. This document from Texas Instruments[5] provides the necessary math to compute the resistors value. However, it is quite common to use resistors with a value close to 4.7KΩ.



Modern microcontrollers, like STM32 ones, allow to configure GPIO lines as *open-drain pull-up*, enabling internal pull-up resistors. It is quite common to read around in the web that you can use internal pull-ups to pull I²C lines, avoiding the usage of dedicated resistors. However, in all STM32 devices the internal pull-up resistors have a value close to (or even higher than) 20KΩ to avoid unwanted power leaks. Such a value increases the time needed by the bus to reach the HIGH state, reducing the transmission speed. If speed is not important for your application and if (very important) you are not using long traces between the MCU and the IC (less then 2cm), then it is ok to use internal pull-up resistors for a lot of applications. But, if you have sufficient room on the PCB to place a couple of resistors, then it is strongly suggested to use external and dedicated ones.

F1



Read Carefully

STM32F1 microcontrollers do not provide the ability to pull-up SDA and SCL lines. Their GPIOs must be configured as *open-drain*, and two external resistors are required to pull-up I²C lines.

Being the I²C a protocol based on just two wires, there should be a way to address an individual slave device on the same bus. For this reason, I²C defines that each slave device provides a unique *slave*

[5] https://bit.ly/29URjoy

address for the given bus[6]. The address may be 7- or 10-bit wide (this option is quite uncommon).

I²C bus speeds are well-defined by the protocol specification, even if it is not so uncommon to find chips able to talk with custom (and often fuzzy) speeds. Common I²C bus speeds are the 100kHz[7], also known as *standard mode*, and the 400kHz, known as *fast mode*. Recent revisions of the standard can run at faster speeds (1MHz, known as *fast mode plus*, and 3.4MHz, known as *high speed mode*, and 5MHz, known as *ultra fast mode*).

I²C protocol is a sufficiently simple protocol so that a MCU can "simulate" a dedicated I²C peripheral if it does not provide one: this technique is called *bit-banging* and it is commonly used in low-cost 8-bit architectures, which sometimes do not provide a dedicated I²C interface to reduce pin-count and/or IC cost.

14.1.1 The I²C Protocol

In the I²C protocol all transactions are always initiated and completed by the master. This is one of the few rules of this communication protocol to keep in mind while programming (and, especially, debugging) I²C devices. All messages exchanged over the I²C bus are broken up into two types of frames: an *address frame*, where the master indicates to which slave the message is being sent, and one or more *data frames*, which are 8-bit data messages passed from master to slave or vice versa. Data is placed on the SDA line after SCL goes low, and it is sampled after the SCL line goes high. The time between clock edges and data read/write is defined by devices on the bus and it vary from chip to chip.

As said before, both SDA and SCL are bidirectional lines, connected to a positive supply voltage via a current-source or pull-up resistors (see **Figure 14.1**). When the bus is free, both lines are HIGH. The output stages of devices connected to the bus must have an open-drain or open-collector to perform the wired-AND function. The bus capacitance limits the number of interfaces connected to the bus. For a single master application, the master's SCL output can be a push-pull driver design if there are no devices on the bus that would stretch the clock (more about this later).

We are now going to analyze the fundamental steps of an I²C communication.

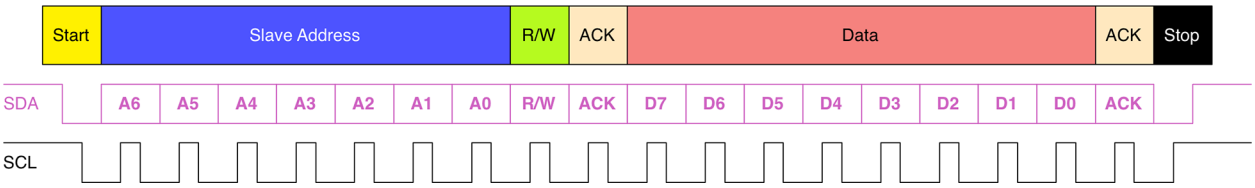


Figure 14.2: The structure of a base I²C message

[6]This constitutes one of the most practical limits of the I²C protocol. In fact, IC manufacturers rarely dedicate enough pins to configure the full slave address used on a given board (no more than three pins are dedicated to this feature, if you are lucky, giving only eight choices of slave addresses). When designing a board with several I²C devices, pay attention to their address and in case of collision you will need to use two or more I²C peripherals to drive them.

[7]There exist ICs communicating only at lower speeds, but nowadays are uncommon.

14.1.1.1 START and STOP Condition

All transactions begin with a START and are terminated by a STOP (see **Figure 14.2**). A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition. A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.

START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition. The bus stays busy if a repeated START (also called RESTART condition) is generated instead of a STOP condition (more about this soon). In this case, the START and RESTART conditions are functionally identical.

14.1.1.2 Byte Format

Every word transmitted on the SDA line must be eight bits long, and this also includes the address frame as we will see in a while. The number of bytes that can be transmitted per transfer is unrestricted. Each byte must be followed by an *Acknowledge* (ACK) bit. Data is transferred with the *Most Significant Bit* (MSB) first (see **Figure 14.2**). If a slave cannot receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL.

14.1.1.3 Address Frame

The address frame is always first in any new communication sequence. For a 7-bit address, the address is clocked out most significant bit (MSB) first, followed by a R/W bit indicating whether this is a read (1) or write (0) operation (see **Figure 14.2**).

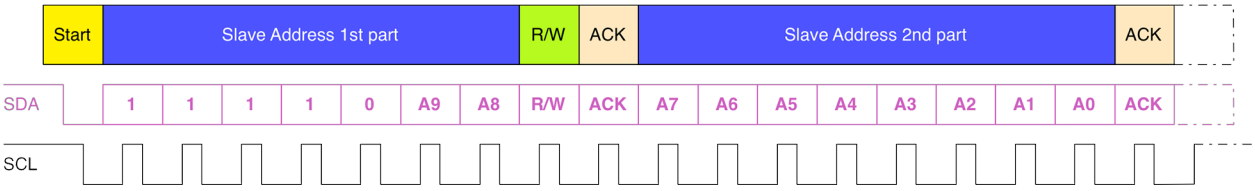


Figure 14.3: The message structure in case if 10-bit addressing is used

In a 10-bit addressing system (see **Figure 14.3**), two frames are required to transmit the slave address. The first frame will consist of the code `1111 0XXD`$_2$ where `XX` are the two MSB bits of the 10-bit slave address and `D` is the R/W bit as described above. The first frame ACK bit will be asserted by all slaves matching the first two bits of the address. As with a normal 7-bit transfer, another transfer begins immediately, and this transfer contains bits [7:0] of the address. At this point, the addressed slave should respond with an ACK bit. If it doesn't, the failure mode is the same as a 7-bit system.

Note that 10-bit address devices can coexist with 7-bit address devices, since the leading `11110` part of the address is not a part of any valid 7-bit addresses.

14.1.1.4 Acknowledge (ACK) and Not Acknowledge (NACK)

The ACK takes place after every byte. The ACK bit allows the *receiver* to signal the *transmitter*[8] that the byte was successfully received and another byte may be sent. The master generates all clock pulses over the SCL line, including the ACK ninth clock pulse.

The ACK signal is defined as follows: the transmitter releases the SDA line during the acknowledge clock pulse so that the receiver can pull the SDA line LOW, and it remains stable LOW during the HIGH period of this clock pulse. When SDA remains HIGH during this ninth clock pulse, this is defined as the *Not Acknowledge* (NACK) signal. The master can then generate either a STOP condition to abort the transfer, or a RESTART condition to start a new transfer. There are five conditions leading to the generation of a NACK:

1. No receiver is present on the bus with the transmitted address so there is no device to respond with an acknowledge.
2. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master.
3. During the transfer, the receiver gets data or commands that it does not understand.
4. During the transfer, the receiver cannot receive any more data bytes.
5. A master-receiver must signal the end of the transfer to the slave transmitter.



The effectiveness of the ACK/NACK bit is due to the *open-drain* nature of the I²C protocol. *Open-drain* means that both master and slave involved in a transaction can pull the corresponding signal line LOW, but cannot drive it HIGH. If one between the transmitter and receiver releases a line, it is automatically pulled HIGH by the corresponding resistor if the other does not pull it LOW. The *open-drain* nature of the I²C protocol also ensures that can be no bus contention where one device is trying to drive the line HIGH while another tries to pull it LOW, eliminating the potential for damage to the drivers or excessive power dissipation in the system.

14.1.1.5 Data Frames

After the address frame has been sent, data can begin being transmitted. The master will simply continue generating clock pulses on SCL at a regular interval, and the data will be placed on SDA by either the master or the slave, depending on whether the R/W bit indicated a read or write operation. Usually, the first or the first two bytes contains the address of the slave register to write to/read from. For example, for I²C EEPROMs the first two bytes following the address frame represent the address of the memory location involved in the transaction.
Depending on the R/W bit, the successive bytes are filled by the master (if the R/W bit is set to 1) or the slave (if R/W bit is 0). The number of data frames is arbitrary, and most slave devices will

[8] Please, take note that here we are generically talking about *receiver* and *transmitter* because ACK/NACK bit can be set by both the master and the slave.

auto-increment the internal register, meaning that subsequent reads or writes will come from the next register in line. This mode is also called *sequential* or *burst* mode (see **Figure 14.4**) and it is a way to speed up transfer speed.

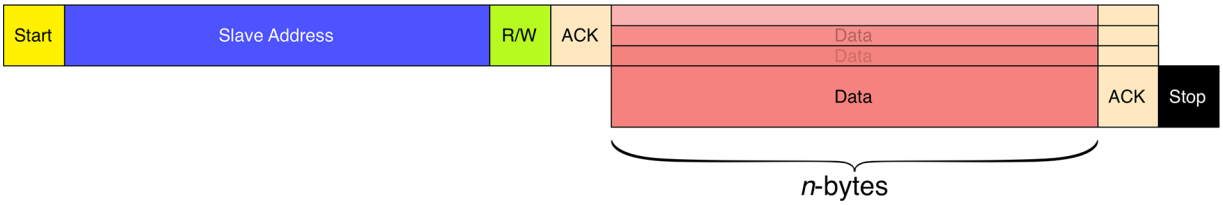


Figure 14.4: A transmission in *burst mode* where multiple bytes are exchanged in one transaction

14.1.1.6 Combined Transactions

The I²C protocol essentially has a simple communication pattern:

- a master sends on the bus the address of the slave device involved in the transaction;
- the R/W bit, which is the LSB bit in the slave address byte, establishes the direction of data flow (*from master to slave* - **W** - or *from slave to master* - **R**)
- a number of bytes are sent, each one interleaved with an ACK bit, by one of the two peers according to the transfer direction, until a STOP condition occurs.

This communication schema has a great pitfall: if we want to ask something specific to the slave device, we need to use two separated transactions. Let us consider this example. Suppose we have an I²C EEPROM. Usually, this kind of devices has several addressable memory locations (a 64Kbits EEPROM is addressable in the range `0 - 0x1FFF`[9]). To retrieve the content of a memory location, the master should perform the following steps:

- start a transaction in write mode (last bit of the slave address set to 0) by sending the slave address on the I²C bus so that the EEPROM begins sampling the messages over the bus;
- send two bytes representing the memory location we want to read;
- end a transaction by sending a STOP condition;
- start a new transaction in read mode (last bit of the slave address set to 1) by sending the slave address on the I²C bus;
- read *n*-bytes (usually one if reading the memory in random mode, more than one if reading it in sequential mode) sent by the slave device and then ending the transaction with a STOP condition.

[9] That values come from the fact that 64Kbits are equal to 65536 **bits**, but every memory location is 8-bit wide, so 65536/8 = 8196 = 0x2000. Since the memory locations starts from 0, then the last one has the `0x1FFF` address.

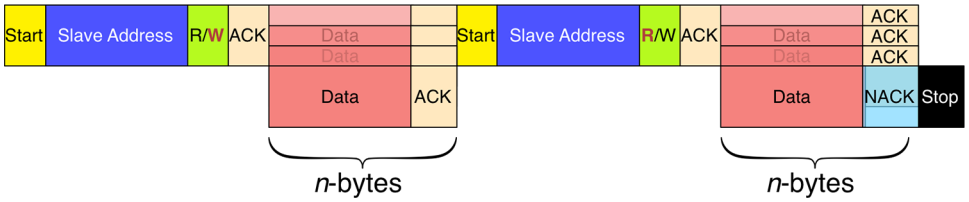


Figure 14.5: The structure of a *combined transaction*

To support this common communication schema, the I²C protocol defines the *combined transactions*, where the direction of data flow is inverted (usually *from slave to master*, or vice versa) after a number of bytes have been transmitted. **Figure 14.5** schematizes this way to communicate with slave devices. The master starts sending the slave address in write mode (note the **W** in red-bold in **Figure 14.5**) and then sends the addresses of registers we want to read. Then a new START condition is sent, without terminating the transaction: this additional START condition is also called *repeated START condition* (or RESTART). The master sends again the slave address but this time the transaction is started in read mode (note the **R** in bold in **Figure 14.5**). The slave now transmits the content of wanted registers, and the master acknowledges every byte sent. The master ends the transaction by issuing a NACK (this is important, as we will see next) and a STOP condition.

14.1.1.7 Clock Stretching

Sometimes, the master data rate will exceed the slave ability to provide that data. This happens because the data isn't ready yet (for example, the slave hasn't completed an analog-to-digital conversion) or because a previous operation hasn't yet completed (say, an EEPROM which hasn't completed writing to non-volatile memory yet and needs to finish that before it can service other requests).

In this case, some slave devices will execute what is referred to as *clock stretching*. In *clock stretching* the slave pauses a transaction by holding the SCL line LOW. The transaction cannot continue until the line is released HIGH again. Clock stretching is optional, and most slave devices do not include an SCL driver so they are unable to stretch the clock (mainly to simplify the hardware layout of the I²C interface). As we will discover later, an STM32 MCU configured in I²C slave mode can optionally implement the *clock stretching* mode.

14.1.2 Availability of I²C Peripherals in STM32 MCUs

Depending on the family type and package used, STM32 microcontrollers can provide up to four independent I²C peripherals. **Table 14.1** summarizes the availability of I²C peripherals in STM32 MCUs equipping all nine Nucleo boards we are considering in this book.

Nucleo P/N	I2C1		I2C2		I2C3		I2C4		100KHz	400KHz	1MHz
	SDA	SCL	SDA	SCL	SDA	SCL	SDA	SCL			
NUCLEO-G474RE	PA14	PA13	PF0	PC4	PC9	PC8	PC7	PC6	Yes	Yes	Yes
	PB7	PA15	PA8	PA9	PB5	PA8	PB7	PA13			
	PB9	PB8	-	-	PC11	-	-	-			
NUCLEO-F446RE	PB7	PB6	PB9	PB10	PC9	PA8	-	-	Yes	Yes	No
	PB9	PB8	PB3	-	PB4	-	-	-			
NUCLEO-F401RE	PB7	PB6	PB3	PB10	PC9	PA8	-	-			
	PB9	PB8	-	-	PB4	-	-	-			
NUCLEO-F303RE	PB7	PB6	PA10	PA9	PC9	PA8	-	-	Yes	Yes	Yes
	PB9	PB8	PF0	PF1	PB5	-	-	-			
NUCLEO-F103RB	PB7	PB6	PB11	PB10	-		-		Yes	Yes	No
	PB9	PB8	-	-							
NUCLEO-F072RB	PB7	PB6	PB11	PB10	-		-		Yes	Yes	Yes
	PB9	PB8	PB14	PB13							
NUCLEO-L476RG	PB7	PB6	PB11	PB10	PC1	PC0	-	-	Yes	Yes	Yes
	PB9	PB8	PB14	PB13	-		-				
NUCLEO-L152RE	PB7	PB6	PB11	PB10	-		-		Yes	Yes	No
	PB9	PB8	-								
NUCLEO-L073RZ	PB7	PB6	PB11	PB10	PC1	PC0	-	-	Yes	Yes	Yes
	PB9	PB8	PB14	PB13	PC9	PA8	-	-			

Table 14.1: Effective availability of I²C peripherals in MCUs equipping all nine Nucleo boards

For every I²C peripheral, and a given STM32 MCU, **Table 14.1** shows the pins corresponding to SDA and SCL lines. Moreover, darker rows show alternate pins that can be used during the layout of the board. For example, given the STM32F401RE MCU, we can see that I2C1 peripheral is mapped to PB7 and PB6, but PB9 and PB8 can be also used as alternate pins. Note that the I2C1 peripheral uses the same I/O pins in all STM32 MCUs with LQFP-64 package. This is a paramount example of the pin-to-pin compatibility offered by STM32 microcontrollers.

We are now ready to see how-to use the CubeHAL APIs to program this peripheral.

14.2 `HAL_I2C` Module

To program the I²C peripheral, the CubeHAL defines the C `struct I2C_HandleTypeDef`, which is defined in the following way:

```
typedef struct {
  I2C_TypeDef                *Instance;  /* I²C registers base address     */
  I2C_InitTypeDef            Init;       /* I²C communication parameters   */
  uint8_t                    *pBuffPtr;  /* Pointer to I²C transfer buffer */
  uint16_t                   XferSize;   /* I²C transfer size              */
  __IO uint16_t              XferCount;  /* I²C transfer counter           */
  DMA_HandleTypeDef          *hdmatx;    /* I²C Tx DMA handle parameters   */
  DMA_HandleTypeDef          *hdmarx;    /* I²C Rx DMA handle parameters   */
  HAL_LockTypeDef            Lock;       /* I²C locking object             */
  __IO HAL_I2C_StateTypeDef  State;      /* I²C communication state        */
  __IO HAL_I2C_ModeTypeDef   Mode;       /* I²C communication mode         */
  __IO uint32_t              ErrorCode;  /* I²C Error code                 */
} I2C_HandleTypeDef;
```

Let us analyze the most important fields of this C `struct`.

- `Instance`: is the pointer to the I²C descriptor we are going to use. For example, `I2C1` is the descriptor of the first I²C peripheral.
- `Init`: is an instance of the C `struct I2C_InitTypeDef` used to configure the peripheral. We will study it more in depth in a while.
- `pBuffPtr`: pointer to the internal buffer used to temporarily store data transferred to and from the I²C peripheral. This is used when the I²C works in interrupt mode and should be not modified from the user code.
- `hdmatx`, `hdmarx`: pointer to instances of the `DMA_HandleTypeDef struct` used when the I²C peripheral works in DMA mode.

The setup of the I²C peripheral is performed by using an instance of the C `struct I2C_InitTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t ClockSpeed;      /* Specifies the clock frequency */
  uint32_t DutyCycle;       /* Specifies the I²C fast mode duty cycle. */
  uint32_t OwnAddress1;     /* Specifies the first device own address. */
  uint32_t OwnAddress2;     /* Specifies the second device own address if dual addressing
                               mode is selected */
  uint32_t AddressingMode;  /* Specifies if 7-bit or 10-bit addressing mode is selected. */
  uint32_t DualAddressMode; /* Specifies if dual addressing mode is selected. */
  uint32_t GeneralCallMode; /* Specifies if general call mode is selected. */
  uint32_t NoStretchMode;   /* Specifies if nostretch mode is selected. */
} I2C_InitTypeDef;
```

These are the functions of the most relevant fields of this C `struct`.

- `ClockSpeed`: this field specifies the speed of the I²C interface and it should correspond to bus speeds defined in the I²C specifications (*standard mode*, *fast mode*, and so on). However, the

exact value of this field is also a function of the `DutyCycle` one, as we will see next. The maximum value for this field is 400000 (400kHz) for that STM32 MCUs supporting up to the *fast mode*. More recent STM32 families support also the *fast mode plus* (1MHz). In these other MCUs, `ClockSpeed` field is replaced with another one called `Timing`. The configuration value for the `Timing` field is computed differently, and we will not cover it here. ST provides a dedicated application note (AN4235[10]) that explains how to compute the exact value for this field according to the wanted I²C bus speed. However, CubeMX is able to generate the right configuration value for you.

Symbol	Parameter	Conditions	Standard-mode		Fast-mode		Fast-mode Plus		Unit
			Min	Max	Min	Max	Min	Max	
f_{SCL}	SCL clock frequency		0	100	0	400	0	1000	kHz
$t_{HD;STA}$	hold time (repeated) START condition	After this period, the first clock pulse is generated.	4.0	-	0.6	-	0.26	-	µs
t_{LOW}	LOW period of the SCL clock		4.7	-	1.3	-	0.5	-	µs
t_{HIGH}	HIGH period of the SCL clock		4.0	-	0.6	-	0.26	-	µs

Table 14.2: Characteristics of the SDA and SCL bus lines for *standard*, *fast*, and *fast-mode plus* I²C-bus devices

- `DutyCycle`: this field, which is available only in those MCU not supporting the *fast mode plus* communication speed, specifies the ratio between t_{LOW} and t_{HIGH} of the I²C SCL line. It can assume the values `I2C_DUTYCYCLE_2` and `I2C_DUTYCYCLE_16_9` to indicate a duty cycle equal to 2:1 and 16:9. By choosing a given clock duty we can "prescale" the peripheral clock to achieve the wanted I²C clock speed. To better understand the role of this configuration parameter, we need to review some fundamental concepts of the I²C bus. In Chapter 11 we have seen that the *duty cycle* is the percentage of one period of time (for example, 10µs) in which a signal is active. For every I²C bus speed, the I²C specification precisely defines the minimum t_{LOW} and t_{HIGH} values. **Table 14.2**, extracted from the UM10204 by NXP[11], shows t_{LOW} and t_{HIGH} values for the given communication speed (values have been highlighted in yellow in **Table 14.2**). The ratio of these two values is the duty cycle, which is independent of the communication speed. For example, a 100kHz period corresponds to 10µs, but $t_{HIGH} + t_{LOW}$ from the **Table 14.2** is less than 10µs (4µs+4.7µs=8.7µs). Thus, the ratio of the actual values can vary as long as the t_{LOW} and t_{HIGH} minimum timings are met (4.7µs and 4µs respectively). The point of these ratios is to illustrate that I²C timing constraints are different between I²C modes. They aren't mandatory ratios that STM32 I²C peripherals need to keep. For example, $t_{HIGH} = 4s$ and $t_{LOW} = 6s$ would be a 0.67 ratio, which is still compatible with timings of the *standard mode* (100kHz) (because $t_{HIGH} = 4s$ and $t_{LOW} > 4.7s$, and their sum is equal to 10µs). The I²C peripherals in STM32 MCUs define the following duty cycles (ratios). For *standard mode* the ratio is fixed to 1:1. This means that $t_{LOW} = t_{HIGH} = 5s$. For *fast mode* we can use two ratios: 2:1 or 16:9. 2:1 ratio means that 4µs (=400kHz) are obtained with $t_{LOW} = 2.66s$ and $t_{HIGH} = 1.33s$ and both the values are higher than the one reported in **Table 14.2** (0.6µs and 1.3µs). A 16:9 ratio means that 4µs are obtained with $t_{LOW} = 2.56s$ and $t_{HIGH} = 1.44s$ and both the values are still higher than the one reported in **Table 14.2**. When to use a 2:1 ratio instead of the 16:9 one and vice

[10] https://bit.ly/2bxBoP1
[11] https://bit.ly/3E18iPF

versa? It depends on the *peripheral clock* (PCLK1) frequency. A 2:1 ratio means that 400MHz are achieved by dividing the clock source by three (1+2). This means that the PCLK1 must be a multiple of 1.2MHz (400kHz * 3). Using a 16:9 ratio means that we are dividing the PCLK1 by 25. That means we can obtain the maximum I²C bus frequency when the PCLK1 is a multiple of 10MHz (400kHz * 25). So, the right selection of the duty cycles depends on the effective speed of the APB1 bus, and the wanted (maximum) I²C SCL frequency. It is important to underline that, even if the SCL frequency is lower than 400kHz (for example, using a ratio equal to 16:9 while having a PCLK1 frequency of 8MHz we can reach a maximum communication speed equal to 360kHz) we still satisfy the requirements of the I²C fast mode specification (400kHz are an upper limit).

- `OwnAddress1`, `OwnAddress2`: the I²C peripheral in STM32 MCUs can be used to develop both master and slave I²C devices. When developing I²C slave devices, the `OwnAddress1` field allows to specify the I²C slave address: the peripheral automatically detects the given address on the I²C bus, and it automatically triggers all the related events (for example, it can generate the corresponding interrupt so that the firmware code can start a new transaction on the bus). I²C peripheral supports 7- or 10-bit addressing, as well as the *7-bit dual addressing* mode: in this case we can specify two distinct 7-bit slave addresses, so that the device is able to answer to requests sent to both addresses.
- `AddressingMode`: this field can assume the values `I2C_ADDRESSINGMODE_7BIT` or `I2C_ADDRESSINGMODE_10BIT` to specify 7- or 10-bit addressing mode respectively.
- `DualAddressMode`: this field can assume the values `I2C_DUALADDRESS_ENABLE` or `I2C_DUALADDRESS_DISABLE` to enable/disable the *7-bit dual addressing mode*.
- `GeneralCallMode`: the *General Call* is a sort of broadcast addressing in the I²C protocol. A special I²C slave address, `0x0000 000`, is used to send a message to all devices on the same bus. General call is an optional feature and, by setting this field to the `I2C_GENERALCALL_ENABLE` value, the I²C peripheral will generate events when the general call address is matched. We will not treat this mode in this book.
- `NoStretchMode`: this field, which can assume the values `I2C_NOSTRETCH_ENABLE` or `I2C_NOSTRETCH_DISABLE` is used to disable/enable the optional clock stretching mode (take note that by setting it to `I2C_NOSTRETCH_ENABLE` you disable the clock stretching mode). For more information about this optional I²C mode, refer to UM10204 by NXP[12] and to the reference manual for your MCU.

As usual, to configure the I²C peripheral we use the function:

```
HAL_StatusTypeDef HAL_I2C_Init(I2C_HandleTypeDef *hi2c);
```

which accepts a pointer to an instance of the `I2C_HandleTypeDef` seen before.

[12] https://bit.ly/3E18iPF

14.2.1 Using the I²C Peripheral in *Master Mode*

We are now going to analyze the main routines provided by the CubeHAL to use the I²C peripheral in master mode. To perform a transaction over the I²C bus in write mode, the CubeHAL provides the function:

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                          uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

where:

- `hi2c`: it is the pointer to an instance of the struct I2C_HandleTypeDef seen before, which identifies the I²C peripheral;
- `DevAddress`: it is the address of the slave device, which can be 7- or 10-bits long depending on the specific IC;
- `pData`: it is the pointer to an array, with a length equal to the `Size` parameter, containing the sequence of bytes we are going to transmit;
- `Timeout`: represents the maximum time, expressed in milliseconds, we are willing to wait for the transmit completion. If the transmission does not complete in the specified timeout time, the function aborts and returns the `HAL_TIMEOUT` value; otherwise, it returns the `HAL_OK` value if no other errors occur. Moreover, we can pass a timeout equal to `HAL_MAX_DELAY` (`0xFFFF FFFF`) to wait indefinitely for the transmit completion.

To perform a transaction in read mode we can use, instead, the following function:

```
HAL_StatusTypeDef HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                         uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

Both the previous functions perform the transaction in *polling mode*. For interrupt based transactions, we can use the functions:

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                             uint8_t *pData, uint16_t Size);                \

HAL_StatusTypeDef HAL_I2C_Master_Receive_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                            uint8_t *pData, uint16_t Size);
```

These functions work in the same way of other routines seen in previous chapters (for example, those one related to UART transmission in interrupt mode). To use them correctly, we need to enable the corresponding ISR and to place a call to the `HAL_I2C_EV_IRQHandler()` routine, which in turn calls the `HAL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef *hi2c)` to signal the completion of the transfer in write mode, or the `HAL_I2C_MasterRxCpltCallback(I2C_HandleTypeDef *hi2c)`

to signal the end of a transfer in read mode. Except for STM32F0 and STM32L0 families, the I²C peripheral in all STM32 MCUs uses a separated interrupt to signal error conditions (look at the *vector table* related to your MCU). For this reason, in the corresponding ISR we need to call the `HAL_I2C_ER_IRQHandler()`, which in turn calls the `HAL_I2C_ErrorCallback(I2C_HandleTypeDef *hi2c)` in case of an error. There exist ten different callbacks invoked by the CubeHAL. The **Table 14.3** lists all of them, together with the ISR that invokes the callback.

Table 14.3: CubeHAL available callbacks when an I²C peripheral works in interrupt or DMA mode

Callback	Calling ISR	Description
`HAL_I2C_MasterTxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Signals that the transfer from master to slave is completed (peripheral working in master mode).
`HAL_I2C_MasterRxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Signals that the transfer from slave to master is completed (peripheral working in master mode).
`HAL_I2C_SlaveTxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Signals that the transfer from slave to master is completed (peripheral working in slave mode).
`HAL_I2C_SlaveRxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Signals that the transfer from master to slave is completed (peripheral working in slave mode).
`HAL_I2C_MemTxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Signals that the transfer from master to an external memory is completed (this is called only when `HAL_I2C_Mem_xxx()` routines are used and the peripheral works in master mode).
`HAL_I2C_MemRxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Signals that the transfer from an external memory to the master is completed (this is called only when `HAL_I2C_Mem_xxx()` routines are used and the peripheral works in master mode).
`HAL_I2C_AddrCallback()`	`I2Cx_EV_IRQHandler()`	Signals that the master has placed the peripheral slave address on the bus (peripheral working in slave mode).
`HAL_I2C_ListenCpltCallback()`	`I2Cx_EV_IRQHandler()`	Signals that the listen mode is completed (this happens when a STOP condition is issued and the peripheral works in slave mode - more about this later).
`HAL_I2C_ErrorCallback()`	`I2Cx_ER_IRQHandler()`	Signals that an error condition is occurred (peripheral working both in master and slave mode).
`HAL_I2C_AbortCpltCallback()`	`I2Cx_ER_IRQHandler()`	Signals that a STOP condition has been raised and the I²C transaction has been aborted (peripheral working both in master and slave mode).

Finally, the functions:

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit_DMA(I2C_HandleTypeDef *hi2c,uint16_t DevAddress,
                                              uint8_t *pData, uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Master_Receive_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                             uint8_t *pData, uint16_t Size);
```

allow to perform I²C transactions using DMA.

To make complete and full working examples we need an external device able to interact through the I²C bus, since Nucleo boards do not provide such peripherals. For this reason, we will use an external EEPROM memory: the 24LCxx. This is a popular family of serial EEPROMs, which are become a sort of standard in electronics industry. They are cheap (cost usually few tens of cents), they are produced in several packages (ranging from "old" THT P-DIP packages, up to modern and compact WLCP ones), they provide a data retention for more than 200 years and individual pages can be erased more than one million of times. Moreover, a lot of silicon manufacturers have their own compatible versions (ST also provides its own set of 24LCxx compatible EEPROMs). These memories have the same popularity of 555 timers, and I bet that they will survive for a lot of years to technology innovation.

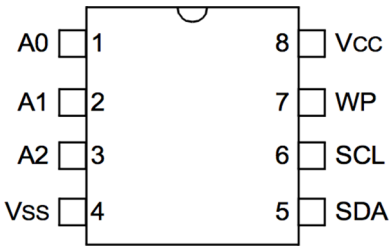


Figure 14.6: The pinout of a 24LCxx EEPROM with a PDIP-8 package

Our examples will be based on the 24LC64 model, which is a 64Kbits EEPROM (this means that the memory is able to store 8KB or, if you prefer, 8192 bytes). The pinout of the PDIP-8 version is shown in **Figure 14.6**. **A0**, **A1** and **A2** are used to set the LSB bits of the I²C address, as shown in **Figure 14.7**: if one of those pins is tied to the ground, then the corresponding bit is set to 0; if tied to VDD, then the bit is set to 1. If all three pins are tied to the ground, then the I²C address corresponds to 0xA0.

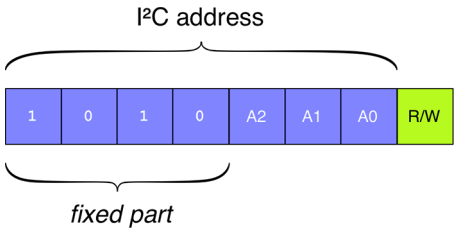


Figure 14.7: How the 24LCxx I²C address is composed.

WP pin is the *write protection* pin: if tied to the ground, we can write inside individual memory cells. On the contrary, if connected to VDD, write operations have no effects. Since I2C1 peripheral

is mapped to the same pins in all Nucleo boards, **Figure 14.8** shows the right way to connect a 24LCxx EEPROM to the Arduino connector in all nine Nucleo boards we are considering in this book.

F1



Read Carefully

STM32F1 microcontrollers do not provide the ability to pull-up SDA and SCL lines. Their GPIOs must be configured as *open-drain*. So, you have to add two additional resistors to pull-up I²C lines. Something between 4K and 10K is a proven value.

As said before, a 64Kbits EEPROM has 8192 addresses, ranging from `0x0000` up to `0x1FFF`. An individual byte write is performed by sending over the I²C bus the EEPROM address, the upper half of the memory address followed by the lower half, and the value to store in that cell, closing the transaction with a STOP condition.

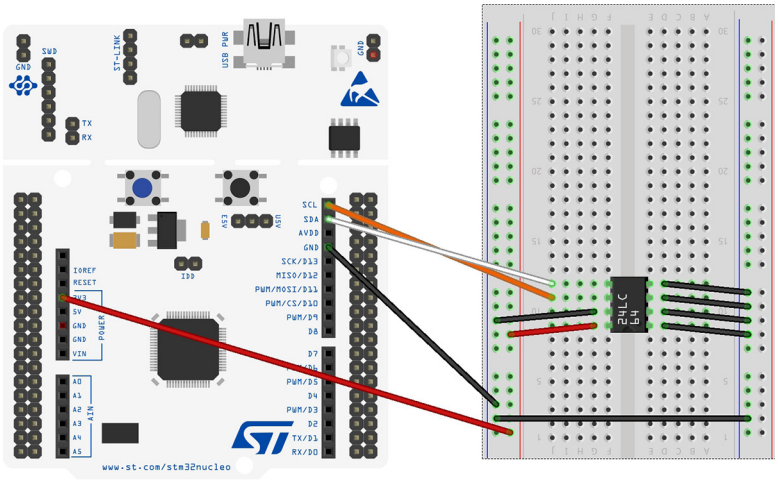


Figure 14.8: How to connect a Nucleo to a 24LCxx EEPROM

Assuming we want to store the value 0x4C inside the memory location 0x320, then **Figure 14.9** shows the right transaction sequence. The address 0x320 is split in two parts: the upper part, equal to 0x3 is transmitted first, and the lower part equal to 0x20 is sent right after. Then the data to store is sent. We can also send multiple bytes in the same transaction: an internal *address counter* automatically increments at every byte sent. This allows us to reduce the transaction time and increase the total throughput.

The ACK bit set by the I²C EEPROM after the last sent byte does not mean that data has been effectively stored inside the memory. Sent data is stored in a temporarily buffer, since EEPROM location memories are erased page-by-page and not individually. The whole page (which is composed by 32 bytes) is refreshed at every write operation, and the transferred bytes are stored only at the end of this operation. During the erase time, every command sent to the EEPROM will

be ignored. To detect when a write operation has been completed, we need to use the *acknowledge polling*. This involves the master sending a START condition followed by slave address plus the control byte for a write command (R/W bit set to 0). If the device is still busy with the write cycle, then no ACK will be returned. If no ACK is returned, the START bit and control byte must be re-sent. If the cycle is complete, the device will return the ACK and the master can then proceed with the next read or write command.



Figure 14.9: How to perform a write operation with a 24LCxx EEPROM

Read operations are initiated in the same way as write operations, with the exception that the R/W bit of the control byte is set to 1. There are three basic types of read operations: current address read, random read and sequential read. In this book we will focus our attention on the random read mode only, leaving to the reader the responsibility to deepen the other modes.

Random read operations allow the master to access any memory location in a random manner. To perform this type of read operation, the memory address must be sent first. This is accomplished by sending the memory address to the 24LCxx as part of a write operation (R/W bit set to '0'). Once the memory address is sent, the master generates a RESTART condition (*repeating START*) following the ACK[13]. This terminates the write operation, but not before the internal address counter is set. The master then issues the slave address again, but with the R/W bit set to a 1 this time. The 24LCxx will then issue an ACK and transmit the 8-bit data word. The master will not acknowledge the transfer and generates a STOP condition, which causes the EEPROM to discontinue transmission (see **Figure 14.10**). After a random read command, the internal address counter will point to the address location following the one that was just read.

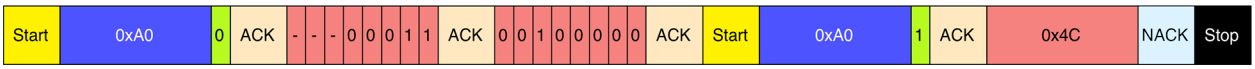


Figure 14.10: How to perform a random read operation with a 24LCxx EEPROM

We are finally ready to arrange a complete example. We will create two simple functions, named `Read_From_24LCxx()` and `Write_To_24LCxx()` that allow to write/read data from a 24LCxx memory, using the CubeHAL. We will then test these routines by simply storing a string inside the EEPROM, and then reading it back: if the original string is equal to the one read from the EEPROM, then the Nucleo LD2 LED starts blinking.

[13]The 24LCxx EEPROM memories are designed so that they work in the same way even if we end the transaction by issuing a STOP condition, and then we immediately start a new one in read mode. This degree of flexibility we will allow us to build the first example of this chapter, as we will see in a while.

Filename: `src/main-ex1.c`

```c
int main(void) {
  const char wmsg[] = "We love STM32!";
  char rmsg[20] = {0};

  HAL_Init();
  Nucleo_BSP_Init();

  MX_I2C1_Init();

  Write_To_24LCxx(&hi2c1, 0xA0, 0x1AAA, (uint8_t*)wmsg, strlen(wmsg)+1);
  Read_From_24LCxx(&hi2c1, 0xA0, 0x1AAA, (uint8_t*)rmsg, strlen(wmsg)+1);

  if(strcmp(wmsg, rmsg) == 0) {
    while(1) {
      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
      HAL_Delay(100);
    }
  }

  while(1);
}

/* I2C1 init function */
static void MX_I2C1_Init(void) {
  GPIO_InitTypeDef GPIO_InitStruct;

  /* Peripheral clock enable */
  __HAL_RCC_I2C1_CLK_ENABLE();

  hi2c1.Instance = I2C1;
  hi2c1.Init.ClockSpeed = 100000;
  hi2c1.Init.DutyCycle = I2C_DUTYCYCLE_2;
  hi2c1.Init.OwnAddress1 = 0x0;
  hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c1.Init.OwnAddress2 = 0;
  hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;

  GPIO_InitStruct.Pin = GPIO_PIN_8|GPIO_PIN_9;
  GPIO_InitStruct.Mode = GPIO_MODE_AF_OD;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
  GPIO_InitStruct.Alternate = GPIO_AF4_I2C1;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

```

```
60   HAL_I2C_Init(&hi2c1);
61 }
```

Let us analyze the above fragment of code starting from the `MX_I2C1_Init()` routine. It starts enabling the I2C1 peripheral clock, so that we can program its registers. Then we set the bus speed (100kHz in our case - the duty cycle setting is ignored in this case, because the duty cycle is fixed to 1:1 when the bus runs at speeds lower or equal to 100kHz). We then configure PB8 and PB9 pins so that they act as SCL and SDA lines respectively.

The `main()` routine is simple: it stores the string `"We love STM32!"` at the 0x1AAA memory location; the string is then read back from the EEPROM and compared with the original one. We need to explain just why we are storing and reading a buffer with a length equal to `strlen(wmsg)+1`. This because the C `strlen()` routines gives back the length of the string skipping the string terminator char (`'\0'`). Without storing this char, and then reading it back from the EEPROM, the `strcmp()` at line 26 wouldn't be able to compute the exact length of the string.

Filename: `src/main-ex1.c`

```
63 HAL_StatusTypeDef Read_From_24LCxx(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemA\
64 ddress, uint8_t *pData, uint16_t len) {
65   HAL_StatusTypeDef returnValue;
66   uint8_t addr[2];
67
68   /* We compute the MSB and LSB parts of the memory address */
69   addr[0] = (uint8_t) ((MemAddress & 0xFF00) >> 8);
70   addr[1] = (uint8_t) (MemAddress & 0xFF);
71
72   /* First we send the memory location address where start reading data */
73   returnValue = HAL_I2C_Master_Transmit(hi2c, DevAddress, addr, 2, HAL_MAX_DELAY);
74   if(returnValue != HAL_OK)
75     return returnValue;
76
77   /* Next we can retrieve the data from EEPROM */
78   returnValue = HAL_I2C_Master_Receive(hi2c, DevAddress, pData, len, HAL_MAX_DELAY);
79
80   return returnValue;
81 }
82
83 HAL_StatusTypeDef Write_To_24LCxx(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAd\
84 dress, uint8_t *pData, uint16_t len) {
85   HAL_StatusTypeDef returnValue;
86   uint8_t *data;
87
88   /* First we allocate a temporary buffer to store the destination memory
89    * address and the data to store */
90   data = (uint8_t*)malloc(sizeof(uint8_t)*(len+2));
91
```

```
92    /* We compute the MSB and LSB parts of the memory address */
93    data[0] = (uint8_t) ((MemAddress & 0xFF00) >> 8);
94    data[1] = (uint8_t) (MemAddress & 0xFF);
95
96    /* And copy the content of the pData array in the temporary buffer */
97    memcpy(data+2, pData, len);
98
99    /* We are now ready to transfer the buffer over the I2C bus */
100   returnValue = HAL_I2C_Master_Transmit(hi2c, DevAddress, data, len + 2, HAL_MAX_DELAY);
101   if(returnValue != HAL_OK)
102     return returnValue;
103
104   free(data);
105
106   /* We wait until the EEPROM effectively stores data in memory */
107   while(HAL_I2C_Master_Transmit(hi2c, DevAddress, 0, 0, HAL_MAX_DELAY) != HAL_OK);
108
109   return HAL_OK;
110 }
```

We can now focus our attention on the two routines to use the 24LCxx EEPROM. Both of them are designed to accept:

- the I²C slave address of the EEPROM memory (`DevAddress`);
- the memory address where start storing/reading data (`MemAddress`);
- the pointer to the memory buffer used to exchange data with the EEPROM (`pData`);
- the amount of data to store/read (`len`);

The `Read_From_24LCxx()` function starts computing the two halves of the memory address (MSB and LSB part). It then sends the two parts over the I²C bus using the `HAL_I2C_Master_Transmit()` routine (line 73). As said before, the 24LCxx memory is designed so that it sets the internal address counter to the passed address. We can so start a new transaction in read mode to retrieve the amount of data from the EEPROM (line 78).

The `Write_To_24LCxx()` functions does a similar thing, but in a different way. It must adhere to the 24LCxx protocol described in **Figure 14.9**, which slightly differs from the one in **Figure 14.8** . This means that we cannot use two separated transactions for the memory address and the data to store, but we have to perform a unique I²C transaction. For this reason, we use a temporary and dynamic buffer (line 90), which contains the two halves of the memory address plus the data to store in the EEPROM. We can so perform a transaction over the I²C bus (line 98) and then wait until the EEPROM completes the memory transfer (line 107).

14.2.1.1 I/O MEM Operations

The protocol used by the 24LCxx EEPROM is indeed common to all I²C devices that have memory-addressable registers to read to and to write from. For example, a lot of I²C sensors, like the HTS221

by ST, adopt the same protocol. For this reason, ST engineers have already implemented specific routines inside the CubeHAL that do the same job of `Read_From_24LCxx()` and `Write_To_24LCxx()` better and faster. The functions:

```
HAL_StatusTypeDef HAL_I2C_Mem_Write(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                    uint16_t MemAddress, uint16_t MemAddSize,
                                    uint8_t *pData, uint16_t Size, uint32_t Timeout);
HAL_StatusTypeDef HAL_I2C_Mem_Read(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                   uint16_t MemAddress, uint16_t MemAddSize,
                                   uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

allow to store and retrieve data from memory-addressable I²C devices, with just one notably difference: the `HAL_I2C_Mem_Write()` function is not designed to wait for the write-cycle completion, as we have done in the previous example at line 107. But, even for this operation the HAL provides a dedicated and more portable routine:

```
HAL_StatusTypeDef HAL_I2C_IsDeviceReady(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                        uint32_t Trials, uint32_t Timeout);
```

This function accepts a maximum number of `Trials` before returning back an error condition, but if we pass to the function the `HAL_MAX_DELAY` as `Timeout` value, then we can pass 1 to the `Trials` argument. When the polled I²C device is ready the function returns `HAL_OK`. Otherwise, it returns the `HAL_BUSY` value.

So, the `main()` function seen before can be rearranged in the following way:

```
14  int main(void) {
15    char wmsg[] ="We love STM32!";
16    char rmsg[20];
17
18    HAL_Init();
19    Nucleo_BSP_Init();
20
21    MX_I2C1_Init();
22
23    HAL_I2C_Mem_Write(&hi2c1, 0xA0, 0x1AAA, I2C_MEMADD_SIZE_16BIT, (uint8_t*)wmsg,
24                      strlen(wmsg)+1, HAL_MAX_DELAY);
25    while(HAL_I2C_IsDeviceReady(&hi2c1, 0xA0, 1, HAL_MAX_DELAY) != HAL_OK);
26
27    HAL_I2C_Mem_Read(&hi2c1, 0xA0, 0x1AAA, I2C_MEMADD_SIZE_16BIT, (uint8_t*)rmsg,
28                     strlen(wmsg)+1, HAL_MAX_DELAY);
29
30    if(strcmp(wmsg, rmsg) == 0) {
31      while(1) {
32        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
33        HAL_Delay(100);
```

```
34      }
35    }
36
37    while(1);
38  }
```

The above APIs works in polling mode, but the CubeHAL provides also corresponding routines to perform transactions in interrupt and DMA mode. As usual, these other APIs have a similar function signature, with just one thing to note: the callback functions used to signal the end of transfers are the `HAL_I2C_MemTxCpltCallback()` and `HAL_I2C_MemRxCpltCallback()`, as reported in **Table 14.3**.

14.2.1.2 Combined Transactions

The transmission sequence of a read operation in a 24LCxx EEPROM memory is a combined transaction. A RESTART condition is used before inverting the direction of the I²C transmission (from write to read). In the first example we were able to use two separated transactions inside the `Read_From_24LCxx()` because 24LCxx EEPROMs are designed to work in the same way. This is possible thanks to the internal address counter: the first transaction sets the address counter to the wanted location; the second one, performed in read mode, retrieves the data from the EEPROM starting from that location. However, this not only reduces the maximum throughput that may be reached but, more important, often leads to not portable code: there exist several I²C devices that strictly adhere to the I²C protocol and implement combined transactions according to the specification using a RESTART condition (so they do not tolerate a STOP condition in the middle).

The CubeHAL provides two dedicated routines to handle combined transaction or, as they are called in the Cube HAL, *sequential transmissions*:

```
HAL_I2C_Master_Sequential_Transmit_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                      uint8_t *pData, uint16_t Size,uint32_t XferOptions);
HAL_I2C_Master_Sequential_Receive_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                     uint8_t *pData, uint16_t Size, uint32_t XferOptions);
```

Compared to the other routines seen before, the only relevant parameter to highlight here is `XferOptions`. It can assume one of the values reported in **Table 14.4** and it is used to drive the generation of **START/RESTART/STOP** conditions in a single transaction. Both functions work in this way. Let us assume that we want to read *n-bytes* from the 24LCxx EEPROM. According to the I²C protocol, we must execute the following operations (refer to **Figure 14.10**):

1. we have to begin a new transaction in write mode issuing a START condition followed by the slave address;
2. we then transfer two bytes containing MSB and LSB parts of the memory address;
3. we so issue a RESTART condition and transmit the slave device address with the last bit set to 1 to indicate a read transaction.
4. the slave device starts sending data byte-by-byte until we end the transaction by issuing a **NACK** or a STOP condition.

Table 14.4: Values for the `XferOptions` parameter to drive the generation of STAR/RESTART/STOP conditions

Transfer option	Description
`I2C_FIRST_FRAME`	This option allows to generate just the START condition, without generating the final STOP condition at the end of transmission.
`I2C_NEXT_FRAME`	This option allows to generate a RESTART before transmitting data if the direction changes (that is we call `HAL_I2C_Master_Sequential_Transmit_IT()` after `HAL_I2C_Master_Sequential_Receive_IT()` or vice versa), or it allows to manage only the new data to transfer if no direction changes and **without** a final STOP condition in both cases.
`I2C_LAST_FRAME`	This option allows to generate a RESTART before transmitting data if the direction changes (that is we call `HAL_I2C_Master_Sequential_Transmit_IT()` after `HAL_I2C_Master_Sequential_Receive_IT()` or vice versa), or it allows to manage only the transfer of new data if no direction changes and **with** a final STOP condition in both cases.
`I2C_FIRST_AND_LAST_FRAME`	No sequential usage. Both the routine work in the same way of `HAL_I2C_Master_Transmit_IT()` and `HAL_I2C_Master_Receive_IT()` functions.

Using *sequential transmission* routines, we can proceed in the following way:

1. we invoke the `HAL_I2C_Master_Sequential_Transmit_IT()` routine by passing the slave address and the two bytes forming the memory location address; we invoke the function by passing the value `I2C_FIRST_FRAME`, so that it generates a START condition without issuing a STOP condition after the two bytes have been sent;
2. we so call the `HAL_I2C_Master_Sequential_Receive_IT()` routine by passing the slave address, the pointer to the buffer used to store read bytes, the amount of bytes to read from the EEPROM and the value `I2C_LAST_FRAME`, so that the function generates a RESTART condition and terminates the transaction at the end of transfer by issuing a STOP condition.

14.2.1.3 A Note About the Clock Configuration in STM32F0/L0/L4 families

In STM32F0/L0 families it is possible to select different clock sources for the I2C1 peripheral. This because in those families the I2C1 peripheral is able to work even in some low-power modes, allowing to wake-up the MCU when the I²C works in slave mode and the configured slave address is placed on the bus. Refer to the *Clock view* in CubeMX for more about this.

In STM32L4/5 MCUs it is possible to select the clock source for all I²C peripherals.

14.2.2 Using the I²C Peripheral in *Slave Mode*

Nowadays there are a lot of *System-on-Board* (SoB) modules on the market. These are usually small PCBs already populated with several ICs and specialized in doing something relevant. GPRS and GPS modules or multi-sensors boards are examples of SoB modules. These modules then are

soldered to the main board, thanks to the fact that they expose solderable pads on their sides also know as "castellated vias" or "castellations". **Figure 14.11** shows the STEVAL-STLCS01V1[14] Sensor Development Kit module by ST, which is an integrated and programmable module with an STM32L476, three highly-integrated sensors (6-axis digital e-compass and a 3-axis digital gyroscope, a barometer, a microphone) plus a *Bluetooth Low Energy* (BLE) network processor BlueNRG-MS.

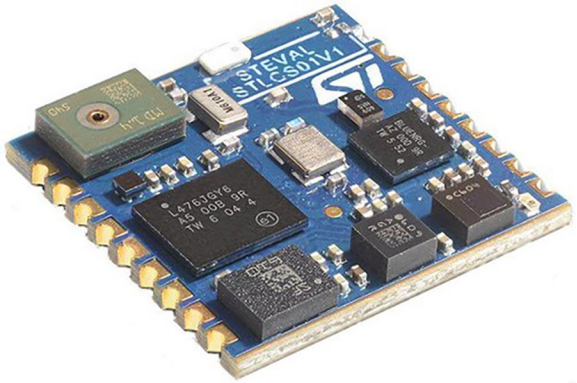


Figure 14.11: The STEVAL-STLCS01V1 Sensor Development Kit by ST

The MCU on these boards usually comes pre-programmed with a firmware, which is specialized in doing a well-established task. The host board also contains another programmable IC, maybe another MCU or something similar. The main board interacts with the SoB using a well-known communication protocol, which usually are the UART, the CAN bus, the SPI or the I²C bus. For this reason, it is quite common to program STM32 devices to make they work in I²C slave mode.

The CubeHAL provides all the necessary glue to develop I²C slave applications easily. The slave routines are identical to the one used to program I²C peripherals in master mode. For example, the following routines are used to transmit/receive data in interrupt mode when the I²C peripheral is used in slave mode:

```
HAL_StatusTypeDef HAL_I2C_Slave_Transmit_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData,
                                            uint16_t Size);
HAL_StatusTypeDef HAL_I2C_Slave_Receive_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData,
                                           uint16_t Size);
```

In the same way, the callback routines invoked at the end of data transmission/reception are the following ones:

```
void HAL_I2C_SlaveTxCpltCallback(I2C_HandleTypeDef *hi2c);
void HAL_I2C_SlaveRxCpltCallback(I2C_HandleTypeDef *hi2c);
```

We are now going to analyze a complete example that shows how to develop I²C slave applications using the CubeHAL. We will realize a sort of digital temperature sensor with an I²C interface similar to the majority of digital temperature sensors on the market (for example, the popular TMP275 by TI and the HT221 by ST). This "sensor" will provide just three registers:

[14] https://bit.ly/3vCGKND

- a `WHO_AM_I` register, used by master code to check that the I²C interface works correctly; this register returns the fixed value 0xBC.
- two temperature-related registers, named `TEMP_OUT_INT` and `TEMP_OUT_FRAC`, which contains the integer and fractional part of the acquired temperature; for example, if the detected temperature is equal to 27.34°C, then the `TEMP_OUT_INT` register will contain the value 27 and the `TEMP_OUT_FRAC` the value 34.



Figure 14.12: The I²C protocol used to read internal register of our slave device

Our sensor will be designed to answer to a really simple protocol, based on *combined transactions*, which is shown in **Figure 14.12**. As you can see, the only notably difference with the protocol used by 24LCxx EEPROMs, when accessing to memory in random read mode, is the size of the memory register, which is just one byte in this case.

The example provides both a "slave" and a "master" implementation: the macro `SLAVE_BOARD`, defined at project level, drives the compilation of the two parts. The example requires two Nucleo boards[15].

Filename: `src/main-ex2.c`

```
11  #ifdef SLAVE_BOARD
12  static void MX_ADC1_Init(void);
13  #endif
14  static void MX_I2C1_Init(void);
15
16  volatile uint8_t transferDirection, transferRequested;
17
18  #define TEMP_OUT_INT_REGISTER   0x0
19  #define TEMP_OUT_FRAC_REGISTER  0x1
20  #define WHO_AM_I_REGISTER       0xF
21  #define WHO_AM_I_VALUE          0xBC
22  #define TRANSFER_DIR_WRITE      0x1
23  #define TRANSFER_DIR_READ       0x0
24  #define I2C_SLAVE_ADDR          0x33
25
26  int main(void) {
27    char uartBuf[30];
28    uint8_t i2cBuf[2];
29    float ftemp;
30    int8_t t_frac, t_int;
31
32    HAL_Init();
```

[15] Unfortunately, when I started designing this example, I thought that it were possible to use just one board, connecting the pins associated with an I²C peripheral to those ones of another I²C peripheral (for example, I2C1 pins directly connected to the I2C3 pins). But, after a lot of struggling, I reached to the conclusion that I²C peripherals in an STM32 are not "truly asynchronous" and it is not possible to use two I²C peripherals concurrently. So, to run this examples you will need two Nucleo boards, or just one Nucleo and another development kit: in this case, you need to rearrange the master part accordingly.

```
33    Nucleo_BSP_Init();
34
35    MX_I2C1_Init();
36
37  #ifdef SLAVE_BOARD
38    uint16_t rawValue;
39    uint32_t lastConversion;
40
41    MX_ADC1_Init();
42    HAL_ADCEx_Calibration_Start(&hadc1);
43    HAL_ADC_Start(&hadc1);
44
45    while(1) {
46      HAL_I2C_EnableListen_IT(&hi2c1);
47      while(!transferRequested) {
48        if(HAL_GetTick() - lastConversion > 1000L) {
49          HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
50
51          rawValue = HAL_ADC_GetValue(&hadc1);
52          ftemp = ((float)rawValue) / 4095 * 3300;
53          ftemp = ((ftemp - 1430.0) / 4.3) + 25;
54
55          t_int = ftemp;
56          t_frac = (ftemp - t_int)*100;
57
58          sprintf(uartBuf, "Temperature: %.2f\r\n", ftemp);
59          HAL_UART_Transmit(&huart2, (uint8_t*)uartBuf, strlen(uartBuf), HAL_MAX_DELAY);
60
61          sprintf(uartBuf, "t_int: %d - t_frac: %d\r\n", t_int, t_frac);
62          HAL_UART_Transmit(&huart2, (uint8_t*)uartBuf, strlen(uartBuf), HAL_MAX_DELAY);
63
64          lastConversion = HAL_GetTick();
65        }
66      }
67
68      transferRequested = 0;
69
70      if(transferDirection == TRANSFER_DIR_WRITE) {
71        /* Master is sending register address */
72        HAL_I2C_Slave_Sequential_Receive_IT(&hi2c1, i2cBuf, 1, I2C_FIRST_FRAME);
73        while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_LISTEN);
74
75        switch(i2cBuf[0]) {
76          case WHO_AM_I_REGISTER:
77            i2cBuf[0] = WHO_AM_I_VALUE;
78            break;
79          case TEMP_OUT_INT_REGISTER:
```

```
80            i2cBuf[0] = t_int;
81            break;
82          case TEMP_OUT_FRAC_REGISTER:
83            i2cBuf[0] = t_frac;
84            break;
85          default:
86            i2cBuf[0] = 0xFF;
87        }
88
89        HAL_I2C_Slave_Sequential_Transmit_IT(&hi2c1, i2cBuf, 1, I2C_LAST_FRAME);
90        while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
91      }
92    }
```

The most relevant part of the `main()` function starts at line 46. The `HAL_I2C_EnableListen_IT()` routine enables all the I²C peripheral-related interrupts. This means that a new interrupt will fire when the master places the slave device address (which is defined by the macro `I2C_SLAVE_ADDR`). The `HAL_I2C_EV_IRQHandler()` routines so will automatically call the `HAL_I2C_AddrCallback()` function, that we will analyze later.

The `main()` function then starts performing an A/D conversion of the internal temperature sensor every second, and it splits the acquired temperature (stored in the `ftemp` variable) in two 8-bit integers, `t_int` and `t_frac`: these represent the integer and fractional parts of the temperature. The main function temporarily stops the A/D conversion as soon as `transferRequested` variable becomes equal to 1: this global variable is set by the `HAL_I2C_AddrCallback()` function, together with the `transferDirection` one, which contains the transfer direction (read/write) of the I²C transaction.

If the master is starting a new transaction in write mode, then it means that it is transferring the register address. The `HAL_I2C_Slave_Sequential_Receive_IT()` function is then invoked at line 72: this will cause that the register address is received from the master. Since the function works in interrupt mode, we need a way to wait until the transfer is completed. The `HAL_I2C_GetState()` returns the internal status of the HAL, which is equal to `HAL_I2C_STATE_BUSY_RX_LISTEN` until the transfer finishes. When this happens, the status goes back to `HAL_I2C_STATE_LISTEN` and we can proceed by transferring to the master the content of the wanted register.

This is performed at line 89, where the function `HAL_I2C_Slave_Sequential_Transmit_IT()` is called: the function inverts the transfer direction, and sends to the master the content of the wanted register. The tricky part is represented by the line 90. Here we do a busy spin until the I²C peripheral state is equal to `HAL_I2C_STATE_READY`. Why we do not check the peripheral status against the `HAL_I2C_STATE_LISTEN` state, as we have performed at line 73? To understand this aspect, we need to remember an important thing of *combined transactions*. When a transaction inverts the transfer direction, the master starts acknowledging every byte sent. Remember that only the master knows how long a transaction lasts, and it decides when to stop the transaction. In *combined transactions*, a master ends the transfer from the slave to the master by issuing a NACK, which causes the slave to issue a STOP condition. From the I²C peripheral point of view, a STOP condition causes the peripheral to

exit from *listen mode* (technically speaking, it generates an abort condition - if you implement the `HAL_I2C_AbortCpltCallback()` callback, you can track when this happens), and that is the reason why we need to check against the `HAL_I2C_STATE_READY` state and to place again the peripheral in *listen mode* at line 46.

Filename: `src/main-ex2.c`

```
94  #else //Master board
95    i2cBuf[0] = WHO_AM_I_REGISTER;
96    HAL_I2C_Master_Sequential_Transmit_IT(&hi2c1, I2C_SLAVE_ADDR, i2cBuf,
97                                          1, I2C_FIRST_FRAME);
98    while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
99
100   HAL_I2C_Master_Sequential_Receive_IT(&hi2c1, I2C_SLAVE_ADDR, i2cBuf,
101                                        1, I2C_LAST_FRAME);
102   while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
103
104   sprintf(uartBuf, "WHO AM I: %x\r\n", i2cBuf[0]);
105   HAL_UART_Transmit(&huart2, (uint8_t*) uartBuf, strlen(uartBuf), HAL_MAX_DELAY);
106
107   i2cBuf[0] = TEMP_OUT_INT_REGISTER;
108   HAL_I2C_Master_Sequential_Transmit_IT(&hi2c1, I2C_SLAVE_ADDR, i2cBuf,
109                                         1, I2C_FIRST_FRAME);
110   while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
111
112   HAL_I2C_Master_Sequential_Receive_IT(&hi2c1, I2C_SLAVE_ADDR, (uint8_t*)&t_int,
113                                        1, I2C_LAST_FRAME);
114   while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
115
116   i2cBuf[0] = TEMP_OUT_FRAC_REGISTER;
117   HAL_I2C_Master_Sequential_Transmit_IT(&hi2c1, I2C_SLAVE_ADDR, i2cBuf,
118                                         1, I2C_FIRST_FRAME);
119   while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
120
121   HAL_I2C_Master_Sequential_Receive_IT(&hi2c1, I2C_SLAVE_ADDR, (uint8_t*)&t_frac,
122                                        1, I2C_LAST_FRAME);
123   while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
124
125   ftemp = ((float)t_frac)/100.0;
126   ftemp += (float)t_int;
127
128   sprintf(uartBuf, "Temperature: %.2f\r\n", ftemp);
129   HAL_UART_Transmit(&huart2, (uint8_t*) uartBuf, strlen(uartBuf), HAL_MAX_DELAY);
130
131 #endif
132
133   while (1);
134 }
```

Finally, it is important to underline that the implementation of the "slave part" is still not sufficiently robust. In fact, we should handle all the possible wrong cases that may happen. For example, the master may shutdown the connection just in the middle of the two transactions. This would complicate a lot the example, and it is left to exercise to the reader.

The "master part" of the example starts at line 94. The code is straightforward. Here we use the `HAL_I2C_Master_Sequential_Transmit_IT()` function to start a combined transaction and the `HAL_I2C_Master_Sequential_Receive_IT()` to retrieve the content of the wanted register from the slave. The integer and fractional part of the temperature are then combined again in a `float`, and the acquired temperature is printed on the UART2.

Filename: `src/main-ex2.c`

```
136 void I2C1_EV_IRQHandler(void) {
137   HAL_I2C_EV_IRQHandler(&hi2c1);
138 }
139
140 void I2C1_ER_IRQHandler(void) {
141   HAL_I2C_ER_IRQHandler(&hi2c1);
142 }
143
144 void HAL_I2C_AddrCallback(I2C_HandleTypeDef *hi2c, uint8_t TransferDirection, uint16_t AddrMat\
145 chCode) {
146   UNUSED(AddrMatchCode);
147
148   if(hi2c->Instance == I2C1) {
149     transferRequested = 1;
150     transferDirection = TransferDirection;
151   }
152 }
```

The last part we need to analyze is represented by ISR handlers. The ISR `I2C1_EV_IRQHandler()` invokes the `HAL_I2C_EV_IRQHandler()`, as said before. This causes that the `HAL_I2C_AddrCallback()` function is called every time the master transmit the slave address on the bus. When invoked, the callback will receive the pointer to the `I2C_HandleTypeDef` representing the specific I²C descriptor, the direction of the transfer (`TransferDirection`) and the matched I²C address (`AddrMatchCode`): this is required because an STM32 I²C peripheral working in slave mode can answer to two different addresses, and so we have a way to write conditional code depending on the I²C address issued by the master.

14.3 Using CubeMX to Configure the I²C Peripheral

As usual, CubeMX reduces to the minimum the effort needed to configure the I²C peripheral. Once the peripheral is enabled in the *Category list pane* (from the *Pinout view*), we can configure all settings from the *Configuration pane*, as shown in **Figure 14.13**.

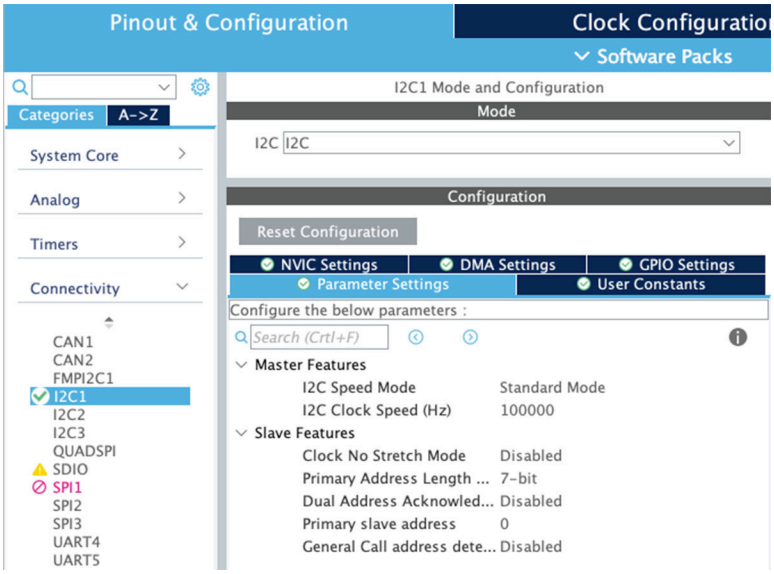


Figure 14.13: The CubeMX configuration view to setup the I²C peripheral



Read Carefully

By default, when enabling the I2C1 peripheral in STM32 MCUs with LQFP-64 packages, CubeMX enables as default peripheral I/Os PB7 and PB6 pins (SDA and SCL respectively). These aren't the pins latched to the Arduino connector on the Nucleo, but you need to select the two alternative pins PB9 and PB8 by clicking on them and then selecting the corresponding function from the drop-down menu, as shown in the following picture.

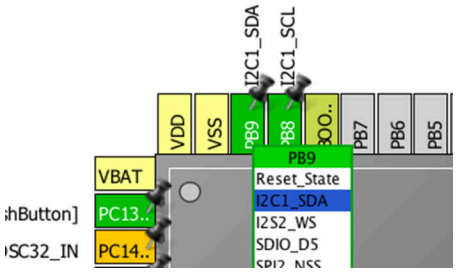


Figure 14.14: How to select the right I2C1 pins in a Nucleo-64 board

15. SPI

In the previous chapter we have analyzed one of the two most widespread communication standards that rule the "market" of intra-boards communication systems: the I²C protocol. Now it is time to analyze the other player: the SPI protocol.

All STM32 microcontrollers provide at least one SPI interface, which allows to develop both master and slave applications. The CubeHAL implements all the necessary stuff to program such peripherals easily. This chapter gives a quick overview of the `HAL_SPI` module after, as usual, a brief introduction to the SPI specification.

15.1 Introduction to the SPI Specification

The *Serial Peripheral Interface* (SPI) is a specification about serial, synchronous and full-duplex communications between a master controller (which is usually implemented with an MCU or something with programmable functionalities) and several slave devices. As we will see next, the nature of the SPI interface allows full duplex as well as half duplex communications over the same bus. SPI specification is a *de facto* standard, and it was defined by Motorola[1] in late '70, and it is still largely adopted as communication protocol for many digital ICs. Different from the I²C protocol, the SPI specification does not force a given message protocol over its bus, but it is limited to bus signaling giving to slave devices total freedom about the structure of exchanged messages.

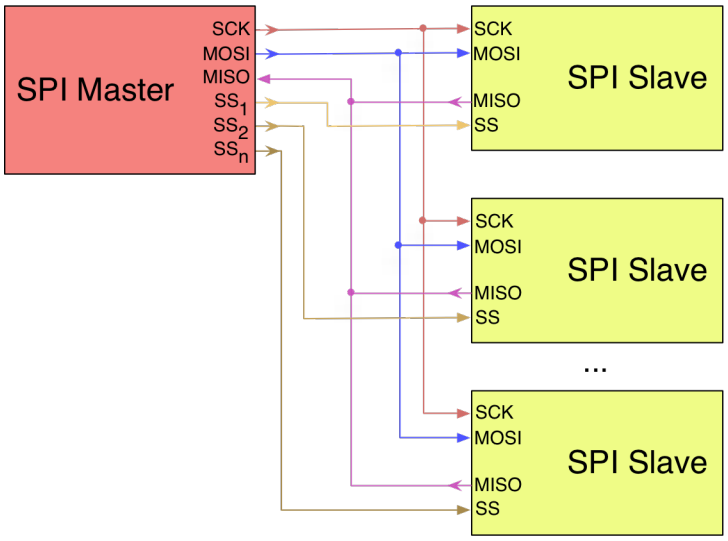


Figure 15.1: The structure of a typical SPI bus

[1]Motorola was a pioneer company in the semiconductor industry that has been split in several sub-companies over the years. The semiconductor division of Motorola flowed into ON Semiconductor, while the microcontrollers division became Freescale Semiconductor. This last one was acquired by NXP in 2015.

A typical SPI bus is formed by four signals, as shown in **Figure 15.1**, even if it is possible to drive some SPI devices with just three I/Os (in this case we talk about *3-wire SPI*):

- **SCK**: this signal I/O is used to generate the clock to synchronize data transfer over the SPI bus. It is generated by the master device, and this means that in an SPI bus every transfer is always started by the master. Different from the I²C specification, the SPI is intrinsically faster, and the SPI clock speed is usually several MHz. Nowadays is quite common to find SPI devices able to exchange data at a rate up to 100MHz. Moreover, the SPI protocol allows to devices with different communication speeds to coexist over the same bus.
- **MOSI**: the name of this signal I/O stands for *Master Output Slave Input*, and it is used to send data from the master device to a slave one. Different from the I²C bus, where just one wire is used to exchange data both the ways, the SPI protocol defines two distinct lines to exchange data between master and slaves.
- **MISO**: it stands for *Master Input Slave Output* and it corresponds to the I/O line used to send data from a slave device to the master.
- ***SSn***: it stands for *Slave Select* and in a typical SPI bus there exist '*n*' separated lines used to address the specific SPI devices involved in a transaction. Different from the I²C protocol, the SPI does not use slave addresses to select devices, but it demands this operation to a physical line that is asserted LOW to perform a selection. In a typical SPI bus only one slave device can be active at same time by asserting low its SS line. This is the reason why devices with different communication speeds can coexist on the same bus[2].

Having two separated data communication lines, MOSI and MISO, the SPI intrinsically allows full-duplex communications, since a slave device can send data to the master while it receives new one from it. In a one-to-one SPI bus (just one master and one slave), the SS signal can be omitted (the corresponding slave's I/O is tied to the ground), and MISO/MOSI lines are fused in a single line called *Slave In/Slave Out* (SISO). In this case we can talk about *2-wire SPI*, even if it is essentially a *3-wire* bus.

[2]For the sake of completeness, we have to say that this is not the exact reason why it is possible to have devices with different communication speeds on the same bus. The main reason is due to the fact that slave I/Os are implemented with tri-state I/Os, that is they are placed in high-impedance state (disconnected) when the SS line is not asserted LOW.

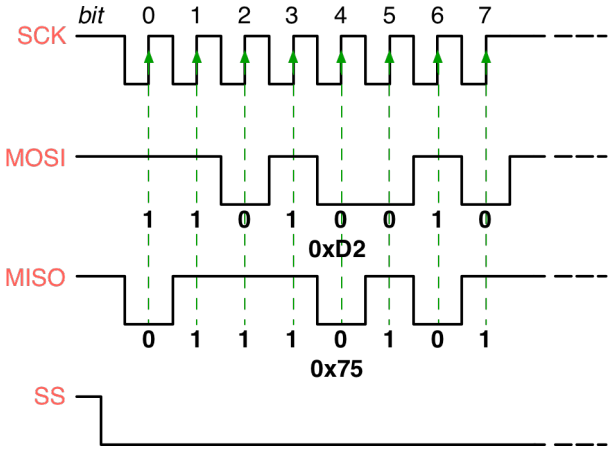


Figure 15.2: How data is exchanged over a SPI bus in a full-duplex transmission

Every transaction over the bus is started by enabling the SCK line according to the maximum slave frequency. Once the clock line starts generating the signal, the master asserts the SS line LOW and data transmission can begin. Transmissions normally involve two registers of a given word size[3], one in the master and one in the slave. Data is usually shifted out with the most-significant bit first, while shifting a new least-significant bit into the same register. At the same time, data from the slave is shifted into the least-significant bit register. After the register bits have been shifted out and in, the master and slave have exchanged data. If more data needs to be exchanged, the shift registers are reloaded and the process repeats. Transmission may continue for any number of clock cycles. When complete, the master stops toggling the clock signal, and typically deselects the slave.

Figure 15.2 shows the way data is transferred in a full-duplex transmission, while **Figure 15.3** shows the way it is typically exchanged in a half-duplex connection.

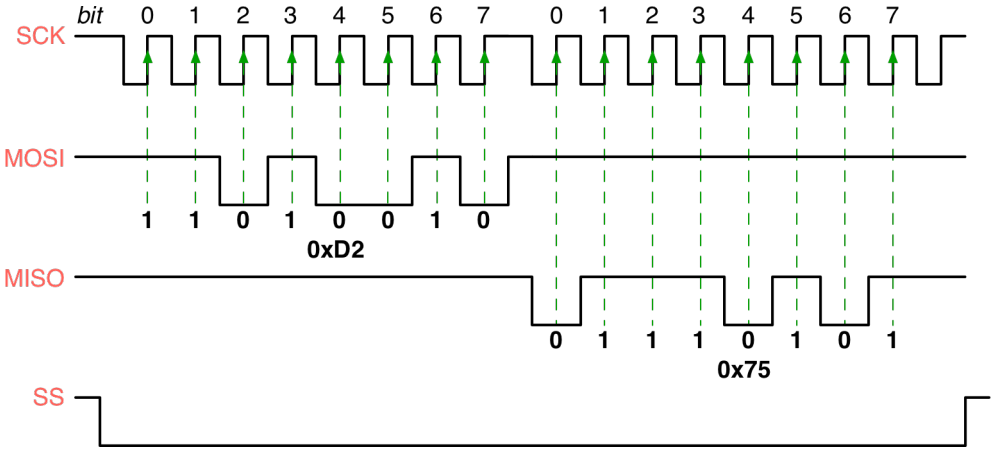


Figure 15.3: How data is exchanged over a SPI bus in a half-duplex transmission

[3] 8-bit data transmissions are the rule, but some slave devices support even 16-bit ones.

15.1.1 Clock Polarity and Phase

In addition to setting the bus clock frequency, the master and slaves must also agree on the clock *polarity* and *phase* with respect to the data exchanged over MOSI and MISO lines. SPI Specification by Motorola[4] names these two settings as CPOL and CPHA respectively, and most silicon vendors have adopted that convention.

The combinations of polarity and phase are often referred to as SPI *bus modes* which are commonly numbered according to **Table 15.1**. The most common mode are *mode 0* and *mode 3*, but the majority of slave devices support at least a couple of bus modes.

Table 15.1: SPI bus modes according to CPOL and CPHA configuration

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

The timing diagram is shown in **Figure 15.4**, and it is further described below:

- At **CPOL=0** the base value of the clock is zero, i.e., the active state is 1 and idle state is 0.
  - For **CPHA=0**, data is captured on the SCK rising edge (LOW ⟶ HIGH transition) and data is output on a falling edge (HIGH ⟶ LOW clock transition).
  - For **CPHA=1**, data is captured on the SCK falling edge and data is output on a rising edge.
- At **CPOL=1** the base value of the clock is one (inversion of CPOL=0), i.e., the active state is 0 and idle state is 1.
  - For **CPHA=0**, data is captured on SCK falling edge and data is output on a rising edge.
  - For **CPHA=1**, data is captured on SCK rising edge and data is output on a falling edge.

That is, CPHA=0 means sampling on the first clock edge, while CPHA=1 means sampling on the second clock edge, regardless of whether that clock edge is rising or falling. Note that with CPHA=0, the data must be stable for a half cycle before the first clock cycle.

[4] http://bit.ly/2cc3T3S

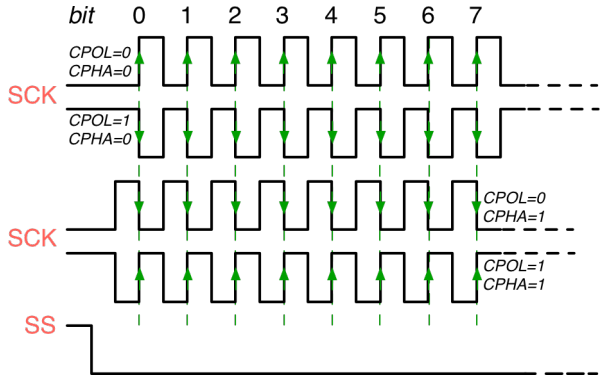


Figure 15.4: The SPI timing diagram according to CPOL and CPHA settings

15.1.2 Slave Select Signal Management

As said before, the SPI slave devices do not have an address that identifies them on the bus, but they start exchanging data with the master as long as the *Slave Select* (SS) signal is LOW. STM32 microcontrollers provide two distinct modes to handle the SS signal, which is called NSS in the ST documentation. Let us analyze them.

- **NSS software mode**: The SS signal is driven by the firmware and any free GPIO can be used to drive an IC when the MCU works in master mode, or to detect when another master is starting a transfer if the MCU works in slave mode.
- **NSS hardware mode**: a specific MCU I/O is used to drive the SS signal, and it is internally managed by the SPI peripheral. Two configurations are possible depending on the NSS output configuration:
  - **NSS output enabled**: this configuration is used only when the device operates in master mode. The NSS signal is driven LOW when the master starts the communication and is kept LOW until the SPI is disabled. It is important to remark that this mode is suitable when there is just one SPI slave device on the bus and its SS I/O is connected to the NSS signal. This configuration does not allow multi-master mode.
  - **NSS output disabled**: this configuration allows multi-master capability for devices operating in master mode. For devices set as slave, the NSS pin acts as a classical NSS input: the slave is selected when NSS is LOW and deselected when NSS HIGH.

15.1.3 SPI *TI Mode*

SPI peripherals in STM32 microcontrollers support the *TI Mode* when working in master mode and when the NSS signal is configured to work in hardware. In *TI mode* the clock polarity and phase are forced to conform to the Texas Instruments protocol requirements whatever the values set. NSS management is also specific to the TI protocol, which makes the configuration of NSS management transparent for the user. In *TI mode*, in fact, the NSS signal "pulses" at the end of every transmitted byte (it goes from LOW to HIGH from the beginning of the LSB bit and goes from HIGH to LOW

at the starting of the MSB bit forming the next transferred byte). For more information about this communication mode, refer to the reference manual for the MCU you are considering.

Nucleo P/N	SPI1			SPI2			SPI3		
	MOSI	MISO	SCK	MOSI	MISO	SCK	MOSI	MISO	SCK
NUCLEO-G474RE	PA7	PA6	PA5	PA11	PA10	PF1	PC12	PC11	PC10
	PB5	PB4	PB3	PB15	PB14	PB13	PB5	PB4	PB3
NUCLEO-F446RE	PA7	PA6	PA5	PC1	PC2	PB10	PB0	PC11	PC10
	PB5	PB4	PB3	PB15	PB14	PB13	PB2	PB4	PB3
NUCLEO-F401RE	PA7	PA6	PA5	PC3	PC2	PB10	PC12	PC11	PC10
	PB5	PB4	PB3	PB15	PB14	PB13	PB5	PB4	PA3
NUCLEO-F303RE	PA7	PA6	PA5	PB15	PB14	PB13	PC12	PC11	PC10
	PB5	PB4	PB3	PA11	PA10	PF1	PB5	PB4	PB3
NUCLEO-F103RB	PA7	PA6	PA5	PB15	PB14	PB13	-	-	-
	PB5	PB4	PB3	-	-	-	-	-	-
NUCLEO-F072RB	PA7	PA6	PB3	PC3	PC2	PB10	-	-	-
	PB4	PB5	PA5	PB15	PB14	PB13	-	-	-
NUCLEO-L476RG	PA7	PA6	PA5	PC3	PC2	PB10	PC12	PC11	PC10
	PB5	PB4	PB3	PB15	PB14	PB13	PB5	PB4	PA3
NUCLEO-L152RE	PA7	PA6	PB3	PB15	PB14	PB13	PC12	PC11	PC10
	PB4	PB5	PA5	-	-	-	PB5	PB4	PB3
NUCLEO-L073RZ	PA7	PA6	PA5	PC3	PC2	PB10	-	-	-
	PB5	PB4	PB3	PB15	PB14	PB13	-	-	-

Table 15.2: Effective availability of SPI peripherals in MCUs equipping all nine Nucleo boards

15.1.4 Availability of SPI Peripherals in STM32 MCUs

Depending on the family type and package used, STM32 microcontrollers can provide up to six independent SPI peripherals. **Table 15.2** summarizes the availability of SPI peripherals in STM32 MCUs equipping all nine Nucleo boards we are considering in this book.

For every SPI peripheral, and a given STM32 MCU, **Table 15.2** shows the pins corresponding to MOSI, MISO and SCK lines. Moreover, darker rows show alternate pins that can be used during the layout of the board. For example, given the STM32F401RE MCU, we can see that SPI1 peripheral is mapped to PA7, PA6 and PA5, but PB5, PB5 and PB3 can be also used as alternate pins. Note that the SPI1 peripheral uses the same I/O pins in all STM32 MCUs with LQFP-64 package. This is another clear example of the pin-to-pin compatibility offered by STM32 microcontrollers.

We are now ready to see how-to use the CubeHAL APIs to program this peripheral.

15.2 HAL_SPI Module

To program the SPI peripheral, the HAL defines the C `struct SPI_HandleTypeDef`, which is defined in the following way[5]:

```
typedef struct __SPI_HandleTypeDef {
  SPI_TypeDef                *Instance;    /* SPI registers base address */
  SPI_InitTypeDef            Init;         /* SPI communication parameters */
  uint8_t                    *pTxBuffPtr;  /* Pointer to SPI Tx transfer Buffer */
  uint16_t                   TxXferSize;   /* SPI Tx Transfer size */
  __IO uint16_t              TxXferCount;  /* SPI Tx Transfer Counter */
  uint8_t                    *pRxBuffPtr;  /* Pointer to SPI Rx transfer Buffer */
  uint16_t                   RxXferSize;   /* SPI Rx Transfer size */
  __IO uint16_t              RxXferCount;  /* SPI Rx Transfer Counter */
  DMA_HandleTypeDef          *hdmatx;      /* SPI Tx DMA Handle parameters   */
  DMA_HandleTypeDef          *hdmarx;      /* SPI Rx DMA Handle parameters   */
  HAL_LockTypeDef            Lock;         /* Locking object                 */
  __IO HAL_SPI_StateTypeDef  State;        /* SPI communication state */
  __IO uint32_t              ErrorCode;    /* SPI Error code */
} SPI_HandleTypeDef;
```

Let us analyze the most important fields of this struct.

- `Instance`: is the pointer to the SPI descriptor we are going to use. For example, `SPI1` is the descriptor of the first SPI peripheral.
- `Init`: is an instance of the C `struct SPI_InitTypeDef` used to configure the peripheral. We will study it more in depth in a while.
- `pTxBuffPtr`, `pRxBuffPtr`: pointer to the internal buffers used to temporarily store data transferred to and from the SPI peripheral. This is used when the SPI works in interrupt mode and should be not modified from the user code.
- `hdmatx`, `hdmarx`: pointer to instances of the `DMA_HandleTypeDef struct` used when the SPI peripheral works in DMA mode.

The setup of the SPI peripheral is performed by using an instance of the C `struct SPI_InitTypeDef`, which is defined in the following way:

[5]Some fields have been omitted for simplicity. Refer to the CubeHAL source code for the exact definition of the `SPI_HandleTypeDef struct`.

```
typedef struct {
  uint32_t Mode;              /* Specifies the SPI operating mode. */
  uint32_t Direction;         /* Specifies the SPI bidirectional mode state. */
  uint32_t DataSize;          /* Specifies the SPI data size. */
  uint32_t CLKPolarity;       /* Specifies the serial clock steady state. */
  uint32_t CLKPhase;          /* Specifies the clock active edge for the bit capture. */
  uint32_t NSS;               /* Specifies whether the NSS signal is managed by
                                 hardware (NSS pin) or by software */
  uint32_t BaudRatePrescaler; /* Specifies the Baud Rate prescaler value which will be
                                 used to configure the SCK clock. */
  uint32_t FirstBit;          /* Specifies whether data transfers start
                                 from MSB or LSB bit. */
  uint32_t TIMode;            /* Specifies if the TI mode is enabled or not. */
  uint32_t CRCCalculation;    /* Specifies if the CRC calculation is enabled or not. */
  uint32_t CRCPolynomial;     /* Specifies the polynomial used for the CRC calculation. */
} SPI_InitTypeDef;
```

- `Mode`: this parameter sets the SPI in master or slave mode. It can assume the values `SPI_MODE_MASTER` and `SPI_MODE_SLAVE`.
- `Direction`: it specifies whatever the slave peripheral works in *4-wire* (two separated lines for input/output) or *3-wire* (just one line for I/O). It can assume the value `SPI_DIRECTION_2LINES` to configure a full-duplex *4-wire* mode; the value `SPI_DIRECTION_2LINES_RXONLY` to setup a half-duplex *4-wire* mode; the value `SPI_DIRECTION_1LINE` to configure a half-duplex *3-wire* mode.
- `DataSize`: configures the size of the transferred data over the SPI bus, and it can assume the values `SPI_DATASIZE_8BIT` and `SPI_DATASIZE_16BIT`.
- `CLKPolarity`: it configures the SCK CPOL setting and it can assume the values `SPI_POLARITY_LOW` (which corresponds to CPOL=0) and `SPI_POLARITY_HIGH` (which corresponds to CPOL=1).
- `CLKPhase` this related field sets the clock phase, and it can assume the values `SPI_PHASE_1EDGE` (which corresponds to CPHA=0) and `SPI_PHASE_2EDGE` (which corresponds to CPHA=1).
- `NSS`: this field handles the behaviour of the NSS I/O. It can assume the values `SPI_NSS_SOFT` to configure NSS signal in *software mode*; the values `SPI_NSS_HARD_INPUT` and `SPI_NSS_HARD_OUTPUT` to configure the NSS signal in input and output *hardware mode* respectively.
- `BaudRatePrescaler`: it sets the prescaler of the APB clock and it establishes the maximum SCK clock speed. It can assume the values `SPI_BAUDRATEPRESCALER_2`, `SPI_BAUDRATEPRESCALER_4`, ..., `SPI_BAUDRATEPRESCALER_256` (all two's powers from 2^1 up to 2^8).
- `FirstBit`: specifies the data transmission ordering, and it can assume the values `SPI_FIRSTBIT_MSB` and `SPI_FIRSTBIT_LSB`.
- `TIMode`: it is used to enable/disable the *TI mode*, and it can assume the values `SPI_TIMODE_DISABLE` and `SPI_TIMODE_ENABLE`.
- `CRCCalculation` and `CRCPolynomial`: the SPI peripheral in all STM32 microcontrollers supports the CRC generation in hardware. A CRC value can be transmitted as last byte in Tx mode, or automatic CRC error checking can be performed for last received byte. The CRC value is

calculated using an odd programmable polynomial on each bit. The calculation is processed on the sampling clock edge defined by the CPHA and CPOL configurations. The calculated CRC value is checked automatically at the end of the data block as well as for transfer managed by CPU or by the DMA. When a mismatch is detected between the CRC calculated internally on the received data and the CRC sent by the transmitter, an error condition is set. The CRC feature is not available when the SPI is driven in DMA circular mode. For more information about this option, refer to the reference manual for the STM32 MCU you are considering.

As usual, to configure the SPI peripheral we use the function:

```
HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef *hspi);
```

which accepts a pointer to an instance of the `SPI_HandleTypeDef struct` seen before.

15.2.1 Exchanging Messages Using SPI Peripheral

Once the SPI peripheral is configured, we can start exchanging data with slave devices. Since the SPI specification does not forces a given communication protocol, there is no difference among the CubeHAL routines when using the SPI peripheral in *slave* or *master* mode. The only difference resides in the peripheral configuration, setting the `Mode` parameter of the `SPI_InitTypeDef` structure accordingly.

As usual, the CubeHAL provides three ways to communicate over a SPI bus: *polling*, *interrupt* and *DMA* mode.

To send a number of bytes to a slave device in *polling* mode, we use the function:

```
HAL_StatusTypeDef HAL_SPI_Transmit(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size,
                                   uint32_t Timeout);
```

The function signature is almost identical to other communication routines seen so far (for example, those used for the UART manipulation), so we will not describe its parameters here. This function can be used if the SPI peripheral is configured to work both in `SPI_DIRECTION_1LINE` or `SPI_DIRECTION_2LINES` modes. To receive a number of bytes in *polling* mode, we use the function:

```
HAL_StatusTypeDef HAL_SPI_Receive(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size,
                                  uint32_t Timeout);
```

This function can be used in all three `Direction` modes.

If the slave device supports the full-duplex mode, then we can use the function:

```
HAL_StatusTypeDef HAL_SPI_TransmitReceive(SPI_HandleTypeDef *hspi, uint8_t *pTxData,
                                          uint8_t *pRxData, uint16_t Size,
                                          uint32_t Timeout);
```

which allows to transmit a given number of bytes while receiving the same quantity simultaneously. Clearly it works only when the SPI `Direction` is set to `SPI_DIRECTION_2LINES`.

To exchange data over the SPI in *interrupt* mode, the CubeHAL provides the functions:

```
HAL_StatusTypeDef HAL_SPI_Transmit_IT(SPI_HandleTypeDef *hspi, uint8_t *pData,
                                      uint16_t Size);
HAL_StatusTypeDef HAL_SPI_Receive_IT(SPI_HandleTypeDef *hspi, uint8_t *pData,
                                     uint16_t Size);
HAL_StatusTypeDef HAL_SPI_TransmitReceive_IT(SPI_HandleTypeDef *hspi, uint8_t *pTxData,
                                             uint8_t *pRxData, uint16_t Size);
```

The CubeHAL routine to exchange data over the SPI in *DMA* mode are identical to the three one before, except for the fact that they end in `_DMA`.

Once using interrupt- and DMA-based routines, we must be prepared to be notified when the transmission is ended, since it is performed asynchronously. This means that we need to enable the corresponding interrupt at NVIC level and to call the function `HAL_SPI_IRQHandler()` from the ISR. There exist six different callbacks we can implement, as reported in **Table 15.3**.

Table 15.3: CubeHAL available callbacks when an SPI peripheral works in interrupt or DMA mode

Callback	Description
`HAL_SPI_TxCpltCallback()`	Signals that a given amount of bytes have been transmitted
`HAL_SPI_RxCpltCallback()`	Signals that a given amount of bytes have been received
`HAL_SPI_TxRxCpltCallback()`	Signals that a given amount of bytes have been transmitted and received
`HAL_SPI_TxHalfCpltCallback()`	Signals that the DMA SPI half transmit process is complete
`HAL_SPI_RxHalfCpltCallback()`	Signals that the DMA SPI half receive process is complete
`HAL_SPI_TxRxHalfCpltCallback()`	Signals that the DMA SPI half transmit and receive process is complete

When the SPI peripheral is configured in DMA circular mode, we can use the following routines to pause/resume/abort a DMA circular transaction:

```
HAL_StatusTypeDef HAL_SPI_DMAPause(SPI_HandleTypeDef *hspi);
HAL_StatusTypeDef HAL_SPI_DMAResume(SPI_HandleTypeDef *hspi);
HAL_StatusTypeDef HAL_SPI_DMAStop(SPI_HandleTypeDef *hspi);
```

When the SPI works in DMA circular mode, the following restriction apply:

- the DMA circular mode cannot be used when the SPI is accessed exclusively in receive mode;

- the CRC feature is not managed when the DMA circular mode is enabled
- when the SPI DMA pause/stop features are used, we must use the function `HAL_SPI_DMAPause()`/ `HAL_SPI_DMAStop()` only under the SPI callbacks.

In this chapter we will not analyze any concrete example. In Chapter 26 we will use the SPI peripheral to program a hardwired TCP/IP embedded Ethernet controller, which allows us to build Internet-based applications with Nucleo boards.

15.2.2 Maximum Transmission Frequency Reachable using the CubeHAL

The SCK frequency is derived from the PCLK frequency using a programmable prescaler. This prescaler ranges from 2^1 up to 2^8. However, as said several other times before, the CubeHAL adds an unavoidable overhead when driving peripherals. And this also applies to the SPI one. In fact, using the CubeHAL it is not possible to reach all supported SPI frequencies with the different SPI modes.

ST engineers have clearly documented this in the CubeHAL. If you open the **stm32XXxx_hal_spi.c** file, you can see (about at line 120) two tables that report the maximum reachable transmission frequency given the direction mode (half-duplex or full-duplex) and the way to program and use the peripheral (*polling*, *interrupt* and *DMA*).

For example, in an STM32F4 MCU we can reach a SCK frequency equal to $f_{PCLK}/8$ if the SPI peripheral works in *slave* mode and we program it using CubeHAL in *interrupt* mode.

15.3 Using CubeMX to Configure SPI Peripheral

To use CubeMX to enable the wanted SPI peripheral, we have to proceed in the following order. First, we need to select the wanted communication, as shown in **Figure 15.5**. Next, we need to specify the behavior of the NSS signal in the same configuration view. Once these two parameters are set, we can proceed by configuring other SPI settings in the CubeMX *Configuration* pane.

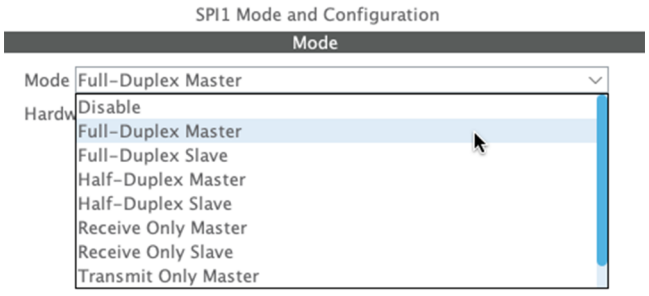


Figure 15.5: How to select the SPI communication mode in CubeMX

16. Cyclic Redundancy Check

In digital systems it is perfectly possible that data gets corrupted, especially if it flows through a communication medium. In digital electronics, a message is a stream of bits either equal to 0 or 1 and it becomes corrupted when one of more of these bits accidentally change during transmission. For this reason, messages are always exchanged with some additional data used to detect if the original message was corrupted. In Chapter 8 we have analyzed an early form of error detection related to data transmission: the *parity bit* is an additional bit added to the message used to keep track if the number of bits equal to 1 is odd or even (depending on the type of parity). However, this method is not able to detect errors if two or more bits change at the same time.

The *Cyclic Redundancy Check* (CRC) is a widely used technique for detecting errors in digital data, both during transmission and storage. In the CRC method, several check bits, called the *checksum*[1], are appended to the message being transmitted. The receiver can determine whether the check bits agree with the data, to assert with a certain degree of probability if an error occurred in transmission. If so, the receiver can ask to the sender to retransmit the message again.
This technique is also applied in some data storage devices, such as Hard Disk Drives. In this case each block on the disk would have certain check bits, and the hardware might automatically initiate a reread of the block when an error is detected, or it might report the error to software. It is important to underline that CRC is a good method to identify corrupted messages, but not for making corrections when errors are detected.

Being the CRC method used by a lot of communication peripherals and protocols (like the Ethernet, MODBUS, etc.), it is quite common to find in microcontrollers dedicated hardware peripherals able to compute CRC checksum of byte streams, freeing the CPU from performing this operation in software. All STM32 microcontrollers provide a dedicated CRC peripheral, and this chapter briefly explains how to use the corresponding CubeHAL module.

As usual, before going into the implementation details, we will first give a brief introduction to the math behind the CRC technique[2].

16.1 Introduction to CRC Computing

CRC technique is based on well-known properties of polynomial arithmetic. To compute the checksum of a stream of bits, the message is seen as a polynomial that is divided by another fixed polynomial, called *generator polynomial*. The remainder of this operation is the *checksum*, which is

[1] The *checksum* is often called the *CRC*. This is not entirely correct, because the CRC is a specific error-detecting method, which uses a well-characterized algorithm plus a checksum sequence of bits to detect if a message is corrupted. However, it is quite common to refer to the checksum as the *CRC*, or the *CRC code*.

[2] An excellent dissertation of CRC algorithms is represented by this on-line document by Ross N. Williams (http://www.zlib.net/crc_v3.txt)

added to original message. The receiver will use it, together with the generator polynomial, to check if the message is correct.

In practice, all CRC methods use polynomials in *GF(2ⁿ)*. *GF(pⁿ)* stands for *Galois field*, also known as *finite field*, that is a field with a finite number of elements. As with any field, a *Galois field* is a set on which the operations of multiplication, addition, subtraction and division are defined and satisfy certain basic rules. The most common examples of finite fields are given by the *integers modulo* p, where p is a prime number. In our case, p is equal to 2 and this implies that the *GF(2ⁿ)* field contains only two elements, when *n=1*: 0 and 1.

In *GF(2ⁿ)* addition and subtraction are performed modulo 2, that is they correspond to the XOR logical operation.

$\oplus$	**0**	**1**
0	0	1
1	1	0

The multiplication, instead, corresponds to the AND logical operation.

$\wedge$	**0**	**1**
0	0	0
1	0	1

Polynomials in *GF(2ⁿ)* are polynomials in a single variable x whose coefficients are either 0 or 1. The CRC technique interprets the bits of a data message as coefficients of a polynomial in *GF(2ⁿ)* with a degree equal to $n-1$, where n is the length of the message. For example, assuming the message 11100110_2, whose length is equal to 8, this corresponds to the polynomial:

$$x^7 \cdot \mathbf{1} + x^6 \cdot \mathbf{1} + x^5 \cdot \mathbf{1} + x^4 \cdot \mathbf{0} + x^3 \cdot \mathbf{0} + x^2 \cdot \mathbf{1} + x^1 \cdot \mathbf{1} + x^0 \cdot \mathbf{0} = x^7 + x^6 + x^5 + x^2 + x$$

As said before, in *GF(2ⁿ)* addition and subtraction correspond to XOR logical operation. This means that the sum of the polynomials $x^4 + x^3 + 1$ and $x^3 + x + 1$ is equal to $x^4 + x$[3]. Clearly, this is also the same of the subtraction of the two polynomials.

Multiplication of polynomials in *GF(2ⁿ)* is, as usual, much like multiplying decimal integers keeping track of powers of x instead of decimal places. For example, multiplying the previous two polynomials we have:

[3] Instead, in normal algebra the addition would be equal to $x^4 + 2x^3 + x + 2$.

$$
\begin{array}{r}
x^4+x^3+\quad 1 \\
\hline
x^3+x+1 \\
\hline
x^4+x^3+\quad 1 \\
x^5+x^4+\quad x \\
\hline
x^7+x^6+\quad x^3 \\
\hline
x^7+x^6+x^5+\quad x+1
\end{array}
$$

As you can see, each term in the first multiplies each term in the second, and then we add them following the addition rules in *GF(2ⁿ)*.

Division of one polynomial by another in *GF(2ⁿ)* is analogous to long division (with remainder) of integers, except there is no borrowing nor carrying. For example, let us divide the polynomial $x^7+x^6+x^5+x^2+x$ by the polynomial x^3+x+1.

$$x^3+x+1 \,\overline{)\, x^7+x^6+x^5+\quad x^2+x}$$

We start by dividing the first term of the dividend by the highest term of the divisor (meaning the one with the highest power of x, which in this case is x^3). Next, we multiply the divisor by the result just obtained (the first term of the eventual quotient).

$$
\begin{array}{r}
x^4 \\
x^3+x+1 \,\overline{)\, x^7+x^6+x^5+\quad x^2+x} \\
\underline{x^7+\quad x^5+x^4} \\
\end{array}
$$

Now we subtract the product just obtained from the appropriate terms of the original dividend applying the rules of subtraction in *GF(2ⁿ)*.

$$
\begin{array}{r}
x^4 \\
x^3+x+1 \,\overline{)\, x^7+x^6+x^5+\quad x^2+x} \\
\underline{x^7+\quad x^5+x^4} \\
\cancel{x^7}\quad x^6+\cancel{x^7}+x^4
\end{array}
$$

We repeat the previous steps, except this time use the two terms that have just been written as the dividend.

$$
\begin{array}{r}
x^4+x^3 \\
\hline
x^3+x+1 \;\big)\; x^7+x^6+x^5+\qquad\qquad x^2+x \\
\underline{x^7+\qquad x^5+x^4} \\
\quad x^6+x^4 \\
\underline{x^6+\qquad x^4+x^3} \\
\qquad\quad x^3+x^2+x \\
\underline{\qquad\quad x^3+\qquad x+1} \\
\qquad\quad\; x^2+\qquad 1
\end{array}
$$

The process continues until the obtained dividend has a degree lower than the divisor. We have so obtained the remainder of the division, which represents the checksum to append to the original message.

$$
\begin{array}{r}
x^4+x^3+1 \\
\hline
x^3+x+1 \;\big)\; x^7+x^6+x^5+\qquad\quad x^2+x \\
\underline{x^7+\qquad x^5+x^4} \\
\quad x^6+\quad\; +x^4 \\
\underline{\quad x^6+\qquad x^4+x^3} \\
\qquad\qquad\quad x^3+x^2+x \\
\underline{\qquad\qquad\quad x^3+\qquad x+1} \\
\qquad\qquad\qquad\; x^2+\qquad 1
\end{array}
$$

There are two ways for the receiver to assess the correctness of the transmission. It can compute the checksum from the first n bits of the received data, and verify that it agrees with the last r received bits. Alternatively, and following usual practice, the receiver can divide all the received bits by the generator polynomial and check that the r-bit remainder is 0.

However, the exact algorithm of CRC calculation usually differs from the normal polynomial division. Moreover, the generator polynomial may define specific initial and final condition, as we will see soon. This means that the generator polynomial cannot be left to change, but it is kept from a portfolio[4] of well-studied polynomials. For example, the widely adopted CRC-32 polynomial has the form:

$$x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$$

which can be represented in binary with the sequence $00000100110000010001110110110111_2$ and in hexadecimal with the number `0x04C1 1DB7`. It is adopted by many transmission and storage protocols, like Ethernet, Serial ATA, MPEG-2, BZip2 and PNG.

16.1.1 CRC Calculation in STM32F1/F2/F4/L1 MCUs

The long division of polynomial is suitable to perform manual calculations. However, another more efficient CRC algorithm is the polynomial division with the bitwise message XORing technique,

[4] http://bit.ly/293h2Hd

which is suited to be implemented with dedicated hardware circuitry: shift registers.

The process of CRC calculation in STM32 microcontrollers is related to the algorithm defined by the CRC-32 polynomial, which is the following one[5]:

- Initialize the CRC register with `0xFFFF FFFF` XORed with the *data value*.
- Shift in the input stream bit by bit. If the popped out MSB is a '1', XOR the CRC register value with the generator polynomial.
- If all input bits are handled, the CRC shift register contains the CRC value.

Assuming a *data value* equal to 10100101_2(0xAD), a CRC polynomial equal to 00010110_2 (0x16), the algorithm implemented by STM32 MCUs works in this way (**Figure 16.1** schematizes this process):

1. The initial content of CRC register is computed by XORing 0xFF and 0xAD.
2. Being the MSB bit 0, the CRC register is just left-shifted.
3. Now the MSB bit of CRC register is 1. We so first left-shift the register and then we perform XOR with the CRC polynomial (0x16).
4. Being the MSB bit 0, the CRC register is just left-shifted.
5. Now the MSB bit of CRC register is 1. We so first left-shift the register and then we perform XOR with the CRC polynomial (0x16).
6. Being the MSB bit 0, the CRC register is just left-shifted.
7. Now the MSB bit of CRC register is 1. We so first left-shift the register and then we perform XOR with the CRC polynomial (0x16).
8. Now the MSB bit of CRC register is 1. We so first left-shift the register and then we perform XOR with the CRC polynomial (0x16).
9. Finally, the MSB is again 0. We so perform a left-shift of the CRC register. The final value represents the *checksum* to prepend to the message.

The above algorithm is just a simplification of the actual one implemented in STM32 MCUs. In fact, it differs for two main reasons:

- The CRC polynomial is fixed and corresponds to the CRC-32 (`0x04C1 1DB7`).
- The single input/output data register is 32-bit wide, and the CRC checksum is computed on the whole 32-bit register and not byte-by-byte[6].

This dramatically limits the effective usability of this peripheral.

[5] As we will see next, STM32F0/F3/F7/L0/L4 MCUs use a slightly different and powerful CRC peripheral not limited to the CRC-32 polynomial.

[6] This is an important distinction compared to algorithms implemented in several libraries and on-line calculators, which usually perform CRC computation by splitting the word in sub-bytes. Refer to this post(http://bit.ly/29303sh) and this other one(http://bit.ly/293067u) on the official ST forum by the *clive1* user (the most active and experienced user on STM32 related sub-forum).

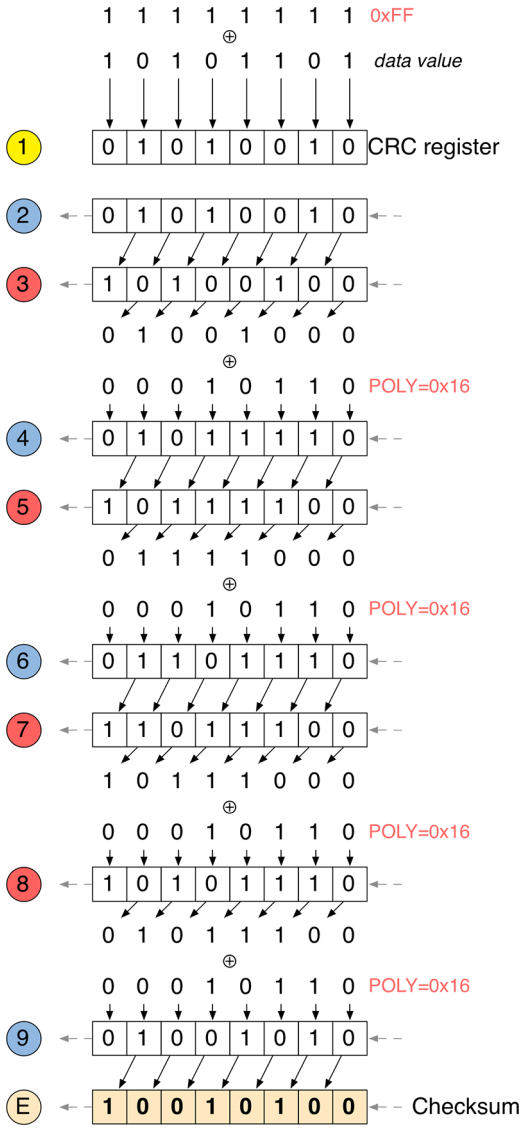


Figure 16.1: How CRC checksum is computed in an STM32

16.1.2 CRC Peripheral in STM32F0/F3/F7/L0/L4/L5/G0/G4 MCUs

In the previous paragraph we have seen that the STM32 peripheral provided by some STM32 MCUs is limited to the computation of the CRC using the CRC-32 Ethernet polynomial. Moreover, the handled data size for every computation is 32-bit.

In more recent STM32-series this limitation has been superseded. In fact, STM32F0/F3/F7/L0/L4/L5/G0/G4 MCUs provide a more advanced CRC peripheral, as shown in **Table 16.1**.

CRC Peripheral Feature	STM32F2/4	STM32L1	STM32F1	STM32F0/3	STM32G0/4	STM32F7	STM32L0	STM32L4/5
Single input/output 32-bit data register	YES							
General-purpose 8-bit register	YES							
Input buffer to avoid bus stall during calculation	NO			YES				
Reversibility option on I/O data	NO			YES				
CRC initial value	Fixed to 0xFFFF FFFF			Programmable on 32 bits	Programmable on 8, 16, 32 bits		Programmable on 32 bits	Programmable on 8, 16, 32 bits
Handled data size	32 bits				8, 16, 32 bits			
Polynomial size	32 bits				Programmable on 7, 8, 16, 32 bits			
Polynomial coefficients	Fixed to 0x4C11DB7				Programmable			

Table 16.1: Effective implementation of the CRC peripheral in STM32 MCUs

In these MCUs, the CRC peripheral is designed to be compatible by default with the simpler CRC peripheral provided by STM32F1/F2/F4/L1 MCUs. This means that, without an explicit configuration, the code designed to run on STM32F1/F2/F4/L1 MCUs will run on STM32F0/F3/F7/L0/L4 MCUs without any change.

16.2 HAL_CRC Module

The CubeHAL provides a dedicated module to manipulate CRC peripheral registers: the `HAL_CRC`. The CRC peripheral is referenced by using an instance of the `CRC_HandleTypeDef struct`. In STM32F1/F2/F4/L1 MCUs providing the simplest CRC peripheral, this `struct` is defined in the following way:

```
typedef struct {
  CRC_TypeDef                *Instance;  /* CRC registers base address   */
  HAL_LockTypeDef            Lock;       /* CRC locking object */
  __IO HAL_CRC_StateTypeDef  State;      /* CRC communication state */
} CRC_HandleTypeDef;
```

The only relevant field is the `Instance` one, which is the pointer to the CRC peripheral descriptor (whose base address is defined by the `CRC` macro).

Instead, in STM32F0/F3/F7/L0/L4 MCUs the `CRC_HandleTypeDef struct` is defined in the following way:

```
typedef struct {
  CRC_TypeDef                 *Instance;    /* Register base address        */
  CRC_InitTypeDef             Init;        /* CRC configuration parameters */
  HAL_LockTypeDef             Lock;        /* CRC Locking object           */
  __IO HAL_CRC_StateTypeDef   State;       /* CRC communication state      */
  uint32_t InputDataFormat;                /* Specifies input data format. */
} CRC_HandleTypeDef;
```

The only relevant difference is the existence of the `Init` field, which is used to configure the CRC peripheral as we will see in a while, and the `InputDataFormat` field, which specifies the data size of the input data: it can assume a value from **Table 16.2**.

Table 16.2: Input data formats for the CRC peripheral

Data format	Description
`CRC_INPUTDATA_FORMAT_BYTES`	Input data is a stream of bytes (8-bit data)
`CRC_INPUTDATA_FORMAT_HALFWORDS`	Input data is a stream of half-words (16-bit data)
`CRC_INPUTDATA_FORMAT_WORDS`	Input data is a stream of words (32-bits data)

To configure the CRC peripheral in those MCUs we use an instance of the `CRC_InitTypeDef struct`, which is defined in the following way:

```
typedef struct {
  uint8_t DefaultPolynomialUse;     /* Indicates if default polynomial is used */
  uint8_t DefaultInitValueUse;      /* Indicates if default init value is used */
  uint32_t GeneratingPolynomial;    /* Set CRC generating polynomial */
  uint32_t CRCLength;               /* Indicates CRC length */
  uint32_t InitValue;               /* Set the initial value to start CRC computation */
  uint32_t InputDataInversionMode;  /* Specifies input data inversion mode */
  uint32_t OutputDataInversionMode; /* Specifies output data (i.e. CRC) inversion mode */
} CRC_InitTypeDef;
```

Let us analyze the fields of this `struct`:

- `DefaultPolynomialUse`: this field indicates if the default polynomial (that is, the CRC-32) or a custom one is used. It can assume the values `DEFAULT_POLYNOMIAL_ENABLE` or `DEFAULT_POLYNOMIAL_DISABLE`. In this last case, the fields `GeneratingPolynomial` and `CRCLength` must be set.
- `DefaultInitValueUse`: this field indicates if the default CRC initialization value (that is, `0xFFFFFFFF`) or a custom one is used. It can assume the values `DEFAULT_INIT_VALUE_ENABLE` or `DEFAULT_INIT_VALUE_DISABLE`. In this last case, the field `InitValue` must be set.
- `GeneratingPolynomial`: sets CRC generating polynomial. 7, 8, 16 or 32-bit long value for a polynomial degree equal to 7, 8, 16 or 32. This field is written in normal representation, e.g., for a polynomial of degree 7, $X^7 + X^6 + X^5 + X^2 + 1$ is written 0x65.
- `CRCLength`: this field indicates the length of the CRC, and it can assume a value from **Table 16.3**.

- `InitValue`: sets the custom initial value to start CRC computation.
- `InputDataInversionMode`: specifies if the input data must be inverted or not. It can assume a value from **Table 16.4**.
- `OutputDataInversionMode`: specifies if the output data (the computed CRC) must be inverted or not. It can assume the values `CRC_OUTPUTDATA_INVERSION_DISABLE` and `CRC_OUTPUTDATA_INVERSION_ENABLE`. In this last case, the operation is done at bit level: for example, output data `0x1122 3344` is converted into `0x22CC 4488`.

Table 16.3: CRC length

CRC Length	Description
`CRC_POLYLENGTH_32B`	32-bit CRC
`CRC_POLYLENGTH_16B`	16-bit CRC
`CRC_POLYLENGTH_8B`	8-bit CRC
`CRC_POLYLENGTH_7B`	7-bit CRC

Table 16.4: Input data inversion mode

Inversion mode	Description
`CRC_INPUTDATA_INVERSION_NONE`	No input data inversion
`CRC_INPUTDATA_INVERSION_BYTE`	Byte-wise inversion, `0x1A2B 3C4D` becomes `0x58D4 3CB2`
`CRC_INPUTDATA_INVERSION_HALFWORD`	Halfword-wise inversion, `0x1A2B 3C4D` becomes `0xD458 B23C`
`CRC_INPUTDATA_INVERSION_WORD`	Word-wise inversion, `0x1A2B 3C4D` becomes `0xB23C D458`

Once an instance of the `CRC_InitTypeDef struct` is defined, and its fields are correctly populated, we configure the CRC peripheral by calling the function:

```
HAL_StatusTypeDef HAL_CRC_Init(CRC_HandleTypeDef *hcrc);
```

To compute the CRC checksum of a data buffer, we use the function:

```
uint32_t HAL_CRC_Calculate(CRC_HandleTypeDef *hcrc, uint32_t pBuffer[],
                                                    uint32_t BufferLength);
```

which accepts a pointer to an `uint32_t` array and its length. This function sets the default CRC initial value to `0xFFFF FFFF` or the specified value if we are working with an STM32F0/F3/F7/L0/L4 MCU. If, instead, we need to compute the CRC starting with the previous computed CRC as initial value, then we can use the function:

```
uint32_t HAL_CRC_Accumulate(CRC_HandleTypeDef *hcrc, uint32_t pBuffer[],
                                                     uint32_t BufferLength);
```

This is useful especially when we are computing CRC checksum of large block of data, using a temporary buffer having a size smaller than the source block.

17. IWDG and WWDG Timers

Anything that can go wrong, will go wrong, states the Murphy's Law[1]. And this is dramatically true especially for embedded systems. Apart from hardware faults, which can also impact on the software, even the most careful design may have some unexpected conditions that lead to abnormal behavior of our device. And this may have important costs, especially if the device is designed to work in dangerous and critical contexts.

Almost all embedded microcontrollers on the market provide a *WatchDog Timer* (WDT)[2]. A watchdog is usually implemented as a free-running and down-counter timer, which causes a reset of the MCU when it reaches zero or if the timer counter is not reloaded to its initial value within a well-defined temporal window (in this case we talk about *windowed watchdog*). Once enabled, the firmware needs to "constantly" refresh the watchdog counter register to its initial value, otherwise the MCU reset line is asserted low by the timer and a hard reset keeps going.

A smart management of the WDT can help us to handle all those unwanted situations that lead to faulty conditions of the embedded firmware (un-handled exceptions, stack overflows, access to invalid memory locations, corruption of the SRAM due to unstable power source, unconditional loops and so on). Moreover, if the WDT can warn us when the timer is running out, we can try to recover the regular activities of the firmware or, at least, to put the device in a safe state.

STM32 microcontrollers provide two independent watchdog timers: the *Independent Watchdog* (IWDG) and the System *Window Watchdog* (WWDG). They share almost the same characteristics, except for a couple of them that make one timer more suitable than the other in some specific applications. This chapter shows how to use the CubeHAL to take advantage of these two important, and often underused, peripherals.

17.1 The *Independent Watchdog* Timer

The IWDG is a 12-bit down-counter timer clocked by the *Low-Speed Internal* (LSI) oscillator: this explains the adjective *independent*, meaning that this peripheral is not fed by the peripheral clock, which is in turn generated by HSI or HSE oscillators. This is an important characteristic, which allows to the IWDG timer to work even if the main clock fails: the watchdog keeps counting even if the CPU is halted, and it will reset the MCU when reaches zero. If the firmware is properly designed to address this issue, the MCU can recover from a faulted clock using the internal oscillator (HSE).

In STM32F0/F3/F7/L0/L4/L5/G0/G4 families, this timer can optionally work in *windowed* mode. This means that we can setup a temporal window (ranging from 0x0 up to 0xFFF) that establishes when

[1] This author prefers the less famous Smith's Law, which states: *Murphy was an optimist.*

[2] There exist really low-cost MCUs that do not provide this feature, or which implement it in an unreliable way, requiring to adopt external and dedicated ICs.

it is ok to refresh the timer counter. If we refresh the timer before the counter reaches the window value, then the MCU is reset in the same way when the timer reaches zero. This allows to ensure that "things" are moving in the right way, especially when the MCU accomplishes repetitive tasks that work in a well-defined temporal window.

How long does the IWDG timer take before the MCU is reset? The following formula establishes the update event of the IWDG timer:

$$UpdateEvent_{IWDG} = \frac{(2^{IWDG_{PSC}})(Period + 1)}{LSI_{clock}} \quad [1]$$

where *Period* ranges from 0 up to 4095 and $IWDG_{PSC}$ corresponds to a dedicate 3-bit prescaler that ranges from 2 to 8. For example, assuming an LSI clock running at 32kHz, a *Period* equal to 0xFFF and a $IWDG_{PSC}$ equal to 2, we have that the IWDG timer will underflow after:

$$UpdateEvent_{IWDG} = \frac{(2^2)(4096)}{32000} \approx 0.5s$$

The IWDG timer also supports the *hardware watchdog* feature. A special bit of the *option bytes* region, an area of the flash memory that we will study in Chapter 21, configures the timer so that it automatically starts counting after every system reset.

Different from regular timers, and from other microcontroller architectures, once an STM32 watchdog timer is started there is no way to stop it. This is an important constraint to keep in mind while developing low-power applications[3].

17.1.1 Using the CubeHAL to Program IWDG Timer

To manipulate the IWDG peripheral, the HAL defines the C `struct IWDG_HandleTypeDef`, which is defined in the following way:

```
typedef struct {
  IWDG_TypeDef                *Instance; /* Pointer to IWDG descriptor */
  IWDG_InitTypeDef            Init;      /* IWDG initialization parameters */
  HAL_LockTypeDef             Lock;      /* IWDG locking object      */
  __IO HAL_IWDG_StateTypeDef  State;     /* IWDG communication state */
} IWDG_HandleTypeDef;
```

To configure the IWDG peripheral we use an instance of the C `struct IWDG_InitTypeDef`, which is defined in the following way:

[3] In Chapter 19 about power management, we will see how to address this limitation.

```
typedef struct {
  uint32_t Prescaler; /* Selects the prescaler of the IWDG */
  uint32_t Reload;    /* Specifies the IWDG down-counter reload value */
  uint32_t Window;    /* Specifies the window value to be compared to the down-counter */
} IWDG_InitTypeDef;
```

Let us study the fields of this C `struct`.

- `Prescaler`: this field specifies the prescaler value, and it can assume all powers of two ranging from 2^2 up to 2^8. To specify this value, the CubeHAL defines seven different macros - `IWDG_PRESCALER_4`, `IWDG_PRESCALER_8`, ..., `IWDG_PRESCALER_256`.
- `Reload`: specifies the timer period, that is the auto-reload value when the timer is refreshed. It can range from 0x0 up to 0xFFF (the default value).
- `Window`: for those STM32 MCUs providing a *windowed* IWDG, this field sets the corresponding window value within which it is allowed to refresh the timer. It can range from 0x0 up to 0xFFF (the default value).

To configure and to start the IWDG timer, we use the CubeHAL function:

```
HAL_StatusTypeDef HAL_IWDG_Init(IWDG_HandleTypeDef *hiwdg);
```

while to refresh it before it reaches zero, we use the function:

```
HAL_StatusTypeDef HAL_IWDG_Refresh(IWDG_HandleTypeDef *hiwdg);
```

17.2 The *System Window Watchdog* Timer

The WWDG is a 7-bit down-counter timer clocked by the APB clock. Different from the IWDG timer, the WWDG one is designed to be refreshed within a given temporal window, otherwise it triggers the MCU reset. The way the WWDG timer works may seem a little bit counterintuitive for newcomers. Let us explain the way it works step-by-step.

The WWDG is a 7-bit timer (see **Figure 17.1**). Its counter register can be set from 0x7F down to 0x40. This value will be used to reload the counter register upon refresh (we are going to call this value T_S).

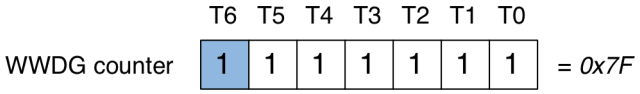


Figure 17.1: The content of the WWDG counter register upon reset

The WWDG timer has this characteristic: when the counter's 7th bit (T6 in **Figure 17.1**) switches from 1 to 0, a system reset takes place. This means that when the counter reaches the value $T_E = 0x3F$, which corresponds to 0111111_2, the MCU is reset.

The WWDG is fed by the APB bus main clock. The clock is prescaled by a fixed factor (4096) plus a programmable one, according to the following formula:

$$WWDG_{PSC} = 4096 \cdot 2^i \qquad where \quad 0 \leq i \leq 3$$

For example, assuming a $WWDG_{PSC} = 4096 \cdot 8$ and an APB clock of 48MHz, we have that the counter is decremented by 1 every 682.6µS.

As said before, the WWDG timer can be refreshed only within a given temporal window: this programmable value can range from T_S down to 0x40: the closer to T_S is, the wider is the window. For example, if we configure the window register with the value $T_W = 0x5F$, we can refresh the WWDG timer only when its counter goes from 0x5F down to 0x40. The **Figure 17.2** clearly shows the role of the temporal window. If we try to refresh the WWDG timer in those greyed regions, that is between 0x7F and 0x60, or when the counter goes below 0x3F, then the MCU will be reset.

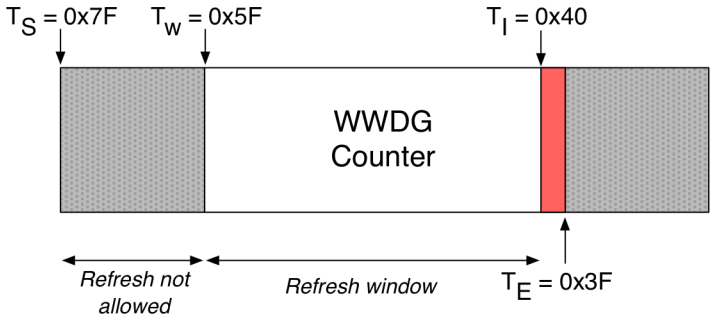


Figure 17.2: How the temporal window defines the counter interval within which it is allowed to refresh the WWDG timer

How long does the temporal window last? This is defined by the following formula:

$$WWDG_{Window} = \frac{WWDG_{PSC} \cdot (Period + 1)}{APB_{clock}} \quad [2]$$

where

$$Period = T_S - T_W \quad [3]$$

For example, let us suppose to set the counter refresh value (that is, T_S) to 0x7F and the window value (that is, T_W) to 0x5F. Moreover, let us assume an APB clock equal to 48MHz and a programmable prescaler factor equal to 8. We have that:

$$WWDG_{W_{MIN}} = \frac{4096 \cdot (2^3) \cdot (0x20 + 1)}{48000000} \approx 22.5ms$$

This represents the minimum timeout we have to wait before we can refresh the WWDG counter. The maximum timeout, instead, is represented by the lower and fixed value 0x40. Using again [2], we have that:

$$WWDG_{W_{MAX}} = \frac{4096 \cdot (2^3) \cdot (0x3F + 1)}{48000000} \approx 43.6ms$$

This means that refreshing the WWDG timer before 22.5ms or after 43.6ms since the last refresh will cause a system reset.

WWDG has another important characteristic: when the counter reaches the value $T_I = 0x40$, just one "tick" before the 0x3F value that will cause the MCU reset, a dedicate IRQ fires, if enabled. This interrupt, called *Early Wakeup Interrupt* (EWI), can be used to eventually refresh the WWDG timer *in extremis*, or to place the device in a safe state. The dedicated ISR is called `WWDG_IRQHandler()`, and it is the first ISR after the fifteen Cortex-M exceptions.

Finally, even the WWDG supports the *hardware watchdog* feature like the IWDG timer.

17.2.1 Using the CubeHAL to Program WWDG Timer

To manipulate the WWDG peripheral, the HAL defines the C `struct WWDG_HandleTypeDef`, which is defined in the following way:

```
typedef struct {
  WWDG_TypeDef                 *Instance; /* Pointer to WWDG descriptor */
  WWDG_InitTypeDef             Init;      /* WWDG initialization parameters */
  HAL_LockTypeDef              Lock;      /* WWDG locking object      */
  __IO HAL_WWDG_StateTypeDef   State;     /* WWDG communication state */
} WWDG_HandleTypeDef;
```

To configure the WWDG peripheral we use an instance of the C `struct WWDG_InitTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t Prescaler; /* Select the prescaler of the WWDG */
  uint32_t Window;    /* Specifies the window value to be compared to the down-counter */
  uint32_t Counter;   /* Specifies the WWDG down-counter reload value */
  uint32_t EWIMode;   /* Specifies if WWDG Early Wakeup Interupt is enable or not.
                         This parameter can be a value of @ref WWDG_EWI_Mode */
} WWDG_InitTypeDef;
```

Let us study the fields of this C `struct`.

- `Prescaler`: this field specifies the prescaler value, and it can range from all powers of two between 1 and 8. To specify this value, the CubeHAL defines four different macros - `WWDG_PRESCALER_1`, `WWDG_PRESCALER_2`, ..., `WWDG_PRESCALER_8`.

- `Window`: this field sets the corresponding window value within which it is allowed to refresh the timer. It can range from the value of the `Counter` field (the default one) down to 0x3F.
- `Counter`: specifies the timer period, that is the reload value when the timer is refreshed. It can range from 0x7F (the default value) down to 0x3F.
- `EWIMode`: this fields enables the *Early Wakeup Interrupt* (EWI) and it can assume the values `WWDG_EWI_ENABLE` and `WWDG_EWI_DISABLE`.

To configure and to start the WWDG timer, we use the CubeHAL function:

```
HAL_StatusTypeDef HAL_WWDG_Init(WWDG_HandleTypeDef *hwwdg);
```

When the WWDG timer EWI mode is enabled, we have to implement the `WWDG_IRQHandler()` ISR and to place a call to the function:

```
void HAL_WWDG_IRQHandler(WWDG_HandleTypeDef *hwwdg);
```

The proper way to be notified when the interrupt fires consists in implementing the callback routine:

```
void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef* hwwdg);
```

To refresh the WWDG timer within the temporal window, we use the function:

```
HAL_StatusTypeDef HAL_WWDG_Refresh(WWDG_HandleTypeDef *hwwdg, uint32_t Counter);
```

where the `Counter` parameter corresponds to the value to reload inside the WWDG counter register.

Finally, being the WWDT timer clocked by the APB clock, we need to enable the peripheral clock by using the macro `__HAL_RCC_WWDG_CLK_ENABLE()`.

17.3 Detecting a System Reset Caused by a Watchdog Timer

It could be useful to detect when a system reset is caused by the expiring of a watchdog timer. This may help us understanding what is going wrong during a debug session. Two special bits in a register of the *Reset and Clock Control* (RCC) peripheral allows to detect this event.

To detect if a reset has been caused by the IWDG timer, we can check the corresponding flag using the following macro:

```
__HAL_RCC_GET_FLAG(RCC_FLAG_IWDGRST);
```

while for the WWDG timer we can check this other flag:

```
__HAL_RCC_GET_FLAG(RCC_FLAG_WWDGRST));
```

17.4 Freezing Watchdog Timers During a Debug Session

During a debug session, both WWDG and IWDG timers will keep counting. This will prevent us to carry out a step-by-step debugging. We can configure debug interface so that it halts watchdog timers when the MCU is halted using the following macros:

```
__HAL_DBGMCU_FREEZE_IWDG();
__HAL_DBGMCU_FREEZE_WWDG();
```

17.5 Selecting the Right Watchdog Timer for Your Application

Both the watchdog timers have similar functionalities, and both do the same thing: to reset the MCU if we do not refresh their counter register in a given amount of time. But when it is best to prefer a timer over to other?

The IWDG timer is to prefer when we need to be sure that the main clock is working. Being the IWDG clocked by the independent LSI, it is useful to detect such malfunctions. Moreover, if we are using an RTOS, we can setup an independent thread configured with the maximum priority and that uses a software timer to refresh the IWDG timer periodically. This also helps us understanding that the kernel is properly scheduling threads.

The WWDG timer must be preferred to the IWDG one when we must be sure that some operations are carried out in a fixed and well-characterized temporal window. If that procedure takes less or more time, it will not be able to refresh the timer in the temporal window, causing a system reset. Moreover, the WWDG is the right choice if we want to perform critical operations (like putting the machine in a safe state or saving special data in non-volatile memory): thanks to the early-warning IRQ, we can get notified of the ongoing system reset.

18. Real-Time Clock

There exist a significant number of embedded applications that need to keep track of the current time and date. Data-loggers, timers, home appliances and control devices are just a limited example. Traditionally, microcontrollers are interfaced with dedicated ICs, which are able to communicate using the SPI or the I²C bus. For example, the same ST Microelectronics sells the M41T81[1] IC, a popular *Real-Time Clock* (RTC) that requires a couple of passives and a 32kHz oscillator to keep track of the current time. Moreover, the same IC is also able to generate alarm events and to act as a watchdog timer.

All STM32 microcontrollers provide an integrated RTC unit that is not limited to keeping track of the current date/time. In fact, RTC provides some additional and relevant features such as anti-tampering detection, generation of alarm events and the ability to wake-up the MCU from deeper *low-power* modes. This chapter shows how to program this peripheral using the related CubeHAL module.

18.1 Introduction to the RTC Peripheral

The STM32 RTC is an independent *Binary Coded Decimal* (BCD) counter. BCD is one type of binary encoding where each digit of a decimal number is represented independently by a fixed number of bits. For example, the RTC timer represents the current hour in the following way:

- two bits are used to encode the hour tens;
- four bits are used to encode the hour units;
- three bits are used to encode the minute tens;
- four bits are used to encode the minute units.

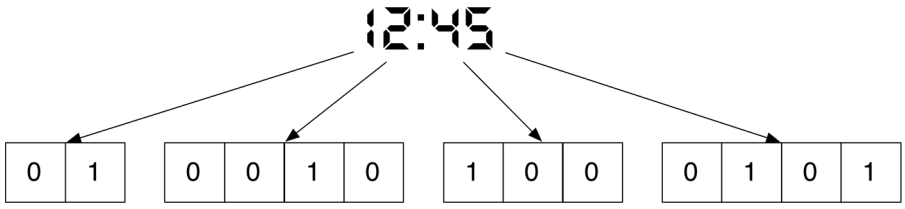


Figure 18.1: How the time is encoded in BCD format in an STM32 MCU

Figure 18.1 shows how the STM32 RTC encodes the current hour in BCD format. Why to use this approach to encode the date/time? This way to keep track of current date/time is typical of small embedded systems and it allows to represent the time in human-readable format without

[1] http://bit.ly/2fj3vCM

performing any type of conversion. Traditionally, high-level Operating Systems keep track of the time using an `unsigned long` variable, which is automatically incremented at every second. For example, the UNIX time is represented with the number of seconds elapsed since the *Epoch*, which corresponds to the 00:00:00, Thursday, January 1st, 1970. However, a lot of CPU power and firmware room is required to convert the number of seconds elapsed since that date to the current date/time. Conversion routines need to keep track of several factors, like how many days are in a month, leap years and seconds, and so on. BCD encoding allows to immediately arrange current date/time in a way that is understandable for a lot of people on the earth, at the cost of a more complex internal circuitry.

The STM32 RTC peripheral allows to easily configure and to display the calendar data fields:

- Calendar with:
  - sub-seconds (not programmable)
  - seconds
  - minutes
  - hours in 12-hour or 24-hour format
  - day of the week (day)
  - day of the month (date)
  - month
  - year
- Automatic management of 28-, 29- (leap year), 30-, and 31-day months
- Daylight saving time adjustment programmable by software

Different from the majority of STM32 peripherals, the RTC can be clocked independently from three distinct clock sources: LSI, LSE and HSE. A series of dedicated prescalers allows delivering a 1Hz clock to calendar unit, regardless of the clock source. When the clock source for the RTC (RTCCLK) is the HSE, it is user responsibility to properly configure the prescalers so that the right clock frequency can fed the RTC. However, CubeMX is designed to handle that automatically according to the specified HSE crystal frequency.

Even if the RTC provides tools to correct imprecisions of the clock, as we will see later, not all clock sources are suitable to achieve a good precision of the RTC, especially if the MCU works at temperatures different from the ambient one. If precision is important for your application, then it is strongly suggested to use a dedicated external LSE crystal, tuned according to the crystal specifications and PCB layout.

The RTC functionalities are not limited to the time/date management. The RTC provides two independent alarm units, named **Alarm A** and **Alarm B**, which can be used to generate events when the RTC counter reaches the configured alarm value. Alarm units are highly customizable: sub-second, seconds, minutes, hours and date fields can be independently selected or masked to provide a rich combination of alarms. Together with the two alarm units, the RTC provides an independent, programmable and dedicated wakeup unit used to wake up the MCU from deeper *sleep* states. In

fact, in the next chapter we will see that the RTC is the only peripheral able to wake up the MCU from the *standby* sleep state on a programmable basis.

Finally, the RTC provides the ability to sample a number of given inputs to detect tampering: since the RTC peripheral can be powered by a battery[2] in several STM32 microcontrollers, it is also able to detect tampering even if the device is powered off. On tamper detection, a particular register is set, and the content of the *backup memory* is also zeroed.

18.2 `HAL_RTC` Module

To program the RTC peripheral, the HAL defines the C `struct RTC_HandleTypeDef`, which is defined in the following way:

```
typedef struct {
  RTC_TypeDef                 *Instance;  /* Register base address    */
  RTC_InitTypeDef             Init;       /* RTC required parameters  */
  HAL_LockTypeDef             Lock;       /* RTC locking object       */
  __IO HAL_RTCStateTypeDef    State;      /* Time communication state */
} RTC_HandleTypeDef;
```

The only notably fields of this `struct` are `Instance`, which is the pointer to the RTC peripheral descriptor, and the `Init` field used to configure the peripheral. This field is an instance of the C `struct RTC_InitTypeDef`, which is defined in the following way:

```
typedef struct {
  uint32_t HourFormat;      /* Specifies the RTC Hour Format. */
  uint32_t AsynchPrediv;    /* Specifies the RTC Asynchronous Predivider value. */
  uint32_t SynchPrediv;     /* Specifies the RTC Synchronous Predivider value. */
  uint32_t OutPut;          /* Specifies which signal will be routed to the RTC output. */
  uint32_t OutPutPolarity;  /* Specifies the polarity of the output signal. */
  uint32_t OutPutType;      /* Specifies the RTC Output Pin mode. */
} RTC_InitTypeDef;
```

- `HourFormat`: this field specifies the hour format, and it can assume the values `RTC_HOURFORMAT_12` to setup the AM/PM hour format ant the `RTC_HOURFORMAT_24` to specify the 24 hour/day format.
- `AsynchPrediv` and `SynchPrediv`: two prescalers are used to derive the 1Hz clock to feed the RTC peripheral from the LSI/LSE/HSE oscillator sources. The first one is the *asynchronous prescaler*, a 7-bit counter that feeds in turn the *synchronous prescaler*, another 15-bit counter. The values of these two fields must be set so that the 1Hz frequency is reached according to

[2] In the next chapter we will see that STM32 MCUs with high pin count provide several independent power domains. The RTC belongs to the VBAT domain, that is the set of all peripherals that are powered through the VBAT pin. This domain is especially designed to be tied to a battery, and all peripherals belonging to this domain keep working even when the main power, and hence the MCU core, is OFF.

equation [1], where $Calendar_{CLK}$ is one of LSI/LSE/HSE. At the time of writing this chapter, the latest CubeMX release (4.22) is not able to automatically derive the proper values for the `AsynchPrediv` and `SynchPrediv` fields. You can use the values reported in **Table 18.1** for most relevant oscillator frequencies.

$$Calendar_{CLK} = \frac{RTCCLK}{(\texttt{AsynchPrediv} + 1)(\texttt{SynchPrediv} + 1)} \quad [1]$$

Table 18.1: Correct values for the `AsynchPrediv` and `SynchPrediv` fields according to the most common clock sources

$Calendar_{CLK}$	**AsynchPrediv**	**SynchPrediv**
HSE_RTC = 1MHz	124	7999
LSE = 32.768kHz	127	255
LSI = 32kHz	127	249
LSI = 37kHz	127	295

- `OutPut`: specifies the signal I/O routed to the RTC output. It can assume the values `RTC_OUTPUT_ALARMA`, `RTC_OUTPUT_ALARMB`, `RTC_OUTPUT_WAKEUP` and `RTC_OUTPUT_DISABLE` to route the output to the signal related to Alarm A, B, Wakeup or to disable the output signal. Please, take note that the actual GPIO associated to a given alarm is designed during the MCU development and it is fixed. Depending on the type of package used, just one signal I/O may be available and shared between the three alarm sources. For example, all STM32 MCU with LQFP-64 package have just one alarm I/O named AF1 and connected to the PC13 pin.
- `OutPutPolarity`: this field specifies the output polarity of the signal, and it can assume the values `RTC_OUTPUT_POLARITY_HIGH` and `RTC_OUTPUT_POLARITY_LOW`.
- `OutPutType`: this field specifies the type of the output signal, and it can assume the values `RTC_OUTPUT_TYPE_OPENDRAIN` and `RTC_OUTPUT_TYPE_PUSHPULL`.

As usual, to configure the RTC peripheral we use the function:

```
HAL_StatusTypeDef HAL_RTC_Init(RTC_HandleTypeDef *hrtc);
```

which accepts a pointer to an instance of the `RTC_HandleTypeDef struct` seen before.

18.2.1 Setting and Retrieving the Current Date/Time

The CubeHAL implements separated routines and C structs to set and retrieve the current date and time. The functions:

```
HAL_StatusTypeDef HAL_RTC_SetTime(RTC_HandleTypeDef *hrtc,
                                  RTC_TimeTypeDef *sTime, uint32_t Format);
HAL_StatusTypeDef HAL_RTC_GetTime(RTC_HandleTypeDef *hrtc,
                                  RTC_TimeTypeDef *sTime, uint32_t Format);
```

are used to set/get the current time, while the functions:

```
HAL_StatusTypeDef HAL_RTC_SetDate(RTC_HandleTypeDef *hrtc,
                                  RTC_DateTypeDef *sDate, uint32_t Format);
HAL_StatusTypeDef HAL_RTC_GetDate(RTC_HandleTypeDef *hrtc,
                                  RTC_DateTypeDef *sDate, uint32_t Format);
```

are used to set/get the current date.

The `RTC_TimeTypeDef struct`, used to set/get the current time, is defined in the following way:

```
typedef struct {
  uint8_t Hours;           /* Specifies the RTC Time Hour.
                              This parameter must be a number between 0 and 12 if the
                              12 hours format is selected. Otherwise, it must be a
                              number between 0 and 23 if the 24 hours format is selected */
  uint8_t Minutes;         /* Specifies the RTC Time Minutes.
                              This parameter must be a number between 0 and 59 */
  uint8_t Seconds;         /* Specifies the RTC Time Seconds.
                              This parameter must be a number 0 and 59 */
  uint8_t TimeFormat;      /* Specifies the RTC AM/PM Time. */
  uint32_t SubSeconds;     /* Specifies the RTC_SSR RTC Sub Second register content.
                              Not used when setting the timer */
  uint32_t SecondFraction; /* Specifies the range or granularity of Sub Second register */
  uint32_t DayLightSaving; /* Specifies DayLight Save Operation. */
  uint32_t StoreOperation; /* Specifies Store Operation value  */
} RTC_TimeTypeDef;
```

Let us analyze the role of the most important fields:

- `Hours`, `Minutes`, `Seconds`: these fields are used to set the current time.
- `TimeFormat`: it is used to set the time format (12/24 hours) and it can assume the values `RTC_HOURFORMAT_12` or `RTC_HOURFORMAT_24`.
- `SubSeconds`: when the `struct RTC_TimeTypeDef` is populated by the `HAL_RTC_GetTime()` routine, this field contains the current sub-second value. It is ignored by the `HAL_RTC_SetTime()` routine. This field corresponds to a time unit range between [0-1] second, with granularity equal to 1s/(`SecondFraction`+1).
- `SecondFraction`: specifies the granularity of `SubSeconds` field, and it corresponds to Synchronous prescaler factor value. This field will be used only by `HAL_RTC_GetTime()` function.
- `DayLightSaving`: this field specifies the DayLight saving and it can assume the values `RTC_DAYLIGHTSAVING_SUB1H`, `RTC_DAYLIGHTSAVING_ADD1H`, `RTC_DAYLIGHTSAVING_NONE`.

The `RTC_DateTypeDef struct`, used to set/get the current date, is defined in the following way:

```
typedef struct {
  uint8_t WeekDay;  /* Specifies the RTC Date WeekDay. */
  uint8_t Month;    /* Specifies the RTC Date Month (in BCD format). */
  uint8_t Date;     /* Specifies the RTC Date.  */
  uint8_t Year;     /* Specifies the RTC Date Year. */
} RTC_DateTypeDef;
```

All the four time/date related functions accept as last parameter the format of time/date related fields. This parameter can assume the values `RTC_FORMAT_BIN` and `RTC_FORMAT_BCD`. If the `RTC_FORMAT_BIN` constant is passed, then the time/date related fields are expressed in conventional binary format. For example, the time "12:45" is expressed as is. If, instead, the `RTC_FORMAT_BCD` constant is passed, then the values are expressed in BCD. This means that every time/date related field (which occupies one byte) must be interpreted as two sub-nibbles, which correspond to the digits of a decimal number. So, following the same previous example, we have that the decimal number "12" is expressed as 18_{10} in binary format, which corresponds to 12 in the hexadecimal representation (see **Figure 18.2**).

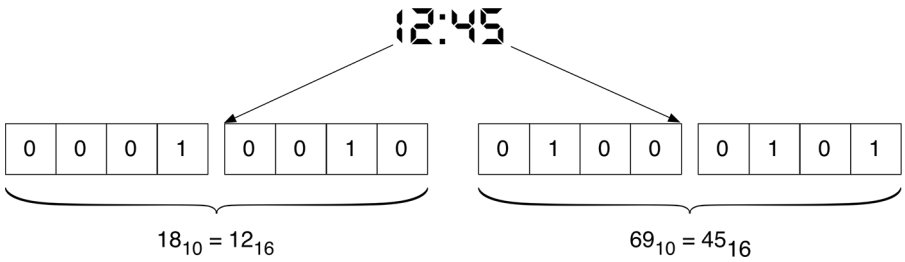


Figure 18.2: How the time encoded in BCD format is returned by the

18.2.1.1 Correct Way to Read Date/Time Values

The current date/time value cannot be read freely, but there is a well-defined procedure to follow. This because, by default, we do not directly access to RTC internal date/time registers. The RTC is a peripheral that runs by its own and that it is not clocked through the APB bus. When the code reads the calendar fields, it accesses to *shadow registers* that contain a copy of the real calendar time and date clocked by the RTC clock (RTCCLK). The copy is performed every two RTCCLK cycles, synchronized with the system clock (SYSCLK). Moreover, we must call the `HAL_RTC_GetDate()` after `HAL_RTC_GetTime()` even if we are not interested to the current date. This because the call to the `HAL_RTC_GetDate()` unlocks the values in the higher-order calendar shadow registers to ensure consistency between the time and date values. Reading RTC current time locks the values in calendar shadow registers until current date is read. This is a really frequent error made by novices of the STM32 platform: when accessing the time-related fields using the `HAL_RTC_GetTime()` we receive the last transferred time unless we read the content of the date-related fields in the corresponding shadow register.

After a system reset or after exiting *low-power* modes, the application must wait the synchronization between the RTC internal and shadow registers, before reading the calendar shadow registers. To perform this operation, the CubeHAL provides the function:

```
HAL_StatusTypeDef HAL_RTC_WaitForSynchro(RTC_HandleTypeDef* hrtc);
```

However, calling this function is required if and only if we want to access to shadow registers immediately after a system reset or a wake up from a *low-power* mode, when the SYSCLK speed is still at its minimum frequency (since it is fed by the HSI). If the HCLK speed is at least eight time faster than the RTCCLK, then the synchronization of shadow registers happens in a few clock cycles. When using the `HAL_RTC_WaitForSynchro()` routine, it is important to keep in mind that the access in write mode to the so called *backup domain* (which include the RTC peripheral) is disabled by default, to prevent corruption of peripheral registers due to an unstable power source. However, the `HAL_RTC_WaitForSynchro()` routine needs to access in write mode to RTC registers, and so we need to enable the access in write mode to the *backup domain* by using the macro `__HAL_RTC_WRITEPROTECTION_DISABLE()`, as shown below:

```
1 /* Disable the write-protection */
2 __HAL_RTC_WRITEPROTECTION_DISABLE(&hrtc);
3 /* Wait until the shadow registers are synchronized */
4 HAL_RTC_WaitForSynchro(&hrtc);
5 /* Enable again the write-protection to prevent registers corruption */
6 __HAL_RTC_WRITEPROTECTION_ENABLE(&hrtc);
```

Finally, it is possible to bypass the access to shadow registers. In this case, it is not mandatory to wait for the synchronization time, but the calendar registers consistency must be checked by the software. The user must read the required calendar field values twice. The results of the two read sequences are then compared. If the results match, the read result is correct. If they do not match, the fields must be read one more time, and the third read result is valid. To bypass the shadow registers, the CubeHAL provides the function:

```
HAL_StatusTypeDef HAL_RTCEx_EnableBypassShadow(RTC_HandleTypeDef* hrtc);
```

To re-enable shadow registers access again, we can use the function:

```
HAL_StatusTypeDef HAL_RTCEx_DisableBypassShadow(RTC_HandleTypeDef* hrtc);
```

18.2.2 Configuring Alarms

STM32 RTC provides two alarms, named **Alarm A** and **Alarm B**, which have the same functionalities. An alarm can be generated at a given time or/and date programmed by the user. To setup an alarm we use the function:

```
HAL_StatusTypeDef HAL_RTC_SetAlarm(RTC_HandleTypeDef *hrtc,
                                   RTC_AlarmTypeDef *sAlarm, uint32_t Format);
```

We can eventually poll an alarm until the event has occurred by using the function:

```
HAL_StatusTypeDef HAL_RTC_PollForAlarmAEvent(RTC_HandleTypeDef *hrtc, uint32_t Timeout);
```

An alarm can be configured so that it asserts a dedicated interrupt when it fires. The IRQ associated to both the alarms is the `RTC_Alarm_IRQn`, and to configure an alarm in interrupt mode we can use the following dedicated routine:

```
HAL_StatusTypeDef HAL_RTC_SetAlarm_IT(RTC_HandleTypeDef *hrtc,
                                      RTC_AlarmTypeDef *sAlarm, uint32_t Format);
```

Like all CubeHAL interrupt handler routines, we need to invoke the `HAL_RTC_AlarmIRQHandler()` from the `RTC_Alarm_IRQn` ISR. To be notified from the alarm event, we can implement the corresponding callback:

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
```

An alarm can be deactivated by using the function:

```
HAL_StatusTypeDef HAL_RTC_DeactivateAlarm(RTC_HandleTypeDef *hrtc, uint32_t Alarm);
```

The `struct RTC_AlarmTypeDef`, used to setup an alarm, is defined in the following way:

```
typedef struct {
  RTC_TimeTypeDef AlarmTime;     /* Specifies the RTC Alarm Time members */
  uint32_t AlarmMask;            /* Specifies the RTC Alarm Masks. */
  uint32_t AlarmSubSecondMask;   /* Specifies the RTC Alarm SubSeconds Masks. */
  uint32_t AlarmDateWeekDaySel; /* Specifies the RTC Alarm is on Date or WeekDay. */
  uint8_t AlarmDateWeekDay;      /* Specifies the RTC Alarm Date/WeekDay. */
  uint32_t Alarm;                /* Specifies the alarm (A or B). */
} RTC_AlarmTypeDef;
```

- `AlarmTime`: this field is an instance of the `RTC_TimeTypeDef struct` seen before, and it is used to setup the alarm time.
- `AlarmMask`: an alarm consists of a register with the same length as the RTC time counter. When the RTC counter matches the value configured in the alarm register, it generates an event. The `AlarmMask` field defines the comparison criteria between the alarm and the RTC time register. It can assume one or more values (by bit-masking them) from those reported in **Table 18.2**. For example, if we want that the alarm occurs at 12:45:03, we use the `RTC_ALARMMASK_NONE` value. If, instead, we want to generate an alarm every hour, at a given minute and second, we can use the value `RTC_ALARMMASK_HOURS`.

Table 18.2: Available alarm masks to set up the alarm behaviour

Mask value	Alarm behaviour
RTC_ALARMMASK_NONE	All fields are used in alarm comparison (e.g., alarm occurs at 12:45:03)
RTC_ALARMMASK_SECONDS	Seconds do not matter in alarm comparison(e.g., alarm occurs at every seconds of 12:45)
RTC_ALARMMASK_SECONDS	Seconds do not matter in alarm comparison(e.g., alarm occurs at every seconds of 12:45)
RTC_ALARMMASK_MINUTES	Minutes do not matter in alarm comparison (e.g., alarm occurs at the 3rd second of every minute of 12:XX)
RTC_ALARMMASK_HOURS	Hours do not matter in alarm comparison (e.g., alarm occurs at the 3rd second of every 45th minute)
RTC_ALARMMASK_DATEWEEKDAY	Week day (or date, if selected) do not matter in alarm comparison (e.g., alarm occurs all days at 12:45:03)
RTC_ALARMMASK_ALL	Alarm occurs every second

- `AlarmDateWeekDaySel`: specifies if the alarm is set on a date (day of the month) or on a weekday (monday, tuesday, etc.). It can assume the value `RTC_ALARMDATEWEEKDAYSEL_DATE` or `RTC_ALARMDATEWEEKDAYSEL_WEEKDAY`.
- `AlarmDateWeekDay`: if the `AlarmDateWeekDaySel` field is set to `RTC_ALARMDATEWEEKDAYSEL_DATE`, then this field must be set to a value in the 1-31 range. Instead, if the `AlarmDateWeekDaySel` field is set to `RTC_ALARMDATEWEEKDAYSEL_WEEKDAY`, then this field must be set to symbolic constants `RTC_WEEKDAY_MONDAY`, `RTC_WEEKDAY_TUESDAY` and so on.
- `AlarmSubSecondMask`: the sub-seconds register of the RTC time can be used to generate events with granularity lower than the second. By masking individual bits of the sub-seconds register it is possible to generate events every 1/128s, 1/64s, and so on. For more information about the masking possibilities, and their effect on the alarm behaviour, refer to the official AN3371 from ST[3]. This functionality allows, for example, to use the RTC as timebase generator for the HAL. ST provides a such example in the CubeHAL projects. Refer to them for more about this.
- `Alarm`: it specifies the configured alarm, and it can assume the values `RTC_ALARM_A` and `RTC_ALARM_B`.

18.2.3 Periodic Wakeup Unit

In the next chapter we will see that STM32 microcontrollers provide the ability to selectively disable internal functionalities in order to reduce the power consumption. Several *low-power* modes give to programmers the possibility to decide the power consumption level that best fits his needs, especially when developing battery-powered devices.

The STM32 RTC features a periodic timebase and wakeup unit that can wakeup the system when the microcontroller operates in *low-power* modes. This unit is a programmable 16-bit down-counting and auto-reload timer. When this counter reaches zero, a flag is set and an interrupt (if enabled) is generated. The wakeup unit has the following features:

[3] http://bit.ly/2fcR1uE

- Programmable down-counting auto-reload timer.
- Specific flag and interrupt able of waking up the device from low power modes.
- Wakeup alternate function output that can be routed to RTC alarm output (the output is shared between Alarm A, Alarm B or Wakeup unit) with configurable polarity.
- A full set of prescalers to select the desired waiting period.

The wakeup counter counting frequency can be derived either by the RTCCLK source, and eventually further prescaled, or by the calendar clock (that is, after the *asynchronous* and *synchronous* prescalers). This gives the possibility to generate wakeup events with a frequency ranging from 122µs up to more than 48 days when an external clock is chosen for the LSE oscillator.

To setup a wakeup event, the CubeHAL provides the function:

```
HAL_StatusTypeDef HAL_RTCEx_SetWakeUpTimer(RTC_HandleTypeDef *hrtc,
                                           uint32_t WakeUpCounter, uint32_t WakeUpClock);
```

where the `WakeUpCounter` parameter sets the autoreload value (that is, the period) of the wakeup counter, and the `WakeUpClock` parameters sets the counter frequency, and it can assume one of the values listed in **Table 18.3**.

Table 18.3: Available values for the `WakeUpClock` parameter

Wakeup counter clock source	Description
`RTC_WAKEUPCLOCK_RTCCLK_DIV2`	The wakeup counter clock source is set to RTCCLK/2
`RTC_WAKEUPCLOCK_RTCCLK_DIV4`	The wakeup counter clock source is set to RTCCLK/4
`RTC_WAKEUPCLOCK_RTCCLK_DIV8`	The wakeup counter clock source is set to RTCCLK/8
`RTC_WAKEUPCLOCK_RTCCLK_DIV16`	The wakeup counter clock source is set to RTCCLK/16
`RTC_WAKEUPCLOCK_CK_SPRE_16BITS`	The wakeup counter clock source is set to $Calendar_{CLK}$
`RTC_WAKEUPCLOCK_CK_SPRE_17BITS`	The wakeup counter clock source is set to $Calendar_{CLK}$ and the wakeup counter increases of an additional bit (so it can count up to `0x1FFFF`).

An independent IRQ (`RTC_WKUP_IRQn`) is associated with the wakeup counter, and it can be enabled by using the function:

```
HAL_RTCEx_SetWakeUpTimer_IT(RTC_HandleTypeDef *hrtc, uint32_t WakeUpCounter,
                            uint32_t WakeUpClock);
```

As usual, we must call the `HAL_RTCEx_WakeUpTimerIRQHandler()` from the ISR, and be prepared to be notified of the wakeup event by implementing the `HAL_RTCEx_WakeUpTimerEventCallback()`. Otherwise, if using the wakeup counter in polling mode, we can use the `HAL_RTCEx_PollForWakeUpTimerEvent()` to detect the wakeup event (not that useful to be honest).

18.2.4 Timestamp Generation and Tamper Detection

The RTC peripheral is hardwired to several signal I/Os depending on the package used. These I/Os can be used to generate a timestamp when their state changes. Current date/time is so saved inside dedicated registers, and the corresponding interrupt is also fired if enabled.

To set the timestamp generation, the CubeHAL provides the function:

```
HAL_RTCEx_SetTimeStamp(RTC_HandleTypeDef *hrtc, uint32_t TimeStampEdge,
                       uint32_t RTC_TimeStampPin);
```

The `TimeStampEdge` parameter specifies the pin edge on which the timestamp is activated. This parameter can be one of the following values: `RTC_TIMESTAMPEDGE_RISING` and `RTC_TIMESTAMPEDGE_FALLING`. The `RTC_TimeStampPin` specifies the I/Os used to generate the timestamp, and it can assume the value `RTC_TIMESTAMPPIN_DEFAULT` (which usually corresponds to PC13 pin), or the value `RTC_TIMESTAMPPIN_PA0` or `RTC_TIMESTAMPPIN_POS1` to indicate an alternative pin (usually PA0 or PI8).

To enable the corresponding interrupt, which is associated with the dedicated `TAMP_STAMP_IRQn` IRQ, we can use the function:

```
HAL_RTCEx_SetTimeStamp_IT(RTC_HandleTypeDef *hrtc, uint32_t TimeStampEdge,
                          uint32_t RTC_TimeStampPin);
```

The `HAL_RTCEx_TamperTimeStampIRQHandler()` is the handler to call from the ISR, while the `HAL_RTCEx_TimeStampEventCallback()` is the corresponding callback. If, instead, we want to use the timestamp feature in polling mode, we can use the function:

```
HAL_RTCEx_PollForTimeStampEvent(RTC_HandleTypeDef *hrtc, uint32_t Timeout);
```

to poll for timestamp event. To retrieve the date/time saved in the timestamp registers, we can use the function:

```
HAL_RTCEx_GetTimeStamp(RTC_HandleTypeDef *hrtc, RTC_TimeTypeDef *sTimeStamp,
                       RTC_DateTypeDef *sTimeStampDate, uint32_t Format);
```

The same I/Os can be configured to detect tampering. The CubeHAL provides dedicated routines and C structures to program this feature. We will not address them here. Refer to the CubeHAL source code (especially to the module `HAL_RTCEx`) for more about this.

18.2.5 RTC Calibration

RTC can be calibrated to compensate imprecisions of the RTCCLK source. This is especially useful for applications that need an elevate RTC precision and for those applications that demand RTC stability when temperature changes.

RTC peripheral offers two types of calibration: *coarse* and *smooth* calibration. Let us analyze them.

18.2.5.1 RTC Coarse Calibration

The digital coarse calibration can be used to compensate crystal inaccuracy by adding (positive calibration) or masking (negative calibration) clock cycles at the output of the asynchronous prescaler. A negative calibration can be performed with a resolution of about 2 ppm, and a positive calibration can be performed with a resolution of about 4 ppm. The maximum calibration ranges from -63 ppm to 126 ppm.

We can measure the output frequency before the Asynchronous prescaler by routing it to a dedicated pin (which usually coincides with the AF1 pin). When this I/O is used for such operation, it is also referred to as AFO_CALIB pin. By measuring the output frequency with an oscilloscope, we can evaluate the quality of the RTCCLK. The AFO_CALIB is expected to emit a square wave with a fixed 512Hz frequency.

To set the coarse calibration, the HAL provides the function:

```
HAL_RTCEx_SetCoarseCalib(RTC_HandleTypeDef *hrtc, uint32_t CalibSign, uint32_t Value);
```

The `CalibSign` parameter can accept the values `RTC_CALIBSIGN_POSITIVE` and `RTC_CALIBSIGN_NEGATIVE`, while the `Value` parameter can range from 0 to 63 with a 2 ppm step when using negative sign or from 0 to 126 with a 4 ppm step when using positive sign.

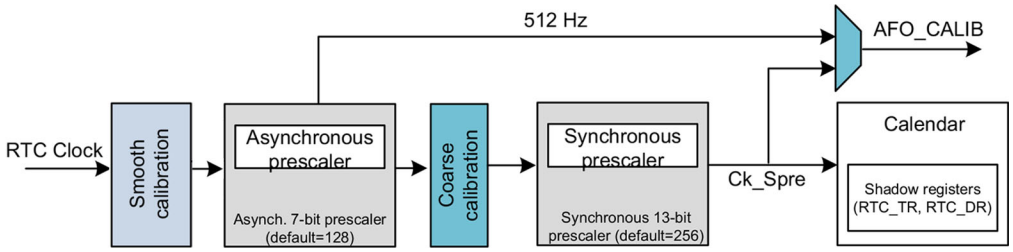


Figure 18.3: The structure of the RTC clock distribution

When calibrating the RTC using coarse calibration, it is important to underline the following points.

- It is not possible to check the calibration result, as the 512Hz output is before the calibration block (see **Figure 18.3**[4]). You can check the calibration in some STM32 MCUs, as the 1Hz `CK_Spre` output is after the coarse calibration block. Refer to the reference manual for your MCU.
- The calibration settings can only be changed during initialization. So use coarse calibration only for static correction.

18.2.5.2 RTC Smooth Calibration

RTC frequency can be calibrated with a resolution of about 0.954 ppm with a range from -487.1 ppm to +488.5 ppm. The correction of the frequency is performed using series of small adjustments

[4]The figure is taken from the AN3371 from ST(http://bit.ly/2fcR1uE).

(adding and/or simultaneously subtracting individual RTCCLK pulses). These adjustments are well distributed on a range of several seconds (8, 16 or 32 seconds), so that the RTC is well calibrated even when observed over short durations of time.

Two RTC registers, named `CALP` and `CALM`, are used to add and/or subtract a given number of RTCCLK pulses over the selected range (8, 16 or 32 seconds). While `CALM` allows the RTC frequency to be reduced by up to 487.1 ppm with fine resolution, the bit `CALP` can be used to increase the frequency by 488.5 ppm. Setting `CALP` register to '1' effectively inserts an extra RTCCLK pulse every 2^{11} RTCCLK cycles, which means that 512 clocks pulses are added during every 32-second cycle. By specifying the number of RTCCLK pulses to be masked during the 32-second cycle using the `CALM` register, the number of clock pulses to add can be decreased up to 0.

Using `CALM` together with `CALP`, an offset ranging from -511 to +512 RTCCLK cycles can be added during the 32-second cycle, which translates to a calibration range of -487.1 ppm to +488.5 ppm with a resolution of about 0.954 ppm. The formula to calculate the effective calibrated frequency (F_{CAL}) given the input frequency (F_{RTCCLK}) is as follows:

$$F_{CAL} = F_{RTCCLK} \times [1 + \frac{(CALP \times 512 - CALM)}{(2^{20} + CALM - CALP \times 512)}] \quad [2]$$

To set the smooth calibration, the HAL provides the function:

```
HAL_RTCEx_SetSmoothCalib(RTC_HandleTypeDef* hrtc, uint32_t SmoothCalibPeriod,
                         uint32_t SmoothCalibPlusPulses, uint32_t SmouthCalibMinusPulsesValue);
```

`SmoothCalibPeriod` parameter can assume the values listed in **Table 18.4** and it defines the distribution interval. `SmoothCalibPlusPulses` parameter can assume the values `RTC_SMOOTHCALIB_PLUSPULSES_SET` and `RTC_SMOOTHCALIB_PLUSPULSES_RESET` and it is used to set/reset the single bit inside the `CALP` register.
The `SmouthCalibMinusPulsesValue` parameter sets the number of clock pulses to subtract, and it can be any value from 0 to 511.

Table 18.4: Available values for the `SmoothCalibPeriod` parameter

Wakeup counter clock source	Description
`RTC_SMOOTHCALIB_PERIOD_8SEC`	The smooth calibration period is 8s
`RTC_SMOOTHCALIB_PERIOD_16SEC`	The smooth calibration period is 16s
`RTC_SMOOTHCALIB_PERIOD_32SEC`	The smooth calibration period is 32s

Different from the RTC coarse calibration, the smooth calibration effects on the calendar clock (RTC Clock) can be easily checked by checking the output of the AFO_CALIB pin. A smooth calibration can be also performed on the fly so that it can be changed when the temperature changes or if other factors are detected.

18.2.5.3 Reference Clock Detection

In some applications, the RTC can be actively calibrated using an external reference clock. The reference clock (at 50Hz or 60Hz - the typical mains frequency) should have a higher precision than a 32.768kHz LSE clock. That is the reason why the RTC in STM32 MCUs with higher pin count provides a reference clock input (named RTC_50Hz pin), which can be used to compensate the imprecision of the calendar frequency (1Hz).

The RTC_50Hz pin should be configured in input floating mode. This mechanism enables the calendar to be as precise as the reference clock. The reference clock detection is enabled by using the function:

```
HAL_StatusTypeDef HAL_RTCEx_SetRefClock(RTC_HandleTypeDef* hrtc);
```

When the reference clock detection is enabled, both *asynchronous* and *synchronous* prescalers must be set to their default values: `0x7F` and `0xFF`. When the reference clock detection is enabled, each 1Hz clock edge is compared to the nearest reference clock edge (if one is found within a given time window). In most cases, the two clock edges are properly aligned. When the 1Hz clock becomes misaligned due to the imprecision of the LSE clock, the RTC shifts the 1Hz clock a bit so that future 1Hz clock edges are aligned. The update window is three `ck_calib` periods (`ck_calib` is the output of the coarse calibration block - see **Figure 18.3**).

If the reference clock halts, the calendar is updated continuously based solely on the LSE clock. The RTC then waits for the reference clock using a detection window centered on the *synchronous* prescaler output clock (`ck_spre`) edge. The detection window is seven `ck_calib` periods.

The reference clock can have a large local deviation (for instance in the range of 500ppm), but in the long term it must be much more precise than a 32kHz quartz. The detection system is used only when the reference clock needs to be detected back after a loss. As the detection window is a bit larger than the reference clock period, this detection system brings an uncertainty of 1 `ck_ref` period (20ms for a 50Hz reference clock) because we can have 2 `ck_ref` edges in the detection window. Then the update window is used, which brings no error as it is smaller than the reference clock period. We assume that `ck_ref` is not lost more than once a day. So, the total uncertainty per month would be 20ms * 1 * 30 = 0.6s, which is much less than the uncertainty of a typical quartz (1.53 minute per month for 35ppm quartz).

18.3 Using the Backup SRAM

The majority of STM32 microcontrollers provide an additional memory region called *backup memory* (or *RTC backup data memory*). This memory is powered-on by VBAT when VDD is switched off, if the VBAT pin is connected to a backup power source, so that its content is not lost upon a system reset. Content of the *backup memory* remains valid even when the device operates in *low-power* mode. Instead, backup registers are reset when a tamper detection event occurs.

By default, after a system reset, the access in write mode to the so-called *backup domain* (which includes the backup memory and the RTC registers) is disabled to protected it from possible and unwanted write accesses due to unstable power source. To modify the whole domain, and hence the backup memory, we need to explicitly follow this procedure:

- enable the power interface clock by using the macro `__HAL_RCC_PWR_CLK_ENABLE()`;
- call the `HAL_PWR_EnableBkUpAccess()` function to enable the access to the *backup domain* (RTC registers, RTC backup data memory).
- Use the functions `HAL_RTCEx_BKUPWrite()` and `HAL_RTCEx_BKUPRead()` function to write/read inside the available backup registers (the number of registers differs among the several STM32 MCUs).

III Advanced topics

19. Power Management

Energy efficiency is one of the trend topics in the electronics industry. Even if you are not designing a battery-powered device, probably you have to address power-related requirements anyway. A well-designed device, from the power point of view, not only consumes less energy, but it also allows to simplify and minimize its power-section, reducing the overall dimension of the PCB, the BOM and the power dissipation.

Often, we think that the power management of an electronic board is all related to its powering stage. In the last two decades, power-conversion has been the hot topic. The research and development made by IC vendors did generate a lot of integrated devices able to boost the overall power efficiency in a lot of applications fields, ranging from low-power solutions to high-load power conversion units able to supply thousands of amperes. Instead, as embedded developers, we have great responsibility in ensuring that our firmware can minimize the energy consumption of devices we make.

Modern microcontrollers provide to developers a lot of tools to minimize the energy used. Cortex-M cores aren't an exception, and they provide an "abstract" power management model that is rearranged by silicon manufacturers to create their own power management scheme. This is exactly the case of STM32 MCUs: even if power management is addressed in all STM32-series, it reaches a very sophisticated implementation in STM32L and STM32U families, which provide to developers a scalable power model to precisely tune-up the energy needed. This allows to design electronic devices able to run even for years while powered by a coin-cell battery.

In this chapter we will give a quick look at the way power management is implemented in STM32 MCUs, analyzing the STM32F-series, STM32L-series and STM32G-series separately. We will start examining which features are provided by the Cortex-M core and then we will discover how ST engineers have specialized them to provide up to twelve different power modes in the recent STM32L5-series.

19.1 Power Management in Cortex-M Based MCUs

Before we study the features provided by Cortex-M based microcontrollers to programmatically select the *power mode* of the MCU, it is best to do some considerations about the power consumption sources in a digital device.

First of all, the complexity of the device itself impacts on the energy consumed. The more peripherals and features our board provides the more power is needed. Moreover, some peripherals are intrinsically energy intensive. For example, TFT displays consume a lot of power if compared with other parts of the electronic board. Finally, a low-power design needs a careful selection of all components in the BOM. For example, in applications where the *Real-Time Clock* (RTC)

is maintained active at all conditions[1], including *sleep*, *shutdown* and VBAT modes, the current consumption of the LSE becomes more critical in overall system-level application design.

Focusing our attention exclusively on the MCU, the first aspect that affects the power consumption is its running frequency: the faster goes the CPU, the higher it consumes. And this a law written in the stone that all firmware developers must know: even if the MCU we are using is able to run up to 200MHz, if we do not need all that speed then we can save a lot of energy by simply reducing the clock frequency. And this is one of the main reasons why STM32 microcontrollers have a complex clock distribution tree.

Another implication of this aspect is that the more peripherals are actively running, the more power the MCU eats. This means that a well-designed firmware always immediately disables a peripheral that becomes unnecessary. For example, if we need an I²C EEPROM only during the bootstrap process (because it stores some configuration parameters that we retain in RAM during the firmware life-cycle), then we have to disable the I²C peripheral once finished[2]. This is the reason why STM32 MCUs offer the ability to selectively disable every peripheral, gating its clock source, by calling the `__HAL__RCC_<PPP>_CLK_DISABLE()`, where `<PPP>` is the specific peripheral (for example, the `__HAL_RCC_DMA1_CLK_DISABLE()` allows to gate the clock of the DMA1, while the `__HAL_RCC_DMA1_CLK_ENABLE()` to enable it).

When talking about microcontrollers, it is best to talk about energy efficiency instead of just their power consumption. While the power consumption of a device talks just about how many mA or µA it uses, the energy efficiency measures "how much work" it can do with a limited amount of energy, for example, in the form of DMIPS/mW or CoreMark/mW. We can so discover that for an STM32L4 MCU the best energy compromise is reached when it runs in *Low-Power RUN* (LPRUN) mode, as shown in **Figure 19.8**.

Finally, the design itself of the MCU and its peripherals impact on the overall power consumption. This is the reason why STM32L microcontrollers are expressly designed to provide the best-in class power consumption while providing the best performances according to the specific sub-family. For example, some of the communication peripherals in an STM32L4 MCU (the LPUART is one of these) allow exchanging data in DMA mode while the MCU is in STOP2 mode[3].

19.2 How Cortex-M MCUs Handle *Run* and *Sleep* Modes

When a Cortex-M based microcontroller resets, its power mode is set to the *run*[4] one. In this mode the energy needed is certainly established by the whole MCU design, but mainly from the running frequency and the number of active peripherals. Here it is important to remark that also the flash

[1] As we will discover next, in some really "deep" sleep modes the MCU can be woken up only by few peripherals, which always include the RTC.

[2] The I²C peripheral consumes up to 720µA in an "old" STM32F103 running at its maximum clock speed. This might seem not that much for a device powered from the mains, but it has a dramatic impact on a battery-powered device.

[3] In this mode, the core of an STM32L4 MCU consumes about 1.1µA.

[4] Official ARM documentation talks about *active* mode, which is opposed to the *sleep* one used to indicate a *non-running* core. However, since this book is all about STM32 microcontrollers, and since the power scheme of a Cortex-M MCU is left to the specific vendor implementation, we will use in this book the term *run* mode, which is what ST uses to indicate a CPU actively running.

and the SRAM memories are "peripherals" external to the Cortex-M core. Moreover, the adoption of advanced flash prefetch technologies, like the ART™ Accelerator, impact on the overall power consumption too.

In this mode the developer can change the way the MCU consumes energy by regulating the clock speed and by disabling the unneeded peripherals. This may seem obviously, but it is important to remark that this is the best power optimization we can do in a lot of real situations. As we will see later in this chapter, STM32G/L MCUs structure the *run* mode in several sub-modes, offering more control on the power consumption while guaranteeing the majority of functionalities and the best CPU performances.

If we know that we do not need to process anything for a given period, then Cortex-M cores allow us to put them in *sleep* mode without doing busy-waits. In this mode the core is halted, and it can be woken up only by "external events" coming from the EXTI controller (for example, a push-button connected to a GPIO). Again, STM32G/L MCUs expand this mode offering up to eight different sub-modes, as we will see next.

It is important to underline that the Cortex-M core enters in sleep mode on "a voluntary basis": two distinct ARM instructions, that we are going to see in while, halts the CPU while leaving some of its event lines active. By triggering these lines, the CPU resumes the execution in a given *wake-up* time, which depends on the effective *sleep* level and the Cortex-M core type (M0, M3, and so on).

The wake-up latency can be expressed in term of CPU cycles for "lightweight" sleep modes, and in µs for *deep sleep* modes. This means that the deeper is the sleep mode, the longer is the wake-up time. Developers need to decide which sleep mode should be used for their specific applications: the energy and time consumed entering and then exiting a deep low power state may outweigh any potential power saving gains. In a wearable device energy efficiency is the most important factor, while in some industrial control applications the wake-up latency can be really critical.

There are also different approaches to designing low power systems. Nowadays a lot of embedded systems are designed to be interrupt driven. This means that the system stays in sleep mode when there are no requests to be processed. When an interrupt request arrives, the processor wakes up and processes it, and goes back into sleep mode when the work is done. Alternatively, if the data processing request is periodic and has a constant duration, and if the data processing latency is not an issue, you could run the system at the slowest possibly clock speed to reduce the power. There is no clear answer to which approach is better, as the choice will be dependent on the data processing requirements of the application, the microcontroller being used, and other factors like the type of power source available.

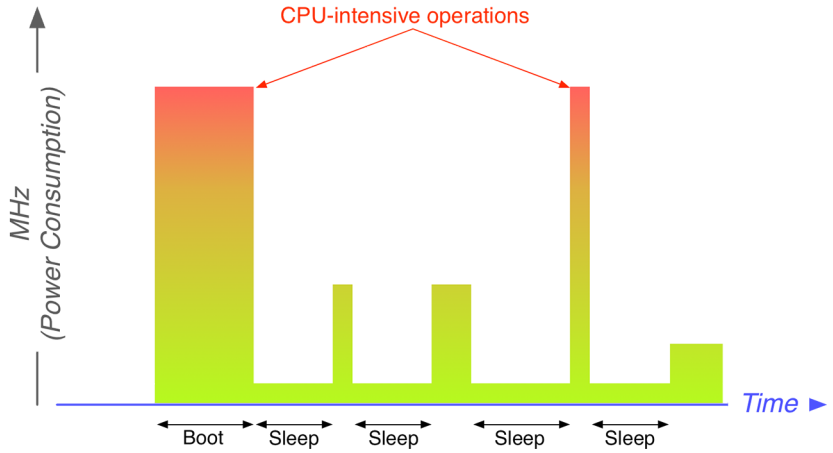


Figure 19.1: How a firmware could potentially manage clock speed and power modes during its activity

The **Figure 19.1** shows a possible strategy for the minimization of power consumption. During the microcontroller booting process, the MCU runs at its maximum speed to allow a fast completion of all initialization activities. When all peripherals are configured, the clock speed is lowered and the MCU enters in sleep modes. In this period, the MCU is woken-up by interrupts that can be processed at lower CPU speeds. When CPU-intensive operations need to be carried out, the clock speed can be increased up to the maximum, and then decreased again once finished.

So, when to go into sleep mode? As said before, it is up to us to decide the right time to place the MCU in one of the possible sleep modes. If we know that the MCU is waiting for asynchronous events notified with interrupts, then it could be the right time to go into sleep mode instead of doing busy-wait. Let us consider the classical blinking LED application we have seen several times in this book.

```
...
while(1) {
  HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
  HAL_Delay(500);
}
```

This apparently innocent code has a dramatic impact on the power consumption of our device. Even if we do not have too much to do during those 500ms, we waste a lot of power checking the value of the global *SysTick* tick count to see if that time has been elapsed. Instead, we can rearrange that code to stay in sleep mode for most of the time, and we can set up a timer that wakes up the MCU after 100ms.

Letting other software components decide when to place the MCU in sleep mode could represent another approach. As we will discover in Chapter 23, a Real-Time Operating System may be programmed to automatically put the MCU in sleep mode when there is nothing to do[5].

[5]We will discover in that chapter that one possible strategy consists in placing the MCU in sleep mode when the *idle* thread is scheduled. The *idle* thread is that thread executed by an RTOS when all other threads are "un-runnable". This clearly means that the MCU has nothing relevant to do, and it can be placed in sleep mode safely.

19.2.1 Entering/exiting sleep modes

As said in the previous paragraph, the CPU enters in sleep mode exclusively on a voluntary basis, by using specific ARM assembly instructions. This means that, as programmers, we have all the responsibility of the power consumption of devices we make[6].

Cortex-M based MCUs offer two instructions to place the MCU in sleep mode: `WFI` and `WFE`. The *Wait For Interrupt* (`WFI`) instruction is also called the un-conditional sleep instruction. When the CPU executes that instruction, it immediately halts the core execution. The CPU will be resumed only by an interrupt request, depending on the interrupt priority and the effective sleep level (more about this later), or in case of debug events. If an interrupt is pending while the MCU executes the `WFI` instruction, it enters in sleep mode and wakes up again immediately.

The *Wait For Event* (`WFE`) is the other instruction that allows to place the MCU in sleep mode. It differs from the `WFI` since it checks the status of a particular event register[7] before it halts the core: if this register is set, the `WFE` clears it and does not halt the CPU, continuing the program execution (this allows us to manage the pending event, if needed). Otherwise, it halts the MCU until this event register is set again.

But what is exactly the difference between an event and an interrupt? Events are a source of confusion in the STM32 world (also in the Cortex-M world in general). They appear like something intangible, compared to the interrupts that we have learned to handle in Chapter 7. Before we clarify what we mean with the term *events*, we need to better explain the role of the EXTI controller in an STM32 MCU. The *Extended Interrupts and Events Controller* (EXTI) is the hardware component internal to the MCU that manages the external and internal asynchronous interrupts/events and generates the event request to the CPU/NVIC controller and a wake-up request to the Power Controller (see **Figure 19.2**). The EXTI allows the management of several event lines, which can wake up the MCU from some sleep modes (not all events can wake up MCU). The lines are either configurable or direct and hence hardwired inside the MCU:

- **The lines are configurable**: the active edge can be chosen independently, and a status flag indicates the source of the interrupt. The configurable lines are used by the I/Os external interrupts, and by few peripherals (more about this soon).
- **The lines are direct and hardwired**: they are used by some peripherals to generate a wakeup from stop event or interrupt. The status flag is provided by the peripheral itself. For example, the RTC can be used to generate an event to wake up the MCU.

[6]Clearly, we are talking about the power consumption of the MCU core and all integrated peripherals. The power consumption of the overall board is determined by other things that we will not address here.

[7]This register is internal to the core and not accessible to the user.

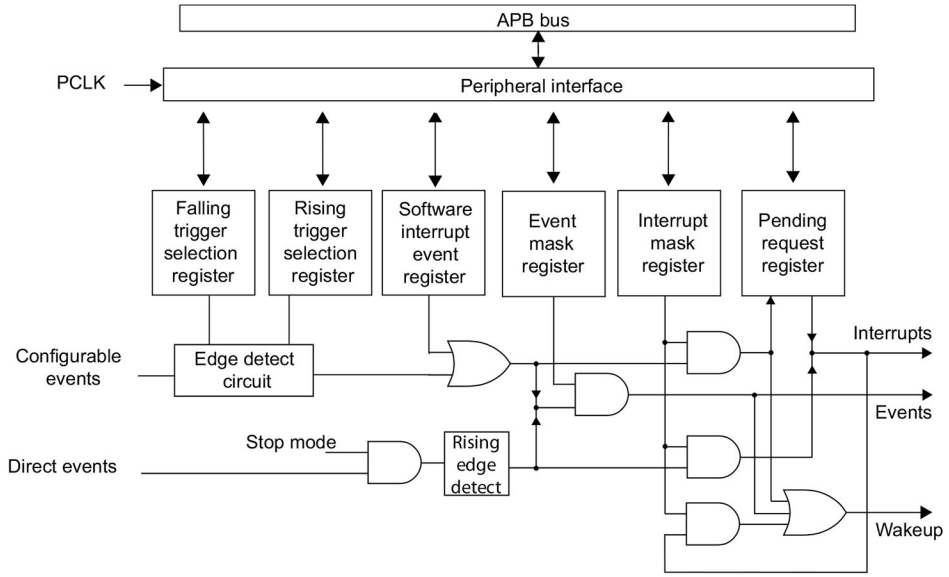


Figure 19.2: How events can be used to wake-up the core

Another important aspect to clarify about EXTI and NVIC controllers is that each line can be masked independently for an interrupt or an event generation. For example, in Chapter 6 we have seen that a GPIO can be configured to work in `GPIO_MODE_EVT_*` mode, which is different from the `GPIO_MODE_IT_*` mode: in the first case, when an I/O is triggered it will not generate an IRQ request, but it will set the event flag. This will cause the MCU to wake up if it has entered a low-power mode using the `WFE` instruction.

So, the `WFE` instruction checks that no event is pending, and for this reason it is also called the conditional sleep instruction. This event register can be set by:

- exception entrance and exit;
- when *SEV-On-Pend* feature is enabled, the event register can be set when an interrupt pending status is changed from `0` to `1` (more about this soon);
- a peripheral sets its dedicated event line (this is peripheral-specific);
- execution of the `SEV` (Send Event) instruction;
- debug event (e.g., halting request).

In Chapter 7 we have seen that in Cortex-M3/4/7/33 cores we can temporarily mask the execution of those interrupts having a priority lower than a value set in the `BASEPRI` register. However, these interrupts are still enabled and marked as pending if they fire. We can configure the MCU to set the event register in case of pending interrupts, by setting the `SCR->SEVONPEND` bit. As the name suggest, this register will cause to "set the event register if interrupts are pending". This means that, if the processor was placed in sleep mode by the `WFE` instruction, the CPU is immediately awakened, and we can eventually process pending interrupts. Instead, the `WFI` instruction would never wake up the core. The Cube HAL provides two convenient functions, `HAL_PWR_EnableSEVOnPend()` and `HAL_PWR_DisableSEVOnPend()`, to perform this setting.

If, instead, interrupts are masked by setting the `PRIMASK` register, a pending interrupt can wake up the processor, regardless for the sleep instruction used (`WFI` or `WFE`): this characteristic allows some parts of the MCU to be turned OFF by software by gating its clock, and the software can turn it back on after waking up before executing the ISR.

So, to recap, the `WFI` and `WFE` have the same following behaviour:

- wake up on interrupt/exception requests that are enabled and with higher priority than current level[8];
- can be woken up by debug events;
- can be used to produce both *sleep* and *deep sleep* modes (more about this soon).

Instead, the `WFI` and `WFE` differ for the following reasons:

- execution of `WFE` does not enter *sleep* mode if the internal event register is set, while the execution of `WFI` always results in sleep;
- new pending of a disabled or masked interrupt can wake up the processor from `WFE` if `SEVONPEND` is set;
- `WFE` can be woken up by en external event;
- `WFI` can be woken up by an enabled interrupt when `PRIMASK` is set.

19.2.1.1 Sleep-On-Exit

The *Sleep-On-Exit* feature is useful for interrupt-driven applications where all operations (apart from the initialization stage) are performed inside interrupt handlers. This is a programmable feature and can be enabled or disabled setting a bit of the `SCB->SCR` register. When enabled, the Cortex-M core automatically enters sleep mode (with the same behavior of `WFI` instruction) when exiting from an exception/interrupt handler. The *Sleep-On-Exit* feature should be enabled at the end of the initialization stage. Otherwise, if an interrupt event happens during the initialization stage while the *Sleep-On-Exit* feature is already enabled, the processor will enter sleep even if the initialization stage was not yet completed.

The CubeHAL provides two convenient routines to enable/disable this mode: `HAL_PWR_EnableSleepOnExit()` and `HAL_PWR_DisableSleepOnExit()`.

19.2.2 *Sleep* Modes in Cortex-M Based MCUs

So far, we have talked broadly about *sleep* mode. This mainly because the power management scheme defined by ARM is further specialized by chip vendors, like ST does with its products. Cortex-M based microcontrollers architecturally support two *sleep* modes: *normal sleep* and *deep sleep*. As we will discover later in this chapter, STM32F microcontrollers calls them *sleep* and *stop* modes

[8]Disabling interrupt on a priority basis is only applicable to Cortex-M3/4/7 based MCUs.

and add a third even deeper mode called *standby*. The STM32L-series further specializes these two "main" operative modes in several sub-modes.

Both *sleep* and *deep sleep* modes are reached using the `WFI` and `WFE` instructions seen before. The only difference is that the *deep sleep* mode is achieved by setting the `SLEEPDEEP` bit to 1 in the `PWR->SCR` register. However, we do not need to deal with these details since the CubeHAL is designed to abstract them.

Usually, STM32 microcontrollers are designed so that in *sleep* mode only the CPU clock is turned OFF, while there are no effects on other clocks or analog clock sources (this means that all enabled peripherals remain active). In *stop* mode, instead, all peripherals belonging to the 1.8V (or 1.2V for more recent STM32 MCUs) domain clock are turned OFF, while the VDD domain is left ON[9], except for HSI and HSE oscillator that are turned OFF. In *standby* mode both the 1.8V domain and the VDD domain are turned OFF. However, in the next paragraph we will deepen these topics.

19.3 Power Management in STM32F Microcontrollers

The concepts illustrated so far are common to all STM32 microcontrollers. However, the STM32 portfolio is divided in several branches: STM32F, STM32G, STM32L, STM32U series (without considering the STM32W families that are not covered in this book). STM32G/L/U are addressed to low-power applications, and they provide a lot of more operative modes to minimize the power consumption.

We will start by analyzing how to manage power modes in STM32F microcontrollers. However, it is important to underline that, as often happens for the other features offered by this large portfolio, some STM32 families, and even some certain part numbers, offer specific peculiarities that differ from the way the power management is handled in the majority of STM32F microcontrollers. For this reason, always keep on hand the reference manual for the MCU you are considering.

19.3.1 Power Sources

Figure 19.3 shows the power sources of an STM32F microcontroller[10]. As said before, even if we are used to supply the MCU by just one power source (more about this in Chapter 27), the MCU has an internal power distribution network that defines several *voltage domains* used to power those peripherals that share the same powering characteristics. For example, the *VDDA domain* includes those analog peripherals that need a separated (better filtered) power source, fed through the VDDA pins.

[9] As we will discover next, STM32 microcontroller can be powered by a variable voltage source ranging from 2.0V to 3.6V (some of them allow to be powered even down to 1.62V). This voltage source is also called *VDD domain* and all components inside the MCU powered from this source are said to be part of the *VDD domain*. However, the internal MCU core and some other peripherals are powered by a dedicated 1.8V (or even 1.0V in some STM32L MCUs) internal voltage regulator. This defines the *1.8V domain* or *VCORE domain*. The low-voltage internal regulator can be independently turned OFF. More about this later.

[10] It is important to remark that the diagram in **Figure 19.3** is just a scheme. Some STM32F MCUs, especially those providing a TFT-LCD controller or other communication interfaces like the Ethernet, introduce other power source domains. In the same way, STM32 MCUs with lower pin count (especially those ones with less than 32 pins) have a simplified power distribution network. However, the concepts illustrated here remain valid.

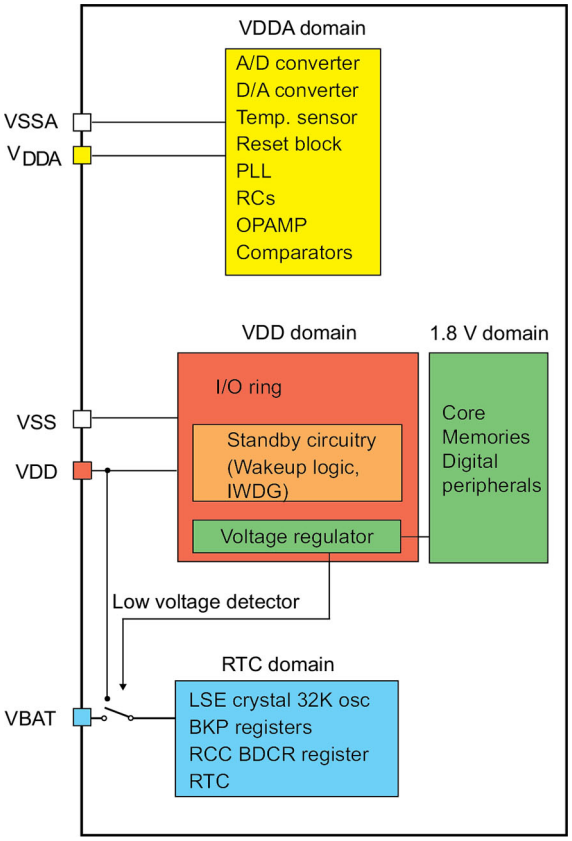


Figure 19.3: The power sources in an STM32F microcontroller

The *VDD* and *VDD18* domains are the most relevant one. The *VDD domain* is supplied by an external power source, while the *VDD18 domain* is supplied by a voltage regulator internal to the MCU. This regulator can be configured to work in low-power mode, as we will see next. To retain the content of the backup registers[11] and supply the RTC function when VDD is turned OFF, VBAT pin can be connected to an optional standby voltage supplied by a battery or by another source. The VBAT pin powers the RTC unit, the LSE oscillator and one or two pins used to wake up the MCU from deep sleep modes, allowing the RTC to operate even when the main power supply is turned OFF. For this reason, the VBAT power source is said to power the *RTC domain*. The switch to the VBAT supply is controlled by the *Power Down Reset* (PDR) embedded in the *reset block*.

19.3.2 Power Modes

In the first part of this chapter, we have seen that a Cortex-M MCU provides three main power modes: *run*, *sleep* and *deep sleep*. Now it is the right time to see how ST engineers have rearranged them in STM32F MCUs. **Table 19.1** summarizes these modes and shows the three main functions provided by the HAL to place the MCU in the corresponding power mode. We will analyze them more in depth later.

[11]Backup registers are a dedicated memory area, with a typical size of 4Kb, that is powered by a different power source usually connected to a battery or a super-capacitor. This is used to store volatile data that remains valid even when the MCU is powered OFF, either if the whole device is turned OFF or the MCU is placed in *standby* mode.

Mode Name	HAL Function to enter	Wake up condition	Effect on 1.8V domain clocks	Effect on VDD domain clocks	Main voltage regulator
Sleep **Sleep-On-Exit**	`HAL_PWR_EnterSLEEPMode()`	Any interrupt (`WFI`) Wake up event (`WFE`)	CPU clock OFF no effect on other clocks or analog clock sources	N/A	ON
Stop	`HAL_PWR_EnterSTOPMode()`	Any EXTI line (configured in the EXTI registers) Specific communication peripherals on reception events (USART, I2C)	All 1.8V domain clocks OFF	HSI and HSE oscillators OFF	Configurable (depends on the specific MCU)
Standby	`HAL_PWR_EnterSTANDBYMode()`	WKUP pin rising edge, RTC alarm, external reset in NRST pin, IWDG reset			OFF

Table 19.1: The three power modes supported by STM32F MCUs

19.3.2.1 Run Mode

By default, and after power-on or a system reset, STM32F MCUs are placed in *run* mode, which is a fully active mode that consumes much power even when performing minor tasks. Consumptions of both the *run* and the *sleep* modes depend on the operating frequency[12].

The **Figure 19.4**[13] shows the power consumption levels of some of the most recent STM32F4 MCUs.

In *run* mode, the main regulator supplies full power to the 1.8-1.2V domain (CPU core, memories and digital peripherals). In this mode, the regulator output voltage (around 1.8-1.2V depending on the given STM32F MCU) can be scaled by software to different voltage values (more about this soon). Some recent STM32F4 MCUs provide two *run* modes:

- **Normal mode**: the CPU and core logic operate at maximum frequency at a given voltage scaling (*scale 1*, *scale 2* or *scale 3*).
- **Over-drive mode**: this mode allows the CPU and the core logic to operate at a higher frequency than the normal mode for the voltage scaling *scale 1* and *scale 2*. More about this mode later.

[12] Don't forget that in *sleep* mode only the CPU clock is turned OFF, while other peripherals remain active. So, the speed of the HCLK clock source continues to affect the overall power consumption.

[13] The figure is taken from the ST AN4365(https://bit.ly/1XzmF2o) application note.

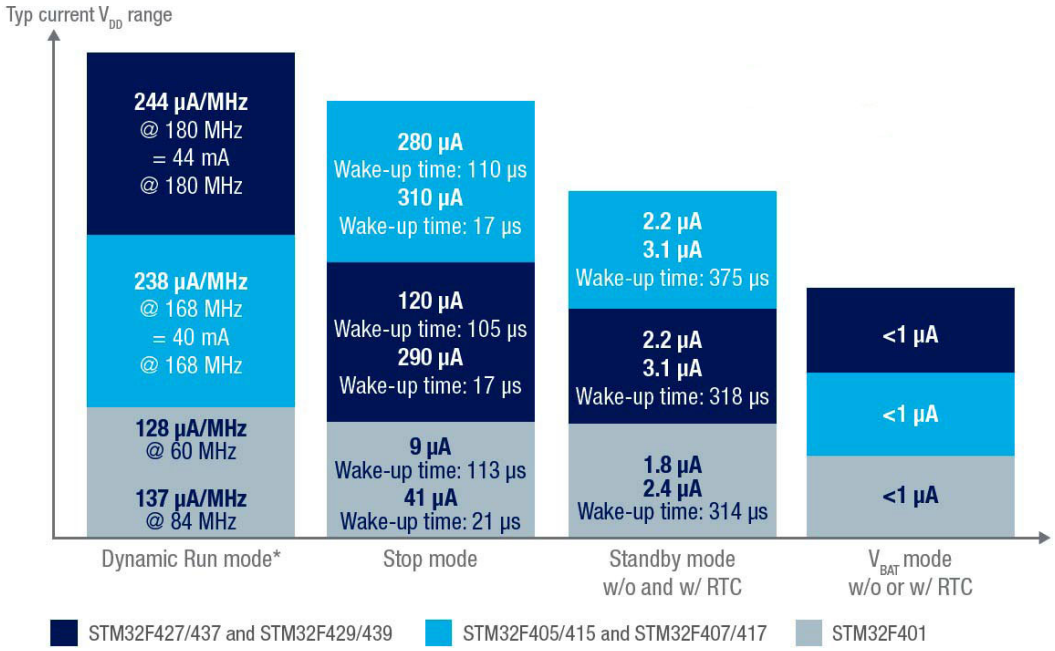


Figure 19.4: The power consumption of some STM32F4 MCU

19.3.2.1.1 Dynamic Voltage Scaling in STM32F4/F7 MCUs

The power used by a DC circuit is given by the current drawn and the voltage of the circuit. This means that we can reduce the power needed by the circuit by reducing the voltage. STM32F4/F7 provides a smart powering technology named *Dynamic Voltage Scaling* (DVS) distinctive of STM32L-series. The idea behind DVS is that many embedded systems do not always require the system's full processing capabilities, because not all subsystems are always active. When this is the case, the system can remain in the active mode without the processor running at its maximum operating frequency. The voltage supplied to the processor can be then decreased when a lower frequency is sufficient. With such power management, we reduce the power drawn battery by monitoring the processor input voltage in response to the system's performance requirements.

That consists in scaling the STM32F4 regulator output voltage that supplies the 1.2V domain (core, memories and digital peripherals) when we lower the clock frequency based on processing needs. STM32F4/F7 offer three voltages scales (*scale 1*, *scale 2* and *scale 3*). The maximum achievable core frequency for a given voltage scale is determined by the specific STM32 MCU. For example, the STM32F401 provides only two voltage scales, *scale 2* and *scale 3*, that allow to run the core up to 84MHz and 60MHz respectively. To control the voltage scaling, the CubeHAL provides the function:

```
HAL_StatusTypeDef HAL_PWREx_ControlVoltageScaling(uint32_t VoltageScaling);
```

which accepts the symbolic constants `PWR_REGULATOR_VOLTAGE_SCALE1`, `PWR_REGULATOR_VOLTAGE_SCALE2` and `PWR_REGULATOR_VOLTAGE_SCALE3`. The voltage scale can be changed only if the source clock for the *System Clock Multiplexer* is the HSI or HSE. So, to increase/reduce the voltage scale you can follow this procedure:

- Set the HSI or HSE as system clock frequency using the `HAL_RCC_ClockConfig()`.
- Call the `HAL_RCC_OscConfig()` to configure the PLL.
- Call `HAL_PWREx_ConfigVoltageScaling()` API to adjust the voltage scale.
- Set the new system clock frequency using the `HAL_RCC_ClockConfig()`.

For more information about this topic, refer to the AN4365[14].

19.3.2.1.2 Over/Under-Drive Mode in STM32F4/F7 MCUs

Some MCUs from the STM32F4 family and all STM32F7 ones provide two or even several sub-running modes. These modes are called *over-drive* and *under-drive*. The first one consists in increasing the core frequency with a sort of "overclocking". It is recommended to enter *over-drive* mode when the application is not running critical tasks and when the system clock source is either HSI or HSE. These features are useful when we want to temporarily increase/decrease the MCU clock speed without reconfiguring the clock tree, which usually introduces a non-negligible overhead. The HAL provides two convenient functions, `HAL_PWREx_EnableOverDrive()` and `HAL_PWREx_DisableOverDrive()` to perform this operation.

The *under-drive* mode is the opposite of the *over-drive* one and consists in lowering the CPU frequency and by disabling some peripherals. In this mode it is possible to place the internal voltage regulator in low-power mode. In some STM32F4/F7 MCUs the *under-drive* mode is available even in *stop* mode.

19.3.2.2 Sleep Mode

The *sleep* mode is entered by executing the `WFI` or `WFE` instruction. In the *sleep* mode, all I/O pins keep the same state as in the *run* mode. However, we should not care to deal with assembly instructions, since the CubeHAL provides the function:

```
void HAL_PWR_EnterSLEEPMode(uint32_t Regulator, uint8_t SLEEPEntry);
```

The first parameter, `Regulator`, is meaningless in *sleep* mode for all STM32F-series, and it is left for compatibility with STM32L-series. The second parameter, `SLEEPEntry`, can assume the values `PWR_SLEEPENTRY_WFI` or `PWR_SLEEPENTRY_WFE`: as the names suggest, the former performs a `WFI` instruction and the latter a `WFE`.



If you take a look at the `HAL_PWR_EnterSLEEPMode()` function you discover that, if we pass the parameter `PWR_SLEEPENTRY_WFE`, it executes two `WFE` instructions consecutively. This causes that the `HAL_PWR_EnterSLEEPMode()` enters in the *sleep* mode in the same way as it would be called with the parameter `PWR_SLEEPENTRY_WFI` (calling `WFE` twice causes that if the event register is set, then it is cleared by the first `WFE` instruction, and the second one place the MCU in *sleep* mode). I do not know why ST has adopted this approach. If you want full control over the way the MCU is placed in low-power modes, then you have to rearrange the content of that function at your need. Clearly, the MCU will exit from *sleep* mode following the exit condition of the `WFE` instruction.

[14] https://bit.ly/1XzmF2o

If the WFI instruction is used to enter in *sleep* mode, any peripheral interrupt acknowledged by the nested vectored interrupt controller (NVIC) can wake up the device from *sleep* mode.
If the WFE instruction is used to enter *sleep* mode, the MCU exits *sleep* mode as soon as an event occurs. The wakeup event can be generated either by:

- enabling an interrupt in the peripheral control register but not in the NVIC, and enabling the SEVONPEND bit in the System Control Registe - When the MCU resumes from WFE, the peripheral interrupt pending bit and the peripheral NVIC IRQ channel pending bit (in the NVIC interrupt clear pending register) have to be cleared;
- or configuring an external or internal EXTI line in event mode - When the CPU resumes from WFE, it is not necessary to clear the peripheral interrupt pending bit or the NVIC IRQ channel pending bit as the pending bit corresponding to the event line is not set.

This mode offers the lowest wakeup time as no time is wasted in interrupt entry/exit.

19.3.2.3 Stop Mode

The *stop* mode is based on the Cortex-M *deep sleep* mode combined with peripheral clock gating. In *stop* mode all clocks in the 1.8V domain are stopped, the PLL, the HSI and the HSE oscillators are disabled. SRAM and register contents are preserved. In the *stop* mode, all I/O pins keep the same state as in the *run* mode. The voltage regulator can be configured either in normal or low-power mode. To place the MCU in *stop* mode the HAL provides the function:

```
void HAL_PWR_EnterSTOPMode(uint32_t Regulator, uint8_t STOPEntry);
```

where the Regulator parameter accepts the value PWR_MAINREGULATOR_ON to leave the internal voltage regulator ON, or the value PWR_LOWPOWERREGULATOR_ON to place it in low-power mode. The parameter STOPEntry can assume the values PWR_STOPENTRY_WFI or PWR_STOPENTRY_WFE.

To enter *stop* mode, all EXTI-line pending bits, all peripherals interrupt pending bits and RTC Alarm flag must be reset. Otherwise, the *stop* mode entry procedure is ignored, and program execution continues. If the application needs to disable the external high-speed oscillator (HSE) before entering *stop* mode, the system clock source must be first switched to HSI and then clear the HSEON bit. Otherwise, if before entering *stop* mode the HSEON bit is kept at 1, the security system (CSS) feature must be enabled to detect any external oscillator (external clock) failure and avoid a malfunction when entering *stop* mode.

Any EXTI-line configured in interrupt or event mode forces the CPU to exit from *stop* mode, according if it entered in low-power mode using the WFI or WFE instruction. Since both HSE and PLL are disabled before entering in *stop* mode, when exiting from this low-power mode the MCU source clock is set to the HSI. This means that our code shall reconfigure the clock tree according to wanted SYSCLK speed.

19.3.2.4 Standby Mode

The *standby* mode allows to achieve the lowest power consumption. It is based on the Cortex-M *deep sleep* mode, with the voltage regulator disabled. The 1.8-1.2V domain is consequently powered OFF. PLL multiplexer, HSI and HSE oscillators are also switched OFF. SRAM and register contents are lost except for registers in the standby circuitry. To place the MCU in *standby* mode the HAL provides the function:

```
void HAL_PWR_EnterSTANDBYMode(void);
```

The microcontroller exits the *standby* mode when an external reset (NRST pin), an IWDG reset, a rising edge on one of the enabled WKUPx pins or an RTC event occurs. All registers are reset after wakeup from *standby* except for *Power Control/Status Register* (`PWR->CSR`). After waking up from *standby* mode, program execution restarts in the same way as after a reset (boot pin sampling, option bytes loading, reset vector is fetched, etc.). Using the macro:

```
__HAL_PWR_GET_FLAG(PWR_FLAG_SB);
```

we can check if the MCU is resetting due to an exit from *standby* mode. Since both HSE and PLL are disabled before entering in *stop* mode, when exiting from this low-power mode the MCU source clock is set to the HSI. This means that our code shall reconfigure the clock tree according to wanted SYSCLK speed.



Read Carefully

Some STM32 MCUs have a hardware bug that prevents entering or exiting from *standby* mode. Particular conditions must be met before we enter in this mode. Consult the errata sheet for your MCU for more about this (if applicable).

19.3.2.5 Low-Power Modes Example

The following example, designed to run on a Nucleo-F072RB[15] shows the way low-power modes work.

[15]For other Nucleo boards refer to the book examples.

Filename: `Core/Src/main-ex1.c`

```c
int main(void) {
  char msg[30];

  HAL_Init();
  Nucleo_BSP_Init();

  /* Before we can access to every register of the PWR peripheral we must enable it */
  __HAL_RCC_PWR_CLK_ENABLE();

  while (1) {
    if(__HAL_PWR_GET_FLAG(PWR_FLAG_SB)) {
      /* If standby flag set in PWR->CSR, then the reset is generated from
       * the exit of the standby mode */
      sprintf(msg, "RESET after STANDBY mode\r\n");
      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
      /* We have to explicitly clear the flag */
      __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU|PWR_FLAG_SB);
    }

    sprintf(msg, "MCU in run mode\r\n");
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
    while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_SET) {
      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
      HAL_Delay(100);
    }

    HAL_Delay(200);

    sprintf(msg, "Entering in SLEEP mode\r\n");
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);

    SleepMode();

    sprintf(msg, "Exiting from SLEEP mode\r\n");
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);

    while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_SET);
    HAL_Delay(200);

    sprintf(msg, "Entering in STOP mode\r\n");
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);

    StopMode();

    sprintf(msg, "Exiting from STOP mode\r\n");
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
```

```
60
61      while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_SET);
62      HAL_Delay(200);
63
64      sprintf(msg, "Entering in STANDBY mode\r\n");
65      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
66
67      StandbyMode();
68
69      while(1); //Never arrives here, since MCU is reset when exiting from STANDBY
70    }
71  }
72
73
74  void SleepMode(void)
75  {
76    GPIO_InitTypeDef GPIO_InitStruct;
77
78    /* Disable all GPIOs to reduce power */
79    MX_GPIO_Deinit();
80
81    /* Configure User push-button as external interrupt generator */
82    __HAL_RCC_GPIOC_CLK_ENABLE();
83    GPIO_InitStruct.Pin = B1_Pin;
84    GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
85    GPIO_InitStruct.Pull = GPIO_NOPULL;
86    HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
87
88    HAL_UART_DeInit(&huart2);
89
90    /* Suspend Tick increment to prevent wakeup by Systick interrupt.
91       Otherwise the Systick interrupt will wake up the device within 1ms (HAL time base) */
92    HAL_SuspendTick();
93
94    __HAL_RCC_PWR_CLK_ENABLE();
95    /* Request to enter SLEEP mode */
96    HAL_PWR_EnterSLEEPMode(0, PWR_SLEEPENTRY_WFI);
97
98    /* Resume Tick interrupt if disabled prior to sleep mode entry*/
99    HAL_ResumeTick();
100
101   /* Reinitialize GPIOs */
102   MX_GPIO_Init();
103
104   /* Reinitialize UART2 */
105   MX_USART2_UART_Init();
106 }
```

The macro `__HAL_RCC_PWR_CLK_ENABLE()` at line 21 enables the PWR peripheral: before we can perform any operation related to power management, we need to enable the PWR peripheral, even if we are simply checking if the standby flag is set inside the `PWR->CSR` register. This is a source of a lot of headaches in novice users struggling with power management.
Lines [24:31] check if the standby flag is set: if so, it means that the MCU was reset after exiting from *standby* mode. Lines [33:38] represent the *run* mode: the LD2 LED blinks until we press the Nucleo USER button connected to the PC13 pin. The remaining lines of code in the `main()` just cycle through the three low-power modes at every pressure of the USER button.
Lines [74:106] define the `SleepMode()` function, used to place the MCU in *sleep* mode. All GPIOs are configured as analog, to reduce current consumption on non-used IOs (especially those pins that may be source of leaks). The corresponding peripheral clock is turned OFF, except for the GPIOC peripheral: the PC13 GPIO is used to resume from low-power modes. The same apply for the UART2 interface and the *SysTick* timer, which is halted to prevent the MCU from exiting low-power mode after 1ms. The call to the `HAL_PWR_EnterSLEEPMode()` function at line 96 places the MCU in *sleep* mode, until it wakes up when the USER button is pressed (the MCU wakes up because we configure the corresponding IRQ that causes the `WFI` instruction exiting from the low-power mode). The `StopMode()` function, not shown here, is almost identical to the `SleepMode()` one, except for the fact that it calls the function `HAL_PWR_EnterSTOPMode()` to place the MCU in *stop* mode and it calls agains the `Nucleo_BSP_Init()` function, which in turn calls the `SystemClock_Config()` function to restore the clock at the original settings to allow the USART2 working properly.

Filename: `Core/Src/main-ex1.c`

```
149 void StandbyMode(void) {
150   MX_GPIO_Deinit();
151
152   /* This procedure come from the STM32F030 Errata sheet*/
153   __HAL_RCC_PWR_CLK_ENABLE();
154
155   HAL_PWR_DisableWakeUpPin(PWR_WAKEUP_PIN1);
156
157   /* Clear PWR wake up Flag */
158   __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);
159
160   /* Enable WKUP pin */
161   HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);
162
163   /* Enter STANDBY mode */
164   HAL_PWR_EnterSTANDBYMode();
165 }
```

Finally, lines [144:160] define the `StandbyMode()` function. Here we follow the procedure described in the STM32F072 errata sheet, since that series is affected by a hardware bug that prevents the CPU from entering in *standby* mode: we have to disable the `PWR_WAKEUP_PIN1` pin firstly, then to clear the wake-up flag in the `PWR->CSR` peripheral and to re-enable the wake-up pin, which in an STM32F072 MCU coincides with the PA0 pin.



STM32 MCUs usually have two wake-up pins, named `PWR_WAKEUP_PIN1` and `PWR_WAKEUP_PIN2`. For a lot of STM32 MCUs with LQFP64 package the second wake-up pin coincides with PC13, which is connected to the USER button in all Nucleo boards (except for the Nucleo-F302 where it is connected to PB13 pin). However, we cannot use the `PWR_WAKEUP_PIN2` in our example because that pin is pulled high by a resistor on the PCB. When we configure wake-up pins in conjunction with the *standby* mode, we are not using the corresponding GPIO peripheral, which would allow us to configure the pin input mode, because it is powered down before entering in *standby* mode: the wake-up pins are directly handled by the PWR peripheral, which resets the MCU if one of the two pins goes high. So, in the example we use the `PWR_WAKEUP_PIN1` pin, which corresponds to the PA0 pin in an STM32F072 MCU.

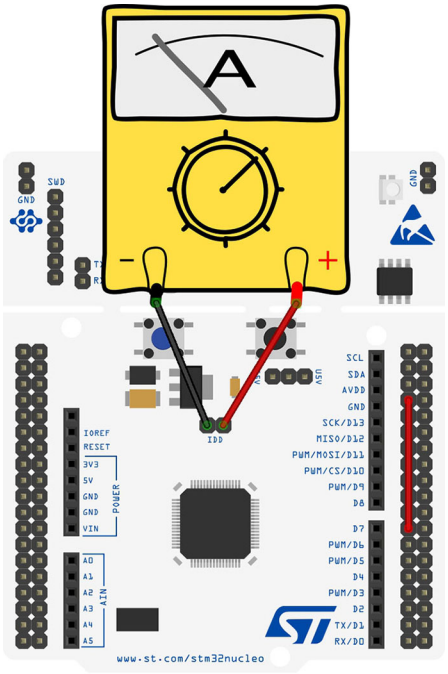


Figure 19.5: How to measure the MCU power consumption in a Nucleo board

Nucleo boards allow to measure the current consumption of the MCU using the IDD pin header. Before you start measurements, you should establish the connection with the board as shown in **Figure 19.5** by removing the IDD jumper and connect the ammeter cables. Ensure that the ammeter is set to the mA scale. In this way you can see the power consumption for every power mode.

F1

19.3.3 An Important Warning for STM32F1 Microcontrollers

During the development of the examples for the FreeRTOS *tickless* mode in the related chapter, I have encountered a nasty behaviour of the STM32F103 MCU when entering in *stop* mode using the `HAL_PWR_EnterSTOPMode()` routine from the CubeF1 HAL. In particular, the problem encountered is

related to the exit from this low-power mode when the MCU enters it using the `WFI` instruction. In that specific scenario, the MCU does enter in *stop* mode correctly, but when it is woken up from an interrupt it immediately generates a *Hard Fault* exception. I have reached to the conclusion that ST developers do not follow what ARM suggests when entering low-power modes in Cortex-M3 processors, as reported here[16].

Modifying the HAL routine in this way fixed the issue:

```
1  void HAL_PWR_EnterSTOPMode(uint32_t Regulator, uint8_t STOPEntry) {
2    /* Check the parameters */
3    assert_param(IS_PWR_REGULATOR(Regulator));
4    assert_param(IS_PWR_STOP_ENTRY(STOPEntry));
5
6    /* Clear PDDS bit in PWR register to specify entering in STOP mode when CPU enter in Deepsle\
7  ep */
8    CLEAR_BIT(PWR->CR,  PWR_CR_PDDS);
9
10   /* Select the voltage regulator mode by setting LPDS bit in PWR register according to Regula\
11 tor parameter value */
12   MODIFY_REG(PWR->CR, PWR_CR_LPDS, Regulator);
13
14   /* Set SLEEPDEEP bit of Cortex System Control Register */
15   SET_BIT(SCB->SCR, ((uint32_t)SCB_SCR_SLEEPDEEP_Msk));
16
17   /* Select Stop mode entry --------------------------------------------------*/
18   if(STOPEntry == PWR_STOPENTRY_WFI)
19   {
20     /* Request Wait For Interrupt */
21     __DSB(); //Added by me
22     __WFI();
23     __ISB(); //Added by me
24   }
25   else
26   {
27     /* Request Wait For Event */
28     __SEV();
29     PWR_OverloadWfe(); /* WFE redefine locally */
30     PWR_OverloadWfe(); /* WFE redefine locally */
31   }
32   /* Reset SLEEPDEEP bit of Cortex System Control Register */
33   CLEAR_BIT(SCB->SCR, ((uint32_t)SCB_SCR_SLEEPDEEP_Msk));
34 }
```

The change simply consists in adding two memory barrier instructions, one before and one after the `WFI` instruction, as shown at lines 21 and 23.

[16] https://bit.ly/3oN7rO5

19.4 Power Management in STM32L/G Microcontrollers

The STM32L-series is an extensive portfolio of MCUs tailored for low-power applications. It is divided in five main families: L0, L1, L4 and the more recent L4+ and L5. These microcontrollers provide more power modes than the STM32F ones, offering the ability to precisely tune the energy consumed by the CPU core and integrated peripherals. Moreover, they provide specific low-power peripherals (like the LPUART or the LPTIM timers). In recent STM32L5 MCU, an integrated *Switched Mode Power Supply* (SMPS) step down converter is also available, reducing the number of external components and increasing overall efficiency. Finally, some recent STM32L MCU provides the possibility to charge an external supercap connected to the VBAT pin[17]. All these features make STM32L MCUs suitable for battery-powered devices.

The STM32 product lineup evolved a lot in recent years, and there exists STM32 MCUs not belonging to the *low-power* series that implement similar capabilities. This is the case of STM32G-series, that provides up to eleven different power modes in some STM32G4 MCUs and an overall power efficiency far better than early STM32L1 microcontrollers.

In this part of the chapter, we will analyze the most relevant power management-related characteristics offered by STM32L/G MCUs, focusing our attention mainly on the STM32L4 family.

19.4.1 Power Sources

Figure 19.6 shows the power sources of an STM32L4 microcontroller. As you can see, to allow a precise tuning of the power consumed by peripherals, these MCUs provide more *voltage domains* compared to the STM32F ones.

[17] ST claims in its datasheet that the VBAT pin can charge a battery. While this is possible in theory, consider that the integrated "charger" is nothing more than a current-limiting resistor connected to the VDD. This means that the charger does not implement any charging algorithm necessary to avoid blowing modern batteries like LiPo or NIMH ones. So, unless you decide to build your own battery charger by implementing adequate charging policies in the firmware, consider that there exists a lot of ICs dedicated to this stuff that cost few cents, especially if you look to the Chinese industry.

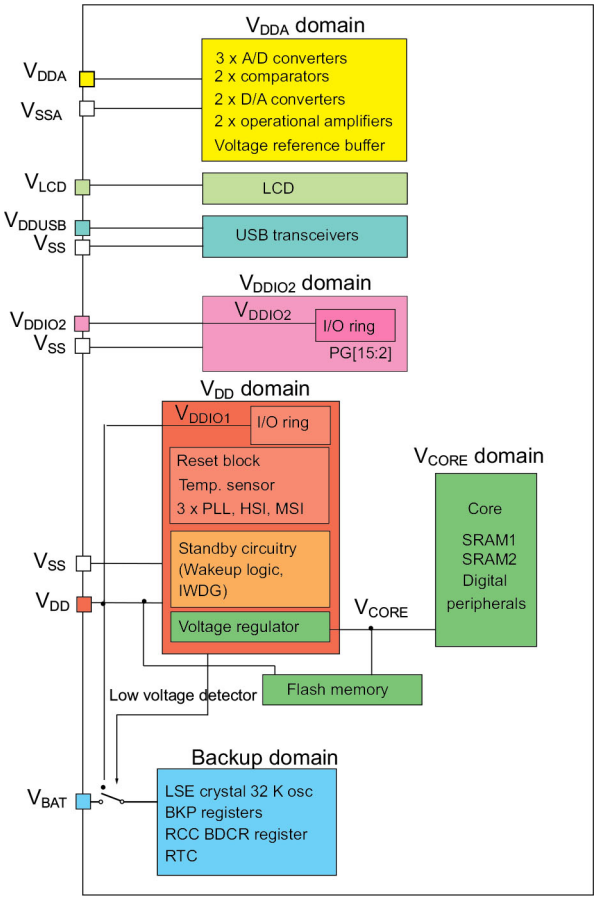


Figure 19.6: The power sources in an STM32L4 microcontroller

Even in these families, the *VDD domain* is the most relevant one. It is used to supply other voltage domains, like the *VDDIO1 domain*, which is used to power most of MCU pins, and the internal voltage regulator used to supply the *VCORE domain*. This can be programmed by software up to three different power ranges (*scale 1*, *scale 2* and *scale 3*) to optimize the consumption depending on the maximum operating frequency of the system (thanks to the voltage scaling technology seen before). It is interesting to remark that for those MCU providing the GPIOG peripheral (that is, those MCUs coming with packages with high pin count), the *VDDIO2 domain* is used to supply the GPIOG peripheral independently. This domain, together with the *USB domain*, can be selectively enabled/disabled by dedicated functions provided by the HAL (`HAL_PWREx_EnableVddIO2()`, `HAL_PWREx_EnableVddUSB()`, etc.).

To retain the content of the backup registers and supply the RTC function when VDD is turned OFF, VBAT pin can be connected to an optional standby voltage supplied by a battery or by another source. The VBAT pin powers the RTC unit, the LSE oscillator and one or two pins used to wake up the MCU from deep sleep modes, allowing the RTC to operate even when the main power supply is turned OFF. For this reason, the VBAT power source is said to power the *RTC domain*. The switch to the VBAT supply is controlled by the PDR. The VLCD pin is provided to control the contrast of the LCD.

19.4.2 Power Modes

Apart from a dedicated design that allows to reduce the power consumption of each component of the MCU, STM32L MCU provide to the user up to twelve different power modes, as shown in **Figure 19.7**. For the first three power modes, consumption values per MHz are an average between the power consumption value when the CPU runs instructions from the flash and from the SRAM[18]. The first three power modes are based on the Cortex-M *run* mode, while the next five modes are based on the *sleep* one. Finally, all other low-power modes rely in the Cortex-M *deep sleep* mode.

Table 19.2 summarizes ten power modes and shows the functions provided by the HAL to place the MCU in the corresponding power mode. We will analyze them more in depth later. Take note that not all those power modes are available in all STM32L/G MCUs. Always consult the reference manual for your datasheet before designing the power transitions in your firmware.

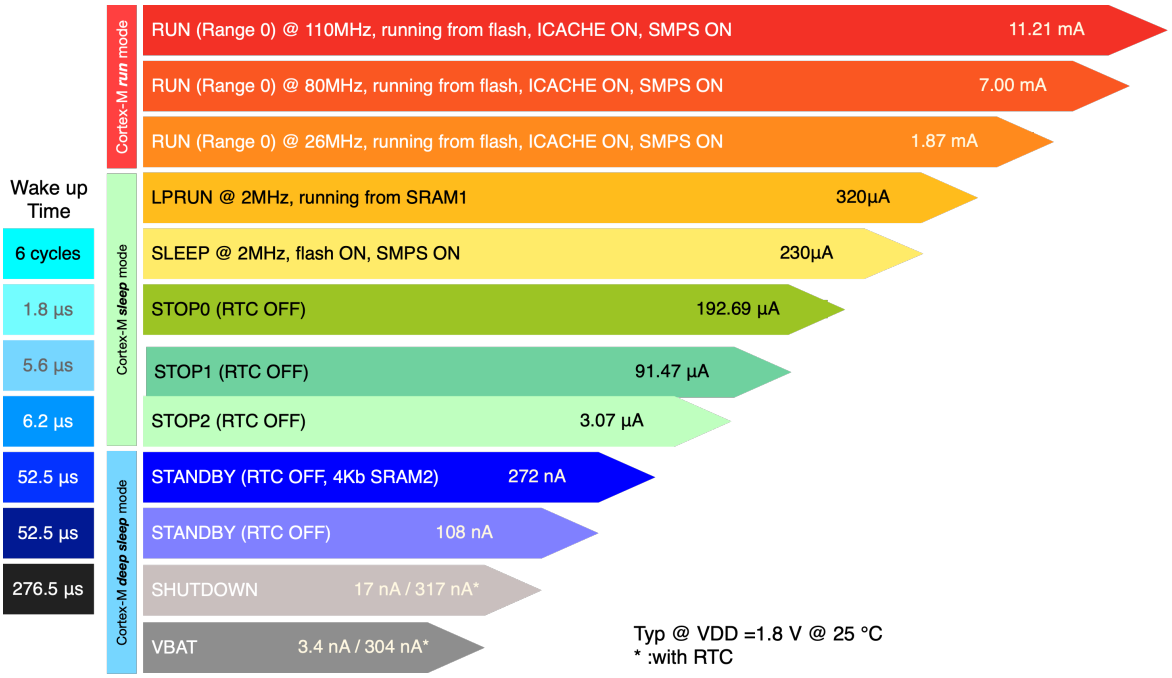


Figure 19.7: The twelve power modes supported by STM32L5 microcontrollers

19.4.2.1 Run Modes

By default, and after power-on or a system reset, STM32L MCUs are placed in *run* mode. The default clock source is set to the MSI, a power-optimized clock source that we have encountered in Chapter 10. STM32L microcontrollers offer to developers more fine-tune capabilities, which allow to reduce the power consumption in this mode. If we do not need too much computing power, then we can leave the MSI as the main clock source, avoiding the powering consumption introduced by the PLL multiplexer. By reducing the clock speed down to 24-26MHz, we can configure the *Dynamic Voltage*

[18]The power consumption values reported in **Figure 19.7** refer to the STM32L562 series with integrated SMPS.

Scaling (DVS) *scale 2* that decreases the *VCORE domain* down to 1.0V in more recent STM32L MCUs. This mode is also called *run range 2* and the overall power consumption can be further decreased by disabling the flash memory.



As said before, the flash in STM32L/G MCU and in some recent STM32F4 MCU (like the STM32F446) can be disabled even in the *run* mode. The CubeHAL function `HAL_FLASHEx_EnableRunPowerDown()` automatically performs this operation for us, while the `HAL_FLASHEx_DisableRunPowerDown()` routine enables again the flash memory. The only required condition is that this function, and all those other routines used when the flash is OFF (***interrupt vector* included**) are placed in SRAM, otherwise a *Bus Fault* occurs as soon as the flash is powered down. This can be easily performed creating a custom *linker script*, as we will see in Chapter 20. For this reason, ST engineers have collected these routines in a separated file named `stm32XXxx_hal_flash_ramfunc.c`.

Mode Name	HAL Function to enter power mode	Wake up condition	Effect on clocks	Main Voltage Regulator	Low-power regulator
Run	`HAL_PWREx_ControlVoltageScaling()`	N/A	None	ON	OFF
LP-Run	`HAL_PWREx_EnableLowPowerRunMode()`	N/A	None	OFF	ON
Sleep **Sleep-On-Exit**	`HAL_PWR_EnterSLEEPMode()`	Any interrupt (`WFI`)	CPU clock OFF no effect on other clocks or analog clock sources	ON	ON
		Wake up event (`WFE`)			
LP-Sleep	`HAL_PWR_EnterSLEEPMode()`	Any interrupt (`WFI`)	CPU clock OFF no effect on other clocks or analog clock sources	OFF	ON
		Wake up event (`WFE`)			
Stop 1	`HAL_PWREx_EnterSTOP0Mode()`	Any EXTI line (configured in the EXTI registers) Specific communication peripherals on reception events (USART, I2C)	All clocks OFF except LSI and LSE	ON	OFF
Stop 2	`HAL_PWREx_EnterSTOP1Mode()`			OFF	ON
Stop 3	`HAL_PWREx_EnterSTOP2Mode()`	The most of peripherals are OFF		OFF	ON

Standby + 32K SRAM	`HAL_PWREx_EnableSRAM2ContentRetention()`
	+
	`HAL_PWR_EnterSTANDBYMode()`
		WKUP pin rising edge, RTC alarm, external reset in NRST pin, IWDG reset	All clocks OFF except LSI and LSE	OFF	ON

Standby	`HAL_PWR_EnterSTANDBYMode()`			OFF	OFF
Shutdown	`HAL_PWREx_EnterSHUTDOWNMode()`	WKUP pin rising edge, RTC alarm, external reset in NRST pin	All clocks OFF except LSE	OFF	OFF

Table 19.2: Ten of the twelve power modes supported by STM32L5 MCUs

To further reduce the energy consumption when the system is in *run* mode, the internal voltage regulator can be configured in *low-power* mode. In this mode, the system frequency should not exceed 2 MHz. The `HAL_PWREx_EnableLowPowerRunMode()` function performs this operation automatically for us. In this mode we can eventually disable the flash memory, to further reduce the overall power consumption.

The *low-power run* mode represents the best compromise in STM32L MCUs from the energy efficiency point of view, as shown in **Figure 19.8**[19]. As you can see, enabling the ART accelerator increases performance but also reduces the dynamic consumption. Best consumption is most often reached when the *Instruction Cache* is ON, *Data Cache* is ON and *Prefetch Buffer* is OFF, as this configuration reduces the number of flash memory accesses.
The small flash dynamic consumption allows a small consumption each time the firmware needs to access the flash memory. Consumptions from SRAM1 and SRAM2 are quite similar, but SRAM2 is much more power efficient than SRAM1, when not remapped at address 0, thanks to its 0-wait state access.

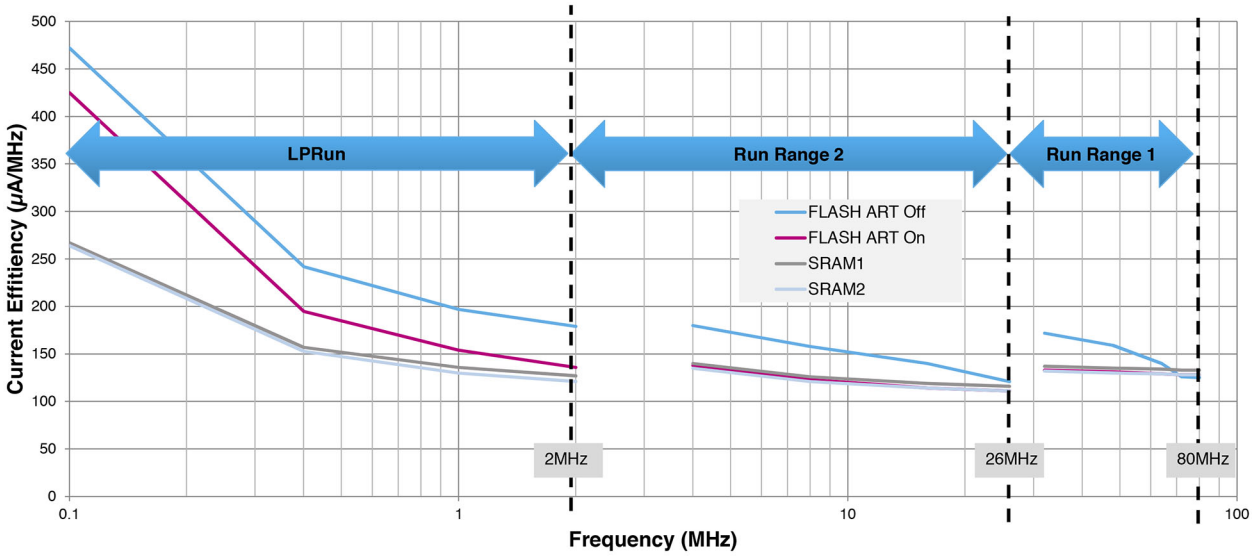


Figure 19.8: Power optimization vs frequency in STM32L4-series

19.4.2.2 Sleep Modes

Sleep modes allow all peripherals to be used, providing the fastest wakeup time at the same time. In these modes, the CPU is stopped, and each peripheral clock can be configured by software to be gated ON or OFF during the *sleep* and *low-power sleep* modes. These modes are entered by executing the assembler instructions `WFI` or `WFE`. To place the MCU in one of the two *sleep* modes, the CubeHAL provides the function:

```
void HAL_PWR_EnterSLEEPMode(uint32_t Regulator, uint8_t SLEEPEntry);
```

The first parameter, `Regulator`, can accept the values `PWR_MAINREGULATOR_ON` and `PWR_LOWPOWERREGULATOR_ON`: the former places the MCU in *sleep* mode, the latter in *low-power sleep* mode. The second parameter, `SLEEPEntry`, can assume the values `PWR_SLEEPENTRY_WFI` or `PWR_SLEEPENTRY_WFE`: as the names suggest, the first one performs a `WFI` instruction and the second one a `WFE`.

[19]The **Figure 19.8** is taken from this ST official document(https://bit.ly/3FwfEfA). ST also provides a useful application note, the AN4746(https://bit.ly/3FAKKCy), about power consumption optimization in STM32L4 MCUs.



Read Carefully

Please, take note that for STM32L MCUs the system frequency should not exceed MSI *range 1* value in this power mode. Please refer to product datasheet for more details on voltage regulator and peripherals operating conditions.

If the `WFI` instruction is used to enter in *sleep* mode, any peripheral interrupt acknowledged by the NVIC can wake up the device from *sleep* mode.
If the WFE instruction is used to enter *sleep* mode, the MCU exits *sleep* mode as soon as an event occurs. The wakeup event can be generated either by:

- enabling an interrupt in the peripheral control register but not in the NVIC, and enabling the `SEVONPEND` bit in the System Control Register - When the MCU resumes from `WFE`, the peripheral interrupt pending bit and the peripheral NVIC IRQ channel pending bit (in the NVIC interrupt clear pending register) have to be cleared;
- or configuring an external or internal EXTI line in event mode - When the CPU resumes from `WFE`, it is not necessary to clear the peripheral interrupt pending bit or the NVIC IRQ channel pending bit as the pending bit corresponding to the event line is not set.

After exiting the *low-power sleep* mode, the MCU is automatically placed in *low-power run* mode.

19.4.2.2.1 Batch Acquisition Mode

Batch Acquisition Mode (BAM) is an implicit and optimized mode for transferring data. Only the needed communication peripheral (e.g., the I²C one), one DMA and the SRAM are configured with clock enable in *sleep* mode. Flash memory is put in power-down mode and the flash memory clock is gated OFF during *sleep* mode. The MCU can enter either *sleep* or *low-power sleep* mode. Take note that the I²C clock can be set at 16 MHz even in *low-power sleep* mode, allowing support for 1MHz fast-mode plus. The USART and LPUART clocks can also be based on the HSI oscillator. Typical applications of BAM are sensor hubs.

19.4.2.3 Stop Modes

STM32L/G MCUs can provide up to 2 different *stop* modes, named *stop1* and *stop2*. *Stop* modes are based on the Cortex-M *deep sleep* mode combined with the peripheral clock gating. The voltage regulator can be configured either in normal[20] or low-power mode. In *stop1* mode, all clocks in the VCORE domain are stopped; the PLL, the MSI, the HSI16 and the HSE oscillators are disabled. Some peripherals with the wakeup capability (I²C, USART and LPUART) can switch ON the HSI16 to receive a frame and switch OFF the HSI16 after receiving the frame if it is not a wakeup frame. In this case, the HSI16 clock is propagated only to the peripheral requesting it. SRAM1, SRAM2 and register contents are preserved. Several peripherals can be functional in *stop1* mode: PVD, LCD controller,

[20] The HAL calls this mode *stop0*, and this achieved by calling the `HAL_PWREx_EnterSTOP0Mode()` function.

digital to analog converters, operational amplifiers, comparators, independent watchdog, LPTIM timers (if available), I²C, UART and LPUART. The *stop2* differs from the *stop1* mode by the fact that only the following peripherals are available: PVD, LCD controller, comparators, independent watchdog, LPTIM1, I2C3, and the LPUART.

The BOR is always available in both in *stop1* and *stop2* modes. The consumption is increased when thresholds higher than VBOR0 are used.

To place the MCU in *stop* mode the HAL provides the function:

```
void HAL_PWREx_EnterSTOPxMode(uint8_t STOPEntry);
```

where the 'x' is equal to 0, 1 and 2 depending on the stop mode. The parameter `STOPEntry` can assume the values `PWR_STOPENTRY_WFI` or `PWR_STOPENTRY_WFE`. For compatibility with the other HALs, the `HAL_PWR_EnterSTOPMode()` is also available.

To enter *stop* mode, all EXTI-line pending bits, all peripherals interrupt pending bits and RTC Alarm flag must be reset. Otherwise, the *stop* mode entry procedure is ignored, and program execution continues. *Stop1* mode can be entered from *run* mode and *low-power run* mode, while it is not possible to enter *stop2* mode from the *low-power run* mode.

Any EXTI-line configured in interrupt or event mode forces the CPU to exit from *stop* mode, according if it entered in low-power mode using the `WFI` or `WFE` instruction. Since both HSE and PLL are disabled before entering in *stop* mode, when exiting from this low-power mode the MCU source clock is set to the HSI. This means that our code shall reconfigure the clock tree according to wanted SYSCLK speed.

19.4.2.4 Standby Modes

STM32L/G MCUs provide two *standby* modes, which are based on the Cortex-M *deep sleep* mode. The *standby* mode is the lowest power mode in which 32 Kbytes of SRAM2 can be retained, the automatic switch from VDD to VBAT is supported and the I/Os level can be configured by independent pull-up and pull-down circuitry. By default, the voltage regulators are in power down mode and the SRAMs and the peripherals registers are lost. The 128-byte backup registers are always retained. The ultra-low-power BOR is always ON to ensure a safe reset regardless of the VDD slope.

To place the MCU in *standby* mode the HAL provides the function:

```
void HAL_PWR_EnterSTANDBYMode(void);
```

If we want to retain 32 Kbytes of SRAM2, then we can call the function:

```
void HAL_PWREx_EnableSRAM2ContentRetention(void);
```

before we call the `HAL_PWR_EnterSTANDBYMode();`

In STM32L microcontrollers each I/O can be configured with or without a pull-up or pull-down resistors, by calling the HAL function `HAL_PWREx_EnablePullUpPullDownConfig()`. This allows to control the inputs state of external components even during *standby* mode. For more information about this topic, refer to the reference manual of your MCU.

The microcontroller exits the *standby* mode when an external reset (NRST pin), an IWDG reset, a rising edge on one of the enabled WKUPx pins or an RTC event occurs. All registers are reset after wakeup from *standby* except for *Power Control/Status Register* (`PWR->CSR`). After waking up from *standby* mode, program execution restarts in the same way as after a reset (boot pin sampling, option bytes loading, reset vector is fetched, etc.). Using the macro:

```
__HAL_PWR_GET_FLAG(PWR_FLAG_SB);
```

we can check if the MCU is resetting due to an exit from *standby* mode. Since both HSE and PLL are disabled before entering in *stop* mode, when exiting from this low-power mode the MCU source clock is set to the HSI. This means that our code shall reconfigure the clock tree according to wanted SYSCLK speed.

19.4.2.5 Shutdown Mode

The *shutdown* mode is the lowest power mode with only 3.4 nA at 1.8 V in STM32L5 MCUs with RTC OFF. This mode is similar to the *standby* one but without any power monitoring: the BOR is disabled and the switch to VBAT is not supported in this mode. The LSI is not available, and consequently the independent watchdog is also not available. A Brown-Out Reset is generated when the device exits *shutdown* mode: all registers are reset except those in the backup domain, and a reset signal is generated on the pad. The 128-byte backup registers are retained in *shutdown* mode. When exiting *shutdown* mode, the wakeup clock is MSI at 4 MHz.

To enter *shutdown* mode the HAL provides the function:

```
void HAL_PWREx_EnterSHUTDOWNMode(void);
```

The microcontroller exits the *shutdown* mode when an external reset (NRST pin), a rising edge on one of the enabled WKUPx pins or an RTC event occurs. All registers are reset after wakeup from *standby*, including the *Power Control/Status Register* (`PWR->CSR`). After waking up from *shutdown* mode, program execution restarts in the same way as after a reset (boot pin sampling, option bytes loading, reset vector is fetched, etc.).

19.4.3 Power Modes Transitions

STM32L/G MCUs offer a lot of power modes. However, it is important to remark that it is not possible to reach every power mode starting from a given one, but the power mode transitions are limited.

The **Figure 19.9** shows the valid power mode transitions in an STM32L4 microcontroller. As you can see, from *run* mode, it is possible to access all low-power modes except the *low-power sleep* one. To go into *low-power sleep* mode, it is required to move first to *low-power run* mode and then to execute a `WFI` or `WFE` instruction while the regulator is the low-power one. On the other hand, when exiting *low-power sleep* mode, the STM32L4 is in *low-power run* mode.
When the device is in *low-power run* mode, it is possible to go into all low-power modes except *sleep* and *stop2* modes. *Stop2* mode can only be entered from the run one.
If the device enters in *Stop1* mode from the *low-power run* one, it will exit in *low-power run* mode.
If the device enters *standby* or *shutdown* from *low-power run* mode, it will exit in *run* mode.

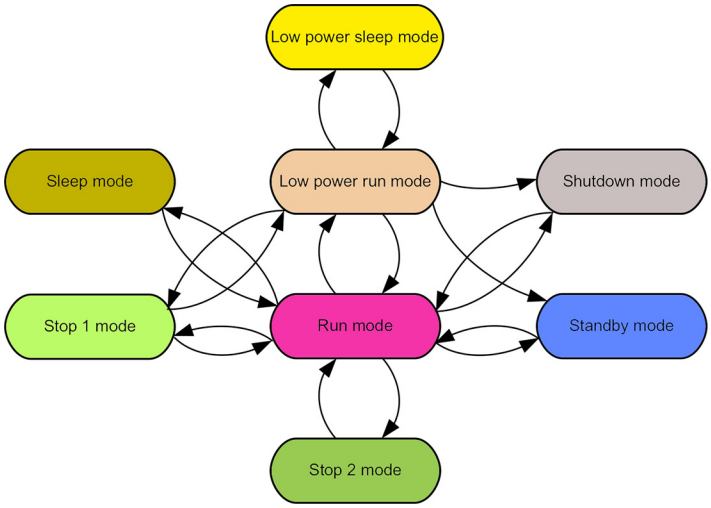


Figure 19.9: The allowable power mode transitions in an STM32L4 MCU

19.4.4 Low-Power Peripherals

Almost all more recent STM32 families (like STM32L, STM32G, STM32H and the latest STM32U) provide dedicated low-power peripherals. Here you can find a brief introduction to them.

19.4.4.1 LPUART

The *Low-Power UART* (LPUART) is an UART that allows bidirectional UART communications with limited power consumption. Only a 32.768 kHz LSE clock is required to allow UART communications up to 9600 baud/s. Higher baud rates can be reached when the LPUART is clocked by clock sources different from the LSE clock. Even when the microcontroller is in *stop* mode, the LPUART can wait for an incoming UART frame while having an extremely low-energy consumption. The LPUART includes all necessary hardware support to make asynchronous serial communications possible with minimum power consumption. It supports half-duplex single wire communications and modem operations (CTS/RTS). It also supports multiprocessor communications. DMA can be used for data transmission/reception even in *stop 2* mode.

To program the LPUART peripheral we use the same functions from the `HAL_UART` module.

19.4.4.2 LPTIM

The *Low-Power Timer* (LPTIM) is a 16-bit timer that benefits from the ultimate developments in power consumption reduction. Thanks to its diversity of clock sources, the LPTIM can keep running whatever the selected power mode, different from standard STM32 timers that do not run during *stop* modes. Given its capability to run even with no internal clock source, the LPTIM can be used as a *pulse counter* which can be useful in some applications. Moreover, the LPTIM capability to wake up the system from low-power modes makes it suitable to realize timeout functions with extremely low-power consumption. In Chapter 23 about FreeRTOS, we will use the LPTIM timer as source timebase for *tickless idle* mode. The LPTIM introduces a flexible clock scheme that provides the needed functionalities and performances, while minimizing the power consumption.

These are the relevant features of a LPTIM peripheral:

- 16 bit upcounter
- 3-bit prescaler with 8 possible dividing factor (1,2,4,8,16,32,64,128)
- Selectable clock source
  - Internal clock sources: LSE, LSI, HSI16 or APB clock
  - External clock source over ULPTIM input (working with no LP oscillator running, used by pulse counter application)
- 16 bit period register
- 16 bit compare register
- Continuous/one shot mode
- Selectable software/hardware input trigger
- Configurable output: Pulse, PWM
- Configurable I/O polarity
- Encoder mode

To program an LPTIM timer we use the dedicated `HAL_LPTIM` module.

19.4.4.3 LPGPIO

ST introduced in Q321 a new STM32 series addressed to *Ultra low-power* application: the STM32U. The STM32U5 family offers advanced power-saving microcontrollers, based on Cortex-M33 to meet the most demanding power/performance requirements for smart applications, including wearables, personal medical devices, home automation, and industrial sensors. At the time of writing this Chapter, still no development boards are available on the market nor the STM32U5 CubeHAL has been released.

STM32U5 family provides new low-power peripherals, compared to STM32L/G families. The *Low-Power GPIO* (LPGPIO) allows the I/O control in *stop* mode (down to *stop 2* mode), using DMA in memory-to-memory transfer mode. LPGPIO is designed to be used in conjunction with the GPIO.

The main LPGPIO feature can be summarized in the following ones:

- 16 I/Os control in low-power modes down to Stop 2 mode.
- Secure clock and reset management.
- Output data from output data register (LPGPIO_ODR).
- Input data to input data register (LPGPIO_IDR).
- Bit set and reset register (LPGPIO_BSRR) for bitwise write access to LPGPIO_ODR.
- TrustZone security support.

19.4.4.4 LPDMA

The STM32U-series provides a DMA able to work even in *stop 3* mode allowing transfers between memory and peripherals while the Cortex-M core is halted. Moreover, the LPDMA is able to perform peripheral-to-peripheral transfers, with autonomous data transfers during *sleep* and *stop* modes.

19.5 Power Supply Supervisors

The majority of STM32 microcontrollers provide two power supply supervisors: BOR and PVD. The *Brownout Reset* (BOR) is a unit that keeps the microcontroller under reset until the supply voltage reaches the specified VBOR threshold. VBOR is configured through device option bytes. By default, BOR is OFF. The user can select between three to five programmable VBOR threshold levels. For full details about BOR characteristics, refer to the "Electrical characteristics" section in the device datasheet. STM32 devices that do not provide a BOR unit, usually have a similar unit named *Power on Reset* (POR)/*Power Down Reset* (PDR), which perform the same operation of the BOR unit but with a fixed and factory-configured voltage threshold.

The power supply can be actively monitored by the firmware by using the *Programmable Voltage Detector (PVD)*. The PVD allows to configure a voltage to monitor, and if this VDD is higher or lower than the given level, a corresponding bit in the *Power Control/Status Register* (`PWR->CSR`) is set. If properly configured, the MCU can generate a dedicated IRQ through the EXTI controller. To enable/disable the PVD in those MCUs with this features, the HAL provides the functions `HAL_PWR_EnablePVD()`/`HAL_PWR_DisablePVD()`, while to configure the voltage level it provides the function `HAL_PWR_ConfigPVD()`. For more information, refer to the `HAL_PWREx` module of the CubeHAL.

Both BOR/POR/PDR and PVD actively monitor the VDD comparing it to given threshold levels. Recent STM32L microcontrollers provide four *Peripheral Voltage Monitoring* (PVM) to monitor other peripheral power domains. Each of the four PVMx is a comparator between a fixed threshold VPVMx and the selected power supply. **Table 19.3** summarizes the PVMx features, including the power domain monitored and the voltage levels.
Each PVM output is connected to an EXTI line and can generate an interrupt if enabled through the EXTI registers. This IRQ is shared with PVD in those STM32 MCUs with PVM suppport. The PVMx output interrupt is generated when the independent power supply drops below the PVMx threshold and/or when it rises above the PVMx threshold, depending on EXTI line rising/falling edge configuration. Each PVM can remain active in *stop 0*, *stop 1* and *stop 2* modes, and the PVM interrupt can wake up from any *stop* mode.

Table 19.3: PVMx features

PVM	Power supply	PVM threshold	EXTI line
PVM1	VDDUSB	VPVM1 (around 1.2 V)	35
PVM2	VDDIO2	VPVM2 (around 0.9 V)	36
PVM3	VDDA	VPVM3 (around 1.65 V)	37
PVM4	VDDA	VPVM4 (around 1.8 V)	38

To configure the PVM, the CubeHAL provides the `HAL_PWREx_ConfigPVM()` routines, while to selectively enable/disable one of the PVMx it provides the `HAL_PWREx_EnablePVMx()`/`HAL_PWREx_DisablePVMx()`. For more information, refer to the CubeL5 documentation.

19.6 Debugging in Low-Power Modes

By default, the debug connection is lost if the application puts the MCU in *sleep*, *stop* and *standby* modes while the debug features are used. This happens because the Cortex-M core is no longer clocked. However, by setting some configuration bits in the `DBGMCU_CR` register of the *MCU debug component* (DBGMCU), the software can be debugged even when using the low-power modes extensively.

The CubeHAL provides convenient functions to enable/disable debug mode in low-power modes. The function `HAL_DBGMCU_EnableDBGSleepMode()` is used to enable debugging during *sleep* mode[21]; the functions `HAL_DBGMCU_EnableDBGStopMode()` and `HAL_DBGMCU_EnableDBGStandbyMode()` allow to use debug interface during *stop* and *standby* modes respectively.

It is important to remark that, if we want to debug the MCU in low-power modes, we also have to leave ON the GPIO peripherals corresponding to SWDIO/SWO/SWCLK pins. In all Nucleo boards these pins coincide with PA13, PA14 and PB3.



Please, take note that, before enabling MCU debugging in low-power modes, DBGMCU interface must be enabled by calling the `__HAL_RCC_DBGMCU_CLK_ENABLE()` macro.

19.7 Using the CubeMX Power Consumption Calculator

It may be a nightmare to manually estimate the power consumption of a microcontroller, with several peripheral enabled and several transition states in its different power modes. Even if MCU datasheets provide all necessary information, it is hard to figure out the exact power consumption levels.

[21] Debugging during the *sleep* mode is not available in STM32F0 microcontrollers and hence the corresponding HAL function is not provided by the HAL.

CubeMX provides a convenient tool, named *Power Consumption Calculator* (PCC), which allows us to build a power sequence and to perform estimations of the MCU power consumption.

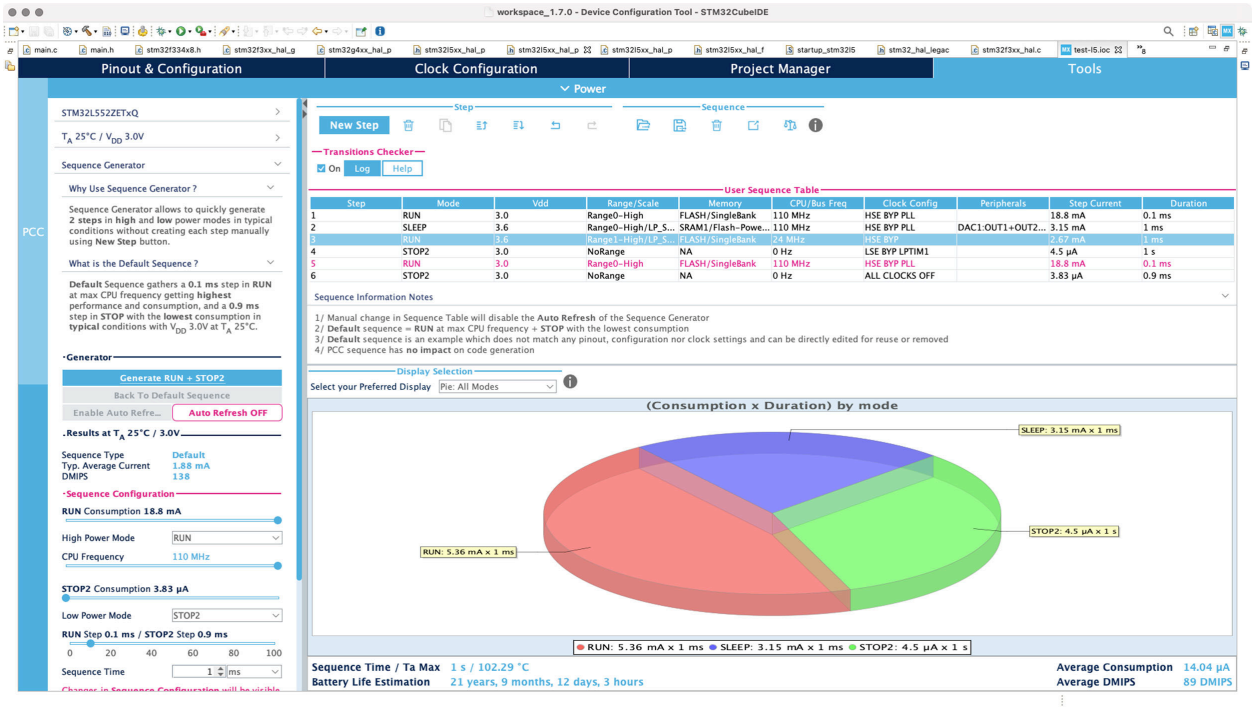


Figure 19.10: The *Power Consumption Calculator* main view

The **Figure 19.10** shows the main PCC view. To use it we have to first select the *Vdd Power Supply* source, otherwise the tool does not allow us to create steps in the power sequence. The next optional step consists in selecting a battery used to power the MCU when the main power is absent. This is useful to evaluate the battery life. We can choose from a portfolio of well-known batteries, or eventually add a custom one.

By clicking on the green **New Step**, we can add a step of the sequence. Here we can specify the power mode (*run*, *sleep*, etc), the memories configuration (flash enabled/disabled, ART enabled/disabled, and so on) and the power voltage level. From the same dialog we can also choose the CPU frequency, the duration of the step and the enabled peripherals.

With this tool we can so figure out how much power is needed by the microcontroller. In STM32L/G MCUs is also possible to enable the **Transitions Checker**, which allows to identify invalid transition states (for example, we cannot switch from the *run* mode to the *low-power sleep* one without passing from the *low-power run* mode). For more information about the PCC view refer to the UM1718[22] from ST.

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

19.8 A Case Study: Using Watchdog Timers With Low-Power Modes

Both IWDG and WWDG timers cannot be stopped once started. The WWDG timer keeps counting until the *stop* mode, while the IWDG timer, being clocked by the LSI oscillator, works even in *shutdown* mode. This means that watchdog timers prevent the MCU from staying in low-power mode for a long time.

If you need to use both watchdog timer and low-power modes in your application, then you need to follow this trick based on the fact that the content of the SRAM memory survives to successive resets (clearly, it does not survive to a power-on reset). So, to keep track of a reset caused by a watchdog timer while staying in a low-power mode, you can use a variable that keeps track of this fact (for example, you set the content of an `uint32_t` variable to a special "key" value before entering in a low-power mode). Once the MCU resets, you can check the content of this variable, and you can avoid starting the watchdog timer if that variable is configured accordingly.

However, we need a "safe" place to store this variable, otherwise it is likely to be overwritten by startup routines. So, the best thing to do is to reduce the size of the SRAM region inside the `mem.ld` file, and to place this sentinel variable at the end of the SRAM memory, where usually the main stack starts:

```
volatile uint32_t *lpGuard = (0x20000000 + SRAM_SIZE);
```

For example, assuming an STM32F030R8 MCU with 8KB of SRAM, and assuming that we define the SRAM region in the `mem.ld` file in the following way:

```
MEMORY {
   FLASH (rx)  : ORIGIN = 0x08000000, LENGTH = 64K
   SRAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 8K - 4
}
```

we have that the macro `SRAM_SIZE` will be equal to `0x2000-4 = 0x1FFC`. The content of the `lpGuard` variable will be so placed at the address `0x2000 1FFC`

I am aware of the fact that these concepts may look totally obscure. A lot of things will be clarified once you read the next chapter about the memory layout of an STM32 application in the next chapter.

20. Memory layout

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

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

20.1 The STM32 Memory Layout Model

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

20.1.1 Flash Memory Typical Organization

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

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

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

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

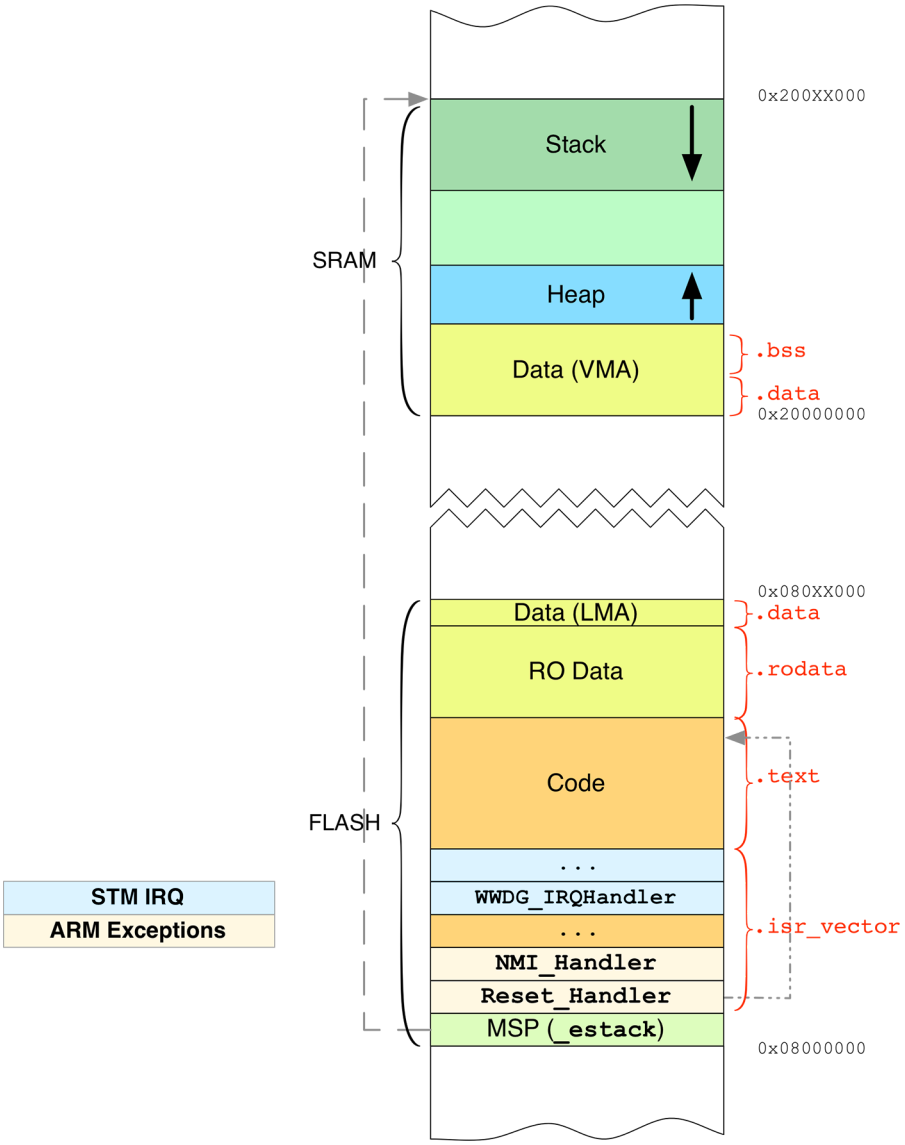


Figure 20.1: The typical layout of flash and SRAM memories

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

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

20.1.2 SRAM Memory Typical Organization

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

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

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

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

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

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

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

20.1.3 Understanding Compilation and Linking Processes

The process that goes from the compilation of the C source code to the generation of the final binary image to flash on our MCU involves several steps and tools provided by the GCC tool-chain. The **Figure 20.2** tries to outline this process. All starts from the C source files. They usually contain the following program structures.

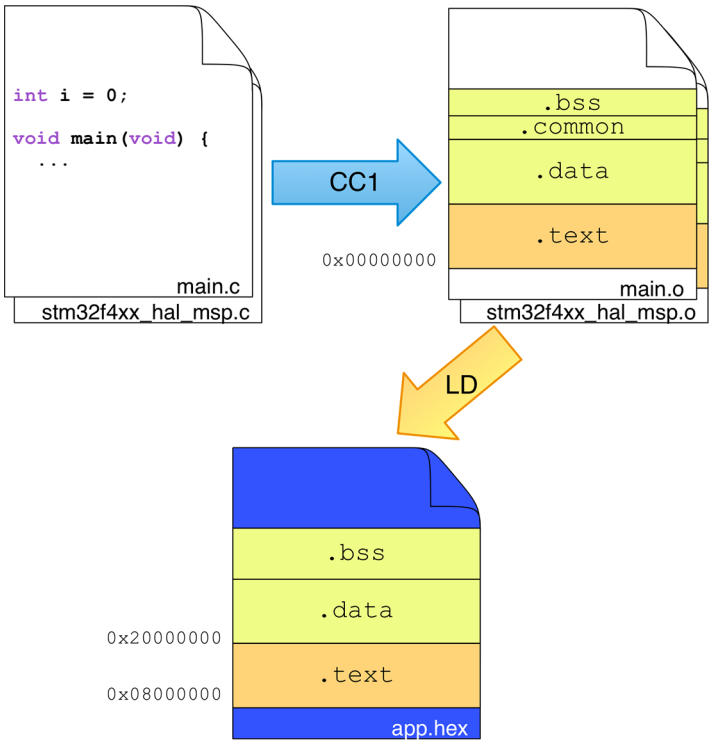


Figure 20.2: The compilation process from the source file to the final binary image

- *Global variables*: these can be in turn divided between un-initialized and initialized variables; a global variable can also defined as `static`, that is its visibility is limited to the current source file.
- *Local variables*: these can be divided between simple local (also called *automatic*) variables and static local variables (that is those variables whose lifetime extends across the entire run of the program).
- *Const data*: these can be in turn divided between const data types (e.g. `const int c = 5`) and string constants (e.g. `"Hello World!"`).
- *Routines*: these constitute the program and they will be translated in assembly instructions.
- *External resources*: these are both global variables (**declared** as `extern`) and routines **defined** in other source files. It will be a linker job to "link" the references to these symbols defined in other source files and to merge the sections coming from the corresponding binary files.

Once a source file is compiled, the above program structures are mapped inside specific sections of the binary file. The **Table 20.1** summarizes the most relevant ones.

Table 20.1: The mapping of program structures and binary file sections

Language structure	Binary file section	Memory region at run-time
Global un-initialized variables	`.common`	Data (SRAM)
Global initialized variables	`.data`	Data (SRAM+Flash)
Global `static` un-initialized variables	`.bss`	Data (SRAM)
Global `static` initialized variables	`.data`	Data (SRAM+Flash)
Local variables	<no specific section>	Stack or Heap (SRAM)
Local `static` un-initialized variables	`.bss`	Data (SRAM)
Local `static` initialized variables	`.data`	Data (SRAM+Flash)
Const data types	`.rodata`	Code (Flash)
Const strings	`.rodata.1`	Code (Flash)
Routines	`.text`	Code (Flash)

For every source file (`.c`) composing our application, the compiler will generate a corresponding *object file* (`.o`), which contains the sections in **Table 20.1**[3]. An *object file* is a type of binary file that adheres to a well-known standard. There are a lot of standards for binary files around (PE, COFF, ELF, etc.). The one used by GCC ARM is the ELF32, an open standard really popular, due its usage in Linux-based Operating Systems, and it is widely supported even by other tools like ST-LINK GDB Server and the STM32CubeProgrammer. File ending with `.o`[4] are, however, a special type of object files. These are also known as *relocatable files*. This name comes from the fact that all the memory addresses contained in this type of file are **relative** to the same file and starts from the `0x0000 0000` address. This means that also `.text` section will start from that address, and we know that this is in contrast with the starting address of flash memory (`0x0800 0000`) in an STM32 MCU[5].

Starting from a series of *relocatable files* (plus some other configuration files that we will see in a while), the linker will assemble their content to form one common object file that will represent our firmware to flash on the MCU. In this process, called *linking*, the linker will *relocate* all relative addresses to the actual memory addresses. This type of file is also known as *absolute file*, because all addresses are **absolute** and **specific** of the given STM32 MCU[6].

How does the linker know where to place in memory the sections contained in the absolute file? It is thanks to *linker scripts* (those files ending with `.ld` in the root STM32CubeIDE project folder - for example **STM32F072RBTX_FLASH.ld** for an STM32F072RB MCU) that we can arrange the content of the absolute file according to the actual memory layout.

[3] It is important to underline that an object file contains much more sections. The most of them are related to debugging, and contain relevant information like the original source code, all the symbols contained in the source file (even those that have been optimized by the compiler), and so on. However, for the purposes of this discussion, it is better to leave them out.

[4] Take in account that, from the compiler point of view, the file extension is just a convention.

[5] Those of you that want to deepen this matter can take a look at the `readelf` tool provided in the GCC ARM tool-chain.

[6] Here, again, the story is more complex. First of all, the linker could assemble other pieces from several external statically linked libraries (those ending with `.a`). These libraries, also known as *archive files*, are nothing more than a merge of several *relocatable files*. During the linking process, only those program structures used in our application will be merged with the final firmware. Another important aspect to remark is that this process is essentially the same for every microprocessor platform (like the x86 and so on), and it is also called *static linking*. More powerful architectures face an advanced linking process, also known as *dynamic linking*, which postpones the linking when the program will be loaded in the OS process. This allows to dramatically reduce the size of executables, and to update the dependency libraries without recompiling the whole application. In *dynamic linking* libraries are called *shared objects* (or *shared libraries*, or DLL in Windows), and in modern Operating Systems it is possible to share the same `.text` section from these libraries among the processes that use them by using `mmap()` or similar *system calls*. This allows reducing disk space as well the SRAM occupation of processes (think to the tons of system libraries that should be "replicated" among the several processes running on a modern PC).

Since it may be really hard to study the content of those script files if we have not mastered several concepts before, it is better to start smoothly creating a bare bone STM32 application.

20.2 The Really Minimal STM32 Application

The majority of applications seen until now seem really simple. Instead, both from the memory organization point of view and from the operations performed when the MCU boots, they already execute a lot of operations under the hood. For this reason, we are going to build a really essential application.

The first step is creating an empty project using STM32CubeIDE. Go to **File->New->C/C++ Project** menu. In the next dialog choose the **C Managed Build** type and click on **Next**. In the next dialog select the **Executable->Empty Project** in the **Project type** tree-view and **MCU ARM GCC** in the **Toolchains** section, as shown in **Figure 20.3**. Chose a Project name and click on **Next**. Skip the next wizard step and go to the **Select default target for the project**. Select the MCU for your development board by clicking on the **Select** button and click on **Finish**.

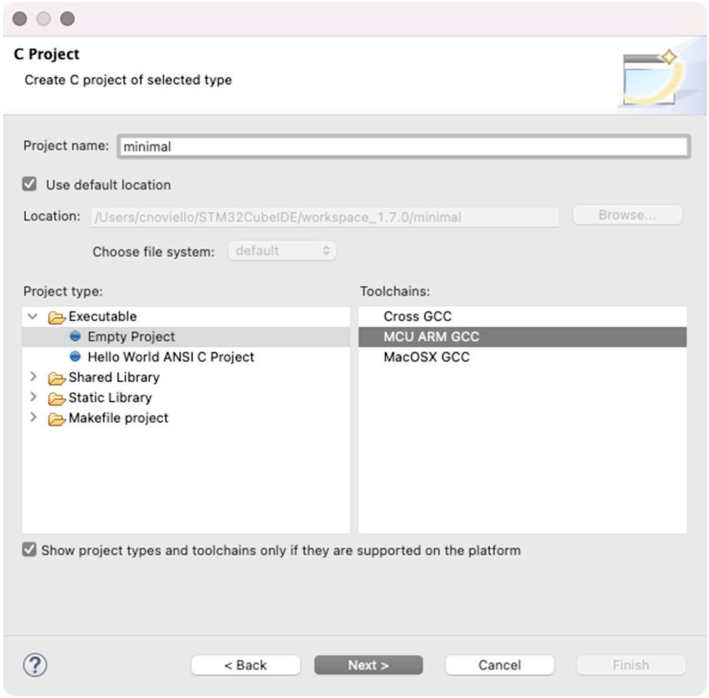


Figure 20.3: The project settings to choose to generate a minimal STM32 application

Create now a new C file named **main.c** (**File->New->Source File**) and place the following code inside it[7].

[7]This code is designed to work with the Nucleo-F401RE. Refer to the book examples for the other Nucleo boards.

Filename: `/main-ex1.c`

```c
typedef unsigned long uint32_t;

/* Memory and peripheral start addresses (common to all STM32 MCUs) */
#define FLASH_BASE      0x08000000
#define SRAM_BASE       0x20000000
#define PERIPH_BASE     0x40000000

/* Work out end of RAM address as initial stack pointer
 * (specific of a given STM32 MCU */
#define SRAM_SIZE       96*1024     // STM32F401RE has 96 KB of RAM
#define SRAM_END        (SRAM_BASE + SRAM_SIZE)

/* RCC peripheral addresses applicable to GPIOA
 * (specific of a given STM32 MCU */
#define RCC_BASE        (PERIPH_BASE + 0x23800)
#define RCC_APB1ENR     ((uint32_t*)(RCC_BASE + 0x30))

/* GPIOA peripheral addresses
 * (specific of a given STM32 MCU */
#define GPIOA_BASE      (PERIPH_BASE + 0x20000)
#define GPIOA_MODER     ((uint32_t*)(GPIOA_BASE + 0x00))
#define GPIOA_ODR       ((uint32_t*)(GPIOA_BASE + 0x14))

/* User functions */
int main(void);
void delay(uint32_t count);

/* Minimal vector table */
uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
    (uint32_t *)SRAM_END,   // initial stack pointer
    (uint32_t *)main        // main as Reset_Handler
};

int main() {
    /* Enable clock on GPIOA peripheral */
    *RCC_APB1ENR = 0x1;
    /* Configure the PA5 as output pull-up */
    *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1

    while(1) {
        *GPIOA_ODR = 0x20;
        delay(200000);
        *GPIOA_ODR = 0x0;
        delay(200000);
    }
}
```

```
47
48 void delay(uint32_t count) {
49     while(count--);
50 }
```

The first 21 lines contain just macros that define the most common STM32 peripheral addresses. Some are generic and some specific of the given MCU. At line 26 we are defining the *vector table*. Being "minimal", it just contains two things: the address in SRAM of the MSP (remember that this it the first entry of the *vector table* and it must be placed at `0x0800 0000` address) and the pointer to the handler of the *Reset* exception. What exactly are we doing?

In Chapter 7 we mentioned that when the MCU resets, the NVIC controller generates a *Reset* exception after few cycles. This means that the corresponding ISR is the real entry point of our application, and the execution of the firmware starts from there. Here we are going to define the `main()` function as the handler of *Reset* exception. The GCC keyword `__attribute__((section(".isr_vector")))` says to the compiler to place the `vector_table` array inside the section named `.isr_vector`, which in turn will be contained in the object file `main.o`. Finally the `main()` routine contains nothing more than the classical blinking LED application.

Once we are ready with the C source file, we can move further with the GNU LD linker script. Create a new file name **LinkerScript.ld** (**File->New->Other**... and then select **General->File**) and put the following content in it.

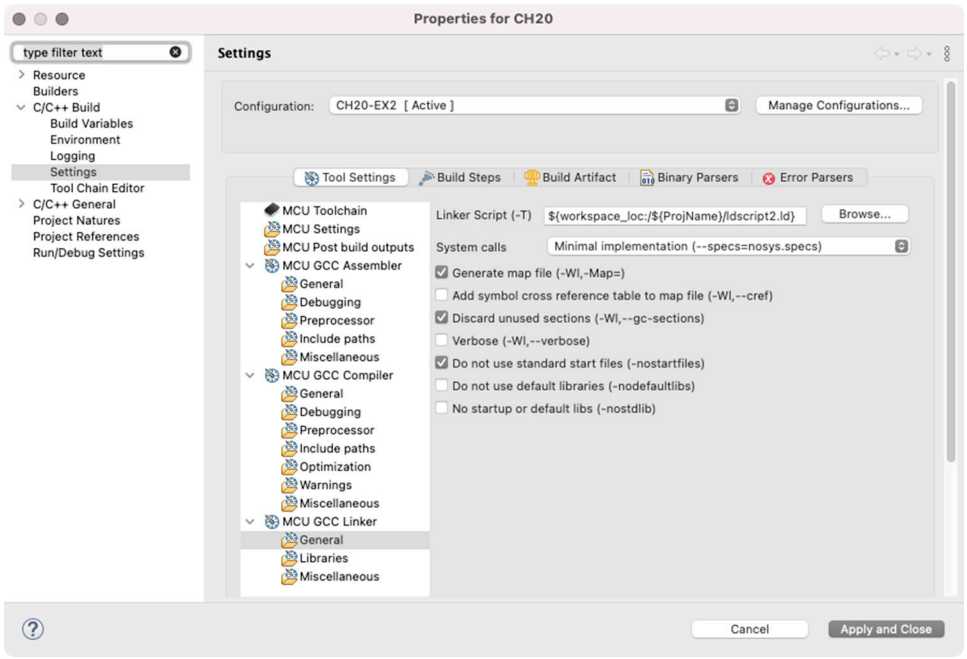


Figure 20.4: The project settings to setup the LD configuration script

Filename: `ldscript.ld`

```
1  /* memory layout for an STM32F401RE */
2
3  MEMORY
4  {
5      FLASH (rx)  : ORIGIN = 0x08000000, LENGTH = 512K
6      SRAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 96K
7  }
8
9  ENTRY(main)
10
11 /* output sections */
12 SECTIONS
13 {
14     /* Program code into FLASH */
15     .text : ALIGN(4)
16     {
17         *(.isr_vector)    /* Vector table */
18         *(.text)          /* Program code */
19         KEEP(*(.isr_vector))
20     } >FLASH
21
22     /* Initialized global and static variables (which
23        we don't have any in this example) into SRAM */
24     .data :
25     {
26         *(.data)
27     } >SRAM
28 }
```

Let us see the content of this file. Lines [3:7] contain the definition of the flash and SRAM memories. Each region can have several attributes (**w**=writable, **r**=readable, **x**=executable). We also specify their starting address and length (in the above example they are related to an STM32F401RE MCU). Line 9 specifies the `main()` function as the entry point of our application[8]. Lines [12:28] define the content of the `.text` and `.data` sections. The `.text` section will be composed first by the *vector table* and then by the program code. With the `ALIGN(4)` directive we are saying that the section is word (4 bytes) aligned, while the `>FLASH` directive specifies that the `.text` section will be placed inside the flash memory. The `KEEP(*(.isr_vector))` says to LD to keep the *vector table* inside the final absolute file, otherwise the section could be "stripped" by other tools that perform optimizations on the final file. Finally, the `.data` section is also defined (even if does not contain nothing in this example), and it is placed inside the SRAM memory.

The project generated by the STM32CubeIDE is already configured to pass to the GNU LD a script file named **LinkerScript.ld**. In case you need to change the file name, go to **Project Properties**-

[8] The `ENTRY()` directive is meaningless in embedded applications, where the actual entry point corresponds to the handler of the *Reset* exception. However, it may be informative for debuggers and simulators, and for this reason you will find it in ST official LD linker scripts.

>C/C++ Build->Settings->MCU GCC Linker->General. You will find the entry **Linker Script (-T)** with this setting: `${workspace_loc:/${ProjName}/LinkerScript.ld}`, as shown in **Figure 20.4**. Moreover, to compile the examples in this chapter without errors, it is important to set the option **Do not use standard start files (–nostartfiles)** as shown in **Figure 20.4**. This will avoid that linker will add to the final binary file the *C Run-Time* (CRT) initialization routine (`_mainCRTStartup()`), which expects a more informative linker script to perform some initializations at start-up.

Congratulation: it is almost impossible to have a smaller STM32 application[9].

20.2.1 ELF Binary File Inspection

An ELF binary file can be inspected using a series of tools provided by the GNU MCU tool-chain[10]. `objdump` and `readelf` are the most common ones. Describing their usage is outside the scope of this book. However, it is strongly suggested to dedicate a little of time playing with their optional parameters to the command-line. Understanding how a binary file is made can dramatically improve the knowledge of what under the hood. For example, running `objdump` with the `-h` parameter shows the content of all sections contained in the firmware binary[11].

```
# ~/gcc-arm/bin/arm-none-eabi-objdump -h CH20.elf
CH20.elf:     file format elf32-littlearm
Sections:
Idx Name            Size      VMA       LMA       File off  Algn
  0 .text           00000008  08000000  08000000  00008000  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text.main      00000040  08000008  08000008  00008008  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text.delay     00000020  08000048  08000048  00008048  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .comment        00000070  00000000  00000000  0000b1d2  2**0
                    CONTENTS, READONLY
  4 .ARM.attributes 00000033  00000000  00000000  0000b242  2**0
                    CONTENTS, READONLY
```

Looking to the above output we see several things regarding the sections contained in the binary file. Every section has a `size`, expressed in bytes. A section has also two addresses: the *Virtual Memory Address* (`VMA`) and the *Load Memory Address* (`LMA`). In embedded systems like the STM32 MCUs, the VMA is the address a section will have when the firmware starts execution. The LMA is the address at which the section will be loaded. In most cases the two addresses will be the same. As we will discover in the next paragraph, they differ for the `.data` region.

Every section has several attributes that say to the loader (in our case, for example, the loader is GDB in conjunction with ST-LINK GDB Server, or the STM32CubeProgrammer tool) what to do with the given section. Let us see what they mean:

[9] Ok, coding it in assembly will allow you to save additional space, but this book is not for masochists ;-D

[10] https://bit.ly/3AiMj4e

[11] When you run the command, you will see much more sections all related to debugging. Here you will not see them because the debug information has been "stripped" from the file using the `arm-none-eabi-strip` command.

- `CONTENTS`: this attribute says to the loader that the section in the binary file contains data to load in the final LMA address. As we will see next, the `.bss` section does not have content in a binary file.
- `ALLOC`: this says to allocate a corresponding space in the LMA memory (which could be both flash and SRAM memory). The dimension of the space allocated is given by the `Size` column.
- `LOAD`: this indicates to load the data from the section contained in the binary file to the final LMA memory.
- `READONLY`: this indicates that the content of the section is read-only.
- `CODE`: this indicates that the content of the section is binary code.

Another interesting thing to remark from the previous output is that the binary file contains a dedicated section for every callable contained in the source code (`.text.main` for the `main()` and `.text.delay` for `delay()`). We have to specify to the linker to merge all the `.text` sections in a whole unique section, modifying the linker script in this way:

```
.text : ALIGN(4)
{
    *(.isr_vector)    /* Vector table */
    *(.text)          /* Program code */
    *(.text*)         /* Merge all .text.* sections inside the .text section */
    KEEP(*(.isr_vector))
} >FLASH
```

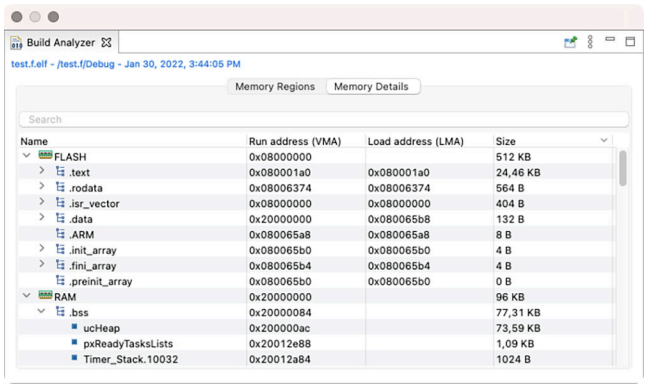
As we will see later, the ability to have separated sections for every callable allow us to selectively place some functions inside different memories (for example, the fast CCM memory in some STM32 MCUs).

Finally, the `File off` column specifies the offset of the section inside the binary file, while the `Algn` column indicates the data align in memory, which is 4-bytes.



STM32CubeIDE Build Analyzer

The STM32CubeIDE provides a very useful tool to visually inspect the ELF binary file. It is called Build Analyzer and it is available as optional view by going to **Window->Show view->Build Analyzer**. This view provides two tabs. The second one, called **Memory Detail** contains detailed program information based on the ELF file. The different linker section names are presented with address and size information. Each section can be expanded and collapsed. When a section is expanded, functions/data in this section is listed. Each presented function/data contains address and size information. For more information regarding the **Build Analyzer**, refer to the chapter dedicated to advanced debugging.



The Build Analyzer view

20.2.2 `.data` and `.bss` Sections Initialization

Let us introduce a minor modification to the previous example.

```
36  volatile uint32_t dataVar = 0x3f;
37
38  int main() {
39      /* enable clock on GPIOA and GPIOC peripherals */
40      *RCC_APB1ENR = 0x1 | 0x4;
41      *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1
42
43      while(dataVar == 0x3f) { // This is always true
44          *GPIOA_ODR = 0x20;
45          delay(200000);
46          *GPIOA_ODR = 0x0;
47          delay(200000);
48      }
49  }
```

This time we use a global initialized variable, `dataVar` to start the blinking loop. The variable has been declared `volatile` just to avoid that the compiler optimizes it (however, when compiling this example, disable all optimizations [-ON] in the project settings). Looking at the code, we can reach to the conclusion that it does the same thing of the previous example. However, if you try to flash your Nucleo, you will see that the LD2 LED does not blink. Why not?

To understand what's happening, we have to review some things from the C programming language. Consider the following code fragment:

```
...
uint32_t globalVar = 0x3f;

void foo() {
        volatile uint32_t localVar = 0x4f;

        while(localVar--);
}
```

Here we have two variables: one defined at global scope, one locally. The `localVar` variable is initialized to the value `0x4f`. When does this exactly happen? The initialization is executed when the **foo()** routine is invoked, as shown by the following assembly code:

```
1  void foo() {
2     0:   b480            push    {r7}            ;Save the current FP
3     2:   b083            sub     sp, #12         ;Allocate 12 bytes on the stack
4     4:   af00            add     r7, sp, #0      ;Save the new FP
5          volatile uint32_t localVar = 0x4f;
6     6:   234f            movs    r3, #79         ;Place 0x4f in r3
7     8:   607b            str     r3, [r7, #4]    ;Store r3 (that is 0x4f) in the 4-th byte
8
9          while(localVar--);
10    a:   bf00            nop
11    c:   687b            ldr     r3, [r7, #4]
12    e:   1e5a            subs    r2, r3, #1
13   10:   607a            str     r2, [r7, #4]
14   12:   2b00            cmp     r3, #0
15   14:   d1fa            bne.n   c <foo+0xc>
16 }
```

Line 0 saves the current *Frame Pointer* (FP)[12] on the stack. Lines [2:4] are the function *prolog*. Each routine is responsible of allocating its own stack frame, saving some CPU internal registers. This is also called *calling convention*, and the way this is performed is defined by a specific standard (in case of ARM based processors, it is defined by the *ARM Architecture Procedure Call Standard* (AAPCS)). We will not go into details of this matter here, because we will better analyze the ARM calling convention in Chapter 24.

The instructions we are interested in are those at lines [5:6]. Here we are storing the value `0x4f` (which is 79 in base 10) inside the general-purpose register R3 and then moving its content inside the second word in the stack, which corresponds to the `localVar` variable [13].

The remaining part of the assembly code contains the `while(localVar--)` and the function *epilog* (not shown here), which is responsible of restoring the state before going back to the caller function.

[12] The *Frame Pointer* is a special pointer to separate the function parameters by the local variables. More details about this register can be found here(https://bit.ly/1ngLrop).

[13] It is important to clarify that the above assembly code is generated with all optimizations disabled.

So, the calling convention defines that local variables are automatically initialized upon function call. What about global variables? Since they are not involved in a calling process, they need to be initialized by some specific code when the MCU resets (remember that the SRAM is volatile, and its content is undefined after a reset). This means that we have to provide a specific initialization function.

The following routine can be used to simply copy the content of the flash region containing the initialization values to the SRAM region dedicated to global initialized variables.

```
void __initialize_data (unsigned int* flash_begin, unsigned int* data_begin,
                        unsigned int* data_end) {
  unsigned int *p = data_begin;
  while (p < data_end)
    *p++ = *flash_begin++;
}
```

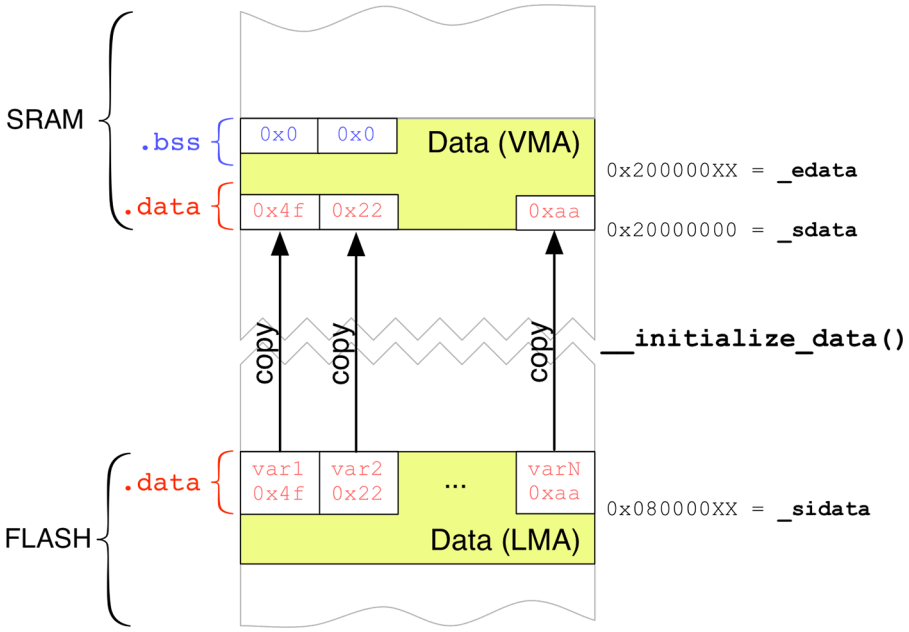


Figure 20.3: The copy process of initialized data from the flash to the SRAM memory

Before we can use this routine, we need to define few other things. First of all, we need to instruct LD to store the initialization values for each variable contained in the `.data` section inside a specific region of the flash memory, which will correspond to the LMA memory address. Second, we need a way to pass to the `__initialize_data()` function the start and the end of `.data` section in SRAM (that we are going call `_sdata` and `_edata` respectively) and the starting location (that we are going to call `_sidata`) where initialization values are stored in the flash memory (it is important to stress that when we initialize a variable to a given value we need to store that value somewhere in the flash, and use it to initialize the SRAM location corresponding to the variable). The **Figure 20.3** schematizes this process.

Once again, all these operations can be performed using the linker script, which we can modify in the following way:

```
25 /* Used by the startup to initialize data */
26 _sidata = LOADADDR(.data);
27
28 .data : ALIGN(4)
29 {
30     . = ALIGN(4);
31     _sdata = .;        /* create a global symbol at data start */
32
33     *(.data)
34     *(.data*)
35
36     . = ALIGN(4);
37     _edata = .;        /* define a global symbol at data end */
38 } >SRAM AT>FLASH
```

The instruction at line 26 defines the symbol `_sidata`, which will contain the LMA address of the `.data` section (that is, the starting address of flash memory containing initialization values). Instructions at line [30:31] use a special operator: the "." operator. It is named *location counter* and it is a counter that keeps track of the memory location reached during the generation of each section. The *location counter* independently counts location memories of every memory region (SRAM, flash and so on). For example, in the above code, it starts from `0x2000 0000` since the `.data` section is the first one loaded in SRAM. When the two instructions `*(.data)` and `*(.data*)` are performed, the *location counter* is incremented by the size of all `.data` sections contained in the file. With the instruction `. = ALIGN(4);` we are just forcing the *location counter* to be word aligned. So, to recap, `_sdata` will contain `0x2000 0000` and `_edata` will be equal to the size of `.data` section (in our example, `.data` section contains only one variable - `dataVar`- and hence its size is `0x2000 0004`). Finally, the directive `>SRAM AT>FLASH` says to the link editor that the VMA address of the `.data` section is bound to the SRAM address space (so `0x2000 0000`), but the LMA address (that is, where the initialization values are stored) is mapped inside the flash memory space.

Thanks to this new memory layout configuration, we can now arrange the **main.c** file in the following way:

Filename: `main-ex2.c`

```c
void _start (void);
int main(void);
void delay(uint32_t count);

/* Minimal vector table */
uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
    (uint32_t *)SRAM_END,   // initial stack pointer
    (uint32_t *)_start        // main as Reset_Handler
};

// Begin address for the initialisation values of the .data section.
// defined in linker script
extern uint32_t _sidata;
// Begin address for the .data section; defined in linker script
extern uint32_t _sdata;
// End address for the .data section; defined in linker script
extern uint32_t _edata;


volatile uint32_t dataVar = 0x3f;

inline void
__initialize_data (uint32_t* flash_begin, uint32_t* data_begin,
                   uint32_t* data_end) {
  uint32_t *p = data_begin;
  while (p < data_end)
    *p++ = *flash_begin++;
}

void __attribute__ ((noreturn,weak))
_start (void) {
  __initialize_data(&_sidata, &_sdata, &_edata);
  main();

  for(;;);
}

int main() {

    /* enable clock on GPIOA and GPIOC peripherals */
    *RCC_APB1ENR = 0x1 | 0x4;
    *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1

    while(dataVar == 0x3f) {
      *GPIOA_ODR = 0x20;
      delay(200000);
```

```
68        *GPIOA_ODR = 0x0;
69        delay(200000);
70      }
71  }
```

The entry point is now the `_start()` routine, which is used as handler for the *Reset* exception. When the MCU resets, it is automatically called, and in turn it calls the `__initialize_data()` function, passing the parameters `_sidata`, `_sdata` and `_edata` computed by the Linker during the linking process. `_start()` then calls the `main()` routine, which now works as expected.

Using the `objdump tool` we can check how the sections are organized in the ELF file.

```
# ~/gcc-arm/bin/arm-none-eabi-objdump -h CH20.elf
CH20.elf:     file format elf32-littlearm
Sections:
Idx Name            Size      VMA       LMA       File off  Algn
  0 .text           000000c0  08000000  08000000  00008000  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data           00000004  20000000  080000c0  00010000  2**2
                    CONTENTS, ALLOC, LOAD, DATA
  2 .comment        00000070  00000000  00000000  00010004  2**0
                    CONTENTS, READONLY
  3 .ARM.attributes 00000033  00000000  00000000  00010074  2**0
                    CONTENTS, READONLY
```

As you can see, the tool confirms that the `.data` section has a size equal to `4` bytes, a VMA address equal to `0x2000 0000` and an LMA address equal to `0x0800 00c0` , which corresponds to the end of `.text` section.

The same applies to the `.bss` section, which is reserved to uninitialized variables. According to the ANSI C standard, the content of this section must be initialized to `0`. However, the `.bss` section does not have a corresponding flash region containing all zeros, but it is up to the startup code to initialize this region. The following linker script fragment shows the definition of the `.bss` section[14]:

[14]Please, take note that the order of sections inside a linker script reflects their order in memory. If we have two sections, named *A* and *B*, both loaded in SRAM, if section *A* is defined before than *B*, then it will be placed in SRAM before then *B*.

```
 /* Uninitialized data section */
    .bss (NOLOAD): ALIGN(4)
    {
      /* This is used by the startup in order to initialize the .bss section */
      _sbss = .;         /* define a global symbol at bss start */
      *(.bss .bss*)
      *(COMMON)

      . = ALIGN(4);
      _ebss = .;         /* define a global symbol at bss end */
    } >SRAM
```

while the following routine, always invoked from the `_start()` one, is used to zero the `.bss` region in SRAM:

```
void __initialize_bss (unsigned int* bss_begin, unsigned int* bss_end) {
  unsigned int *p = bss_begin;
  while (p < bss_end)
    *p++ = 0;
}
```

Changing the `main()` routine in the following way allow us to check that all works correctly:

Filename: `main-ex3.c`

```
volatile uint32_t dataVar = 0x3f;
volatile uint32_t bssVar;

int main() {

    /* enable clock on GPIOA and GPIOC peripherals */
    *RCC_APB1ENR = 0x1 | 0x4;
    *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1

    while(bssVar == 0) {
      *GPIOA_ODR = 0x20;
      delay(200000);
      *GPIOA_ODR = 0x0;
      delay(200000);
    }
}
```

Once again, we can see how the `.bss` section is arranged by invoking the `objdump` tool on the final binary file

```
# ~/gcc-arm/bin/arm-none-eabi-objdump -h CH20.elf
CH20.elf:     file format elf32-littlearm
Sections:
Idx Name            Size      VMA       LMA       File off  Algn
  0 .text           000000e8  08000000  08000000  00008000  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data           00000004  20000000  080000e8  00010000  2**2
                    CONTENTS, ALLOC, LOAD, DATA
  2 .bss            00000004  20000004  20000004  00010004  2**2
                    ALLOC
  3 .comment        00000070  00000000  00000000  00010004  2**0
                    CONTENTS, READONLY
  4 .ARM.attributes 00000033  00000000  00000000  00010074  2**0
                    CONTENTS, READONLY
```

The above output shows that the section has a size equal to four bytes, but it does not occupy room in the final binary file since the section has only the `ALLOC` attribute.

20.2.2.1 A Word About the `COMMON` Section

In the previous linker script we have used the special directive `*(COMMON)` during the definition of the `.bss` section. This simply says to the LD to merge the content of the *common section* inside the `.bss` section. But what is exactly the *common section*? To better understand its role, we need to revise some little-known features of the C language. Suppose that we have two source files, and both of them define two global initialized variables with the same name:

File A.c

```
int globalVar[3] = {0x1, 0x2, 0x3};
...
```

File B.c

```
int globalVar[3] = {0x1, 0x2, 0x3};
...
```

When we try to generate the final application linking the two relocatable files (`.o`), we obtain the following error:

```
B.o:(.data+0x0): multiple definition of 'globalVar'
A.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

The reason why this happens is evident: we are defining the same global variable in two different source files. But what if we declare the two symbols as un-initialized global variables?

File A.c

```
int globalVar[3];
...
```

File B.c

```
int globalVar[6];
...
```

If you try to generate the final binary file you will discover that the linker does not generate errors. Why does the linker complain about both symbol definitions? Because the C Standard says nothing to prohibit it. But if the language essentially allows to define multiple times a global un-initialized variable, how much memory will be allocated? (that is, `globalVar` will be an array containing 3 or 6 elements?). This aspect is leaved to compiler implementation. Recent GCC versions place all un-initialized global variables (not declared as `static`) inside a whole "common" section, and the amount of memory for a given symbol will assume the value of the greatest one (in our case, the array will have room for six elements of type `int` - that is, 12 bytes).

So, to recap, static global un-initialized variables are **local** to a given relocatable, and hence go in its `.bss` section; global un-initialized variables are **global** to the whole application, and go inside the *common* section. The previous linker script places both types of global un-initialized variables inside the `.bss` section, that will be zeroed at run-time by the `__initialize_bss()` routine.

This behavior can be overridden specifying the option `-fno-common` to the GCC command. GCC will allocate global un-initialized variables inside the `.data` section, initializing them to zero. This means that if we are declaring an un-initialized global array of 1000 elements, the `.data` section will contain one thousand times the value 0: this will waste a lot of flash memory. So, for embedded applications is better to avoid using that command line option.

20.2.3 `.rodata` Section

A program usually makes usage of constant data. Strings and numeric constants are just two examples, but also large arrays of data can be initialized as constants (for example, a HTML file used to generate web pages can be converted in an array, using tools like the `xxd` UNIX command). Being immutable, constant data can be placed inside the internal flash memory (or inside external flash memories connected to the MCU through the Quad-SPI interface) to save SRAM space. This can be simply achieved defining the `.rodata` section inside the linker script:

```
/* Constant data goes into flash */
.rodata : ALIGN(4)
{
  *(.rodata)         /* .rodata sections (constants) */
  *(.rodata*)        /* .rodata* sections (strings, etc.) */
} >FLASH
```

For example, considering this C code:

Filename: `main-ex4.c`

```
76  const char msg[] = "Hello World!";
77  const float vals[] = {3.14, 0.43, 1.414};
78
79  int main() {
80      /* enable clock on GPIOA and GPIOC peripherals */
81      *RCC_APB1ENR = 0x1 | 0x4;
82      *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1
83
84      while(vals[0] >= 3.14) {
85        *GPIOA_ODR = 0x20;
86        delay(200000);
87        *GPIOA_ODR = 0x0;
88        delay(200000);
89      }
90  }
```

we have that both the string `msg` and the array `vals` are placed inside the flash memory, as shown by the `objdump` tool:

```
# ~/gcc-arm/bin/arm-none-eabi-objdump -h CH20.elf
CH20.elf:     file format elf32-littlearm
Sections:
Idx Name            Size      VMA       LMA       File off  Algn
  0 .text           00000590  08000000  08000000  00008000  2**3
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata         00000024  08000590  08000590  00008590  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment        00000070  00000000  00000000  000085b4  2**0
                    CONTENTS, READONLY
  3 .ARM.attributes 00000033  00000000  00000000  00008624  2**0
                    CONTENTS, READONLY
```



Pointers to Const Data

Pay attention that declaring a string in this way:

```
char *msg = "Hello World!";
...
```

is completely different from declaring it in this other way:

```
char msg[] = "Hello World!";
...
```

In the first case we are declaring a pointer to a const array, which implies that a word will be allocated inside the `.data` section to store the location in flash memory of the string `"Hello World!"`. In the second case, instead, we are correctly defining an array of chars. **Remember that in C arrays are not pointers**.

20.2.4 Stack and Heap Regions

We have already seen in **Figure 20.1** that heap and stack are two dynamic regions of the SRAM memory that grow in the opposite direction. The stack is a descendant structure, which grows from the end of SRAM up to the end of `.bss` section, or the end of the heap if used. The heap grows in the opposite direction. While the stack is a mandatory structure in C, the heap is used only if dynamic memory allocation is needed. In some application fields (like in automotive area) the dynamic allocation is not used, or at least is strongly suggested not to be used, because of the risk involved. A decent management of the heap introduces a lot of performance penalties, and it is the source of possible leaks and memory fragmentation.

However, if your application needs to allocate dynamically some portions of the memory, you can consider to use the classical `malloc()`[15] routine from the C library. Let us consider this following example:

Filename: `main-ex5.c`

```
107 int main() {
108     /* enable clock on GPIOA and GPIOC peripherals */
109     *RCC_APB1ENR = 0x1 | 0x4;
110     *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1
111
112     char *heapMsg = (char*)malloc(sizeof(char)*strlen(msg));
113     strcpy(heapMsg, msg);
114
115     while(strcmp(heapMsg, msg) == 0) {
116       *GPIOA_ODR = 0x20;
117       delay(200000);
```

[15] There are other better alternatives, however. We will explore them in the next chapter.

```
118         *GPIOA_ODR = 0x0;
119         delay(200000);
120       }
121   }
```

The above code is simple. `heapMsg` is a pointer to a memory region dynamically allocated by the `malloc()` function. We simply copy the content of the `msg` string and check if both strings are equal. If so, the LD2 LED starts blinking.

If you try to compile the above code, you will see the following linking error:

```
Invoking: Cross ARM C++ Linker
arm-none-eabi-g++ ...  ./src/ch10/main-ex5.o
/../../../../arm-none-eabi/lib/armv7e-m/libg_nano.a(lib_a-sbrkr.o): In function `_sbrk_r':
sbrkr.c:(.text._sbrk_r+0xc): undefined reference to `_sbrk'
collect2: error: ld returned 1 exit status
```

What's happening? The `malloc()` function relies on the `_sbrk()` routine, which is a feature OS and architecture dependent. The `newlib` leaves to the user the responsibility of providing this function. The `_sbrk()` is a routine that accepts the amount of bytes to allocate inside the heap memory and returns the pointer to the start of this contiguous "chunk" of memory. The algorithm underlying the `_sbrk()` function is fairly simple:

1. First, it needs to check that there is sufficient space to allocate the desired amount of memory. To accomplish this task, we need a way to provide to the `_sbrk()` routine the maximum heap size.
2. If the heap has sufficient room to allocate the needed memory, it increments the current heap size and returns the pointer to the beginning of the new memory block.
3. If the heap does not have sufficient room (heap overflow), then the `_sbrk()` fails, and it is up to the user to provide an error feedback.

The following code shows a possible implementation for the `_sbrk()` routine. Let us analyze its code.

Filename: `main-ex5.c`

```
81  void *_sbrk(int incr) {
82      extern uint32_t _end_static; /* Defined by the linker */
83      extern uint32_t _Heap_Limit;
84
85      static uint32_t *heap_end;
86      uint32_t *prev_heap_end;
87
88      if (heap_end == 0) {
89        heap_end = &_end_static;
90      }
```

```
91      prev_heap_end = heap_end;
92
93  #ifdef __ARM_ARCH_6M__ //If we are on a Cortex-M0/0+ MCU
94      incr = (incr + 0x3) & (0xFFFFFFFC); /* This ensure that memory chunks are
95                                             always multiple of 4 */
96  #endif
97      if (heap_end + incr > &_Heap_Limit) {
98        asm("BKPT");
99      }
100
101     heap_end += incr;
102     return (void*) prev_heap_end;
```

The `_end_static` and `_Heap_Limit` are provided by the linker, and they correspond to the end of `.bss` section and the highest memory address for the heap region (that is, `_Heap_Limit - _end_static` is the size of the heap). We will see in a while how they are defined inside the linker script. `heap_end` is a statically allocated variable, and it is used to keep track of the first free memory location inside the heap. Since it is a static un-initialized local variable, according to **Table 20.1** it is placed inside the `.bss` section, and hence it is zeroed at run-time. So, the first time `_sbrk()` is called it is equal to zero, and hence it is initialized to the value of `_end_static` variable. The `if` at line 97 ensures that there is sufficient room in the heap memory. If not, the ARM assembly `BKPT` instruction is called, causing that the debugger stops the execution[16]. The tricky part is represented by the instructions at line [93:96]. The preprocessor macro checks if the ARM architecture is the ARMv6-M, that is the architectures of Cortex-M0/0+ based processors. Those processors, in fact, do not allow unaligned memory access. The instruction at line 95 ensures that the allocated memory is always a multiple of 4 bytes.

We have left to analyze the linker script. The part we are interested in starts at line 51.

Filename: `ldscript5.ld`

```
51              _end_static = _ebss;
52              _Heap_Size = 0x190;
53              _Heap_Limit = _end_static + _Heap_Size;
```

`_end_static` is nothing more than an alias to the `_ebss` memory location, that is the end of `.bss` section. `_Heap_Size` is fixed by us, and it establishes the dimension of the heap (400 bytes). Finally, `_Heap_Limit` contains nothing more than the final address of the heap memory.

[16] Here, we may use a different way to signal the heap overflow. For example, a global `error()` function could be called, and take the appropriate actions there. However, this is often a programming style, so feel free to arrange that code at your needs.



A Note About Linker Script Symbols

In this chapter we have extensively used symbols defined in linker scripts from the C source code. For every symbol, we have defined a corresponding `extern uint32_t _symbol` variable. Every time we need to access to the content of that symbol, we use the syntax `&_symbol`. This could be a source of confusion.

The way symbols are handled in linker scripts is different from that of C. In C a symbol is a triple made of the symbol, its memory location and the value. Symbols in liker scripts are tuple, made of the symbol and its memory location. So, symbols are containers for memory locations, as they would be pointers, without no value. So the following instruction:

```
extern uint32_t _symbol;
uint32_t symbol_value = _symbol;
```

is completely meaningless (there is no corresponding value for `_symbol`).

While this way of dealing with linker symbols could be obviously if the `_symbol` is a memory location, it is a source of lot of mistakes in case it is a constant value. For example, to retrieve the `_Heap_Size` value in C we have to use the following code:

```
unsigned int heapSize = (unsigned int)&_Heap_Size;
```

`_Heap_Size`, again contains the heap size as an address (that is 0x00000190), but it is not a valid STM32 address. This fact can be also analyzed by inspecting the symbol table of the final binary file, using the `objdump` tool with the `-t` command line parameter.

20.2.5 Checking the Size of Heap and Stack at Compile-Time

Microcontrollers have limited memory resources. Especially with *Value-lines* STM32 MCUs, it is easy to exceed the maximum SRAM memory. We can use the linker script also to add a sort of "static" checking about the maximum memory usage. The following linker script section helps ensuring that we are not using too much SRAM:

```
_Min_Stack_Size = 0x200;

/* User_heap_stack section, used to check that there is enough RAM left */
._user_heap_stack :
{
  . = ALIGN(4);
  . = . + _Heap_Size;
  . = . + _Min_Stack_Size;
  . = ALIGN(4);
} >SRAM
```

With the above code, we are defining a "dummy" section inside the final binary file. Using the *location counter* operator (".") we increment the size of this section so that it has a dimension equal

to the maximum heap size and the "estimated" minimum stack size. If the sum of `.data`, `.bss`, stack and heap regions is greater than the SRAM size, the linker will emit an error, as shown below:

```
arm-none-eabi-g++ ...  ./src/ch10/main-ex5.o
../../../../arm-none-eabi/bin/ld: nucleo-f401RE.elf section `._user_heap_stack' will not fit i\
n region `SRAM'
../../../../arm-none-eabi/bin/ld: region `SRAM' overflowed by 9520 bytes
collect2: error: ld returned 1 exit status
make: *** [nucleo-f401RE.elf] Error 1
```

It is important to underline that this is a static check and it is not related to the activities of the firmware at run-time. Different strategies are needed to detect a stack overflow, and it is hard to have a complete solution for embedded system. We will analyze this topic in Chapter 22.

20.2.6 Differences With the Tool-Chain Script Files

Every time we make a fresh new project, CubeMX automatically adds to the project a linker script suitable for the given STM32 MCU. For example, the file **STM32F401RETX_FLASH.ld** contains all linker directives to make a full working firmware for an STM32F401RE MCU.

By looking to the auto-generated linker script, you can see a lot of similarities with the linker scripts we used in this Chapter, which are simplified to avoid confusing novice readers. On a first instance, the generated file may appear more complicated and fuller of obscure sections. The role of this paragraph is to just clarify few other things in real-life applications. Let us look to the following code.

Filename: `STM32F401RETX_FLASH.ld`

```
36  /* Entry Point */
37  ENTRY(Reset_Handler)
38
39  /* Highest address of the user mode stack */
40  _estack = ORIGIN(RAM) + LENGTH(RAM); /* end of "RAM" Ram type memory */
41
42  _Min_Heap_Size = 0x200 ; /* required amount of heap */
43  _Min_Stack_Size = 0x400 ; /* required amount of stack */
44
45  /* Memories definition */
46  MEMORY
47  {
48    RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH = 96K
49    FLASH    (rx)    : ORIGIN = 0x8000000,   LENGTH = 512K
50  }
51
52  /* Sections */
53  SECTIONS
```

```
{
  /* The startup code into "FLASH" Rom type memory */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  /* The program code and other data into "FLASH" Rom type memory */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    *(.glue_7)         /* glue arm to thumb code */
    *(.glue_7t)        /* glue thumb to arm code */
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))

    . = ALIGN(4);
    _etext = .;        /* define a global symbols at end of code */
  } >FLASH

  /* Constant data into "FLASH" Rom type memory */
  .rodata :
  {
    . = ALIGN(4);
    *(.rodata)         /* .rodata sections (constants, strings, etc.) */
    *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
    . = ALIGN(4);
  } >FLASH
```

The above linker script is related to an STM32F401RE MCU, but its content is the same to the majority of the STM32 MCU. The first lines are nothing more than what we have seen before. The `Reset_Handler` assembly routine (which is defined in the `Core/Startup/startup_stm32f401retx.s` file) is configured as the entry point for the whole application. By looking to the assembly code, you can easily see that it does nothing more than the functions `__initialize_data()` and `__initialize_bss()` seen before.
Next, the linker script defines the stack and heap boundaries and dimensions, the RAM and FLASH sizes and their base addresses. Next, the `.isr_vector` section is defined, which is dedicated to the *vector table*. The next section is the `.text` one, which contains these additional sections:

- `.glue_7` and `.glue_7t`: these two sections, which are automatically generated by GCC, refer

to the so called *ARM interwork code*. This "obscure" code is a preamble (called "glue") invoked right after the ARM `BX` or `BLX` assembly instructions, which are the instructions used to switch the Cortex-M core between the ARM and the Thumb/Thumb-2 modes. As said in Chapter 1, the Cortex-M cores allow to switch at run-time between the ARM instruction set and the more optimized (in terms of size) Thumb instruction set. The switch between the two modes requires additional operations, which are automatically added to the binary file by the compiler. By default, in a regular STM32CubeIDE application, the instruction mode switching is not used, and those sections will be discarded by the linker since the option `--gc-sections` is set by default (see **Figure 20.4**).

- `.eh_frame`: this section is related to operations carried out during stack *unwinding* operations in C++ after an exception is thrown. If you do not use any C++ code in your application, that section will be silently discarded if `--gc-sections` option is set.
- `.init` and `.fini`: these sections will contain initialization and de-initialization code added by the C *run-time* library. Those sections ensure proper initialization of the C *run-time* library and they are needed unless the options `-nodefaultlibs` and `-nostdlib` are set in project settings (**Figure 20.4**) and so no C standard library functions are used.

Let us continue the analysis of the auto-generated linker script.

Filename: `STM32F401RETX_FLASH.ld`

```
  .ARM.extab   : {
    . = ALIGN(4);
    *(.ARM.extab* .gnu.linkonce.armextab.*)
    . = ALIGN(4);
  } >FLASH

  .ARM : {
    . = ALIGN(4);
    __exidx_start = .;
    *(.ARM.exidx*)
    __exidx_end = .;
    . = ALIGN(4);
  } >FLASH

  .preinit_array     :
  {
    . = ALIGN(4);
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP (*(.preinit_array*))
    PROVIDE_HIDDEN (__preinit_array_end = .);
    . = ALIGN(4);
  } >FLASH

  .init_array :
  {
```

```
114     . = ALIGN(4);
115     PROVIDE_HIDDEN (__init_array_start = .);
116     KEEP (*(SORT(.init_array.*)))
117     KEEP (*(.init_array*))
118     PROVIDE_HIDDEN (__init_array_end = .);
119     . = ALIGN(4);
120   } >FLASH
121
122   .fini_array :
123   {
124     . = ALIGN(4);
125     PROVIDE_HIDDEN (__fini_array_start = .);
126     KEEP (*(SORT(.fini_array.*)))
127     KEEP (*(.fini_array*))
128     PROVIDE_HIDDEN (__fini_array_end = .);
129     . = ALIGN(4);
130   } >FLASH
```

The `.ARM.extab` and `.ARM` sections are once again connected to the stack unwinding in C++, and you can find additional information in the ELF for the ARM Architecture[17] specification. Instead, the `.preinit_array`, `.init_array` and `.fini_array` sections are related to C++ object instantiation. To understand what those sections are used for, consider the following C++ application:

```
1  class MyClass {
2      int i;
3
4  public:
5      MyClass() {
6          i = 100;
7      }
8
9      void increment() {
10         i++;
11     }
12 };
13
14 MyClass instance;
15
16 int main() {
17     instance.increment();
18     for (;;);
19 }
```

Let us focus our attention on line 14. Here we are defining an instance of the class `MyClass`. The instance is defined as global variable. But declaring an instance of a class assumes that the

[17] https://bit.ly/3FoZRz4

constructor of that class is automatically called. So, to be clear, when we call the `increment()` method at line 17, the instance attribute `i` will be equal to `101`. But who takes care of calling the instance constructor? When an instance is created locally (that is, from a global function or another method), it is up to that callable to perform class initialization. But when this happens at global scope, it is up to other initializations routines. Usually, the compiler automatically generates an array of function pointers that will contain initializations routines for all globally and statically allocated objects. These arrays are usually called `__init_array` and `__fini_array` (which contains the call to object destructors).

Let us continue the analysis of the auto-generated linker script.

Filename: `STM32F401RETX_FLASH.ld`

```
132   /* Used by the startup to initialize data */
133   _sidata = LOADADDR(.data);
134
135   /* Initialized data sections into "RAM" Ram type memory */
136   .data :
137   {
138     . = ALIGN(4);
139     _sdata = .;        /* create a global symbol at data start */
140     *(.data)           /* .data sections */
141     *(.data*)          /* .data* sections */
142     *(.RamFunc)        /* .RamFunc sections */
143     *(.RamFunc*)       /* .RamFunc* sections */
144
145     . = ALIGN(4);
146     _edata = .;        /* define a global symbol at data end */
147
148   } >RAM AT> FLASH
149
150   /* Uninitialized data section into "RAM" Ram type memory */
151   . = ALIGN(4);
152   .bss :
153   {
154     /* This is used by the startup in order to initialize the .bss section */
155     _sbss = .;         /* define a global symbol at bss start */
156     __bss_start__ = _sbss;
157     *(.bss)
158     *(.bss*)
159     *(COMMON)
160
161     . = ALIGN(4);
162     _ebss = .;         /* define a global symbol at bss end */
163     __bss_end__ = _ebss;
164   } >RAM
165
166   /* User_heap_stack section, used to check that there is enough "RAM" Ram  type memory left */
```

```
167   ._user_heap_stack :
168   {
169     . = ALIGN(8);
170     PROVIDE ( end = . );
171     PROVIDE ( _end = . );
172     . = . + _Min_Heap_Size;
173     . = . + _Min_Stack_Size;
174     . = ALIGN(8);
175   } >RAM
```

The rest of the linker script file is almost the same to what seen in previous paragraph. `.data` and `.bss` sections are defined. Here the differences are minimal. `.RamFunc` is a section dedicated to functions that are stored in RAM instead in FLASH. We will analyze this possibility in a next chapter. The `.bss` section defines just two additional symbols: `__bss_start__` and `__bss_end__`, which are an alias for `_sbss` and `_ebss`. These two symbols are needed by recent releases of the C *run-time* (by the `_mainCRTStartup()` routine, defined in the `.text.init` section).

20.3 How to Use the CCM Memory

Some microcontrollers from STM32F3 family and all microcontrollers from the STM32G4 family provide an additional SRAM memory named *Core Coupled Memory* (CCM). Different from the regular SRAM, this memory is tightly coupled with the Cortex-M core. A direct path connects both the D-Bus and I-Bus to this memory area (see **Figure 20.5**[18]), allowing 0-wait state execution. Although it is perfectly possible to store data in this memory, like look-up tables and initialization vectors, the best usage of this area is to store critical and computationally intensive routines, which may be executed in *real-time*. For this reason, MCUs with CCM memory are said to implement *routine booster technology*.

[18]The figure has been arranged from the one contained in the AN4296 from ST(https://bit.ly/1QSctkT).

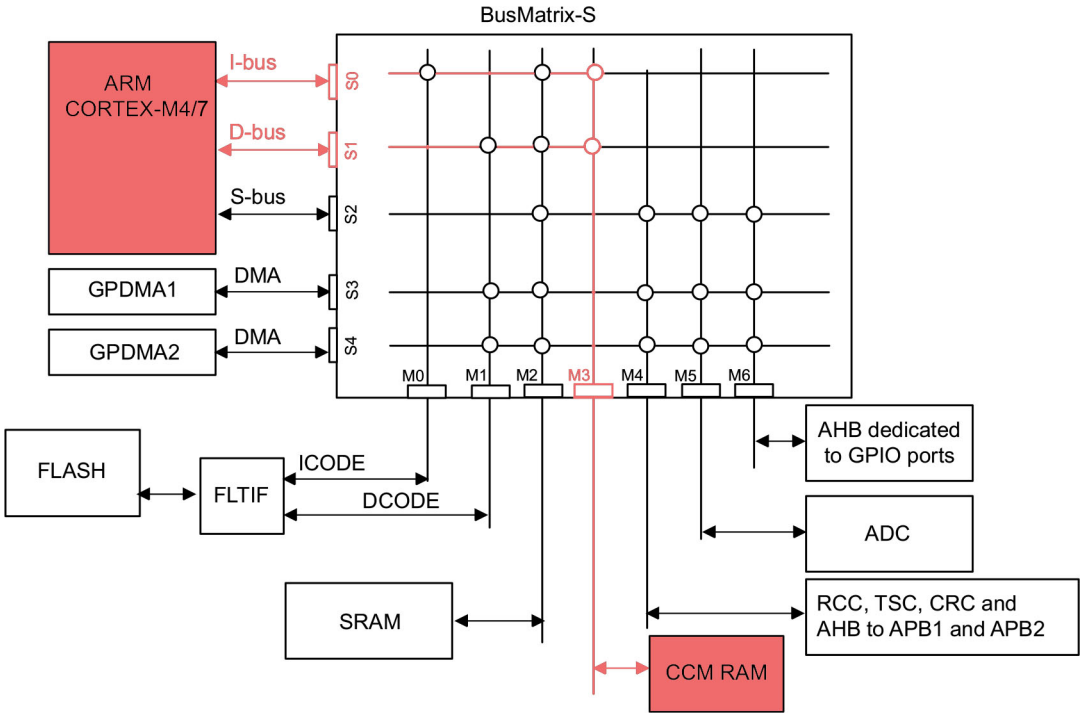


Figure 20.5: The direct connection between the Cortex-M core and the CCM SRAM in an STM32F3 MCU with CCM memory



Why Use CCM to Store Code Instead of Data?

It is quite common to read around on the web that the CCM memory can be used to store critical data. This guarantees a fast access to it from the core. While this is true in theory, it does not give practical advantages. All STM32 MCUs with CCM memory also provide SRAM that can be addressed at maximum system clock frequency without wait states[19]. Moreover, in STM32F3 MCUs SRAM can be accessed by both CPU and DMA, while the CCM only by the Cortex core. Instead, when code is in CCM SRAM and data is stored in the regular SRAM, the Cortex core is in the optimum Harvard configuration, because allows 0-wait states access for the I-Bus (which accesses to CCM) and the D-Bus (which accesses in parallel to the SRAM)[20].

However, if deterministic performances are not important for your application, and you need additional SRAM storage, then the CCM is a good reserve for data memory.

[19]Some STM32 MCUs provide two SRAM memories, with one of these allowing 0-wait access. Always consult the datasheet for your MCU.
[20]Keep in mind that to reach a full parallel access to the SRAM, no other masters (e.g., the DMA) must contend the access to the SRAM through the BusMatrix.

Feature	STM32F303xB/C STM32F358xx	STM32F303x6/8 STM32F334xx STM32F328xx	STM32F303xD/E STM32F398xxs	STM32G47xx STM32G84xx	STM32G431x STM32G441x
Size (Kbytes)	8	4	16	32	10
Mapping	0x1000 0000				
Aliasing Address	No	No	No	0x2001 8000	0x2000 5800
Parity check	Yes				
Write protection	Yes, with 1-Kbyte page granularity				
Read protection	No			Yes	
Erase	No			Yes	
DMA access	No			Yes if aliased at 0x2001 8000	Yes if aliased at 0x2000 5800

Table 20.2: The CCM implementation in STM32F3 and STM32G4 families

Table 20.2 shows the main characteristics of the CCM memory in STM32F3/G4 microcontrollers. In all STM32 MCUs with this additional memory, the CCM SRAM is mapped starting from the `0x1000 0000` address, apart for the STM32G4 MCUs where the CCM is also aliased right after the end of SRAM, so that it can be easily expanded. Moreover, this simplifies the placing of stack in CCM memory.
Once again, to use the CCM memory we need to define this memory region inside the linker script, in the following way[21]:

```
/* memory layout for an STM32F303RE */
MEMORY
{
    FLASH (rx)  : ORIGIN = 0x08000000, LENGTH = 64K
    SRAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 12K
    CCM  (xrw)  : ORIGIN = 0x10000000, LENGTH = 16K
}
```

Obviously, the `LENGTH` attribute has to reflect the size of the CCM memory for the specific STM32 MCU. Once the region is defined, we have to create a specific section inside the linker script:

```
.ccm : ALIGN(4) {
    *(.ccm .ccm*)
} >CCM
```

To relocate a specific routine inside the CCM memory we can use the GCC keyword `__attribute__`, as seen before for the `.isr_vector` section:

[21]The memory configuration refers to a Nucleo-F303 that, together with the Nucleo-G474, provides the CCM memory.

```
void __attribute__((section(".ccm"))) routine() {
    ...
}
```

If, instead, we want to store data inside the CCM memory, than we also need to initialize it as we have seen for `.bss` and `.data` regions in regular SRAM memory. In this case, we need a more articulated liker script:

```
/* Used by the startup to initialize data in CCM */
_siccm = LOADADDR(.ccm.data);

/* Initialized data section in CCM */
.ccm.data : ALIGN(4) {
     _sccmd = .;
     *(.ccm.data .ccm.data*)
     . = ALIGN(4);
     _eccmd = .;
} >CCM AT>FLASH

/* Uninitialized data section in CCM */
.ccm.bss (NOLOAD) : ALIGN(4) {
  _sccmb = .;

  *(ccm.bss ccm.bss*)

  . = ALIGN(4);
  _eccmb = .;
} >CCM
```

Here we are defining two sections: `.ccm.data`, which will be used to store global initialized data in CCM, and `.ccm.bss` used to store global un-initialized data. As done for the regular SRAM, will need to call the `__initialize_data()` and `__initialize_bss()` routines from the `_start()` routine:

```
...
__initialize_data(&_siccm, &_sccmd, &_eccmd);
__initialize_bss(&_sccmb, &_eccmb);
...
```

Then, to place data inside the CCM, we have to instruct the compiler using the **attribute** keyword:

```
uint8_t initdata[] __attribute__((section(".ccm.data"))) = {0x1, 0x2, 0x3, 0x4};
uint8_t uninitdata __attribute__((section(".ccm.bss")));
```

20.3.1 Relocating the *vector table* in CCM Memory

The CCM memory can be also used to store ISR routines, by relocating the whole *vector table* inside the CCM memory. This can be especially useful for ISRs that need to be processed in the shortest possible time. However, relocating the *vector table* requires additional steps, since the Cortex-M architecture is designed so that the *vector table* starts from the `0x0000 0004` address (which corresponds to the `0x0800 0004` address of the internal flash memory). The steps to follow are these ones:

- define the *vector table* to place in the CCM RAM using the `__attribute__((section(".isr_vector_ccm"))` keyword;
- define the exception handlers for the interested exceptions and ISRs and place them in the corresponding section using the `__attribute__((section(".ccm"))` keyword;
- define a minimal *vector table*, composed by the MSP pointer and the address of the *Reset* exception handler, to place in the flash memory starting from `0x0800 0000` address;
- relocate the *vector table* from the *Reset* exception by copying the content of the `.ccm` section from the flash memory into the SRAM.

Let us start defining the *vector table* to place in CCM RAM. Here we are defining a file named **ccm_vector.c** with the following content:

Filename: `ccm_vector.c`

```
1  #include <stdint.h>
2  #include <stm32f3xx_hal.h>
3
4  /* GPIOA peripheral addresses */
5  #define GPIOA_ODR       ((uint32_t*)(GPIOA_BASE + 0x14))
6
7  extern const uint32_t _estack;
8
9  void SysTick_Handler(void);
10
11 uint32_t *ccm_vector_table[] __attribute__((section(".isr_vector_ccm"))) = {
12   (uint32_t *)&_estack,   // initial stack pointer
13   (uint32_t *) 0,         // Reset_Handler not relocatable
14   (uint32_t *) 0,
15   (uint32_t *) 0,
16   (uint32_t *) 0,
17   (uint32_t *) 0,
18   (uint32_t *) 0,
19   (uint32_t *) 0,
20   (uint32_t *) 0,
21   (uint32_t *) 0,
22   (uint32_t *) 0,
23   (uint32_t *) 0,
```

```
24    (uint32_t *) 0,
25    (uint32_t *) 0,
26    (uint32_t *) 0,
27    (uint32_t *) SysTick_Handler
28  };
29
30  void __attribute__((section(".ccm"))) SysTick_Handler(void) {
31    *GPIOA_ODR = *GPIOA_ODR ? 0x0 : 0x20; //Causes LD2 LED to blink
32  }
```

The file contains just the *vector table*, which is placed inside the `.isr_vector_ccm` section, and the handler for the *SysTick* exception, which is placed inside the `.ccm` section. Next, we need to arrange the linker script in the following way:

Filename: `ldscript6.ld`

```
75      /* Used by the startup to load ISR in CCM from FLASH */
76        _slccm = LOADADDR(.ccm);
77
78      .ccm : ALIGN(4)
79      {
80         _sccm = .;
81         *(.isr_vector_ccm)
82         *(.ccm)
83         KEEP(*(.isr_vector_ccm .ccm))
84
85         . = ALIGN(4);
86         _eccm = .;
87      } >CCM AT>FLASH
88
89      /* Size of the .ccm section */
90      _ccmsize = _eccm - _sccm;
```

The linker script does not contain anything different from what seen so far. The `.ccm` section is defined and we instruct the linker to place in it the content of the `.isr_vector_ccm` section first and then the content from the `.ccm` section, which in our case contains just the `SysTick_Handler` routine. We also instruct the linker to store the content of `.ccm` section inside the flash memory (using the directive `CCM AT>FLASH`), while the *VMA* addresses of the `.ccm` section are bound to the CCM range of memory addresses (that is, the starting address is `0x1000 0000`).

Finally, we need to manually copy the content of the `.ccm` section from the flash memory to the CCM one and to relocate the *vector table*. This work is performed again by the `Reset_Handler` exception.

Filename: `main-ex6.c`

```
67  /* Minimal vector table */
68  uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
69      (uint32_t *)&_estack,   // initial stack pointer
70      (uint32_t *)_start        // main as Reset_Handler
71  };
72
73  void __attribute__ ((noreturn,weak))
74  _start (void) {
75    /* Copy the .ccm section from the FLASH memory (_slccm) into CCM memory  */
76    memcpy(&_sccm, &_slccm, (size_t)&_ccmsize);
77
78    SCB->VTOR = (uint32_t)&_sccm;  /* Relocate vector table to 0x1000 0000 */
79    SYSCFG->RCR = 0xF;             /* Enable write protection for CCM memory */
80
81    __initialize_data(&_sidata, &_sdata, &_edata);
82    __initialize_bss(&_sbss, &_ebss);
83    main();
84
85    for(;;);
86  }
87
88  int main() {
89    /* enable clock on GPIOA peripheral */
90    *RCC_APB1ENR |= 0x1 << 17;
91    *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1
92
93    SysTick_Config(4000000); //Underflows every 0.5s
94  }
95
96  void delay(uint32_t count) {
97      while(count--);
98  }
```

Lines [68:72] define the minimal *vector table* used when the CPU resets. It is just composed by the MSP pointer and the address of the `Reset_Handler` exception, which is represented by the `_start()` routine. When the MCU resets, we copy at line 77 the content of the `.ccm` section from the flash memory (the base address is stored inside the `_slccm` variable) to the CCM memory, and then we relocate the whole *vector table* assigning the position in CCM memory of the `ccm_vector_table` array to the register `VTOR` in the *System Control Block* (SCB) - line 81. Next, we enable the write protection on the whole CCM memory to avoid unwanted writings that may corrupt the code.



The CCM RAM is subdivided in pages of 1Kb. Every bit in the RCR register of the *System Configuration Controller* (SYSCFG) is used to set the write protection on individual page basis (bit 1 sets protection of first page, bit 2 sets protection on second page and so on). Here, we are write-protecting the whole CCM memory of an STM32F303 MCU, which has a CCM memory made of four 1Kb pages.

It is important to remark that, if we disable writing of the whole CCM memory, we cannot place global or statically allocated variables in it, otherwise a fault will occur. On the other side, placing both code and data in CCM memory makes us lose the benefits obtained by the CCM memory, due to the simultaneous access to the same memory both by the *D-Bus* and *I-Bus* bus (looking at **Figure 20.5** you can see that the CCM memory is connected to just one master port of the BusMatrix - the port M3; so the access from *D-Bus* and *I-Bus* is disciplined by the BusMatrix).



The *vector table* relocation is not limited to the CCM memory. As we will see in Chapter 22, this technique is also used when the MCU boots from different sources than the internal flash. In this case, the *vector table* is usually placed in SRAM and it has to be relocated.



The *vector table* relocation is a feature not available in Cortex-M0 microcontrollers, while is available in Cortex-M0+. As we will see in Chapter 22, there exists a procedure that tries to address this limitation.

20.4 How to Use the MPU in Cortex-M0+/3/4/7 Based STM32 MCUs

Apart from the Cortex-M0 core, all Cortex-M based microcontrollers can optionally provide a *Memory Protection Unit* (MPU). And the good news is that all STM32 MCUs based on those cores provide it. The MPU should not be confused with the *Memory Management Unit* (MMU), an advanced hardware component available in more performing microprocessors like Cortex-A, which is mostly dedicated to the translation of virtual memory addresses in physical ones.

The MPU is used to protect up to eight memory regions, numbered from 0 to 7. These, in turn can have eight subregions, if the main region is at least 256 bytes. The subregions have all the same size and can be enabled or disabled according to the subregion number. The MPU is used to make an embedded system more robust and more secure, and in some application domains its usage is mandatory (e.g., in automotive and aerospace). The MPU can be used to:

- Prohibit the user applications from corrupting data used by critical tasks (such as the operating system kernel).

- Define the SRAM memory region as a non-executable to prevent code injection attacks.
- Change the memory access attributes.

If the CPU core violates the access definitions of a given memory region (for example, trying to execute code from a non-executable region), the *HardFault* exception (or the more specific *Memory Fault* one as we will see in Chapter 24) is raised.

The MPU regions can spawn the whole 4GB address space, and they can also overlap. The region characteristics are defined by two parameters: the region type and its attributes. There are three memory types:

- **Normal memory**: allows the load and store of bytes, half-words and words[22] to be arranged by the CPU in an efficient manner (the compiler is not aware of memory region types). For the normal memory region, the load/store is not necessarily performed by the CPU in the order listed in the program. SRAM and flash memories are two examples of normal memory.
- **Device memory**: within the device region, the loads and stores are done strictly in order. This is to ensure the registers are set in the proper order, otherwise the device behavior will be impacted.
- **Strongly ordered memory**: everything is always done in the programmatically listed order, where the CPU waits the end of load/store instruction execution (effective bus access) before executing the next instruction in the program stream. This can cause a performance hit.

Table 20.3: Memory region attributes

Region Attribute	Description
XN	Execute never
AP	Access permission (see **Table 20.4**)
TEX	Type Extension field (not available in Cortex-M0+
S	Shareable
C	Cacheable
B	Bufferable
SRD	Subregion disable/enable
SIZE	Size of the memory region

Each memory region has eight attributes, reported in **Table 20.3**:

- **Execute never (XN)**: a memory region marked with this attribute does not allow the execution of program code.
- **Access Permission (AP)**: defines the access permissions to the memory region. Permissions are set both for privileged (e.g., the RTOS kernel) and unprivileged code (e.g., an individual thread). **Table 20.4** lists all possible combinations.

[22] Remember that Cortex-M0/0+ cores are only able to perform word-aligned access.

- **TEX**, **C** and **B**: these fields are used to define cache properties for the region, and to some extent, its shareability. They are encoded according to the **Table 20.5**. Take note that in Cortex-M0+ cores the TEX field is always 0. This because Cortex-M0+ cores support one level of cache policy.
- **S**: this fields configures a shareable memory region. The memory system provides data synchronization between bus masters in a system with multiple bus masters, for example, a processor with a DMA controller. Strongly-ordered memory is always shareable. If multiple bus masters can access a non-shareable memory region, the software must ensure the data coherency between the bus masters. This field is not supported in ARMv6-M architecture and therefore is always set to 0 in the Cortex-M0+ processors.
- **SRD**: defines whether a particular subregion is enabled or disabled. Disabling a subregion means that another region overlapping the disabled range matches instead. If no other enabled region overlaps the disabled subregion the MPU issues a fault.
- **SIZE**: specifies the memory region size. The size cannot be arbitrary, but it can assume a value from a well-known pool of region sizes (it depends on the specific STM32 family).

Table 20.4: Access permissions to a region

Privileged access	Unprivileged access	Description
No access	No access	All accesses to the region generate a permission fault
RW	No access	Access from a privileged software only
RW	RO	Writings by an unprivileged software generate a permission fault
RW	RW	Full access to the region
Unpredictable	Unpredictable	RESERVED
RO	No access	Read by a privileged software only
RO	RO	Read only, by privileged or unprivileged software

STM32F7 microcontrollers provide an integrated L1-cache, as we will see in the next chapter. For these MCUs the following additional memory attributes are available:

- **Cacheable/non-cacheable**: means that the dedicated region can be cached or not.
- **Write through with no write allocate**: on hits it writes to the cache and the main memory, on misses it updates the block in the main memory not bringing that block to the cache.
- **Write-back with no write allocate**: on hits it writes to the cache setting dirty bit for the block, the main memory is not updated. On misses it updates the block in the main memory not bringing that block to the cache.
- **Write-back with write and read allocate**: on hits it writes to the cache setting dirty bit for the block, the main memory is not updated. On misses it updates the block in the main memory and brings the block to the cache.

Table 20.5: Region cache properties and shareability

TEX	C	B	Memory Type	Description	Shareable
000	0	0	Strongly Ordered	Strongly Ordered	Yes
000	0	1	Device	Shared Device	Yes
000	1	0	Normal	Write through, no write allocate	S bit dependent
000	1	1	Normal	Write-back, no write allocate	S bit dependent
001	0	0	Normal	Non-cacheable	S bit dependent
001	0	1	Reserved	Reserved	Reserved
001	1	0	Undefined	Undefined	Undefined
001	1	1	Normal	Write-back, write and read allocate	S bit dependent
010	0	0	Device	Non-shareable device	No
010	0	1	RESERVED	RESERVED	RESERVED

Table 20.6 lists the types and attributes of the memories found in an STM32 microcontroller. As we will see in the next chapter, the integrated L1-cache in STM32F7 MCUs also allows to define as cacheable regions external memories accessible through the FMC controller. This is a great performance improvement that this families of MCUs offers.

Table 20.6: Memory attributes for the typical STM32 memories

Memory	Memory type	Memory attributes
ROM, flash (program memories)	Normal memory	Non-shareable, write-through C=1, B=0, TEX=0, S=0
Internal SRAM	Normal memory	Shareable, write-through C=1, B=0, TEX=0, S=1/S=0
External RAM (through FMC)	Normal memory	Shareable, write-back C=1, B=1, TEX=0, S=1/S=0
Peripherals	Device	Shareable devices C=0, B=1, TEX=0, S=1/S=0

Table 20.7 shows a comparison of the MPU features in Cortex-M0+/3/4/7 cores. The *MPU bypass* is a feature offered by the MPU to bypass access permissions to a region when the processor is running NMI or HardFault exceptions. For example, the MPU might be used as a mechanism to detect stack limit by allocating a small SRAM space at the bottom of the stack as non-accessible. When the stack limit is reached, the HardFault handler can bypass the MPU restriction and utilize the reserved SRAM space for fault handling.

Table 20.7: Comparison of MPU features between the various Cortex-M cores

	Cortex®-M0+	Cortex®-M3/M4	Cortex®-M7
Number of regions	8	8	8
Region address	Yes	Yes	Yes
Region size	256 bytes to 4GB	32 bytes to 4GB	32 bytes to 4 GB
Region memory attributes	S, C, B, XN	TEX, S, C, B, XN	TEX, S, C, B, XN
Region access permission	Yes	Yes	Yes
Subregion disable	Yes	Yes	Yes
MPU bypass for NMI/HardFault	Yes	Yes	Yes

Table 20.7: Comparison of MPU features between the various Cortex-M cores

	Cortex®-M0+	Cortex®-M3/M4	Cortex®-M7
Fault exception	HardFault only	HardFault/MemManage	HardFault/MemManage

20.4.1 Programming the MPU With the CubeHAL

The CubeHAL provides all the necessary abstraction layer to program the MPU. The function

```
void HAL_MPU_ConfigRegion(MPU_Region_InitTypeDef *MPU_Init);
```

allows to configure a memory region. All region settings are specified with an instance of the `MPU_Region_InitTypeDef struct`, which is defined in the following way:

```
typedef struct {
  uint8_t  Enable;           /* Specifies the status of the region. */
  uint8_t  Number;           /* Specifies the number of the region to protect. */
  uint32_t BaseAddress;      /* Specifies the base address of the region to protect. */
  uint8_t  Size;             /* Specifies the size of the region to protect.  */
  uint8_t  SubRegionDisable; /* Specifies the number of the subregion protection
                                to disable. */
  uint8_t  TypeExtField;     /* Specifies the TEX field level. */
  uint8_t  AccessPermission; /* Specifies the region access permission type. */
  uint8_t  DisableExec;      /* Specifies the instruction access status. */
  uint8_t  IsShareable;      /* Specifies the shareability status of the
                                protected region. */
  uint8_t  IsCacheable;      /* Specifies the cacheable status of the region protected. */
  uint8_t  IsBufferable      /* Specifies the bufferable status of the protected region. */
} MPU_Region_InitTypeDef;
```

Let us analyze the most relevant fields of this struct.

- `Enable`: specifies the status of the region, and it can assume the values `MPU_REGION_ENABLE` and `MPU_REGION_DISABLE`.
- `Number`: it is the region ID and it can spawn from 0 up to 7.
- `BaseAddress`: corresponds to the base address of the region. In Cortex-M0+ this address must be word-aligned.
- `Size`: specifies the size of the region and corresponds to all power of two from 2^5 up to 2^{32}. The CubeHAL defines a set of 27 macros, ranging from `MPU_REGION_SIZE_32B` up to `MPU_REGION_SIZE_4GB`. Take a look at the file `stm32XXxx_hal_cortex.h` for the complete list.
- `AccessPermission`: specifies the region permission attributes and it can assume the values listed in **Table 20.8**.

- `DisableExec`: specifies if it is possible to execute code inside the region. It can assume the values `MPU_INSTRUCTION_ACCESS_ENABLE` and `MPU_INSTRUCTION_ACCESS_DISABLE`.
- `IsShareable`: specifies if the region has the *shareable* attribute, and it can assume the values `MPU_ACCESS_SHAREABLE` and `MPU_ACCESS_NOT_SHAREABLE`.
- `IsCacheable`: specifies if the region has the *cacheable* attribute, and it can assume the values `MPU_ACCESS_CACHEABLE` and `MPU_ACCESS_NOT_CACHEABLE`.
- `IsBufferable`: specifies if the region has the *bufferable* attribute, and it can assume the values `MPU_ACCESS_BUFFERABLE` and `MPU_ACCESS_NOT_BUFFERABLE`.

Table 20.8: CuneHAL macros to define access permissions to a region

Access permission	Description
`MPU_REGION_NO_ACCESS`	All accesses to the region generate a permission fault
`MPU_REGION_PRIV_RW`	Access from a privileged software only
`MPU_REGION_PRIV_RW_URO`	Writings by an unprivileged software generate a permission fault
`MPU_REGION_FULL_ACCESS`	Full access to the region
`MPU_REGION_PRIV_RO`	Read by a privileged software only
`MPU_REGION_PRIV_RO_URO`	Read only, by privileged or unprivileged software

The MPU must be disabled before configuring any memory region (or before changing its attributes). To perform this operation the HAL provides the function:

```
void HAL_MPU_Disable(void);
```

while to enable the MPU we use the function:

```
void HAL_MPU_Enable(uint32_t MPU_Control);
```

The `MPU_Control` parameter specifies the control mode of the MPU during *HardFault*, NMI, FAULTMASK and privileged access to the default memory. It can assume a value from those listed in **Table 20.9**. It is important to note that the *MemFault* exception is automatically enabled once the MPU is enabled.

Table 20.9: CubeHAL macros to define MPU control during *HardFault*, NMI and FAULTMASK

Access permission	Description
`MPU_HFNMI_PRIVDEF_NONE`	The default memory map is used for privileged accesses, and it assumes the role of a background region (also called "region -1", where "-1" is the region ID). The access to the whole 4GB is so prohibited by unprivileged code, except in those regions that explicitly allow it.
`MPU_HARDFAULT_NMI`	The MPU is disabled when *HardFault* and NMI exceptions raise.
`MPU_PRIVILEGED_DEFAULT`	The background region is disabled, and any access not covered by any enabled region will cause a fault.
`MPU_HFNMI_PRIVDEF`	The MPU is enabled when *HardFault* and NMI exceptions raise.

```
1   MPU_Region_InitTypeDef MPU_InitStruct;
2
3   /* Disable MPU */
4   HAL_MPU_Disable();
5
6   /* Configure RAM region as Region N°0, 8kB of size and R/W region */
7   MPU_InitStruct.Enable = MPU_REGION_ENABLE;
8   MPU_InitStruct.BaseAddress = 0x20000A00;
9   MPU_InitStruct.Size = MPU_REGION_SIZE_32B;
10  MPU_InitStruct.AccessPermission = MPU_REGION_PRIV_RO_URO;
11  MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
12  MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
13  MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
14  MPU_InitStruct.Number = MPU_REGION_NUMBER0;
15  MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
16  MPU_InitStruct.SubRegionDisable = 0x00;
17  MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
18  HAL_MPU_ConfigRegion(&MPU_InitStruct);
19
20  /* Defines a pointer to the first word of protected region */
21  volatile uint32_t *p = (uint32_t*)0x20000A00;
22  *p = 0xDDEEFF00;
23
24  /* Re-enable the MPU and enable the background region */
25  HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
26
27  if(*p != 0xDDEEFF00)
28    asm("BKPT #0");
29
30  *p = 0xAABBCCDD; //This will generate a MemManage fault
```

The previous code fragment shows how to define a memory region over the SRAM memory and to prevent access to it in write mode, both from privileged and unprivileged code. The region starts from the address `0x2000 0A00` and lasts 32 bytes. A pointer to the beginning of that region is defined (line 21) and the content of the first word is modified (line 22). The MPU is enabled, and the region attributes prevent code from modify its content. The `if` at line 27 will not match, because the first region word effectively contains the value `0xDDEEFF00`. However, the instruction at line 30 will generate a *MemManage* fault, due to read only attribute of the region.

21. Flash Memory Management

Flash memory is a silent peripheral that we use without worrying too much about it. Once we are sure that the flash has sufficient room to store the firmware, we upload the binary image using the debugger or a dedicated flashing tool. And we completely forget it.

However, the internal flash provided by all STM32 microcontrollers works in the same way of other peripherals. It can be programmed directly from the firmware by configuring specific registers, and this allows us to upgrade the firmware using the same on-board code or to store relevant configuration data without using dedicated external hardware (an external I²C EEPROM or an SPI flash).

This chapter shows how to program the internal STM32 flash memory using the dedicated `HAL_FLASH` module from the CubeHAL. It describes how the flash is usually organized in a typical STM32 microcontroller, briefly illustrating the differences among each family and the steps involved to program specific areas of this memory directly from the same microcontroller.

Finally, the role of the ART™ Accelerator is described, together with the evolutions of this ST proprietary technology in STM32F7 microcontrollers.

21.1 Introduction to STM32 Flash Memory

Different from other embedded architectures[1], all STM32 microcontrollers provide a dedicated flash memory to store program code and constant data. There are currently eleven memory sizes, ranging from 16KB up to 2MB. The last digit of the part number of a given STM32 MCU defines the size of the flash memory, as shown in **Table 21.1**. For example, an STM32F401R**E** MCU has 512KB of flash memory.

Table 21.1: The size of the flash memory given the last digit in an STM32 part number

Last digit in P/N	Flash memory size (KB)
4	16
6	32
8	64
B	128
Z	192
C	256
D	384
E	512

[1] This is especially true for Cortex-A microprocessors or FPGAs, where the non-volatile memory is provided by external flash memories connected to the CPU through dedicated bus lines.

Table 21.1: The size of the flash memory given the last digit in an STM32 part number

Last digit in P/N	Flash memory size (KB)
F	768
G	1024
H	1536
I	2048

Depending on the STM32 family, sales type and packaged used, the flash memory of an STM32 MCU can be organized in:

- **one or two banks**: the majority of STM32 microcontrollers provide just one *bank* of flash memory, while the most performing ones up to two banks. The *multi-bank* architecture allows dual and simultaneous operations: while programming or erasing in one bank, read operations are possible in the other one. This approach provides higher flexibility for dual operations especially for high performance applications. In some more recent STM32 MCUs multi-bank is a programmable feature that can be optionally enabled, and the bank sizes can be configured at need.
- **each bank is in turn divided in sectors**: each flash memory bank is partitioned in several sub-blocks, called *sectors*. Some STM32 MCUs provide flash memory having all sectors with the same size (usually equal to 1KB or 2KB). Some other ones provide several sectors with different sizes (usually the first sectors have a smaller size than the remaining ones).
- **each sector can be divided in pages**: in some STM32 MCUs, a sector is further partitioned in several smaller *pages*. Sometimes, this happens only for the first sector, and this allows erasing and then programming only a fraction of the sector.

Table 21.2[2] shows how the flash memory is organized in some STM32F0 microcontrollers. As you can see, they can provide up to seventeen sectors, each one in turn divided in four pages. Moreover, a dedicated area, called *Information Block*, is mapped to another address range: this non-volatile memory is used to store special configuration registers (named *Option bytes*) and some factory pre-programmed bootloaders, which we will study in the next Chapter. In more powerful STM32 MCUs, the *Information Block* region also contains the *One-time Programmable* (OTP) memory (which can range from 512 up to 1024 bytes): this is a non-volatile memory that can be used to store relevant configuration parameters of the device.

Why having such memory organization? Before we can answer to this question, we need to introduce some fundamental concepts regarding flash memory technologies. Without detailing it too much, there are two main families of flash memories: NAND and NOR.

NAND-flash memories offer a more compact physical architecture, allowing to store more memory cell in the same silicon area. NAND memories are available in greater storage densities and at lower costs per bit than NOR-flash (remember that in electronics, apart from the R&D costs, the production cost of an IC is all about the die size). NAND memories also have up to ten times the endurance of

[2]The table is extracted from the ST RM0360 reference manual (https://bit.ly/1GfS3iC)

NOR-flash. NAND is more fit as storage media for large files including video and audio. The USB thumb drives, SD cards and MMC cards are of NAND type.

Flash area	Flash memory addresses	Size (byte)	Name	Description(1)
Main Flash memory	0x0800 0000 - 0x0800 03FF	1 Kbyte	Page 0	Sector 0
	0x0800 0400 - 0x0800 07FF	1 Kbyte	Page 1	
	0x0800 0800 - 0x0800 0BFF	1 Kbyte	Page 2	
	0x0800 0C00 - 0x0800 0FFF	1 Kbyte	Page 3	
	. . .	. . .	. . .	. . .
	0x0800 7000 - 0x0800 73FF	1 Kbyte	Page 28	Sector 7 (1)
	0x0800 7400 - 0x0800 77FF	1 Kbyte	Page 29	
	0x0800 7800 - 0x0800 7BFF	1 Kbyte	Page 30	
	0x0800 7C00 - 0x0800 7FFF	1 Kbyte	Page 31	
	. . .	. . .	. . .	. . .
	0x0800 F000 - 0x0800 F3FF	1 Kbyte	Page 60	Sector 15
	0x0800 F400 - 0x0800 F7FF	1 Kbyte	Page 61	
	0x0800 F800 - 0x0800 FBFF	1 Kbyte	Page 62	
	0x0800 FC00 - 0x0800 FFFF	1 Kbyte	Page 63	
Information block	0x1FFF EC00 - 0x1FFF F7FF	3 Kbyte(2)	-	System memory
	0x1FFF C400 -0x1FFF F7FF	13 Kbyte(3)	-	System memory
	0x1FFF F800 - 0x1FFF F80F	2 x 8 byte	-	Option byte

1. On STM32F030x4 devices, the main Flash memory space is limited to sector 3. On STM32F030x6 and STM32F070x6 devices, the main Flash memory is limited to sector 7.
2. STM32F030x4, STM32F030x6 and STM32F030x8 devices
3. STM32F070x6 devices

Table 21.2: Flash memory organization in F030x4, F030x6, F070x6 and F030x8 devices

NAND-flash does not provide a random-access external address bus so the data must be read on a block-wise basis, where each block holds hundreds to thousands of bits, resembling to a kind of sequential data access. This makes NAND-flash technology not suitable for embedded microcontrollers, because most of the microprocessors and microcontrollers require byte-level random access.

An important thing to know about flash memory technologies is that a write operation in any type of flash device can only be performed on an empty or erased unit. So, in most cases a write operation must be preceded by an erase operation. While the erase operation is straightforward in the case of NAND-flash devices, in NOR-flash it is mandatory that all bytes in the target block should be written with all zeros before they can be erased. Conversely, NOR-flash memories offer complete

address and data buses to randomly access any of its memory location (addressable to every byte). This makes them suitable for store code and constant data because they rarely need to be updated.

NOR memories endurance is 10,000 to 100,000 erase cycles. NOR-flash memories are slower in erase-operation and write-operation compared to NAND-flash. That means the NAND-flash has faster erase and write times. Moreover, NAND has smaller erase units. So, fewer erases are needed, and this makes them more suitable to store filesystems. NOR-flash can read data slightly faster than NAND.

NOR-flash devices are divided into erase units, also called blocks, pages or sectors. This division is necessary to reduce prices and overcome physical limitations. Writing information to a specific block can only be performed if that block is empty/erased, as said before. In the majority of NOR-flash memories, after an erase cycle an individual cell contains the value "1", and a write operation allows to change its value to "0". This means that a word memory location is set to `0xFFFF FFFF` after an erase. There exists, however, some NOR-flash memories where the cell-default value after an erase is "0", and we can set it to "1" with a write operation.

Partitioning the flash memory in several blocks gives us an indirect advantage: we can erase and then reprogram only small fractions of the flash memory. This is especially useful when we use the flash memory to store non-volatile configuration parameters, without using dedicated and external EEPROM memories[3].

To completely avoid unwanted writings in the *Non-Volatile Memory* (NVM), the flash memory in all STM32 MCUs is write protected, and there exists a specific unlocking sequence to follow to disable it: two dedicated key registers are provided in the *Option Bytes* region, which allow to disable flash writing protection by issuing a specific value inside them. In some STM32 MCUs the write protection must be individually disabled for each sector. Depending on the STM32 family, the write access is performed by 8-, 16-, 32- or 64-bit.

To protect the intellectual property, the flash memory can be read-protected against external access from debug interface (clearly, the read access is still permitted from the Cortex-M core and DMA controllers). This avoids other malicious users can save the content of flash memory to disassemble or replicate it on counterfeit devices[4]. We will analyze this topic later.

Depending on the STM32 family, the flash memory can perform several program/erase operations in parallel, allowing to write more bytes at once. Particular conditions must be met to carry out program operations in parallels. Usually, a given VDD voltage is required to reach the maximum parallelism. Always consult the reference manual of your MCU to discover more about this.

21.2 The `HAL_FLASH` Module

Like all other STM32 peripherals, even the flash memory provides several registers used to manipulate its settings, as said before. The `HAL_FLASH` module, together with the related `HAL_FLASHEx`

[3]Several STM32 MCUs from the STM32L-series provide a dedicated and true EEPROM memory, like in other low-cost 8-bit microcontrollers (for example, ATMEL AVR microcontrollers).

[4]However, keep in mind that there exists companies able to bypass read-protection using advanced hardware techniques (this usually involves the usage of lasers that overwrite the read-protection bits inside the *Option Bytes* region - it is not inexpensive, but it is possible ;-))

module, allows to easily erase and reprogram the NVM memory without dealing too much with its implementation details. The next subparagraphs introduce the most relevant functions from those modules.

21.2.1 Flash Memory Unlocking

The flash memory is write-protected by default, to prevent accidental writings caused by electrical disturbances or program malfunctions. To enable write mode a sequence of operations must be performed, and this is specific of the given STM32 family. To accomplish this task, the CubeHAL provides the function:

```
HAL_StatusTypeDef HAL_FLASH_Unlock(void);
```

which allows us to completely ignore the specific flash memory architecture. Once the flash memory write/erase protection is disabled, we can perform an erase or write operation. The reverse of the unlock procedure is performed by using the function:

```
HAL_StatusTypeDef HAL_FLASH_Lock(void);
```

The write protection is automatically set upon a system reset. However, it is strongly suggested to explicitly re-lock the memory when all writing operations are completed. This prevents any accidental writing caused by firmware malfunction or power instability.

21.2.2 Flash Memory Erasing

Before we can change the content of a flash memory location, we need to reset its bits to the default value ("0" or "1" depending on the NOR-flash type). This is performed by an erase operation on sector/page granularity. Alternatively, a mass erase of the whole bank can be performed: this means that on those STM32 MCUs providing two banks we can mass erase each bank at a time.

In the majority of STM32 microcontrollers, the individual cells of a flash memory block (sector or page) are set to "1" after an erase operation, with just two notably exceptions: STM32L0 and STM32L1 microcontrollers, whose default value is instead "0".

The CubeHAL provides two ways to perform a flash erase operation: flash erasing in *polling* and *interrupt* mode.

The function:

```
HAL_StatusTypeDef HAL_FLASHEx_Erase(FLASH_EraseInitTypeDef *pEraseInit,
                                    uint32_t *SectorError);
```

allows to perform a flash erasing in *polling* mode. It accepts a pointer to an instance of the `FLASH_EraseInitTypeDef struct`, that we are going to see in a while, and a pointer to variable (`SectorError`) which returns the id of faulty sectors/pages in case of error during the erasing procedure (for example, if the erasing procedure fails on the 4th page, the `SectorError` parameter will contain the value 3).

The `FLASH_EraseInitTypeDef struct` differs a lot between each STM32 family. For this reason, take a look at the `stm32XXxx_hal_flash_ex.h` file of the CubeHAL for your MCU. Here, we are going to consider the implementation found in CubeHALs for the most performing STM32 MCU like the F2/F4/F7 ones.

```
typedef struct {
  uint32_t TypeErase;   /* Mass erase or sector Erase */
  uint32_t Banks;       /* Select banks to erase when Mass erase is enabled */
  uint32_t Sector;      /* Initial FLASH sector to erase when Mass erase is disabled */
  uint32_t NbSectors;   /* Number of sectors to be erased */
  uint32_t VoltageRange;/* The device voltage range which defines the erase parallelism */
} FLASH_EraseInitTypeDef;
```

- `TypeErase`: specifies if we are performing a mass erase of the whole bank or a sector/page erasing. It can assume the values `FLASH_TYPEERASE_SECTORS` or `FLASH_TYPEERASE_MASSERASE`.
- `Banks`: this parameter, which is available only in those STM32-series providing a multi-bank internal flash memory, specifies the bank involved in a mass-erase. It can assume the values `FLASH_BANK_1`, `FLASH_BANK_2` or `FLASH_BANK_BOTH` to delete both the banks.
- `Sector`(`Page`): this field refers to the sector id involved in a sector-based erasing. It can assume the value `FLASH_SECTOR_0`, `FLASH_SECTOR_1` and so on (the maximum number of sectors depends on the specific microcontroller). In those STM32 MCUs providing a flash memory with page granularity, this fields is replaced by the first address of the page involved in an erasing procedure. Consult the CubeHAL source code for more about this.
- `NbSectors`(`NbPages`): the number of sectors (pages) that will be erased starting from the specified `Sector`.
- `VoltageRange`: even if we are erasing a whole sector (or page), the erasing procedure cycles over a subset of it (usually two bytes). More performing STM32 MCUs allows to erase multiple bytes at once. This feature is called *flash parallelism* and it is related to the MCU operating voltage: the higher is VDD, the more bytes are erased at a time[5]. This field can assume a value from **Table 21.3**. However, always consult the reference manual for your MCU for more about this.

[5]STM32L4-series provides a similar feature named *fast program/erase* mode. It is related to both the VDD and the clock speed. It allows to erase/program the flash on a double word granularity. Consult the reference manual for your MCU for more about this.

Table 21.3: Program/erase parallelism depending on the voltage range

`VoltageRange`	Voltage range	*Parallelism*
`FLASH_VOLTAGE_RANGE_1`	1.7 - 2.1 V	8 bits at a time
`FLASH_VOLTAGE_RANGE_2`	2.1 - 2.4 V	16 bits at a time
`FLASH_VOLTAGE_RANGE_3`	2.4 - 3.6 V	32 bits at a time
`FLASH_VOLTAGE_RANGE_4`	2.7 - 3.6 V with External VPP	64 bits at a time

The `HAL_FLASHEx_Erase()` is a blocking function: it will wait until the erasing procedure has been completed. This may be a quite "long" procedure, depending on the STM32 family, the HCLK speed, the number of sector/pages involved in the erasing and the VDD voltage in those STM32 MCU providing program/erase parallelism. To avoid blocking the firmware activities during this procedure, the HAL provides the function:

```
HAL_StatusTypeDef HAL_FLASHEx_Erase_IT(FLASH_EraseInitTypeDef *pEraseInit,
                                       uint32_t *SectorError);
```

which performs an erasing procedure in *interrupt* mode. We can get notified of the end of the erasing procedure by enabling the `FLASH_IRQn` interrupt and implementing the corresponding ISR.



Read Carefully

Special care must be placed in case we are erasing flash memory location containing program code, especially if we are deleting first sector/page containing the *vector table* (this is always true if we are performing a mass-erase). If this the case, then we need to move the program code and relocate the whole vector table inside the SRAM, as shown in Chapter 20, otherwise a fault will occur once the interrupt fires.

21.2.3 Flash Memory Programming

Once a sector/page is erased, we can proceed programming its content. In theory, it is perfect possible to directly access to a flash location to change its content[6] writing a C code like the following one:

```
...
*(volatile uint16_t*)0x0800AA00 = Data;
...
```

However, this is basically not convenient for two main reasons. First of all, in some STM32 MCUs preliminary operations (like setting specific registers) may be required before we can program a flash location. Secondly, depending on the specific STM32-series and the VDD voltage range, the number of bytes that can be simultaneous transferred to the flash may significantly differ. For these reasons, the HAL defines the function:

[6]Obviously, the flash must be unlocked before we can modify it.

```
HAL_StatusTypeDef HAL_FLASH_Program(uint32_t TypeProgram, uint32_t Address, uint64_t Data);
```

which is designed to abstract all specific implementation details. Let us analyze the function arguments:

- `TypeProgram`: it indicates how many bytes are transferred during the write operation, and it can assume the values `FLASH_TYPEPROGRAM_HALFWORD`, `FLASH_TYPEPROGRAM_WORD` and `FLASH_TYPEPROGRAM_DOUBLEWORD`. Please, take note that this parameter specifies only the amount of data transferred using the `HAL_FLASH_Program()` function. The effective number of bytes transferred in a single transaction depends on the STM32 family and the parallelism degree, if available.
- `Address`: it is the initial memory address where start placing content.
- `Data`: it is the data to store inside the flash memory location (represented as a `double word` variable).

Like for the erase procedure seen before, it is possible to perform a flash programming procedure in *interrupt* mode by using the function:

```
HAL_StatusTypeDef HAL_FLASH_Program_IT(uint32_t TypeProgram,
                                       uint32_t Address, uint64_t Data);
```

21.2.4 Flash Read Access During Programming and Erasing

A read access to the flash memory while an erase or write operation is ongoing will cause a bus stall, at least in the majority of STM32 microcontrollers[7]. This means that if you need to carry out other operations in parallel, you need to relocate in SRAM code to be executed during a flash programming operation. A typical scenario is represented by a custom bootloader: we may program our code so that we exchange the new firmware to flash using the UART in *interrupt* or *DMA* mode. If this the case, we cannot lose asynchronous events (for example, an interrupt that notifies us a data transfer) because the MCU is stalled waiting for the ongoing operation. If so, it is best to relocate the code in SRAM (and eventually to relocate the *vector table* too).

21.3 Option Bytes

Option bytes are two or more bytes whose bits are special configuration values. The concept of *option bytes* is similar to the one found in other microcontroller architectures, like the *fuses* in the AVR series from Atmel or the *Configuration Bits* found in PIC microcontrollers from Microchip.

Each individual bit of these special bytes in the *Information Block* region has a special meaning. The number and type of configuration parameters depend on the specific STM32 MCU. The most common configuration parameters are related to:

[7] In some STM32 MCUs, like the STM32L0-series, a bus fault may occur if we try to access the flash memory while a half-page program operation is ongoing. For more information, consult the reference manual for the MCU you are considering.

- **BOOT**: in the majority of STM32 microcontrollers two option bits allow to select the boot origin (FLASH, *System memory* or SRAM).
- **RDP**: these bits set the flash memory read-protection level, and we will analyze them more in depth later in this chapter.
- **BOR_LEVEL**: these bits contain the supply level threshold that activates/releases the reset. They can be written to program a new BOR level. By default, BOR is off. When the supply voltage (VDD) drops below the selected BOR level, a device reset is generated.
- **MCU behavior when entering in some low-power modes**: in almost all STM32 microcontrollers it is possible to configure the MCU so that it generates a reset when entering in *stop* or *sleep* low-power modes.
- **Hardware watchdog**: in some STM32 MCUs, there exist one or two bits used to configure the WWDG and IWDG in "hardware mode", that is they are automatically started upon a MCU reset.
- **Flash write protection**: these bits allow to individually write-protect some flash sectors/pages, preventing from writing into them even if the flash memory is unlocked. If a given bit is set to '1', the corresponding sector/page is not write-protected; if, instead, the bit is set to '0', then the sector/page is write-protected.

To program the *option bytes* there is a specific procedure to follow, which is independent from the programming of the whole flash memory. So, the CubeHAL provides dedicated routines to use.

First of all, this region must be unlocked by calling the function:

```
HAL_StatusTypeDef HAL_FLASH_OB_Unlock(void);
```

Next, a give option byte is programmed entirely by using the function:

```
HAL_StatusTypeDef HAL_FLASHEx_OBProgram(FLASH_OBProgramInitTypeDef *pOBInit);
```

The value of an option byte is automatically modified by first erasing the information block and then programming all the option bytes with the values passed to the `HAL_FLASHEx_OBProgram()` routine. The function accepts an instance of the C `struct FLASH_OBProgramInitTypeDef`, whose fields represent the content of the given option byte. For more information about the exact type and number of fields consult the source code of the CubeHAL.

Similarly, to retrieve the content of a given option byte we use the function:

```
void HAL_FLASHEx_OBGetConfig(FLASH_OBProgramInitTypeDef *pOBInit);
```

Once an option byte is modified, we have to force the MCU to reload its content by using the function:

```
HAL_StatusTypeDef HAL_FLASH_OB_Launch(void);
```

Please take note that changing some option bits in some STM32 MCUs may cause a reset of the chip.

Finally, the ST STM32CubeProgrammer provides the ability to easily modify the option bytes. Once you have connected the ST-LINK debugger to the target MCU, click on the **Option bytes** icon (the third green icon on the left). The **Option bytes** section appears, as shown in **Figure 21.1**. The same STM32CubeProgrammer tool also allows to erase selected flash sectors/pages.

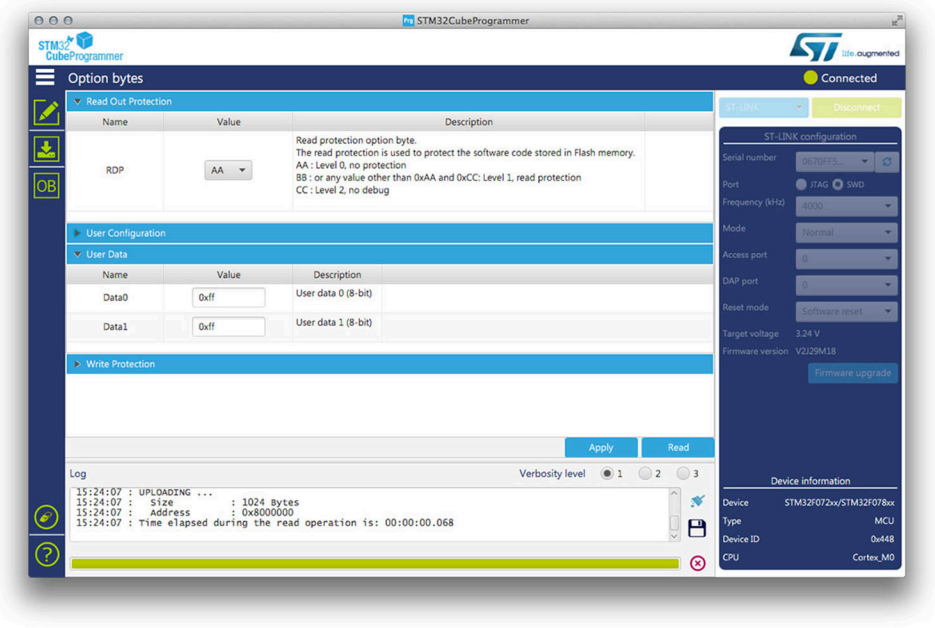


Figure 21.1: The *Option Bytes* configuration dialog in the STM32CubeProgrammer

21.3.1 Flash Memory Read Protection



Read Carefully

Some procedures described in this paragraph may brick your microcontroller preventing you from flashing and erasing it forever. Read carefully the content of this paragraph and avoid performing operations if they are not **totally clear**.

One option byte (called **RDP**) deserves a separated paragraph: the configuration byte related to the flash read protection. To avoid unwanted access to the flash memory through the debug interface it is possible to temporarily or **permanently** disable the read access to this memory from the external world (clearly, the access from the CPU core and the DMA controllers is always possible). There exist three protection levels, which correspond to three different values to store in the option byte:

- **Level 0 (no read protection)**: when the read protection level is set to Level 0 by writing **0xAA** into the read protection option byte (RDP), all read/write operations (if no write protection is set) from/to the flash memory or the backup SRAM are possible in all boot configurations (flash user boot, debug or boot from RAM).
- **Level 1 (read protection enabled)**: it is the default read protection level after option bytes erase (which is automatically performed by the `HAL_FLASHEx_OBProgram()` routine). The read protection Level 1 is activated by writing **any value** (**except for 0xAA and 0xCC used to set Level 0 and Level 2, respectively**) into the RDP option byte. When the read protection Level 1 is set, no access (read, erase, program) to flash memory or backup SRAM can be performed while the debugger is connected or while booting from RAM or system memory bootloader. A bus error is generated in case of read request. Instead, when booting from flash memory, accesses (read, erase, program) to flash memory and backup SRAM from user code are allowed. When Level 1 is active, programming the protection option byte (RDP) to Level 0 causes the flash memory and the backup SRAM to be mass-erased. As result, the user code area is cleared before the read protection is removed. The mass erase only erases the user code area. The other option bytes including write protections remain unchanged before the mass-erase operation. The OTP area is not affected by mass erase and remains unchanged. Mass erase is performed only when Level 1 is active and Level 0 requested. When the protection level is increased (0->1, 1->2, 0->2) there is no mass erase.
- **Level 2 (!!!debug/chip read protection permanently disabled!!!)**: the read protection Level 2 is activated by writing **0xCC** to the RDP option byte. When the read protection Level 2 is set:
  - All protections provided by Level 1 are active.
  - Booting from RAM is no more allowed.
  - Booting system memory bootloader is possible and all the commands are not accessible except Get, GetID and GetVersion. Refer to AN2606.
  - JTAG, SWV (single-wire viewer), ETM, and boundary scan are disabled.
  - User option bytes can no longer be changed.
  - When booting from Flash memory, accesses (read, erase and program) to Flash memory and backup SRAM from user code are allowed.



Memory read protection Level 2 is an irreversible operation. When Level 2 is activated, the level of protection cannot be decreased to Level 0 or Level 1. Just to clarify once again, ***this means that you will be no longer able to flash and debug your MCU***.

Table 21.4 summarizes the effects of a given protection level on:

- the flash memory;
- the option bytes and the OTP memory (when these memories are accessed by the debugger interface);
- the pre-programmed bootloaders;

- code placed in SRAM and in flash memory.

As you can see, the **Level 2** does not prevent user code from writing into flash memory (for example, a custom bootloader is still able to program the MCU).

Memory Area	Protection Level	Debug features, Boot from RAM or from System memory bootloader			Booting from Flash memory		
		Read	Write	Erase	Read	Write	Erase
Main Flash Memory and Backup SRAM	Level 0	YES			YES		
	Level 1	NO	NO	YES	YES		
	Level 2	**NO**			**YES**		
Option Bytes	Level 0	YES			YES		
	Level 1	YES			YES		
	Level 2	**NO**			**NO**		
OTP memory	Level 0	YES		N/A	YES		N/A
	Level 1	NO		N/A	YES		N/A
	Level 2	NO		N/A	YES		N/A

Table 21.4: The effects of read protection levels on the individual NVM memories

21.4 Optional OTP and True-EEPROM Memories

More recent and powerful STM32 microcontrollers provide a *One-Time Programmable* (OTP) memory. This is a dedicated memory with a size ranging from 512 up to 1024 bytes with a unique characteristic: once a bit of this memory turns from 1 to 0 is no longer possible to restore it to 1. This means that this region is not erasable. This memory area is especially useful to store relevant configuration parameters connected with the given device, such as serial numbers, MAC address, calibration values and so on. A typical practice in the electronics industry is to produce devices with different functionalities starting from the same PCB or even the same complete board. This area could be also used to store configuration parameters employed by the firmware to adapt board features.

The OTP area is divided into *N* OTP data blocks of 32 bytes and one lock OTP block of *N* bytes. The OTP data and lock blocks cannot be erased. The lock block contains *N* bytes $LOCKB_i$ ($0 \leq i \leq N-1$) to lock the corresponding OTP data block (blocks 0 to *N*). Each OTP data block can be programmed until the value 0x00 is programmed in the corresponding OTP lock byte (clearly an individual bit already set to 0 cannot be restored to 1). The lock bytes must only contain 0x00 and 0xFF values, otherwise the OTP bytes might not be considered correctly.

Block	[128:96]	[95:64]	[63:32]	[31:0]	Address byte 0
0	OTP0	OTP0	OTP0	OTP0	0x1FFF 7800
	OTP0	OTP0	OTP0	OTP0	0x1FFF 7810
1	OTP1	OTP1	OTP1	OTP1	0x1FFF 7820
	OTP1	OTP1	OTP1	OTP1	0x1FFF 7830
. . .	. . .				. . .
15	OTP15	OTP15	OTP15	OTP15	0x1FFF 79E0
	OTP15	OTP15	OTP15	OTP15	0x1FFF 79F0
Lock block	LOCKB15 ... LOCKB12	LOCKB11 ... LOCKB8	LOCKB7 ... LOCKB4	LOCKB3 ... LOCKB0	0x1FFF 7A00

Table 21.5: The organization of the OTP memory in an STM32F401RE MCU

Table 21.5 shows the organization of the OTP memory in an STM32F401RE MCU, and it is extracted from the related reference manual. As you can see, this MCU provides 16 OTP data blocks, with a total of 512 bytes. Sixteen lock bytes allow to lock the corresponding OTP data block.

Another common practice in digital electronics is to use dedicated and often external EEPROM memories to store configuration parameters. EEPROM memories have several benefits compared to the flash ones:

- Their blocks can be individually erased.
- Each block can be erased up to and even more than 1.000.000 times (flash erase cycles may be limited to 10.000 cycles).
- The rated lifetime is usually higher than flash memories.
- They are usually cheap than flash (NOR and NAND) memories.
- There exist EEPROM memories able to operate up to 200°C.

However, the main drawback of EEPROM memories is that they are usually much slower than flash memories and occupy additional space on PCB.

If your design is all about reducing the BOM cost, then ST provides several application notes that describe how to emulate an EEPROM memory using the STM32 integrated flash memory (these application notes are titled "*EEPROM emulation in STM32Fxx microcontrollers*"). Recently, ST also released a dedicated STM32Cube Extension package with a lot of interesting functionalities including wear leveling algorithm to increase emulated EEPROM cycling capability.
Finally, several MCUs from the STM32L-series provide an integrated true-EEPROM. For more information, consult the datasheet of your MCU.

21.5 Flash Read Latency and the ART™ Accelerator

In Chapter 1 we have seen that Cortex-M cores provide an *n-stage*[8] *instruction pipeline* designed to boost the program execution. However, that pipeline has to be filled with machine instructions normally stored inside the flash memory. This operation is a substantial bottleneck, because flash memories are slower if compared to the CPU clock speed.

If both the CPU and the flash memory run at the same speed, the CPU can feed its internal pipeline without any penalty[9]. For example, an STM32F401RE MCU running at a clock speed lower than 30MHz can access to the flash memory without delays. Unfortunately, in more performing MCUs it is required to interleave two successive accesses to the flash memory with one or more (in some cases even up to ten) delays, called *wait states*. Wait states correspond to hardware "busy waits" performed in one or more CPU cycles, and they are a way to synchronize the CPU with the slower flash memory. Wait states dramatically reduce the effective performances of the CPU. This limitation is usually addressed by using dedicated cache memories.

Configuring the exact number of needed wait states is a critical step that depends on the specific STM32 MCU you are considering. This operation is usually performed during the SYSCLK configuration, because the higher the CPU frequency is the more wait states are needed. Configuring the correct number of wait states is critical especially when we are increasing the CPU speed: we have to setup the right number of wait states before we increase the CPU speed, otherwise a *BusFault* is generated. However, CubeMX is designed to abstract these details, and it generates the right configuration code depending on the specific STM32 MCU and the wanted core speed (take a look at the code inside the `SystemClock_Config()` routine).

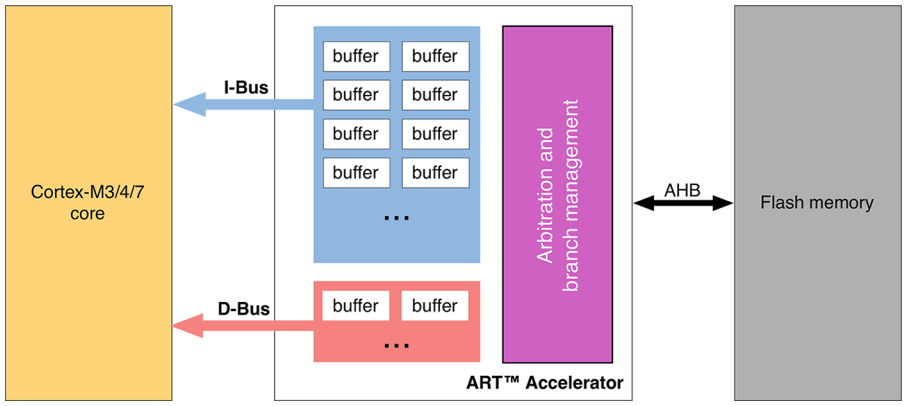


Figure 21.2: The main blocks forming the ART™ Accelerator

ST has developed a distinctive technology available in its more powerful STM32 microcontrollers: the *ART™ Accelerator*. The ART™ Accelerator is a pool of cache technologies (see **Figure 21.2**), external to Cortex-M core, which can zero the effects of wait states. The ART™ Accelerator is

[8]The exact number of pipeline stages depend on the specific Cortex-M core.

[9]Talking about "speed" in this context is improper because we should talk about the "latency" needed to perform a machine operation. This latency is essentially formed by the time needed by the CPU to decode and execute a machine instruction, plus the time needed by the flash controller to retrieve the given instruction from the NVM memory. However, here we are interested to the fact that these two "devices" (the CPU and the flash memory with its controller) may need different amount of time to carry out their activities.

designed so that it preservers the *Harvard architecture* of Cortex-M microcontrollers, providing separated cache pools for the *I-Bus* and the *D-Bus*.

The ART™ Accelerator is composed by:

- an instruction prefetch buffer;
- a dedicated instruction cache to reduce the effects of branching;
- a data cache for literal pools;
- a scheduling policy of the AHB bus that facilitates the access of the CPU to the flash controller through the *D-Bus* bus.

Let us analyze the exact role of these technologies.

The Instruction Prefetch Buffer
When the CPU accesses to the flash memory, it does not fetch one byte at a time, but it usually reads from 64 up to 256 bits at a time depending on the specific STM32 MCU. These bits contain a variable number of instructions and for this reason they are called *instruction lines*: assuming that the CPU reads 128 bits (this is what happens in STM32F4 MCUs), this may contain four 32-bit wide instructions or eight 16-bit wide instructions (it depends if the CPU is running in *thumb mode* or not). So, in case of sequential code, at least four CPU cycles are needed to execute the previous read instruction line. Prefetch on the *I-Bus* bus can be used to read the next sequential instruction line from the flash memory while the current instruction line is being requested by the CPU. This feature is useful if at least one wait state is needed to access the flash memory.

Instruction prefetch buffer can be enabled by setting the `PREFETCH_ENABLE` macro to 1 inside the `stm32XXxx_hal_conf.h` file.

The Instruction Cache Memory
The content of the prefetch buffer can be invalided due branching. To limit the time lost due to jumps, it is possible to retain a given number of instruction lines in an instruction cache memory. Each time a miss occurs (requested data not present in the currently used instruction line, in the prefetched instruction line or in the instruction cache memory), the line read is copied into the instruction cache memory. If the CPU requests data contained in the instruction cache memory, it is provided without inserting any delay. Once all the "empty" instruction cache memory lines have been filled, a *Least Recently Used* (LRU) policy is used to determine the line to replace in the instruction memory cache. This feature is particularly useful in case of code containing loops.

This feature can be enabled by setting the `INSTRUCTION_CACHE_ENABLE` macro to 1 inside the `stm32XXxx_hal_conf.h` file, for those MCU providing the ART™ Accelerator. **Data Cache Memory**
Assembly instructions often move data between memory locations and CPU registers. Sometimes, this data is stored inside the flash memory (they are constant values): in this case, we talk about *literal pools*. Literal pools are fetched from flash memory through the *D-Bus* bus during the execution stage of the CPU pipeline. The CPU pipeline is consequently stalled until the requested literal pool is provided. To limit the time lost due to literal pools, accesses through the AHB data-bus *D-Bus* have priority over accesses through the AHB instruction bus *I-Bus* (this is indeed a bus-arbitration policy over the *D-Bus* bus).

Moreover, a dedicated data cache memory exists between the *D-Bus* bus and the flash memory. This cache is smaller than the instruction cache, but it helps increasing the overall performances of the CPU. This feature can be enabled by setting the `DATA_CACHE_ENABLE` macro to 1 inside the `stm32XXxx_hal_conf.h` file, for those MCU providing the ART™ Accelerator.

21.5.1 The Role of the TCM Memories in STM32F7/H7 MCUs

The memory organization of more recent and powerful STM32F7/H7 MCUs deserves a separate mention. In fact, this family of microcontrollers faces a more complex and flexible memory and bus organization, offering two distinct interfaces to access flash and SRAM memories: the *Advanced eXtensible Interface* (AXI), which is an ARM bus specification that interconnects the CPU core to the other peripherals; the *Tightly-Coupled Memory* (TCM) interface which interconnects the CPU core to volatile and non-volatile memories directly coupled with it. Both the interfaces, AXI and TCM, face a Harvard architecture, providing separated lines for instructions (*I-Bus*) and data (*D-Bus*).

Looking at **Figure 21.3**[10], you can see that the Cortex-M7 core has three distinct paths to access the flash controller (and so the flash memory). Before we describe these three paths, it is important to note a fundamental thing: the Cortex-M7 core already provides an integrated L1-cache. This cache has two dedicated cache pools, each one up to 64KB wide, one dedicated to the *I-Bus* and one for the *D-Bus*: this differs from other STM32 families, where data and instruction caches are implemented exclusively inside the ART™ Accelerator.

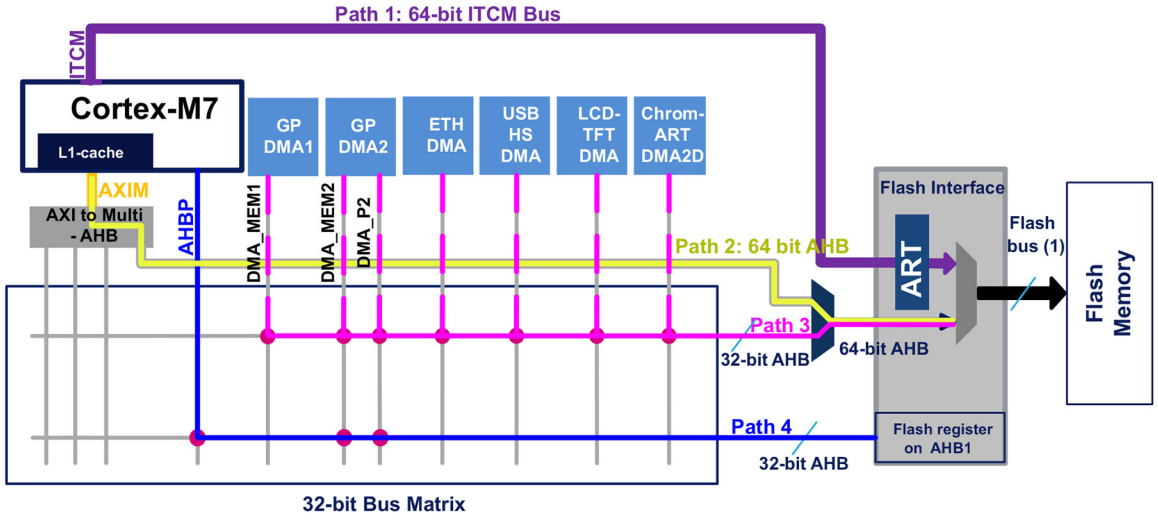


Figure 21.3: How the flash memory is accessed in an STM32F7 MCU

In all STM32F7 MCUs, flash memory is accessible through three main interfaces for read and/or write accesses:

- **A 64-bit ITCM interface**: it connects the embedded flash memory to the Cortex-M7 via the ITCM bus (Path 1 in **Figure 21.3**) and it is used for the program execution and data read

[10]The figure is taken from the AN4667 from ST(https://bit.ly/29gmp61).

access for literal values. **The write access to the flash memory is not permitted via this bus**. The flash memory is accessible by the CPU through ITCM starting from the address `0x0020 0000`. Being the embedded flash memory slower than the CPU core, the ART™ Accelerator allows *0-wait* execution from the flash memory at a CPU frequency up to 216MHz in STM32F7 and up to 480MHz in STM32H7. The ART™ Accelerator is available only for a flash memory access on the ITCM interface. It implements a unified instruction and branch cache up to 256 bits x 64 lines. The ART™ Accelerator is available for both the instruction and data access, which increases the execution speed of sequential code and loops. The ART™ Accelerator also implements an instruction prefetch buffer.

- **A 64-bit AHB interface**: it connects the embedded flash memory to the Cortex-M7 via the AXI/AHB bridge (Path 2 in **Figure 21.3**). It is used for the code execution, read and write accesses. The flash memory is accessible by the CPU through AXI/AHB bridge starting from the address `0x0800 0000` and it is cacheable (that is, it can use the L1-cache) reaching the same *0-wait* performances of the ART™ Accelerator. The L1-cache in Cortex-M7 cores can range from 4KB to 16KB. STM32F7/H7 provide two cache pools, one for the instructions (*I-Bus*) and one for the literal pools (*D-Bus*), each one up to 16KB wide. The L1-caches on all Cortex-M7 cores are divided into lines of 32 bytes. Each line is tagged with an address. The data cache is 4-way set associative (four lines per set) and the instruction cache is 2-way set associative. This is a hardware compromise to keep from having to tag each line with an address.
- **A 32-bit AHB interface**: it is used for DMAs transfers from the flash memory (Path 3 in **Figure 21.3**). The DMAs flash memory access is performed starting from the address `0x0800 0000`.

A fourth path exists (see **Figure 21.3**) through the *Advanced Bus Peripheral* (AHBP) interface, and it is reserved to the access to flash peripheral registers inside the `0x4000 0000` peripheral mapped region.

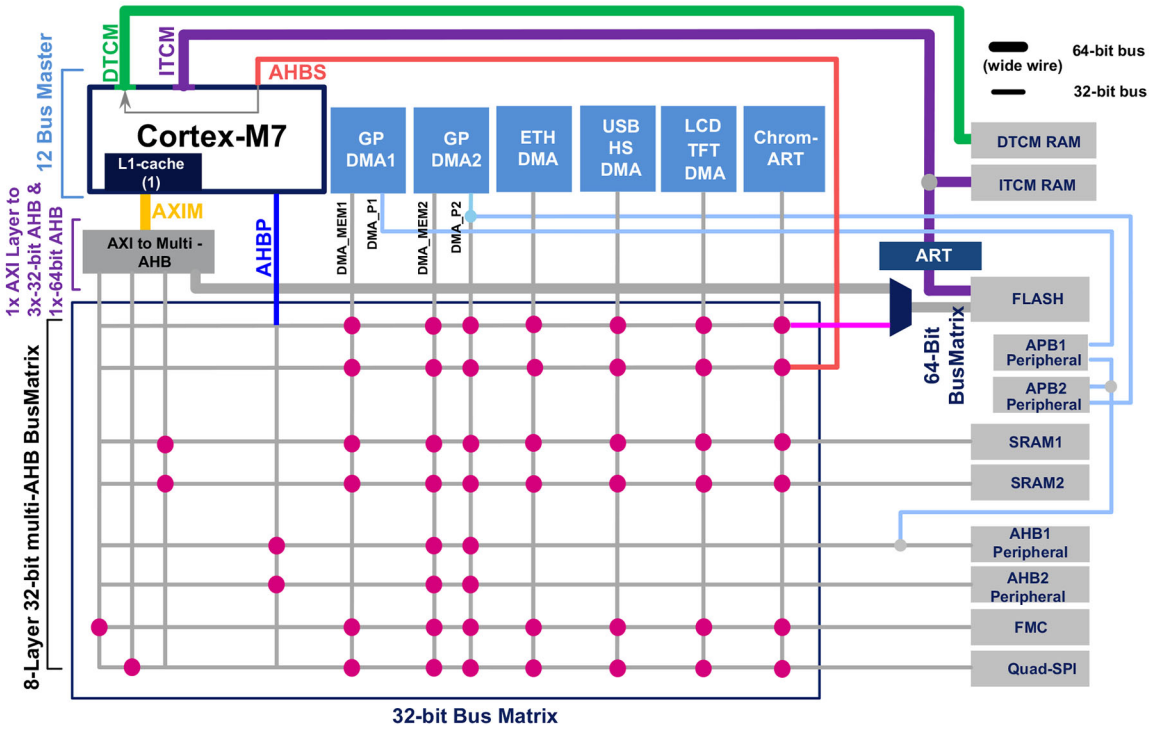


Figure 21.4: The bus matrix in an STM32F7 MCU

What is the advantage of this apparently complex architecture? If both the flash interfaces, that is the AXI/AHB and the ITCM, provide *0-wait* execution (one thanks to internal L1-cache and one thanks to the ART™ Accelerator), why we should deal with this complexity during the firmware design?

The answer comes from the bus-matrix architecture of an STM32F7 MCU, which is shown in **Figure 21.4**[11]. As you can see, the AXI/AHB bus is connected to the internal L1-cache thanks to the AXIM interface. This means that accesses to some peripherals on the bus are *cacheable*. And this is the case of the FMC and QuadSPI controllers. Thanks to this architecture, it is possible to use external NVM memories to store data or program code, taking advantage of the 64K L1-cache, while having parallel access (without the bus arbitration) to the internal flash memory through the ITCM interface and the ART™ Accelerator. This is a great performance boost for devices that make use of a lot of memory to store images, videos and multimedia content in general, but also of large constant data table, like FFT IV.

The CMSIS layer for Cortex-M7 based MCUs defines a dedicated set of routines to manipulate Cortex-M7 L1-cache memory (see **Table 21.6**).

[11]The figure is taken from the AN4667 from ST(https://bit.ly/29gmp61).

Table 21.6: CMSIS functions to manipulate Cortex-M7 L1-caches

CMSIS-F7 Function	Description
`void SCB_EnableICache(void)`	Invalidate and then enable the instruction cache
`void SCB_DisableICache(void)`	Disable the instruction cache and invalidate its contents
`void SCB_InvalidateICache(void)`	Invalidate the instruction cache
`void SCB_EnableDCache(void)`	Invalidate and then enable the data cache
`void SCB_DisableDCache(void)`	Disable the data cache and then clean and invalidate its contents
`void SCB_InvalidateDCache(void)`	Invalidate the data cache
`void SCB_CleanDCache(void)`	Clean the data cache
`void SCB_CleanInvalidateDCache(void)`	Clean and invalidate the data cache

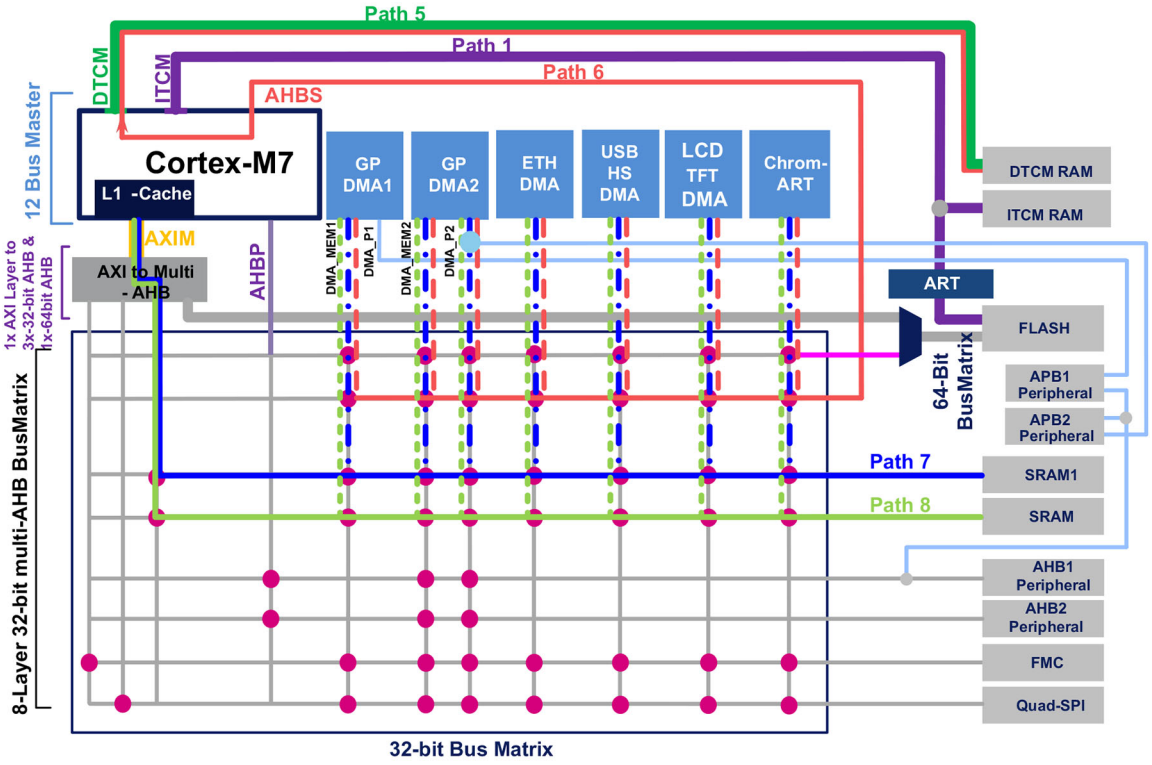


Figure 21.5: The four SRAM memories available in STM32F7 microcontrollers

Looking at **Figure 21.5**[12], there is another important thing to note. As you can see, STM32F7/H7 microcontrollers offer four distinct SRAM memories, accessible through three separated paths:

- **The instruction RAM (ITCM-RAM)**, mapped at the address `0x0000 0000` and accessible only by the core, that is, through Path 1 in **Figure 21.5**. It is accessible by bytes, half-words (16 bits), words (32 bits) or double words (64 bits). The ITCM-RAM can be accessed at a maximum CPU clock speed without latency. The ITCM-RAM is protected from a bus contention since only the CPU can access to this RAM region. The ITCM-RAM plays the same role of the CCM memory

[12]The figure is taken from the AN4667 from ST(https://bit.ly/29gmp61).

in other STM32 MCUs.

- **The data RAM (DTCM-RAM)**, mapped on the TCM interface at the address `0x2000 0000` and accessible by all AHB masters from the AHB bus Matrix: by the CPU through the DTCM bus (Path 5 in **Figure 21.5**) and by DMAs through the specific AHBS "bridge" in the Cortex-M7 core (Path 6 in **Figure 21.5**). It is accessible by bytes, half-words (16 bits), words (32 bits) or double words (64 bits). The DTCM-RAM is accessible at a maximum CPU clock speed without latency. The concurrent accesses to the DTCM-RAM by the masters (core and DMAs) and their priorities can be handled by the slave control register of the Cortex-M7 core (`CM7_AHBSCR` register). A higher priority can be given to the CPU to access the DTCM-RAM versus the other masters (DMAs). For more details of this register, please refer to "ARM Cortex-M7 processor Technical Reference Manual".
- **The SRAM1**, accessible by all the AHB masters from the AHB bus Matrix, that is, all general purpose DMAs as well as dedicated DMAs. The SRAM1 is accessible by bytes, half-words (16 bits) or words (32 bits). Refer to **Figure 21.5** (Path 7) for possible SRAM1 accesses. It can be used for the data load/store as well as the code execution (even if it does not offer any specific performance boost).
- **The SRAM2**, accessible by all the AHB masters from the AHB bus matrix. All the general purpose DMAs as well as the dedicated DMAs can access to this memory region. The SRAM2 is accessible by bytes, half-words (16 bits) or words (32 bits). Refer to **Figure 21.5** (Path 8) for possible SRAM2 accesses. It can be used for the data load/store as well as the code execution (even if it does not offer any specific performance boost).

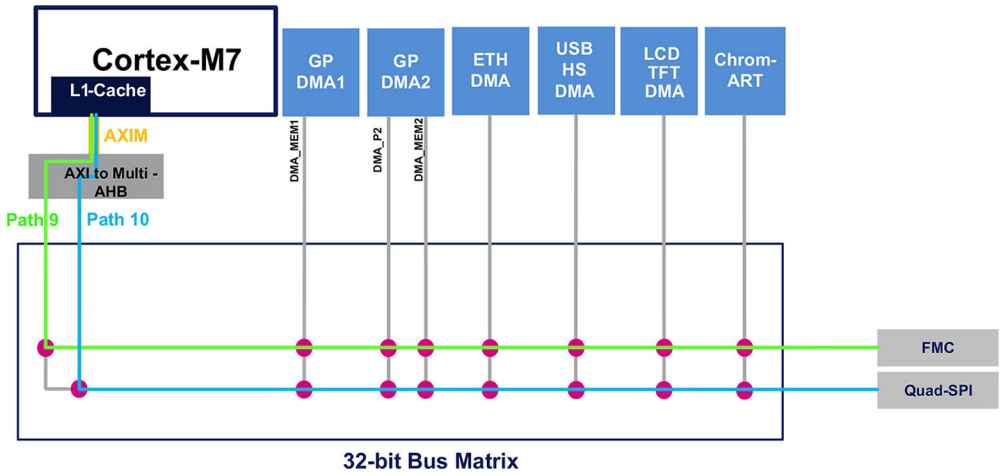


Figure 21.6: FMC and QuadSPI external memory controllers

In addition to the internal flash and SRAM memories, STM32F7 memory pools can be extended using the *Flexible Memory Controller* (FMC) and the Quad-SPI controller. **Figure 21.6**[13] shows the paths that connect the CPU with these external memories via the AXI bus. As shown in **Figure 21.6**, the external memories can benefit of the Cortex-M7 L1-cache, reaching the maximum of the performances both while loading/storing data or during the code execution. The Cortex-M7 L1-cache

[13]The figure is taken from the AN4667 from ST(https://bit.ly/29gmp61).

offers a great performance improvement to STM32F7 microcontrollers compared to the STM32F4 with the same external memory controllers.

Table 21.7 summarizes the memory types, both internal and external to the MCU, available in STM32F74xxx/STM32F75xxx MCUs. The table shows the size of these memories, how they are mapped and the bus interface used to access them. For example, you can see that the address range `0x0020 0000 - 0x002F FFFF` allows to access to the internal flash memory through the ITCM interface, which is *cacheable* thanks to the ART accelerator. **Table 21.8** summarizes the same memories for the STM32F76xxx/STM32F77xxx MCUs (the FMC and QSPI characteristics are the same and so they are not listed in **Table 21.8**).

For more information about these topics, it is strongly suggested to have a look at the AN4667 from ST[14].

Memory Type	Memory region	Address range	Size	Cacheable	Access interfaces
Flash	FLASH-ITCM	**0x0020 0000-0x002F FFFF**	1 MB	YES (ART™)	ITCM (64-bit)
	FLASH-AXIM	**0x0800 0000-0x080F FFFF**		YES (L1-cache)	AHB (64-bit/32-bit)
RAM	DTCM-RAM	**0x2000 0000-0x2000 FFFF**	64 KB	YES (ART™)	DTCM (64-bit)
	ITCM-RAM	**0x0000 0000-0x0000 3FFF**	16 KB	YES (ART™)	ITCM (64-bit)
	SRAM1	**0x2001 0000-0x2004 BFFF**	240 KB	YES (L1-cache)	AHB (32-bit)
	SRAM2	**0x2004 C000-0x2004 FFFF**	16 KB	YES (L1-cache)	
FMC	NOR FLASH/RAM	**0x6000 0000-0x6FFF FFFF**	Up to 256MB	YES (L1-cache)	AHB (32-bit)
	NAND FLASH	**0x8000 0000-0x8FFF FFFF**		YES (L1-cache)	
	SDRAM1	**0xD000 0000-0xDFFF FFFF**		NO	
	SDRAM2	**0xC000 0000-0xCFFF FFFF**		NO	
Quad-SPI	QSPI FLASH	**0x6000 0000-0x6FFF FFFF**	Up to 256MB	YES (L1-cache)	AHB (4-bit/32-bit)

Table 21.7: Memory mapping and sizes in STM32F74xxx/STM32F75xxx MCUs

[14] https://bit.ly/29gmp61

Memory Type	Memory region	Address range	Size	Cacheable	Access interfaces
Flash	FLASH-ITCM	0x0020 0000-0x003F FFFF	2 MB	YES (ART™)	ITCM (64-bit)
	FLASH-AXIM	0x0800 0000-0x081F FFFF		YES (L1-cache)	AHB (64-bit/32-bit)
RAM	DTCM-RAM	0x2000 0000-0x2001 FFFF	128 KB	YES (ART™)	DTCM (64-bit)
	ITCM-RAM	0x0000 0000-0x0000 3FFF	16 KB	YES (ART™)	ITCM (64-bit)
	SRAM1	0x2002 0000-0x2007 BFFF	368 KB	YES (L1-cache)	AHB (32-bit)
	SRAM2	0x2007 C000-0x2007 FFFF	16 KB	YES (L1-cache)	

Table 21.8: Memory mapping and sizes in STM32F76xxx/STM32F77xxx MCUs

21.5.1.1 How to Access Flash Memory Through the TCM Interface

A common question to all novices of the STM32F7 platform is how to take advantage of the TCM interface. This is clearly a linker script job, which has to remap the addresses of `.text`, `.bss` and `.data` regions using as base addresses the ones reported in **Table 21.7** and **21.8**.

However, this operation cannot be easily performed by changing the starting address of the `FLASH` region inside the linker script. This because, as said before, the access in write-mode through the ITCM interface is not permitted. This means that the ST-LINK Debugger, or any equivalent debugger, would not be able to load the program code using the address range `0x0020 0000 - 0x002F FFFF`. To address this limitation, we need to separate the VMA address range from the LMA one, in the same way we have done for the `.data` region. For example, the following linker script fragment shows how to perform this operation.

```
1  /* Specify the memory areas */
2  MEMORY {
3      ITCM_FLASH (rx): ORIGIN = 0x00200000, LENGTH = 1024K
4      AXI_FLASH  (rx): ORIGIN = 0x08000000, LENGTH = 1024K
5      RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 320K
6  }
7
8  /* Define output sections */
9  SECTIONS
10 {
11   /* The startup code goes first into FLASH */
12   .isr_vector :
13   {
14     . = ALIGN(4);
15     KEEP(*(.isr_vector)) /* Startup code */
16     . = ALIGN(4);
17   } >ITCM_FLASH AT>AXI_FLASH
18
```

```
19   /* The program code and other data goes into FLASH */
20   .text :
21   {
22     . = ALIGN(4);
23     *(.text)           /* .text sections (code) */
24     *(.text*)          /* .text* sections (code) */
25
26     KEEP (*(.init))
27     KEEP (*(.fini))
28
29     . = ALIGN(4);
30     _etext = .;        /* define a global symbols at end of code */
31   } >ITCM_FLASH AT>AXI_FLASH
32
33   /* Constant data goes into FLASH */
34   .rodata :
35   {
36     . = ALIGN(4);
37     *(.rodata)         /* .rodata sections (constants, strings, etc.) */
38     *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
39     . = ALIGN(4);
40   } >ITCM_FLASH AT>AXI_FLASH
```

As you can see (look at lines 17, 31 and 40), the VMA address range (that is the address range used by the CPU to fetch program code) is mapped to the ITCM-FLASH interface, while the LMA address range (that is the address range used to store the program in flash memory) is mapped to the AXI interface, which allows to access to flash memory in write-mode.

21.5.1.2 Using CubeMX to Configure Flash Memory Interface

CubeMX simplifies the configuration of the bus used to access flash memory (TCM/AXI), of the ART™ Accelerator and Cortex-M7 L1-cache. Going into *Pinout view* section and then clicking on the **Cortex-M7** entry, it is possible to configure these parameters, as shown in **Figure 21.7**.

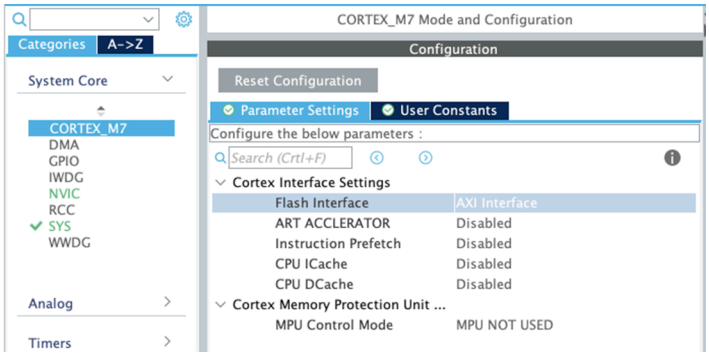


Figure 21.7: The Cortex-M7 configuration view in CubeMX



Please, take note that at the time of writing this chapter (October 2021) the generated linker script is wrong, because it does not specify distinct LMA and VMA addresses, as shown in the previous paragraph.

22. Booting Process

In Chapter 20 we have seen that the handler of the *Reset* exception corresponds to the first routine to be executed when the CPU starts. The fixed memory layout model of Cortex-M based processors establishes that the address in memory of *Reset* exception handler is placed just after the *Main Stack Pointer* (MSP), that is at the address `0x0000 0004`. This memory location usually corresponds to the beginning of flash memory. However, silicon vendors can bypass this limitation by "aliasing" other memories to the `0x0000 0000` address with an operation called *physical remapping*. This operation is performed in hardware after few clock cycles, and it is different from the *vector table* relocation seen in Chapter 20, which is performed by the same code running on the MCU.

Moreover, the STM32 platform provides a factory pre-programmed boot loader, which can be used to load the firmware inside the flash memory from several sources. Depending on the STM32 family and sales type used, an STM32 MCU can load the code using USART, USB, CAN, I²C and SPI communication peripherals. The bootloader is selected thanks to specific boot pins.

This chapter completes the Chapter 20 by showing the booting process performed by STM32 microcontrollers after a system reset. It gives a detailed description of the steps involved during the bootstrap and it briefly shows how to use the factory pre-programmed bootloader in all STM32 MCUs. Finally, a custom bootloader is also shown, which allows to upgrade the on-board firmware using the USART interface and a custom upload procedure.

22.1 The Cortex-M Unified Memory Layout and the Booting Process

Different from more advanced microprocessor architectures, like the ARM Cortex-A, Cortex-M microcontrollers do not provide a *Memory Management Unit* (MMU), which allows to alias logical addresses to actual physical addresses. This means that, from the Cortex-M core point of view, the memory map is fixed and standardized among all implementations.

In Cortex-M based microcontrollers, the code area starts from the `0x0000 0000` address (accessed through the *I-Bus*/*D-Bus*[1] buses in Cortex-M3/4/7 and through the *S-Bus* in Cortex-M0/0+) while the data area (SRAM) starts from address `0x2000 0000` (accessed through the *S-Bus*). Cortex-M CPUs always fetch the *vector table* from the *I-Bus*, which implies that they only boot from the code area (which typically correspond to flash memory).

STM32 microcontrollers implement a special mechanism, called *physical remap*, to boot from other memories than the flash, which consists in sampling two dedicated MCU pins, called BOOT0 and

[1] For more information about these buses, refer to the Chapter 9.

BOOT1[2]. The electrical status of these pins establishes the boot starting address, and hence the source memory.

Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as the boot space
0	1	System memory	System memory is selected as the boot space
1	1	Embedded SRAM	Embedded SRAM is selected as the boot space

Table 22.1: The boot modes available in an STM32F401RE MCU

Table 22.1 shows the boot modes available in an STM32F401RE MCU, and it is extracted from the relative reference manual. The 'x' inside the BOOT1 column means that, when the BOOT0 pin is tied to the ground, the BOOT1 pin logical state can be arbitrary. The first row corresponds to the most common booting mode: the MCU will alias the flash memory to the address `0x0000 0000`. The other two boot modes correspond to booting from the internal SRAM and the *System Memory*, a ROM memory containing a bootloader in all STM32 MCUs and that we will study later.

The status of the BOOT pins is latched on the 4th rising edge of SYSCLK after a reset. It is up to the user to set BOOT pins after a reset to select the required boot mode. BOOT pins are also resampled when exiting the *standby* low-power mode. Consequently, they must be kept in the wanted boot mode configuration when entering in *standby* mode. Once this startup time is elapsed, the CPU fetches the *Main Stack Pointer* (MSP) from the address `0x0000 0000`, and so starts code execution from the boot memory starting from the `0x0000 0004` address. The selected memory (flash, SRAM or ROM) is always accessible with its original address space.

If we configure the MCU to boot from the SRAM memory, which is a volatile memory, we have to upload the program code inside this memory and ensure that a valid *vector table* (made of at least a pointer to the base stack and a pointer to the *Reset* exception) is properly set at the `0x0000 0000` address. This requires that we use a debugger tool, which pre-loads all the necessary code inside the SRAM before starting the execution. Moreover, a custom linker script is also needed. We will see a complete example later.

22.1.1 Software *Physical Remap*

Once the MCU boots up, that is the *Reset* exception is being executed, it is still possible to remap the memory accessible through the code area (that is through *I-Bus* and *D-Bus* lines) by programming some bits of the SYSCFG *memory mapped register*. In Cortex-M0/0+ cores this involves setting two

[2]Depending on the package used, in some STM32 MCUs the BOOT1 pin is absent, and it is replaced by a special bit, called nBOOT1, inside the *option bytes* region. Consult the reference manual for your MCU for more about this. In some other STM32 families, like the STM32F7, the functionality of the BOOT1 pin is completely replaced by two dedicated option bytes. Finally, in those MCUs providing two boot pins, BOOT0 is most of the times a dedicated pin used exclusively to select boot origin, while BOOT1 is shared with a GPIO pin. Once BOOT1 has been sampled, the corresponding GPIO pin is freed and can be used for other purposes. However, there exist exceptions in those MCUs with less than 36 pins where even BOOT0 pin is treated as **input GPIO** once sampled during the first clock cycles (for example, the STM32L011K4T is one of these).

bits of the SYSCFG_CFGR1 (`SYSCFG->CFGR1` in the CMSIS library). In Cortex-M3/4/7/33 cores this involves setting the dedicated SYSCFG_MEMRMP register (`SYSCFG->MEMRMP` in the CMSIS library)

Depending on the specific STM32 MCU, the following memories can be remapped by performing a *software physical remap*:

- Internal flash memory
- Internal SRAM
- FMC NVM bank1
- FMC SDRAM bank

The last two memories are available only in those MCUs providing the *Flexible Memory Controller* (FMC), a peripheral that allows to interface external NVM and SDRAM memories. According to **Table 22.1**, direct boot from external NOR as well as SDRAM memories is not allowed. These memories can only be mapped at the `0x0000 0000` address using *software physical remap* after that the MCU is already started with a minimal firmware loaded from the internal flash memory.

Once an external memory has been *physical remapped* at the address `0x0000 0000`, the CPU can access it via the *I-Bus* and *D-Bus* lines, instead of the crowded *S-Bus*, boosting the overall performances. This is especially important for Cortex-M7 based MCU, where those lines are tightly coupled with a dedicated L1-cache.

When the CPU boots, the content of the `SYSCFG->CFGR1` register in Cortex-M0/0+ or the `SYSCFG->MEMRMP` register in Cortex-M3/4/7/33 is latched to the values of the BOOT pins: this means that the *physical remap* is automatically performed from the MCU when sampling BOOT pins. Before changing the content of this register, to perform a remap, it is important to have into the destination memory a working *vector table*[3].

22.1.2 Vector Table Relocation

In Chapter 20 we have seen how to relocate the *vector table* in CCM memory so that we can take advantage of this core-coupled memory. When we perform *physical remapping*, either setting the BOOT pins or configuring the `SYSCFG->MEMRMP` register accordingly, there is no need to perform *vector table* relocation since the MCU automatically aliases the starting address of the selected memory to `0x0000 0000`. Sometimes, however, we want to move the *vector table* in other memory locations that do not correspond to its origin. For example, we may want to store two independent firmware images inside the flash memory (see **Figure 22.1**) and to select one of these according to a given initial condition. This is the case of *bootloaders*, special "system" programs that carry out important configuration tasks such as upgrading the main firmware, as we will see later in this chapter.

[3] It is important to clarify that the CPU will not restart a reset sequence, invoking the handler of the *Reset* exception once the memory has been remapped using the `SYSCFG->MEMRMP` register. It will be your responsibility to invoke that exception handler, and to ensure that the CPU is placed to the initial conditions that the target firmware expects to find (e.g., all peripherals disabled, and so on).

The *Vector Table Offset Register* (VTOR) is a register in the *System Control Block* (SCB) (`SCB->VTOR` in the CMSIS library) that allows to setup the base address of the *vector table*. Once the content of this register is set, the CPU will treat the addresses starting from the new base location as pointers to interrupt service routines.

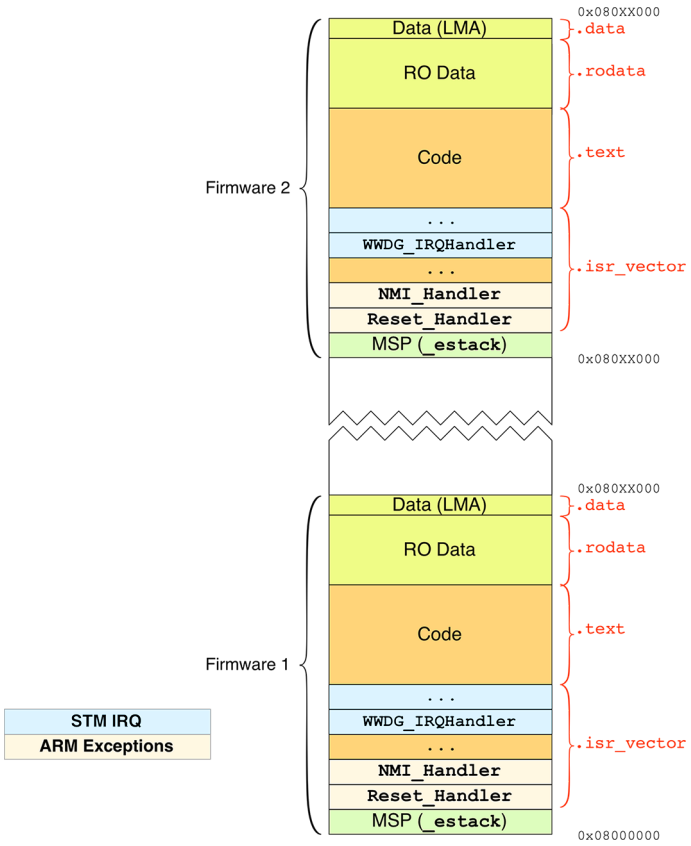


Figure 22.1: Two independent firmware images may be stored inside the flash memory

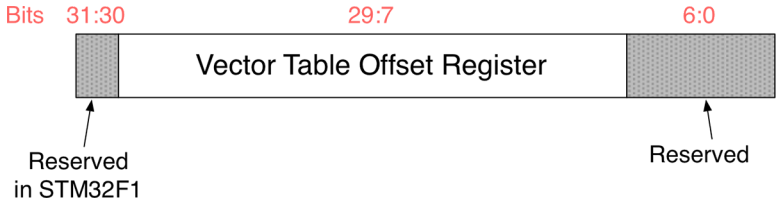


Figure 22.2: The structure of the VTOR register

When modifying the content of the VTOR register, it is important to consider that:

- The VTOR register is not available in Cortex-M0 based MCU and hence it is not possible to relocate the *vector table* without using the *physical remap* (a way to bypass this limitation exists, as we will see later).
- In STM32F1 MCUs, which are based on the Cortex-M3 r1p0 core revision, the bits [31:30] of the VTOR register are reserved (see **Figure 22.2**) and hence it is possible to relocate the *vector table* only in the code memory (`0x0000 0000`) and in SRAM (`0x2000 0000`).

- ARM specification suggests to use a `dmb` (*Data Memory Barrier*) instruction before updating the content of the VTOR register and a `dsb` (*Data Synchronization Barrier*) instruction after the update. Refer to the example 6 in Chapter 20 for a complete example.
- Before changing the content of the VTOR register, ensure that a minimal *vector table* for your application is already in the new location.
- If the application is using peripheral interrupts, suspend all interrupts before starting the relocation procedure.

22.1.3 Running the Firmware From SRAM Using the STM32CubeIDE

Sometimes, it can be useful to load the binary firmware inside the SRAM and to boot from it. This gives some advantages compared to the regular debugging from flash memory:

- it reduces the load time a lot, especially on older and less performant STM32 MCUs, since the RAM memory runs faster than flash one and no erase cycle is needed before writing the firmware;
- it can help increase the flash memory lifecycle, avoiding useless erase cycles just to check a small change to the firmware;
- it can be a lifesaver when debugging code involving advanced operations on the flash memory such as during the development of a custom bootloader.

However, debugging from SRAM sets some limits. The most relevant one is the reduced amount of free SRAM memory. Also the impossibility to reset the MCU while debugging is another serious limit.

The STM32CubeIDE and the ST-LINK Debugger are perfectly able to load the code in RAM and to start debugging from it. Those of you working with some STM32 families should already have noticed that CubeMX automatically adds two distinct linker scripts to the project: one with a filename ending with **_FLASH.ld** and one ending with **_RAM.ld** (for example, owners of Nucleo-F401RE board will find the files **STM32F401RETX_FLASH.ld** and **STM32F401RETX_RAM.ld**). The two files have just one difference: in the file ending with **`_RAM.ld`** all binary sections are generated so that they are mapped in the SRAM memory. It will be a debugger job to handle this configuration e to automatically start the execution from RAM.

To use the linker script for RAM debugging, we need to instruct the GNU LD accordingly. Go inside **Project Properties->C/C++ Build->Settings->MCU GCC Linker->General**. You will find the entry **Linker Script (-T)** with this setting: `${workspace_loc:/${ProjName}/STM32FXXXXXX_FLASH.ld}`, as shown in **Figure 22.3**. Change the linker script according t the filename of the linker script for RAM debugging.

For those projects where CubeMX does not generate the dedicated linker script, it will be easy (after you mastered Chapter 20 concepts) to modify the provided linker script so that all binary sections are mapped in SRAM memory.



Before filing a support request to this author because this procedure may not to work in your case, take in account that this procedure may not work for those of you having Nucleo boards based on STM32 MCUs with few SRAM memory. This because it could happen that the code area falls through the stack area. This procedure essentially works for really small and limited programs.

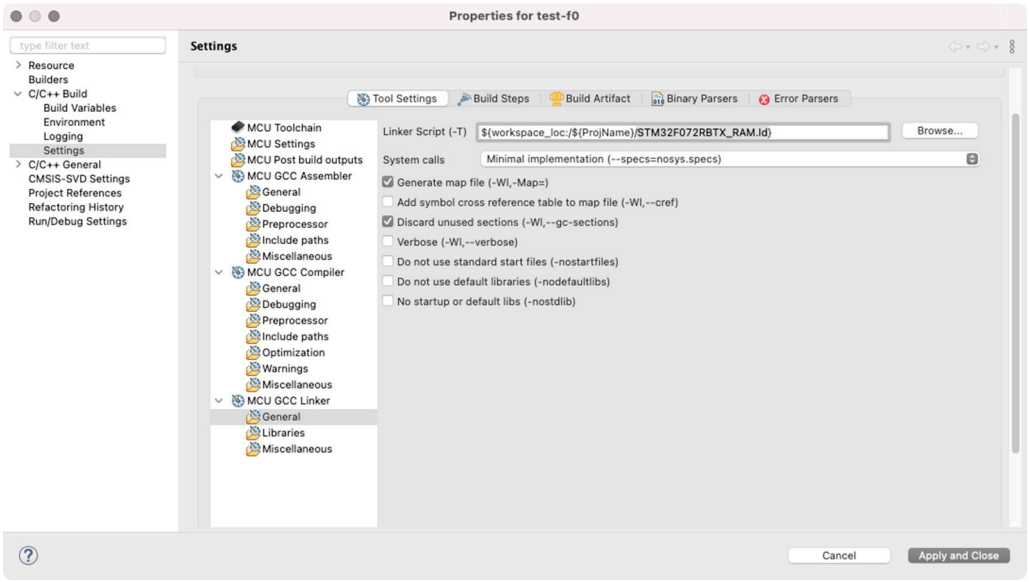


Figure 22.3: The project settings to setup the LD configuration script

22.2 Integrated STM32 Bootloader

In modern digital electronics it is almost impossible to distribute electronic devices without releasing successive upgrades of the firmware. And this is especially true for complex boards with a lot of integrated circuits and peripherals. Soon or later, all embedded developers will need a way to distribute a firmware upgrade and, most important, they will need a way to let customers uploading it on the MCU without a dedicated (and sometimes expensive) debugger. Moreover, often the SWD debug port is not added to the final PCB for a design choice.[4].

A *bootloader* is a piece of software, usually executed first when the MCU boots, which has the ability to upgrade the firmware inside the internal flash. This operation is also known as *In-Application Programming* (IAP), which is distinct from the MCU programming using an external and dedicated debugger: this other way to program MCUs is also known as *In-System Programming* (ISP).

Bootloaders are usually designed so that they accept commands through a communication peripheral (USART, USB, Ethernet and so on), which is used to exchange the firmware binary with the

[4]For those of you wondering how to upload the firmware on a board without the debug port, and without using the integrated bootloader, it could be useful to know that ST can ship to you MCUs with your firmware already pre-programmed during MCU production. This possibility is offered for quite large orders (as far as I know lots with more than 10.000pcs). Ask to your sales representative for more about this. Finally, there exists specialized companies that can do this job for you, by performing even additional operations such as re-reeling service.

MCU. A dedicated program, designed to run on an external PC, is usually also needed.

All STM32 MCUs come from the factory with a pre-programmed bootloader in a ROM memory, called *System memory*, which is mapped inside the address range `0x1FFF 0000` - `0x1FFF 77FF` in the majority of STM32 microcontrollers[5]. Depending on the MCU family and package used, this bootloader can interact with the outside world using:

- USART
- USB (DFU protocol)
- CAN bus
- I^2C
- SPI

For each one of these communication peripherals, ST has defined a standardized protocol that allows to:

- Retrieve the bootloader release and supported commands.[6]
- Get the chip ID.
- Read a number of bytes of memory starting from an address specified by the host application.
- Write a number of bytes to the RAM or flash memory starting from an address specified by the host application.
- Erase one or more flash memory pages/sectors.
- Jump to user application code located in the internal flash memory or in SRAM.
- Enable/disable the read/write protection for some pages/sectors.

For each communication protocol, ST provides a dedicated application note called "*PPP* protocol used in STM32 bootloader", where *PPP* is the peripheral type. For example, the AN3155[7] is about the USART protocol.

The integrated bootloader is designed so that it continuously samples all the supported peripherals for a given "starting condition": once this occurs, that peripheral is chosen to interact with the external world. For example, if the byte 0x7F is received on the UARTx interface, then the bootloader starts working in UART mode.
For the complete description of the bootloader selection sequence for the given STM32 MCU, refer to the AN2606 from ST[8].

Apart from the communication peripheral used, the bootloader uses several other hardware resources:

- The HSI oscillator, which is selected as the clock source.

[5] Figure 4 in Chapter 1 gives you an idea of the *System memory* position inside the Cortex-M 4GB address space.
[6] This is not a secondary feature, since there exist different releases of STM32 bootloaders, and some of them have non negligible differences.
[7] https://bit.ly/2cojjQI
[8] https://bit.ly/29sEb8t

- The *SysTick* timer (not for all communication peripherals).
- About 2K of SRAM memory.
- The IWDG peripheral (prescaler is configured to its maximum value and IWDG is periodically refreshed to prevent reset in case the hardware IWDG option was previously enabled by the user).

Moreover, there are some limitations regarding memory management through the bootloader:

- Some STM32 microcontrollers don't support mass-erase operation. To perform a mass-erase using bootloader, two options are available: to erase all sectors one-by-one using the Erase command or to set flash read protection level to Level 1 and then to set it back to Level 0.
- Bootloader firmware in STM32L1/L0 series allows to manipulate EEPROM in addition to standard memories (internal flash and SRAM, *option bytes* and *system memory*). The starting address and the size of this memory type depend on the specific part number. EEPROM can be read and written but cannot be erased using the Erase Command. When writing in an EEPROM location, the bootloader firmware manages the erase operation of this location before any write. A write to the EEPROM must be word-aligned (address to be written should be a multiple of 4) and the number of data must also be a multiple of 4. To erase an EEPROM location, you can write zeros at this location.
- Bootloader firmware in STM32F2/F4/F7/L4 series supports OTP memory in addition to standard memories (internal Flash, internal SRAM, *option bytes* and *system memory*). The starting address and the size of this area depends on the specific part number. Please refer to the product reference manual for more information. OTP memory can be read and written but cannot be erased using Erase command. When writing in an OTP memory location, make sure that the relative protection bit is not reset.
- For STM32F2/F4/F7 series the internal flash write operation format depends on voltage range. By default, write operations are allowed by one byte format (half-word, word and double-word operations are not allowed). To increase the speed of write operations, the user should apply the adequate voltage range that allows write operations by half-word, word or double-word and update this configuration on the fly by using the bootloader software. Some virtual locations are reserved for this operation. For more information about this, refer to the AN2606 from ST[9].

To interface the integrated bootloader using the USART or the USB protocol, it is possible to use the STM32CubeProgrammer by selecting the corresponding protocol as shown in **Figure 22.4**.

Also consider that the STM32CubeProgrammer has a command-line interface allowing you to use it in production environments. For more information about this feature, consult the corresponding user manual (UM2237[10]). Moreover, if you are considering to use the USB bootloader, take note that some other open source applications, like the `dfu-util`[11] tool, can be also used on Windows as well

[9] https://bit.ly/29sEb8t
[10] https://bit.ly/3E9bMzr
[11] http://dfu-util.sourceforge.net/

as on Linux and MacOS. For more information about USB DFU mode in STM32 bootloaders, consult the UM0412[12] user manual from ST.

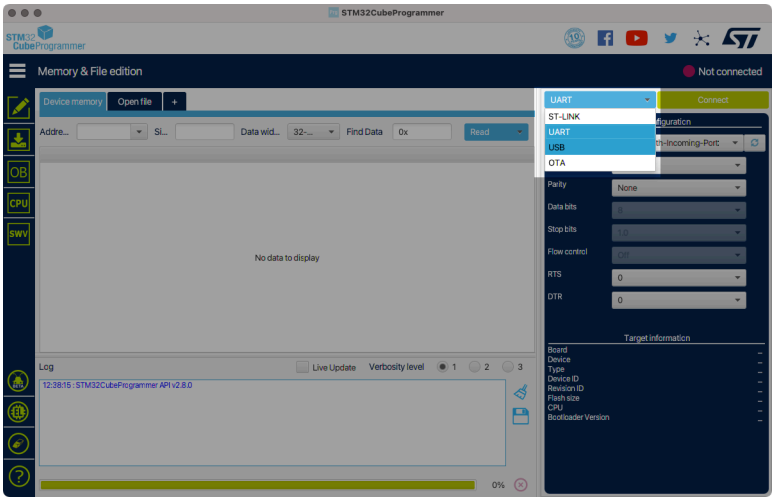


Figure 22.4: How to select the bootloader protocol in the STM32CubeProgrammer

22.2.1 Starting the STM32 Bootloader from the On-Board Firmware

The execution of the integrated bootloader is connected to the status of BOOT pins, which are sampled during the first clock cycles. However, for several design choices, you may not be able to configure BOOT pins as required. For this reason, you can "jump" to the *System memory* from the firmware (for example, the user may be forced to press a hidden switch).

Forcing the bootloader execution from the user code is not that hard: it is just about defining a function pointer.

```
1  __set_MSP(SRAM_END);
2  uint32_t JumpAddress = *(volatile uint32_t*)(0x1FFF0000 + 4);
3  void (*boot_loader)(void) = JumpAddress;
4  SYSCFG->MEMRMP = 0x1; //Remap 0x0000 0000 to System Memory
5  boot_loader();
6  //Never coming here
```

The instruction at line 1 sets the main stack pointer to the end of SRAM (this should not be usually required, but just in case....). Then we create a pointer to a function whose address is set to the beginning of the *System Memory* [13] and we simply jump to the integrated bootloader by calling the function `boot_loader()` after a physical remap to *System Memory*[14].

[12] https://bit.ly/29sJen2

[13] The above address, `0x1FFF 0000`, coincides with the starting address of *System Memory* in an STM32F401RE MCU; consult the reference manual for your MCU for the exact value).

[14] Probably the physical remap is not strictly needed, since the bootloader seems to work well the same.

However, we must place special care when jumping to the *System Memory*. The bootloader, in fact, is designed to be called just after a reset and it assumes that the CPU and its peripherals are set to the default initial state. A better solution could be achieved by storing a special code inside the SRAM memory and then forcing a system reset in software: we may check from the *Reset* exception handler against this special code and jump to the *System Memory* before any other initialization procedure. This guard value must be stored in a memory location outside of `.data` and `.bss` regions, otherwise it may be initialized during firmware booting (alternatively, we can place this code inside *Reset* exception handler before those regions are initialized).

22.2.2 The Booting Sequence in the STM32CubeIDE Tool-chain

Now that the booting process should be clear, we can analyze a really fundamental topic: what are the exact steps performed during boot by an application developed with the STM32CubeIDE tool-chain? The answer is not trivial, and there are several important things an experienced programmer working with this tool-chain must know.

In Chapter 20 we have deeply analyzed the way a *Reset* exception works. However, examples made in that chapter are insulated from the real tool-chain: we have developed a minimal STM32 application that does not use either the CubeHAL nor the startup files generated by CubeMX. So, to understand the actual boot sequence, we have to start from the beginning: the *Reset* exception.

In Chapter 7 we have seen that the assembly file **system/src/cmsis/startup_stm32XXxx.s** contains the definition of the *vector table*. This file is provided by ST and it is specific for the given STM32 MCU. Opening the one fitting your MCU, you can find the definition of the `Reset_Handler`, about at lines 50-60.

```
56    .section  .text.Reset_Handler
57    .weak  Reset_Handler
58    .type  Reset_Handler, %function
59  Reset_Handler:
60    ldr   sp, =_estack                    /* set stack pointer */
61
62  /* Copy the data segment initializers from flash to SRAM */
63    movs  r1, #0
64    b  LoopCopyDataInit
65
66  CopyDataInit:
67    ldr  r3, =_sidata
68    ldr  r3, [r3, r1]
69    str  r3, [r0, r1]
70    adds  r1, r1, #4
71
72  LoopCopyDataInit:
73    ldr  r0, =_sdata
74    ldr  r3, =_edata
75    adds  r2, r0, r1
```

```
76    cmp  r2, r3
77    bcc  CopyDataInit
78    ldr  r2, =_sbss
79    b  LoopFillZerobss
80  /* Zero fill the bss segment. */
81  FillZerobss:
82    movs  r3, #0
83    str  r3, [r2], #4
84
85  LoopFillZerobss:
86    ldr  r3, = _ebss
87    cmp  r2, r3
88    bcc  FillZerobss
89
90  /* Call the clock system intitialization function.*/
91    bl  SystemInit
92  /* Call static constructors */
93      bl __libc_init_array
94  /* Call the application's entry point.*/
95    bl  main
96    bx  lr
97  .size  Reset_Handler, .-Reset_Handler
```

As you can see, the `Reset_Handler` is written in assembly, but it should be easy to understand now that we have mastered a lot of fundamental concepts. The routine body starts at line 60. Here the MSP is set to the content of the `_estack` linker variable (it coincides with the end of SRAM). Then the control is transferred to the `LoopCopyDataInit` routine, which initializes the `.data` section. The control is then transferred to the `LoopFillZerobss` routine, which initializes the `.bss` sections and calls the `SystemInit()` routine (we will analyze it in a while); right after this routine, C++ static constructors are initialized by calling the `__libc_init_array()`. Finally, the control is transferred to the `main()` routine.

The CMSIS `SystemInit()` routine is platform-dependent and it is provided by ST inside the file named **Core/Src/system_stm32XXxx.c**. This routine usually contains the activation of additional coprocessors - if present - like the FPU. In Cortex-M3/4/7/33 based MCUs, this routine also contains *vector table* relocation according to the definition of a specific set of C macros. Please refer to the `SystemInit()` implementation for your MCU.

22.3 Developing a Custom Bootloader



Read Carefully

The bootloader described in this paragraph works correctly if and only if the ST-LINK interface has a firmware version equal or higher than 2.27.15. Older releases have a bug on the VCP preventing the USART interface to work as expected. Ensure that your Nucleo is updated.

Integrated bootloaders work well in a lot of cases. Many real projects can benefit from their usage. Moreover, the free-of-charge tools provided by ST can reduce the effort needed to develop custom applications that upload the firmware on the MCU. However, for some applications you may need additional functionalities not implemented in standard bootloaders. For example, we may want to encrypt the distributed firmware so that only the on-board bootloader is able to decode it using a pre-shared key hardcoded inside the bootloader code.

We are now going to develop a custom bootloader that will allow us to upload a new firmware on the target MCU. This will essentially provide only a fraction of the features implemented by integrated bootloaders, but it will give us the opportunity to review the fundamental steps needed to develop a custom bootloader. It will provide the following functionalities:

- Upload a new firmware using the UART interface (in our case, the UART2 interface provided by all Nucleo boards).
- Retrieve the MCU type.
- Erase a given amount of flash sectors/pages.
- Write a series of bytes starting from a given address.
- Encrypt/Decrypt the exchanged firmware using AES-128 algorithm[15].

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Table 22.2: The flash memory organization in an STM32F401RE MCU

The code that we will analyze here relies on the flash memory layout of STM32F401RE microcontrollers, which is shown in **Table 22.2** and extracted from the corresponding reference manual. As you can see, the 512KB of flash memory are partitioned in seven sectors. The first one, the *sector 0* highlighted in blue in **Table 22.2**, will be used to store the integrated bootloader. If you are working on a different STM32 MCU, refer to the book examples to see how the bootloader has been arranged for your MCU.

[15]In real-life applications, especially if you use a contract manufacturer to produce your boards, you may need a more secure bootloader and a more articulated production environment. ST provides a complete extension package, named X-CUBE-SBSFU, which consists a of complete solution to handle secure bootloader, secure firmware upgrade and distribution, keys management and their distributions.

Once the MCU resets, the bootloader starts its execution[16]. This means that the bootloader is compiled so that it is mapped starting from the `0x0800 0000` address, like any regular STM32 application seen in this book.

A minimal *vector table* is defined, which allows to the MCU to properly start the execution. The bootloader so samples the PC13 pin, which in almost all Nucleo boards corresponds to the blue button on the board. If the button is pressed, then it starts accepting commands on the UART2 interface. Otherwise, it immediately relocates the VTOR register and passes the control to the *Reset* exception handler of the main firmware.

A companion script, written in Python 2, is also provided. It is named `flasher.py` and you can find it inside the book examples. We will describe how to use it in a following paragraph.

Before we go into the details of the commands used to exchange messages with the bootloader, we will start analyzing the procedures executed during the boot process and the way the control is transferred to the main firmware.

Filename: `src/main-bootloader.c`

```
7  /* Global macros */
8  #define ACK                0x79
9  #define NACK               0x1F
10 #define CMD_ERASE          0x43
11 #define CMD_GETID          0x02
12 #define CMD_WRITE          0x2b
13
14 #define APP_START_ADDRESS  0x08004000 /* In STM32F401RE this corresponds with the start
15                                          address of Sector 1 */
16
17 #define SRAM_SIZE          96*1024     // STM32F401RE has 96 KB of RAM
18 #define SRAM_END           (SRAM_BASE + SRAM_SIZE)
19
20 #define ENABLE_BOOTLOADER_PROTECTION 0
21 /* Private variables ---------------------------------------------------------*/
22
23 /* The AES_KEY cannot be defined const, because the aes_enc_dec() function
24  temporarily modifies its content */
25 uint8_t AES_KEY[] = { 0x4D, 0x61, 0x73, 0x74, 0x65, 0x72, 0x69, 0x6E, 0x67,
26                       0x20, 0x20, 0x53, 0x54, 0x4D, 0x33, 0x32 };
27
28 extern CRC_HandleTypeDef hcrc;
29 extern UART_HandleTypeDef huart2;
```

The macro `APP_START_ADDRESS` at line 14 defines the starting address of the main firmware. According to the memory layout of an STM32F401RE MCU, the second sector starts at that address and the main application firmware will be stored there. This means that the MSP will be placed

[16]Clearly, the MCU pins must be configured so that the flash memory is the default boot source.

at `0x0800 4000` and the address in flash memory of the *Reset* exception handler at `0x0800 4004`. The `AES_KEY` array, defined at line 25, contains sixteen bytes forming the AES-128 key used to encrypt/decrypt the uploaded firmware. We will analyze its usage later.

Filename: `src/main-bootloader.c`

```
44 /* Minimal vector table */
45 uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
46     (uint32_t *) SRAM_END, // initial stack pointer
47     (uint32_t *) _start,   // _start is the Reset_Handler
48     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, (uint32_t *) SysTick_Handler };
```

The *vector table* is defined at line 45. It just contains the MSP pointer, which coincides with the end of SRAM memory, the pointer to the *Reset* exception handler (`_start` in this case, which does nothing more than to initialize `.data` and `.bss` sections and to transfer the control to the `main()` routine), and the pointer to the `SysTick_Handler`. This is required because we will use the standard HAL routines to interface peripherals, and the HAL is built around an unique timebase, usually generated using the *SysTick* timer. The HAL so needs to enable that timer and to catch the overflow event so that the global *tick* count is increased.

Filename: `src/main-bootloader.c`

```
93  int main(void) {
94    uint32_t ulTicks = 0;
95    uint8_t ucUartBuffer[20];
96
97    /* HAL_Init() sets SysTick timer so that it overflows every 1ms */
98    HAL_Init();
99    MX_GPIO_Init();
100
101 #if ENABLE_BOOTLOADER_PROTECTION
102   /* Ensures that the first sector of flash is write-protected preventing that the
103    bootloader is overwritten */
104   CHECK_AND_SET_FLASH_PROTECTION();
105 #endif
106
107   /* If USER_BUTTON is pressed */
108   if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET) {
109     /* CRC and UART2 peripherals enabled */
110     MX_CRC_Init();
111     MX_USART2_UART_Init();
112
113     ulTicks = HAL_GetTick();
114
115     while (1) {
116       /* Every 500ms the LD2 LED blinks, so that we can see the bootloader running. */
117       if (HAL_GetTick() - ulTicks > 500) {
```

```
118           HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
119           ulTicks = HAL_GetTick();
120         }
121
122         /* We check for new commands arriving on the UART2 */
123         HAL_UART_Receive(&huart2, ucUartBuffer, 20, 10);
124         switch (ucUartBuffer[0]) {
125         case CMD_GETID:
126           cmdGetID(ucUartBuffer);
127           break;
128         case CMD_ERASE:
129           cmdErase(ucUartBuffer);
130           break;
131         case CMD_WRITE:
132           cmdWrite(ucUartBuffer);
133           break;
134         };
135       }
136     } else {
137       /* USER_BUTTON is not pressed. We first check if the first 4 bytes starting from
138        APP_START_ADDRESS contain the MSP(end of SRAM). If not, the LD2 LED blinks quickly. */
139       if (*((uint32_t*) APP_START_ADDRESS) != SRAM_END) {
140         while (1) {
141           HAL_Delay(30);
142           HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
143         }
144       } else {
145         /* A valid program seems to exist in the second sector: we so prepare the MCU
146          to start the main firmware */
147         MX_GPIO_Deinit(); //Puts GPIOs in default state
148         SysTick->CTRL = 0x0; //Disables SysTick timer and its related interrupt
149         HAL_DeInit();
150
151         RCC->CIR = 0x00000000; //Disable all interrupts related to clock
152         __set_MSP(*((volatile uint32_t*) APP_START_ADDRESS)); //Set the MSP
153
154         __DMB(); //ARM says to use a DMB instruction before relocating VTOR */
155         SCB->VTOR = APP_START_ADDRESS; //We relocate vector table to the sector 1
156         __DSB(); //ARM says to use a DSB instruction just after relocating VTOR */
157
158         /* We are now ready to jump to the main firmware */
159         uint32_t JumpAddress = *((volatile uint32_t*) (APP_START_ADDRESS + 4));
160         void (*reset_handler)(void) = (void*)JumpAddress;
161         reset_handler(); //We start the execution from he Reset_Handler of the main firmware
162
163         for (;;)
164           ; //Never coming here
```

```
165       }
166     }
167   }
```

We are now going to explain the tasks performed by the `main()` routine. Once it is called by the *Reset* exception handler (`_start()` routine), it firstly initializes the CubeHAL, reducing to the minimum the amount of operations performed in this phase: this helps reducing the boot time. The `HAL_Init()` routine also configures the *SysTick* timer so that it expires every 1ms. The PC13 pin is so sampled, and if the user keeps pressed the USER BUTTON, then the routine enters in an infinite loop accepting three commands on the UART2. We will analyze them later. Note that we leave the default clock source as is (that is, the HSI oscillator).

If, instead, the USER BUTTON is left unpressed, then the `main()` routine verifies if the first memory location of the second sector contains the MSP (we simply check that it does contain the SRAM_-END value). If not, the firmware starts blinking LD2 LED very fast to signal that there is no main application to run.

If that memory location contains the MSP pointer (line 144), we can start the boot sequence. GPIOs are so placed to their default state, the HAL is deinitialized and the *SysTick* timer is stopped, and its exception disabled. All clock-related interrupts are disabled at line 151 and the MSP is set to the address specified at the first 4 bytes of the sector 1 (because the *vector table* is placed there, as we will see later). The VTOR base location is so set to the `APP_START_ADDRESS` (that is, `0x0800 4000` for the STM32F401RE bootloader). The address of *Reset* exception for the main firmware is derived from the `0x0800 4004` memory location and a pointer to that function is defined. Finally, at line 161 the *Reset* exception is invoked and the bootloader ends.

Before we analyze the three commands implemented by the bootloader, it is best to give a quick look at the other application shipped with the examples of this chapter. It is named `main-app1.c` and it is nothing more than a simple application that blinks the LD2 LED and prints a message on the UART2. The only relevant thing to note is the related linker script, named **STM32F401RETX_APP.ld**, which defines the FLASH memory region in the following way:

Filename: `src/ldscript-app.ld`

```
14  MEMORY {
15    FLASH (rx) : ORIGIN = 0x08004000, LENGTH = 512K - 16K
16    RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 96K
```

As you can see, the linker will relocate the application code starting from the `0x0800 4000` address. Moreover, the length of this memory region is set to 496KB: since the first sector is 16KB wide, 512-16 is equal to 496. This definition of the flash memory region also allows us to upload and debug the firmware using the ST-LINK Debugger (or the STM32CubeProgrammer) without overwriting the bootloader.



According to what seen in the previous paragraph, ensure that `SystemInit()` routine does not change the VTOR register value.

Now it is the right time to analyze the three commands supported by this bootloader: `CMD_GETID`, `CMD_ERASE` and `CMD_WRITE`.

Get ID Command

The `CMD_GETID` command is used to retrieve the MCU ID[17] and it has the structure shown in **Figure 22.5**. The bootloader so expects to retrieve the byte `0x02` followed by the CRC-32 of this byte. The bootloader answers to the request by sending an ACK (which is defined at line `8` of the `main-bootloader.c` file and it is equal to `0x79`) followed by two bytes containing the MCU ID.

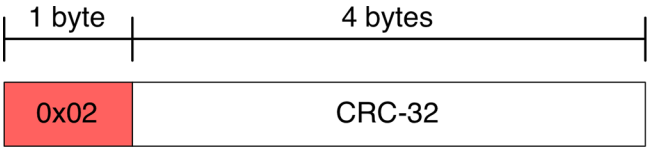


Figure 22.5: The structure of the `CMD_GETID`

Filename: `src/main-bootloader.c`

```
223 void cmdGetID(uint8_t *pucData) {
224   uint16_t usDevID;
225   uint32_t ulCrc = 0;
226   uint32_t ulCmd = pucData[0];
227 
228   memcpy(&ulCrc, pucData + 1, sizeof(uint32_t));
229 
230   /* Checks if provided CRC is correct */
231   if (ulCrc == HAL_CRC_Calculate(&hcrc, &ulCmd, 1)) {
232     usDevID = (uint16_t) (DBGMCU->IDCODE & 0xFFF); //Retrieves MCU ID from DEBUG interface
233 
234     /* Sends an ACK */
235     pucData[0] = ACK;
236     HAL_UART_Transmit(&huart2, pucData, 1, HAL_MAX_DELAY);
237 
238     /* Sends the MCU ID */
239     HAL_UART_Transmit(&huart2, (uint8_t *) &usDevID, 2, HAL_MAX_DELAY);
240   } else {
241     /* The CRC is wrong: sends a NACK */
242     pucData[0] = NACK;
243     HAL_UART_Transmit(&huart2, pucData, 1, HAL_MAX_DELAY);
244   }
245 }
```

The above code shows how the command is implemented. As you can see, the CRC is extracted from the message coming on the UART and compared with the one computed by the CRC peripheral. If the two values match, then the MCU ID is derived from DEBUG interface and it is transmitted over

[17] The MCU ID is different from the CPU ID. The former identifies the STM32 family and chip type (for example, 0x433 identifies the STM32F401RE MCU). The latter is a unique ID that identifies that specific MCU, and it is impossible (or at least really hard) that exist two STM32 microcontrollers with the same CPU ID.

the UART together with the ACK. If the CRC does not match, a NACK (which is equal to 0x1F) is sent.

Erase Command

The `CMD_ERASE` command is used to erase a given sector of the flash memory and it has the structure shown in **Figure 22.6**. The command is composed by the id `0x43` that identifies the command type, followed by the number of sectors to delete (or the value `0xFF` to delete all sector except the first one where the bootloader resides) and the CRC-32. The bootloader answers by sending an ACK when the erasing procedure completes.

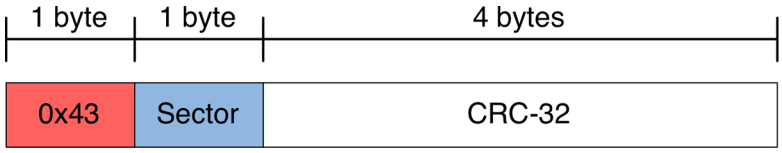


Figure 22.6: The structure of the `CMD_ERASE`

Filename: `src/main-bootloader.c`

```
180 void cmdErase(uint8_t *pucData) {
181   FLASH_EraseInitTypeDef eraseInfo;
182   uint32_t ulBadBlocks = 0, ulCrc = 0;
183   uint32_t pulCmd[] = { pucData[0], pucData[1] };
184
185   memcpy(&ulCrc, pucData + 2, sizeof(uint32_t));
186
187   /* Checks if provided CRC is correct */
188   if (ulCrc == HAL_CRC_Calculate(&hcrc, pulCmd, 2) &&
189       (pucData[1] > 0 && (pucData[1] < FLASH_SECTOR_TOTAL - 1 || pucData[1] == 0xFF))) {
190     /* If data[1] contains 0xFF, it deletes all sectors; otherwise
191      * the number of sectors specified. */
192     eraseInfo.Banks = FLASH_BANK_1;
193     eraseInfo.Sector = FLASH_SECTOR_1;
194     eraseInfo.NbSectors = pucData[1] == 0xFF ? FLASH_SECTOR_TOTAL - 1 : pucData[1];
195     eraseInfo.TypeErase = FLASH_TYPEERASE_SECTORS;
196     eraseInfo.VoltageRange = FLASH_VOLTAGE_RANGE_3;
197
198     HAL_FLASH_Unlock(); //Unlocks the flash memory
199     HAL_FLASHEx_Erase(&eraseInfo, &ulBadBlocks); //Deletes given sectors */
200     HAL_FLASH_Lock(); //Locks again the flash memory
201
202     /* Sends an ACK */
203     pucData[0] = ACK;
204     HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
205   } else {
206     /* The CRC is wrong: sends a NACK */
207     pucData[0] = NACK;
208     HAL_UART_Transmit(&huart2, pucData, 1, HAL_MAX_DELAY);
```

```
209   }
210 }
```

The above code shows how the command is implemented. As you can see, the CRC is extracted from the message coming on the UART and compared with the one computed by the CRC peripheral. Note that, since the CRC peripheral has a 32-bit wide data register and the CRC-32 is computed over the whole register, we covert the first two bytes to two 32-bit values.

If the CRC matches, then an instance of the `FLASH_EraseInitTypeDef struct` is filled so that the flash sectors are erased starting from the second one (line 193) up to the number of sectors specified (line 194). The flash memory is so unlocked (line 198) and the erase procedure is performed by calling the `HAL_FLASHEx_Erase()` routine.

Write Command

The `CMD_WRITE` command is used to store sixteen bytes (that is, four words) starting from a given memory location, and it has the structure reported in **Figure 22.7**. The command is made of two distinct parts. The first one is composed by the command id `0x2b`, followed by the starting address where to place data bytes and the command's CRC-32. If the CRC matches and the specified address is equal or higher than `APP_START_ADDRESS`, the bootloader answers with an ACK. The bootloader so expects to receive another sequence made of sixteen bytes and the CRC-32 checksum of these bytes.

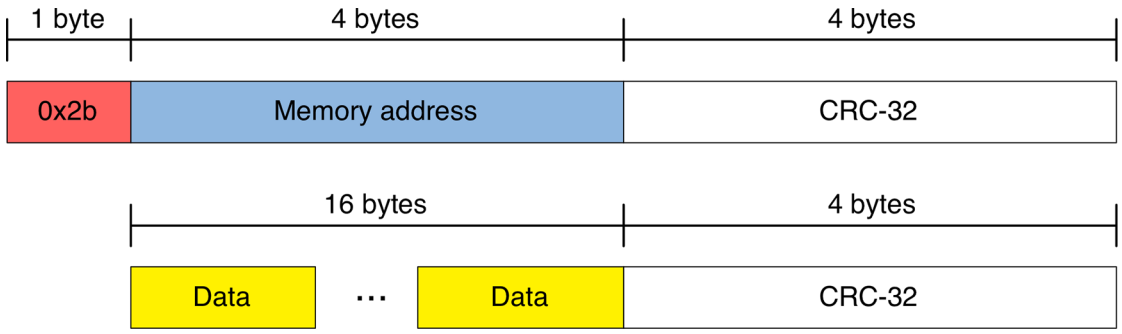


Figure 22.7: The structure of the `CMD_WRITE`

Filename: `src/main-bootloader.c`

```
267 void cmdWrite(uint8_t *pucData) {
268   uint32_t ulSaddr = 0, ulCrc = 0;
269
270   memcpy(&ulSaddr, pucData + 1, sizeof(uint32_t));
271   memcpy(&ulCrc, pucData + 5, sizeof(uint32_t));
272
273   uint32_t pulData[5];
274   for(int i = 0; i < 5; i++)
275     pulData[i] = pucData[i];
276
277   /* Checks if provided CRC is correct */
278   if (ulCrc == HAL_CRC_Calculate(&hcrc, pulData, 5) && ulSaddr >= APP_START_ADDRESS) {
```

```
279     /* Sends an ACK */
280     pucData[0] = ACK;
281     HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
282
283     /* Now retrieves given amount of bytes plus the CRC32 */
284     if (HAL_UART_Receive(&huart2, pucData, 16 + 4, 200) == HAL_TIMEOUT)
285       return;
286
287     memcpy(&ulCrc, pucData + 16, sizeof(uint32_t));
288
289     /* Checks if provided CRC is correct */
290     if (ulCrc == HAL_CRC_Calculate(&hcrc, (uint32_t*) pucData, 4)) {
291       HAL_FLASH_Unlock(); //Unlocks the flash memory
292
293       /* Decode the sent bytes using AES-128 ECB */
294       aes_enc_dec((uint8_t*) pucData, AES_KEY, 1);
295       for (uint8_t i = 0; i < 16; i++) {
296         /* Store each byte in flash memory starting from the specified address */
297         HAL_FLASH_Program(FLASH_TYPEPROGRAM_BYTE, ulSaddr, pucData[i]);
298         ulSaddr += 1;
299       }
300       HAL_FLASH_Lock(); //Locks again the flash memory
301
302       /* Sends an ACK */
303       pucData[0] = ACK;
304       HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
305     } else {
306       goto sendnack;
307     }
308   } else {
309     goto sendnack;
310   }
311
312 sendnack:
313   pucData[0] = NACK;
314   HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
315 }
```

The above code shows how the command is implemented. As you can see, the CRC of the first part of the message is checked against the transmitted value (lines [273:278]). If it corresponds, an ACK is sent, and the next bytes are processed. If the CRC-32 of these other bytes matches (line 290), then the sent data bytes are decrypted using the AES-128 algorithm[18] and the pre-shared key. Data bytes

[18]The `aes_enc_dec()` function is taken from a library made by Eric Peeters, a TI employee. It can be downloaded from the TI website(http://www.ti.com/tool/AES-128) and its license allows to use it freely. ST provides a complete cryptographic library for the STM32 platform, which is also compatible with the Cube framework (https://bit.ly/29zWN81). This library can also take advantage of those STM32 MCUs providing a dedicated hardware crypto unit. However, the license of this library prevents this author from shipping the library with the examples in this book.

are so stored inside the flash memory starting from the given memory location.

There is one more thing to analyze: the function `CHECK_AND_SET_FLASH_PROTECTION()` invoked by the `main()` function if the macro `ENABLE_BOOTLOADER_PROTECTION` is set to 1.

Filename: `src/main-bootloader.c`

```
317 void CHECK_AND_SET_FLASH_PROTECTION(void) {
318   FLASH_OBProgramInitTypeDef obConfig;
319
320   /* Retrieves current OB */
321   HAL_FLASHEx_OBGetConfig(&obConfig);
322
323   /* If the first sector is not protected */
324   if ((obConfig.WRPSector & OB_WRP_SECTOR_0) == OB_WRP_SECTOR_0) {
325     HAL_FLASH_Unlock(); //Unlocks flash
326     HAL_FLASH_OB_Unlock(); //Unlocks OB
327     obConfig.OptionType = OPTIONBYTE_WRP;
328     obConfig.WRPState = OB_WRPSTATE_ENABLE; //Enables changing of WRP settings
329     obConfig.WRPSector = OB_WRP_SECTOR_0; //Enables WP on first sector
330     HAL_FLASHEx_OBProgram(&obConfig); //Programs the OB
331     HAL_FLASH_OB_Launch(); //Ensures that the new configuration is saved in flash
332     HAL_FLASH_OB_Lock(); //Locks OB
333     HAL_FLASH_Lock(); //Locks flash
334   }
335 }
```

This function simply retrieves the current *Option Bytes* configuration and checks if the first sector is write-protected (line 324). If not, the write-protection is enabled so that the bootloader cannot be overwritten.

If you want to experiment with this function, then to disable the write-protection you can use the STM32CubeProgrammer.



Some Considerations on the Custom Bootloader

The custom bootloader presented here is far from to be complete. It lacks some relevant features and, most important, it is not sufficiently robust to cover error conditions. Moreover, the sole bootloader for the STM32F0/L0 platforms is about 13KB when compiled with the GCC `-Os` option, which produces the most size-optimized binary image. This is too much for a bootloader. Unfortunately, the HAL has a non-negligible overhead on the final size of the binary image. A well-designed bootloader is coded reducing to the minimum its footprint. This aspect is outside the scope of this book, which merely shows the fundamental concepts behind the booting process.

22.3.1 *Vector Table* Relocation in STM32F0 Microcontrollers

So far, we have seen that in Cortex-M0 based microcontrollers it is not possible to relocate the *vector table* as it happens in Cortex-M0+/3/4/7 MCUs. This means that we cannot use the code seen before (at lines [154:161]) to pass the control to the main firmware, because Cortex-M0 cores always expect to find the *vector table* at the address `0x0000 0000`, and this one coincides with the *vector table* of the bootloader in our scenario.

We can, however, bypass this limitation in a somewhat craftier manner. The idea that we are going to analyze is based on the fact that the *software physical remap* allows to alias SRAM memory at the `0x0000 0000` address, while the original flash memory is always accessible at the `0x0800 0000` address. We can so relocate the *vector table* of the main firmware before passing the control to its *Reset* exception handler by simply copying the "target" *vector table* inside the SRAM and then performing the *physical remapping*. The addresses contained inside the target *vector table* are still accessible at their original locations, allowing the correct execution of exception handlers and ISRs.

Figure 22.8 tries to represent this procedure. On the left side we have the main application (the bootloader is not shown). Let us suppose for the sake of simplicity that its *vector table* is placed at the address `0x0800 2C00`. This means that, starting from the address `0x0800 2C04` we have the address in memory of Cortex-M0 exception handlers and ISRs. Clearly, these addresses point to other memory locations above the `0x0800 2C00` address (in **Figure 22.8** they are represented as grey arrows).

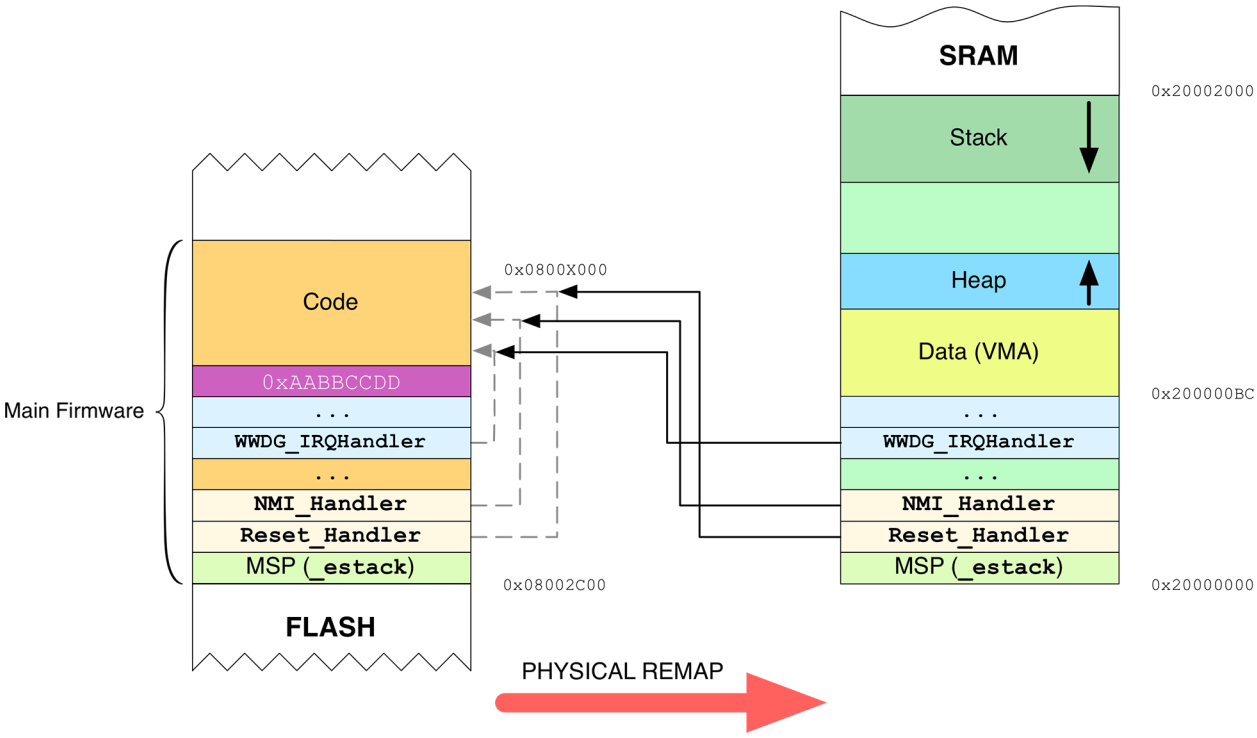


Figure 22.8: How the *vector table* can be relocated in STM32F0 microcontrollers

The bootloader works in the following way. It copies the *vector table* inside the SRAM memory, starting by placing its content from the initial address `0x2000 0000`. This means that from the `0x2000 0004` memory location we have the addresses in flash memory of exception handlers and ISRs. Clearly, these addresses still point to the same original flash memory locations, as indicated by black arrows in **Figure 22.8**. At the end of the copy procedure the memory is remapped, so that the `0x0000 0000` address now coincides with the `0x2000 0000` address. The control is then transferred to the *Reset* exception handler of the main firmware and its execution takes place. In this way we have bypassed the limitation of Cortex-M0 based MCUs, which do not allow to relocate in memory the *vector table*.

The following code shows our bootloader implemented for the STM32F030 microcontroller. Only the part related to *vector table* relocation is shown.

Filename: `src/main-bootloader.c`

```
146     } else {
147       /* A valid program seems to exist in the second sector: we so prepare the MCU
148        to start the main firmware */
149       MX_GPIO_Deinit(); //Puts GPIOs in default state
150       SysTick->CTRL = 0x0; //Disables SysTick timer and its related interrupt
151       HAL_DeInit();
152
153       RCC->CIR = 0x00000000; //Disable all interrupts related to clock
154
155       uint32_t *pulSRAMBase = (uint32_t*)SRAM_BASE;
156       uint32_t *pulFlashBase = (uint32_t*)APP_START_ADDRESS;
157       uint16_t i = 0;
158
159       do {
160         if(pulFlashBase[i] == 0xAABBCCDD)
161           break;
162         pulSRAMBase[i] = pulFlashBase[i];
163       } while(++i);
164
165       __set_MSP(*((volatile uint32_t*) APP_START_ADDRESS)); //Set the MSP
166
167       SYSCFG->CFGR1 |= 0x3; /* __HAL_RCC_SYSCFG_CLK_ENABLE()
168                                already called from HAL_MspInit() */
169
170       /* We are now ready to jump to the main firmware */
171       uint32_t JumpAddress = *((volatile uint32_t*) (APP_START_ADDRESS + 4));
172       void (*reset_handler)(void) = (void*)JumpAddress;
173       reset_handler(); //We start the execution from he Reset_Handler of the main firmware
174
175       for (;;)
176         ; //Never coming here
177     }
178   }
```

The code we are interested in starts at line 155. Two pointers are defined: one starting at the beginning of SRAM memory (`pulSRAMBase`) and one at the beginning of the main firmware (`pulFlashBase`, which is equal to `0x0800 2C00` following the previous example). The loop at lines [158:162] does a copy of the *vector table* in SRAM, until the current flash memory location contains the value `0xAABBCCDD` (more about this soon). The MSP is then set to the end of SRAM (this should be unnecessary, but just in case...) and the *physical remap* is performed (line 167). The control is then transferred to the main firmware.

There are several things to note. First of all, to simplify the copy process and to avoid that the *vector table* is overwritten by the growing stack, the *vector table* is copied in SRAM starting from its beginning, and the rest of the application data (formed by `.data` section, `.bss`, heap and stack) is placed next (see **Figure 22.8**). This requires that the linker script of main firmware is properly configured, as shown below:

```
MEMORY {
  FLASH (rx) : ORIGIN = 0x08002C00, LENGTH = 64K - 10K
  RAM (xrw) : ORIGIN = 0x200000B8, LENGTH = 8K - 0xB8
```

Secondly, we need a way to know where the *vector table* ends. Since not all IRQs are usually enabled in an application, we can place the sentinel value `0xAABBCCDD` inside the first vector entry that comes right after the last used IRQ. For example, assuming that our main firmware uses the USART2 in interrupt mode, we can see that this IRQ is the 46th entry inside the vector table. We can so place that value in the 47th entry. This can be easily performed by modifying the file **startup_stm32f0Xxx.s**, as shown below.

Filename: `src/startup_stm32f030x8.S`

```
180   .word  SPI1_IRQHandler          /* SPI1                */
181   .word  SPI2_IRQHandler          /* SPI2                */
182   .word  USART1_IRQHandler        /* USART1              */
183   .word  USART2_IRQHandler        /* USART2              */
184   .word  0xAABBCCDD               /* Reserved            */
185   .word  0                        /* Reserved            */
186   .word  0                        /* Reserved            */
```

In this way we have a generic and configurable way to set the end of *vector table*. Looking at the previous linker script fragment, we can see that we subtract from SRAM memory size the value 0xB8, which is 184 in base 10. Dividing 184 by 4 bytes, we have 46, which corresponds to the last vector table entry.

Finally, note that the `SYSCFG` is a peripheral separated from the Cortex-M core, and we need to enable it by calling the `__HAL_RCC_SYSCFG_CLK_ENABLE()`.

22.3.2 How to Use the `flasher.py` Tool

As said before, you can find a Python 2 script named `flasher.py` inside the book source files for this chapter. This tool simply allows to upload to the MCU a firmware generated using the Intel

HEX binary format, a specification for binary files developed by Intel several years ago and still widespread especially in low-cost embedded platforms. The source code of this script is not shown here, but it should be easy to understand the way it is made. This script requires three additional modules: `pyserial`, `IntelHex` and `pycryptodome` libraries[19].

You can easily install them using the `pip` command:

```
$ sudo pip install intelhex pycryptodome pyserial
```

The script is designed to accept two arguments at command line:

- The serial port corresponding to the Nucleo VCP
  - In Windows this is equal to "COMx" string, where *'x'* must be replaced with the COM number corresponding to Nucleo VCP (e.g. COM3).
  - In Linux and Mac OS this corresponds to a file mapped in the `/dev` path (usually something similar to `/dev/tty.usbmodemXXXX`).
- The complete path to the HEX file corresponding to the main firmware.

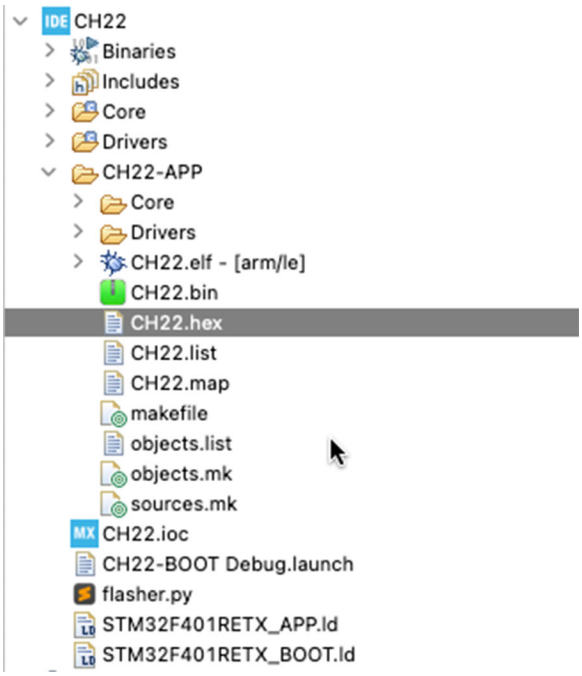


Figure 22.9: The binary file in HEX format inside the Eclipse build folder

By default, the GNU MCU Eclipse tool-chain automatically generates the HEX file of the the compiled firmware. You can find it inside the *build folder*: this is an Eclipse folder with the same name of the active build configuration (usually named **Debug** or **Release**). **Figure 22.9** shows the

[19] `pycryptodome` is a collection of both secure hash functions (such as SHA256 and RIPEMD160), and various encryption algorithms (AES, DES, RSA, etc.). It is the most widespread cryptographic library for Python. `IntelHex` is a small library that allows to easily manipulate Intel HEX files. It is developed by Alexander Belchenko and distributed under the BSD license.

build folder corresponding to active configuration (**CH22-APP1**) if you are working on the official book samples repository.

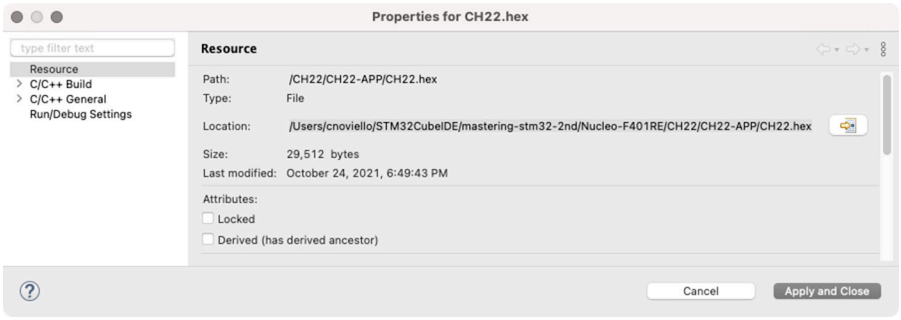


Figure 22.10: How to derive the full path of the HEX file

You can derive the full path to the HEX file by clicking with the right mouse button on it and then selecting **Properties**. You can find the full path inside the **Resource** view, as shown in **Figure 22.10**.

23. Running FreeRTOS

Taking full-advantage of the computing power offered by 32-bit microcontrollers is not easy, especially for powerful STM32F4/G4/F7/H7 series. Unless our device needs to perform simple tasks, the correct allocation of computing resources requires special care during the firmware development. Moreover, the use of improper synchronization structures and poor-designed interrupt service routines could lead to the loss of important asynchronous events and to the overall unpredictable behavior of our device.

Real Time Operating Systems (RTOS) take advantage of the exceptions system provided by Cortex-M cores to bring to programmers the notion of *thread*[1], an independent execution stream which "contends" the MCU with other threads involved in concurrent activities. Moreover, they offer advanced synchronization primitives, which allow both to coordinate the simultaneous access to physical resources from different threads and to avoid wasting CPU cycles while waiting for slower and asynchronous events.

The market segment of RTOSes is quite crowded nowadays, with several commercial as well as free and open source solutions available to programmers. Moreover, due to the increasing diffusion of the IoT, many RTOS evolved with the addition of support to several connectivity solutions and to remote cloud platforms. Being the Cortex-M a standardized architecture among a lot of silicon manufacturers, STM32 developers can choose from a wide portfolio of RTOS systems, depending on their need of complexity handling and dedicated (and maybe commercial) support.

Traditionally, ST Microelectronics has developed full support to one of the most popular and free and Open Source RTOS: FreeRTOS. FreeRTOS has become a sort of standard in the electronics industry, and it is also widely adopted by the Open Source community. According to some statistics[2], FreeRTOS is the most widespread RTOS for embedded systems on the market today.
Recent CubeMX versions provide full integration of FreeRTOS 10.x for all STM32 microcontrollers. It is really easy for developers to setup FreeRTOS in a CubeMX project and start using it in their real-life applications. FreeRTOS was acquired in 2017 by Amazon, and it is now part of the AWS ecosystems. It is now distributed under a more permissive and "commercially friendly" MIT license.

[1] Some RTOSes, like FreeRTOS, use the term *task* to indicate an independent execution stream contending the CPU with other tasks. However, this author considers this terminology not appropriate. Traditionally, in general purpose Operating Systems, *multitasking* is a method by which multiple tasks, also known as *processes*, share common hardware resources (mainly the CPU and the RAM). With a multitasking OS, such as Linux, you can simultaneously run multiple applications. Multitasking refers to the ability of the OS to quickly switch between each computing task to give the impression that different applications are executing multiple actions simultaneously. A process has one relevant characteristic: its memory space is physically insulated from other processes, thanks to features offered by the *Memory Management Unit* (MMU) inside the CPU. *Multithreading* extends the idea of multitasking into single processes, so that you can subdivide specific operations within a single application into individual *threads*. Each thread could run in parallel. The important trait of threads it that they share the same memory address space. True embedded architectures, like the STM32 are, do not provide a MMU (only a features-limited *Memory Protection Unit* - MPU - is available in some of them). The absence of this unit does not allow to have separated address spaces, since it is impossible to alias physical addresses to logical ones. This means that they can carry out just one single application, which can be eventually split in several threads sharing the same memory address space. For this reason, we will talk about *threads* in this book, even if sometimes we will use the word "task" when talking about some FreeRTOS APIs or to indicate an activity of the firmware in general terms.

[2] https://bit.ly/3maUQT0

In the recent years, the evolution of FreeRTOS was mostly focused on the integration of AWS IoT services.

However, ST announced in December 2020[3] a key collaboration with Microsoft, which recently acquired the company behind ThreadX RTOS, renaming it in Azure RTOS. Microsoft is pushing hard the development of a complete ecosystems of IoT solutions for embedded platforms, and Azure RTOS is the central element of Microsoft offering. The collaboration is mutual: ST will integrate the Azure RTOS as STM32Cube expansion pack, and Microsoft will push STM32 microcontrollers[4] as reference "platform" for IoT applications. ST will not limit the support to Azure RTOS to the kernel. The integration will include FileX, a filesystem offering advanced features on NAND and NOR Flash memories like fault tolerance, or wear leveling; NetX, and NetX Duo, which are network stacks that offer TCP/IP, IPv4, and IPv6, as well as many upper-level protocols used in IoT like MQTT or COAP; USBX that facilitates the use of a USB interface, both as a host or as a device, with a complete set of supported USB classes.

At the time of writing this chapter (January 2022), ST released just X-AZURE Cube Extension Pack for STM32H7 and STM32F4 families. According to this author, it will take more than a year before STM will complete the integration to the complete STM32 portfolio. Moreover, it is still not clear how the RTOS market will evolve in the next years. For this reason, this book will cover just FreeRTOS, leaving the possibility to cover Azure RTOS to future updates of the book.

23.1 Understanding the Concepts Underlying an RTOS



This paragraph gives a quick introduction to the main concepts underlying real-time Operating Systems. Experienced users can safely skip it.

Except for the ISRs and exception handlers, all the examples built so far are designed so that our applications are composed by just one execution stream. Typically, starting from the `main()` routine, a large and infinite `while` loop carries out firmware tasks:

```
...
while(1) {
        doOperation1();
        doOperation2();
        ...
        doOperationN();
}
```

The time spent by each `doOperationX()` is broadly estimated by the developer, who has the responsibility to avoid that one of those functions sticks for too much time, preventing other parts

[3] https://blog.st.com/azure-rtos/
[4] https://azure.microsoft.com/it-it/blog/new-azure-rtos-collaborations-with-leaders-in-the-semiconductor-industry/

of the firmware from running correctly. Moreover, the calling order of the functions also *schedules* their execution, defining the sequence of operations performed by the firmware. This, indeed, is a form of *cooperative scheduling*[5], where each function concurs to the execution of the next activity by voluntarily releasing the control periodically.

In this early form of multiprogramming, there is no guarantee that a function cannot monopolize the CPU. The application designer carefully needs to ensure that every function should be carried out in the shortest possible time. In this execution model, an "innocent" *busy loop* can have dramatic effects. Let us consider the following pseudo-code:

```
uint32_t timeKeep = HAL_GetTick();
uint32_t uartData[20];

void blinkTask() {
  while(HAL_GetTick() - timeKeep < 500);
  HAL_GPIO_TooglePin(GPIOA, GPIO_PIN_5);
  timeKeep = HAL_GetTick();
}

uint8_t readUART2Task() {
  if(HAL_UART_Receive(&huart2, &uartData, 20, 1) == HAL_TIMEOUT)
    return 0;
  return 1;
}

while(1) {
  blinkTask();
  readUART2Task();
}
```

This code is quite common among several unexperienced embedded developers and, under certain hypothesis, it is also correct. However, that code has a subtle weird behavior. The `blinkTask()` is designed so that it will busy-spin for 500ms before releasing the control. If data arrives on the UART interface during this period, the `readUART2Task()` will certainly lose some data[6]. A better way to write down the `blinkTask()` is the following one:

[5]Experienced user will point out that it is not correct to talk about *cooperative scheduling* in this context for two fundamental reasons: the execution order of tasks is fixed (the "schedule" is computed by the programmer during the firmware development) and each routine is not able to save its execution context before leaving, that is the stack frame of the `doOperationX()` routine is destroyed when it returns. As we will see in a while, *co-routines* are a generalization of subroutines in *non-preemptive multitasking* systems.

[6]With high *baudrates*, polling the UART is certainly not correct at all, but here we are interested to the point.

```
void blinkTask() {
  if(HAL_GetTick() - timeKeep > 500) {
    HAL_GPIO_TooglePin(GPIOA, GPIO_PIN_5);
    timeKeep = HAL_GetTick();
  }
}
```

A simple modification to that routine ensures that we will not lose data coming from the UART in the majority of situations, unless the UART transfers data quickly.

As you can see, with *cooperative scheduling* programmers have a great responsibility in ensuring their code will not affect the overall activities of the firmware, introducing performance bottlenecks.

The voluntary releasing of the execution flow is not the only limit of the code seen so far. Let us have a closer look at the `blinkTask()` routine. Here we need a global variable[7], `timeKeep`, to keep track of the global tick counter incremented by the CubeHAL every 1ms and to perform a comparison to check if 500ms are elapsed. This is required because every time a routine exits, its execution context (that is, the stack frame) is popped from the main stack and it is destroyed. Unless we do not use some nasty tricks offered by the language[8], there is no way to exit from a function without losing its context.

Continuation routines, abbreviated as *co-routines* or simply *coroutines*, are program structures that generalize the concept of subroutines for non-preemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations. Co-routines require special support from the *run-time* of the language, and they are traditionally provided from more high-level languages like Scheme, but also more widespread languages like Python and Perl provide a form of co-routines. A co-routine is said not *to return* but *to yield* the execution flow. For example, the `blinkTask()` could be rewritten using co-routines in this way:

```
1  void blinkTask() {
2    uint32_t timeKeep = HAL_GetTick();
3    while (1) {
4      if(HAL_GetTick() - timeKeep > 500) {
5        HAL_GPIO_TooglePin(GPIOA, GPIO_PIN_5);
6        timeKeep = HAL_GetTick();
7      }
8      yield; /* Pass the control to another routine, e.g. the scheduler */
9    }
10 }
```

Co-routines work so that, the next time the control passes to `blinkTask()`, the execution will resume from line 3. We will not go into details of how *co-routines* are implemented in languages that support them. However, this usually involves the creation of separated stacks for each *co-routine*, which could call other *co-routines* that in turn may pass the control to other continuations.

[7] A local and static variable would have the same effect, however without changing the concept.
[8] Which involves the use of the C `setjmp()` and `longjmp()` functions.

A *preemptive multitasking* Operating System is a coordinator of physical resources that allows the execution of multiple computing tasks[9], each one with its independent stack, by assigning a limited *quantum time* (also called *slice time*) to each task. Every *task* has a well-defined temporal window, usually large about 1ms in embedded systems, during which it performs its activities before it is *preempted*. The RTOS kernel decides the execution order of the tasks ready to be executed using a scheduling policy: a *scheduler* is an algorithm that characterizes the way the OS plans the execution of tasks.

A task is "moved" in/out from the CPU by a *context switch* operation. A *context switch* is performed by the OS, thanks to hardware features we will explore next, which makes a "snapshot" of the current task state by saving the internal CPU registers (`PC`, `MSP`, `R0..R15`, etc.) before switching to another task, which will be able to "re-use" again the CPU for the same *quantum time* (or even less if "it wants").

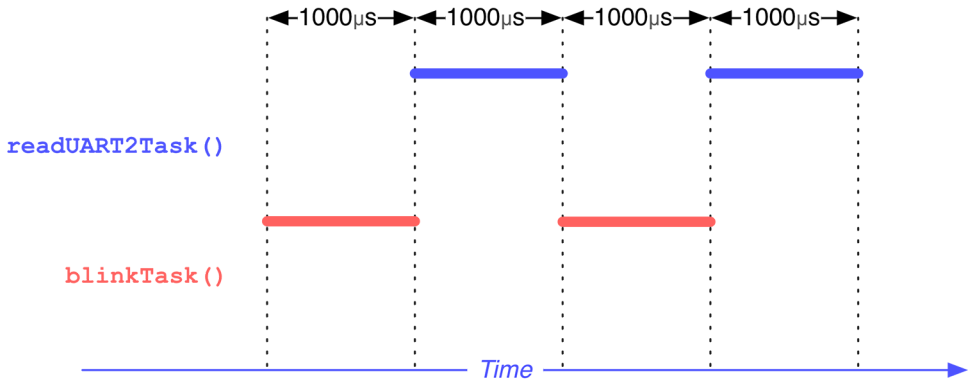


Figure 23.1: How an OS schedules the tasks execution by assigning them a fixed quantum time

Figure 23.1 shows how the task preemption works for the case of the example seen before. Here we are supposing that we have just two tasks: one for the `blinkTask()` routine and one for the `readUART2Task()` one. The OS start scheduling the `blinkTask()` task, which can "use" the CPU for 1000µs (that is, 1ms)[10]. After the time is gone, the OS schedules the execution of the `readUART2Task()` which can now occupy the CPU for the same *quantum time*. After that period, the CPU will reschedule the first task, and so on.

Figure 23.2 shows the way SRAM memory is typically organized by an OS. Each task is represented by a memory segment containing the *Thread Control Block* (TCB), which is nothing more than a descriptor containing all relevant information related to the task execution just "a moment"[11] before it is preempted (the stack pointer, the program counter, CPU registers and other few things), plus the stack itself, that is the stack frame of those routines currently invoked on the thread stack. By jumping between several threads, thanks to *context switch* operations, the OS guarantees the same

[9] In this paragraph, and only in this one, the term *task* and *thread* will be used indiscriminately.

[10] Those values of quantum time are indicative, since the exact duration of a quantum is affected by a lot of things. Not last, the overhead connected with a *context switch*, which is non-negligible. Moreover, here we are assuming that tasks have all the same priority, which usually is not true especially in embedded systems.

[11] This is not true at all, since before a task is preempted several other things take place. However, explaining into details these aspects is outside the scope of this book. Refer to Joseph Yiu books if interested in deepening how *context switch* is performed on Cortex-M based microcontrollers.

execution time to all threads, giving the impression that firmware activities are performed in parallel.

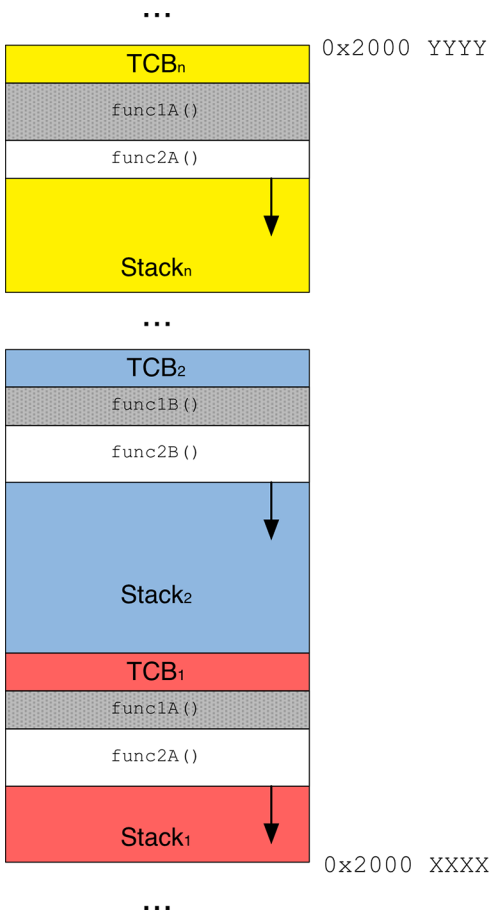


Figure 23.2: How the memory is organized in several tasks by an OS

A *Real Time Operating Systems* (RTOS) is an OS able to offer the notion of *multitasking* (or better, multithreading as seen in note 1) while ensuring response within specified time constraints, often referred to as *deadlines*. Real-time responses are often understood to be in the order of milliseconds, and sometimes microseconds. A system not specified as operating in real-time cannot usually guarantee a response within any timeframe, although actual or expected response times may be given. General-purpose Operating Systems (like Linux, Windows and MacOS) cannot be real-time Operating Systems (even if exist some their derivative releases - especially of Linux - engineered for real-time applications) for two simply reasons: *pagination* and *swapping*. The former allows to segment the task memory in small chunks named *pages*, which can be scattered in the RAM and aliased from the MMU giving the illusion that the process can manage the whole 4GB address space (even if the computer do not provide that amount of SRAM). The latter allows to *swap-in*/*swap-out* those "unused" pages on an external (and slower) memorization unit (typically a hard drive). Those two features are intrinsically non-deterministic and preventing the OS from servicing requests in short and countable time.

An RTOS allows to use the first version of the `blinkTask()` function minimizing the impact of the

busy loop on the UART transfer process[12]. However, as we will see later in this chapter, typically an RTOS also gives us tools to completely avoid busy loops: using software timers it is possible to ask to the OS to re-schedule the `blinkTask()` only when the specified amount of time is elapsed. Moreover, the RTOS also provides ways to voluntary release the control when we know that it is completely useless to wait for an operation that will be performed by another task (or if we are waiting for an asynchronous event).

We have said just one moment before that an RTOS gives a way to voluntary release the control to other threads. But what if one task does not want to release it? For example, the first release of the `blinkTask()` routine could monopolize the CPU up to more than 500ms in the worst case that, given the typical *slice time* of 1ms, is a long time. So, what can to perform the *context switch*? It is impossible to "jump" to other program instructions (a *context switch*, is a sort of *goto* to another program instruction) without losing one relevant information: the value of the program counter itself.

The *context switch* needs a substantial help from the hardware. In Chapter 7 we have seen that interrupts and exceptions are a source of multiprogramming. The way they are handled by the Cortex-M core allows to jump to the exception handler without losing the current execution context. By taking advantage of a dedicated hardware timer, usually the *SysTick* one, the RTOS uses the periodic interrupt generated on the overflow event to perform the *context switch*. This timer is configured to overflow (or underflow in case of the *SysTick*, which is a downcounter timer) every 1ms. The RTOS then captures the exception and saves the current execution context in the TCB, passing the control to the next task in the scheduling list by restoring its execution context and exiting from the timer interrupt. The preempted threads will not know anything that this happened[13].

[12]This does not mean that using an RTOS we can write bad code without impacting on the overall performances. This only means that, a true *preemptive scheduler* can guarantee a higher multiprogramming degree, ensuring that all threads have the same CPU time-slice. Unless we mess with task priorities, as we will see later.

[13]However, this could not correspond to what an RTOS does. The story here is more complex, and it is related to the specific hardware architectures and to the way interrupts are prioritized. During the execution of an interrupt handler, another interrupt with a higher priority could suspend the execution of the current interrupt, as seen in Chapter 7. But when this happens, the CPU cannot switch to the *thread mode* (which is the regular mode when the normal code is executed) by performing the task switch without prior exiting from all interrupts (which run in the *handler mode* - a special mode provided by Cortex-M core during the exception handling). This means that if the *SysTick* IRQ takes place while another IRQ is active, the *SysTick* exception handler cannot perform the *context switch* (that is to pass the control to another task running in *thread mode*), because another code running in *handler mode* has been preempted and needs to complete its activities. Usually this is solved by deferring the effective *context switch* operation to the *PendSV Handler*, which is an exception configured to run at the lowest priority. However, this is just one way to implement the *context switch*. If interested in deepening this topic, you have to consult the source code or the documentation of your RTOS.

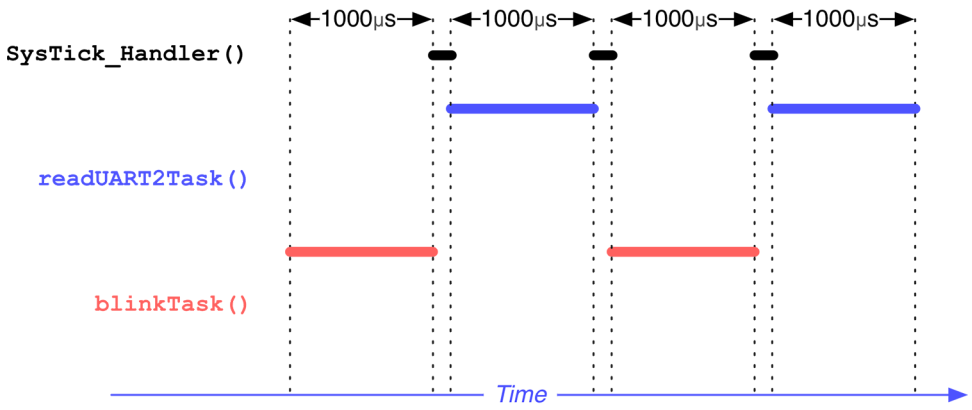


Figure 23.3: The impact of the *context switch* on tasks scheduling

In light of the considerations that we have shown up to this point, the **Figure 23.1** needs to be updated with the one shown in **Figure 23.3** where the time spent by the OS while performing *context switch* is also considered. *Context switches* are usually computationally intensive, and much of the design of operating systems is to optimize the use of *context switches*. Special care must be placed when developers decide to change the underflow frequency of the *SysTick* timer (often increasing it), which also affects the slice time of each individual task, and hence the number of *context switches* per second.

Before we can start doing practical things with an RTOS, we need to explain just one last concept. What about the case when a task wants to voluntary leave the control? In this case often RTOSes use the `SVC` (*SuperVisor Call*) instruction implemented by Cortex-M processors, which causes that the `SVC_Handler` exception handler is called or force the `PendSV` exception to be raised. Explaining when they use one and when the other is outside the scope of this book, and it is also a design choice of OS maker. For more information, refer to Joseph Yiu[14] books if interested in deepening these topics.

This is just an introduction to the complex topics underlying an RTOS. We will analyze several other concepts, mainly related to the synchronization of concurrent tasks, later in this chapter. We will now start seeing the most relevant features of FreeRTOS.

23.2 Configuring FreeRTOS and the CMSIS-RTOS v2 Wrapper

As said at the beginning of this chapter, FreeRTOS is the OS chosen by ST as official RTOS for its Cube distribution. Recent releases of CubeMX offer a good support to this OS and including it as *middleware* component in a project is really easy. A lot of additional modules of the CubeHAL (like the LwIP stack) rely on the services provided by it.

However, ST did not limit its integration in shipping FreeRTOS in its CubeHAL distribution. It has built two complete CMSIS-RTOS wrappers over it, one for CMSIS-RTOS v1 and one for CMSIS-RTOS v2, allowing to develop CMSIS-RTOS compliant applications. We talked about CMSIS-RTOS

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

in Chapter 1, when we have introduced the whole stack. The idea behind the CMSIS initiative is that, using a common standardized set of APIs among several silicon manufacturers and software vendors, it is possible to "easily" port our application on different microcontrollers from other vendors. For this reason, we will introduce the FreeRTOS functionalities using as much as possible the CMSIS-RTOS v2 API.

23.2.1 The FreeRTOS Source Tree

FreeRTOS source code is organized in a compact source tree, which spreads over few folders and files. The **Figure 23.4** shows how FreeRTOS is organized inside the CubeHAL. The files `.c` found in the root folder contain the main OS features (for example, the file `tasks.c` contains all those routines related to the thread management). The sub-folder `include` contains several include files used to define the most of C `struct` and macros used by the OS. The most relevant of those files is the `FreeRTOSConfig.h`[15] one, which includes all the user-defined macros used to configure the RTOS according to user's needs. The other sub-folder contained in the root tree is `portable`. FreeRTOS is designed to run on about many different hardware architectures and compilers, while ensuring the same consistent API. All platform-specific features are organized inside two files[16], `port.c` and `portmarco.h`, which are in turn collected in the sub-folder specific of the given architecture. For example, the folder `portable/GCC/ARM_CMO` contains `port.c` and `portmarco.h` files providing the code specific for the Cortex-M0/0+ architecture and the GCC compiler.
Another important subfolder is `MemMang`. This folder contains 5 different memory allocation schemes used by FreeRTOS. As we will see later, FreeRTOS offers to the user the possibility to select the best memory allocation policy fitting the application specific requirements. Finally, the `CMSIS-RTOS_V2` folder contains the CMSIS-RTOS v2 compliant layer developed by ST on the top of FreeRTOS.

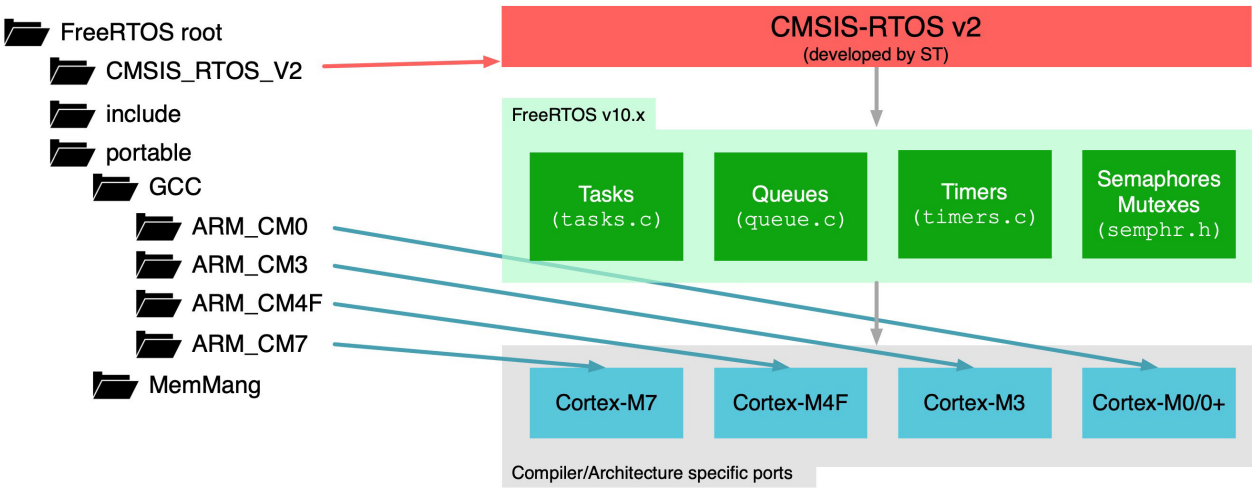


Figure 23.4: The FreeRTOS source tree organization in the CubeHAL

[15]However, that file in the FreeRTOS source tree is just a template. The actual file used by CubeMX to configure FreeRTOS is the **Core/Inc/FreeRTOSConfig.h** one.
[16]This part of FreeRTOS is considered separated from the core FreeRTOS source three, and it is said to implement the *port layer* of FreeRTOS.

23.2.1.1 How to Configure FreeRTOS Using CubeMX

CubeMX allows to easily add FreeRTOS to an existing project. Once you have configured the MCU peripherals in CubeMX, you can easily enable the FreeRTOS middleware by selecting the wanted CMSIS-RTOS wrapper (V1 or V2) in the *Middleware* section of the *Categories* pane, as shown in **Figure 23.5**.

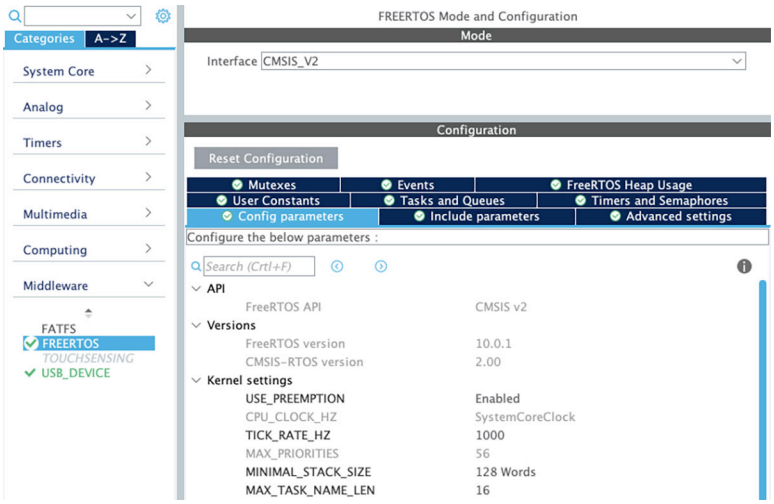


Figure 23.5: How to enable the FreeRTOS middleware in CubeMX

In the configuration section it is possible to set the FreeRTOS configuration parameters. We will analyze the most relevant ones in this chapter. Once you generate the project, CubeMX will show you a warning message (see **Figure 23.6**). Let us explain the meaning of those two messages.

The first message warns you to use a timer different from the *SysTick* one for the HAL timebase generation. CubeMX asks this because FreeRTOS is designed so that it automatically sets the *SysTick* IRQ priority to the lowest one (highest priority number). This is an architectural requirement of FreeRTOS, which unfortunately conflicts with the way the HAL is designed.

As said several other times before, the STM32Cube HAL is built around a unique timebase source, which usually is *SysTick* timer. `SysTick_Handler()` ISR automatically increments the global *tick* counter every 1ms. The HAL uses this unique counter for the `HAL_Delay()` function, which is used often in several HAL routines. These HAL routines can in turn called by the `HAL_<PPP>_IRQHandler()` functions, which are executed in the context of an ISR (for example, the `HAL_TIM_IRQHandler()` is called from the ISR of a timer). If the *SysTick* IRQ is not configured to run at the highest priority interrupt (which is 0 in Cortex-M based processors), then calling the `HAL_Delay()` from an ISR context may lead to *deadlocks*[17] if the priority of the ISR that makes call to the `HAL_Delay()` is higher than the one of the *SysTick* timer (and this is always true if you use FreeRTOS, as said before). So, it is best to use another timer for the HAL. To change the HAL timebase source, follow the instructions written in Chapter 11.

[17]In concurrent programming, a *deadlock* is a situation in which two or more concurrent execution streams are each waiting for the other to finish, and thus neither ever does. Incur in *deadlock* is anything but difficult, and all programmers soon or later will encounter this hard-to-debug event.

The other warning message is related to usage of some functions from the `newlib` library (the *standard* C run-time library) or from the `newlib-nano` library (the *compact* C run-time library) in multi-thread applications. We will deepen this topic later in this chapter. For now, consider that the usage of standard C routines must be handled with special care if you are going to call them from more than a thread (including *interrupt handlers*).

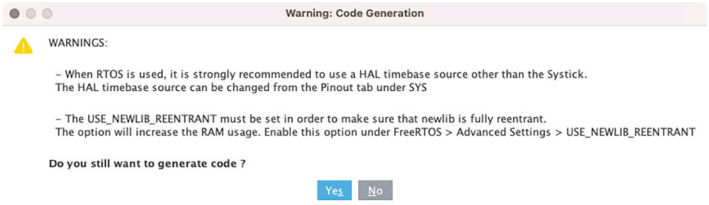


Figure 23.6: The warning messages about timebase generator for the HAL and re-entrancy of `newlib`

23.3 Thread Management

Once we have configured the Eclipse project, we can start coding using the CMSIS-RTOS layer and hence FreeRTOS.

At the base of all RTOSes there is the notion of thread, which we have analyzed in the first paragraph of this chapter. A thread is nothing more than a C function, which FreeRTOS requires to be defined in the following way:

```
void ThreadFunc(void const *argument) {
  while(1) {
    ...
  }
  osThreadTerminate(osThreadGetId());
}
```

The function `osThreadTerminate()` is used to terminate a thread, and it accepts the *Thread ID* (TID), which we are going to see in a while. A thread is usually made of an infinite loop that contains the thread instructions. Placing the `osThreadTerminate()` outside that loop is usually a precaution in case the control exits from that loop, because it is not correct to terminate a thread by simply returning from its function. Passing the `NULL` parameter to the `osThreadTerminate()` function will cause that the function returns the `osErrorParameter` error, without correctly deleting the thread. So ensure that the `osThreadTerminate()` is invoked by passing the correct TID.

To start a new thread with the CMSIS-RTOS v2 API (or simply CMSIS-RTOS2) we use the following function:

```
osThreadId_t osThreadNew (osThreadFunc_t func, void *argument, const osThreadAttr_t *attr);
```

The `osThreadAttr_t` is the thread attribute descriptor, a C `struct` defined in the following way:

```
typedef struct {
  const char     *name;      /* Thread name */
  uint32_t       attr_bits;  /* Bitmask to configure the thread:  this is meaningless in FreeR\
TOS */
  void           *cb_mem;    /* Control block to hold thread's data (default: NULL).
                                Used only for static allocation */
  uint32_t       cb_size;    /* Size of provided memory for control block (default: 0) */
  void           *stack_mem; /* Pointer to the memory holding the thread stack (default: NULL)\
.
                                Used only for static allocation */
  uint32_t       stack_size; /* Size of provided memory for stack (default: 128 * 4) */
  osPriority_t   priority;   /* Initial thread priority (default: osPriorityNormal) */
  TZ_ModuleId_t  tz_module;  /* TrustZone module identifier (used in Cortex-M33 based MCUs) */
  uint32_t       reserved;   /* Reserved (must be 0) */
} osThreadAttr_t;
```

It is possible to pass to the `osThreadNew()` the value `NULL` for the `attr` parameter: in this case the thread will be created with the default stack size specified by the macro `configMINIMAL_STACK_SIZE` inside the **Core/Inc/FreeRTOSConfig.h** file.
Now it is the right time to see a practical example.

Filename: `Core/Src/main-ex1.c`

```
21 #include "nucleo_hal_bsp.h"
22 #include "cmsis_os.h"
23
24 /* Private function prototypes -----------------------------------------------*/
25 void blinkThread(void *argument);
26
27 /* Definitions for blinkThread */
28 osThreadId_t blinkThreadID;
29 const osThreadAttr_t blinkThread_attr = {
30   .name = "blinkThread",
31   .stack_size = 128 * 4, /* In bytes */
32   .priority = (osPriority_t) osPriorityNormal,
33 };
34
35 int main(void) {
36   HAL_Init();
37
38   Nucleo_BSP_Init();
39
40   /* Init scheduler */
41   osKernelInitialize();
42
43   /* Creation of blinkThread */
44   blinkThreadID = osThreadNew(blinkThread, NULL, &blinkThread_attr);
45
```

```
46    /* Start scheduler */
47    osKernelStart();
48
49    /* We should never get here as control is now taken by the scheduler */
50    while (1);
51  }
52
53  void blinkThread(void *argument) {
54    while(1) {
55      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
56      osDelay(500);
57    }
58  }
```

Lines [29:33] define the attribute of the thread, assigning to it the name "blinkThread", specifying a stack size equal to 512 bytes and assigning a normal priority (more about this soon).



Read Carefully

The CMSIS-RTOS API expresses the thread stack size in bytes, and you will find this information in the CMSIS-RTOS layer on the top of FreeRTOS developed by ST. However, FreeRTOS defines the stack size as a multiple of `StackType_t` type, which in a Cortex-M processor corresponds to an `uint32_t`, and hence four bytes. This means that, the value we pass to the `osThreadNew()` macro shall be a multiple of four bytes. This is the reason why it is convenient to specify the stack size as a multiple of 4 explicitly: the `osThreadNew()` will internally divide the passed value by four, and then the corresponding FreeRTOS API `xTaskCreate()` will again multiply the value by four. A useless and somewhat intricate procedure that says it all about the effective portability of these abstraction layers.

The `main()` routine initialize the main OS's scheduler by calling the `osKernelInitialize()`[18] and the `osThreadNew()` at line 44 effectively creates the new thread and asks to the kernel to schedule its execution, returning the *Thread ID* (TID): this is used by other APIs to manipulate the thread status and its configuration. The second parameter of the `osThreadNew()` function is an optional parameter to pass to the thread (through the `*argument` pointer). Finally, we start the kernel scheduler by using the function `osKernelStart()`, which never returns unless something wrong happens.

The function `blinkThread()` is nothing more than the omnipresent blinking application. The only notably difference is the use of the `osDelay()` function instead of the classical `HAL_Delay()`: the `osDelay()` is designed so that the thread will remain in blocked state for 500ms without impacting on the CPU performances. After that time, the thread will be resumed and the LD2 LED will be toggled again. We will talk more about the `osDelay()` function later.

[18] For the sake of completeness, this function does not anything relevant here, since there is no kernel initialization procedure in FreeRTOS, which performs this job in the `vTaskStartScheduler()` routine (wrapped by the CMSIS-RTOS2 `osKernelStart()` routine).



Read Carefully

Note that, since we are using here the *SysTick* as timebase for the FreeRTOS kernel, the ISR for the *SysTick* IRQ is directly defined inside the **Middlewares/Third_Party/FreeRTOS/-Source/CMSIS_RTOS_V2/cmsis_os2.c** file, around at line 159. To setup a custom handler for the *SysTick* IRQ the macro `USE_CUSTOM_SYSTICK_HANDLER_IMPLEMENTATION` must be set to `1`. Instead, the configuration of the *SysTick* timer is performed by FreeRTOS itself with the function `vPortSetupTimerInterrupt()` (inside the corresponding port file - for example, **portable/GCC/ARM_CMO/port.c**), which is called when the scheduler is started by the `osKernelStart()` routine. **This is a paramount thing to take in account when changing the MCU core frequency**.

23.3.1 Thread States

In FreeRTOS a thread can have two major execution states: *running* and *not running*. On a single-core architecture, just one thread at once can be in *running* state.

In FreeRTOS the *not running* state is characterized by several sub-states, as shown in **Figure 23.7**. A *not running* thread can be *ready* (this is also the state of new threads), that is it is ready to be scheduled for execution by the RTOS kernel.

A *running* thread can voluntary suspend its execution, by calling the `osThreadSuspend()` function, which accepts the TID of the thread to suspend. In this case the thread assumes the *suspended* state. To resume a *suspended* thread the `osThreadResume()` is used.

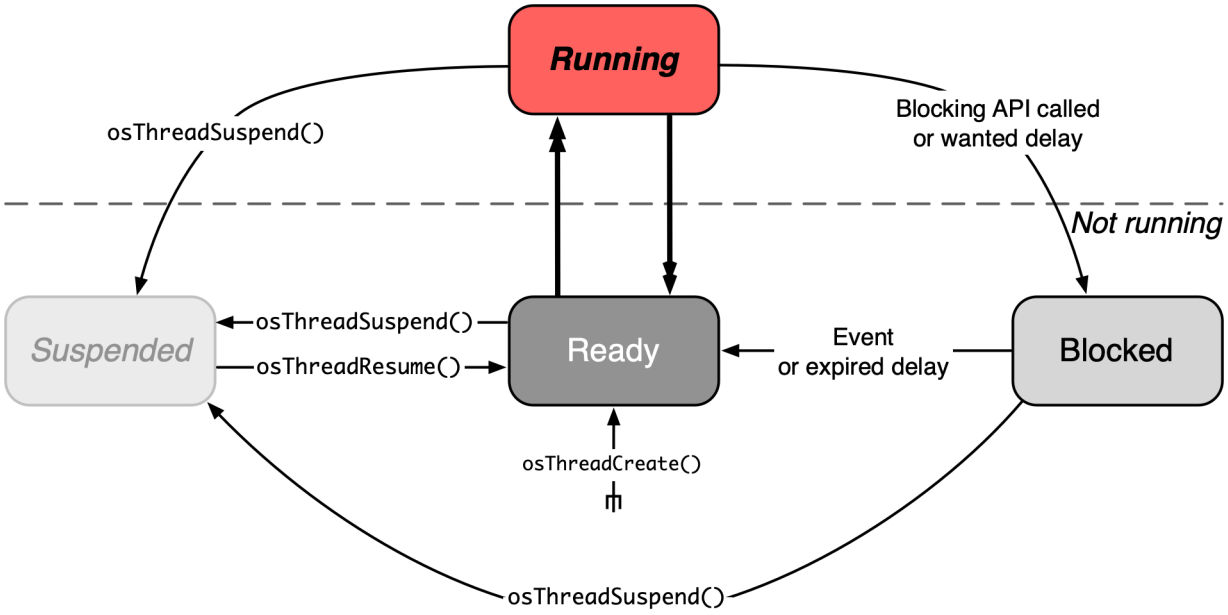


Figure 23.7: The possible states of a thread in FreeRTOS

A running thread can put itself in *blocked* state by start waiting for "an external" event. This event could be, for example, a synchronization primitive (e.g., a semaphore) that will be unlocked from another thread. Another source of blocking state is the `osDelay()` function, which places the thread in blocked state until the specified delay time does not pass. A *blocked* thread can be placed in *ready* state, and hence it becomes ready to be scheduled for execution, or in *suspended* state.

It is important to clarify, to avoid any misunderstanding, that a *suspended* or *blocked* thread needs the intervention of an external entity to return in ready state.

23.3.2 Thread Priorities and Scheduling Policies

In the first example we have seen that each thread has a priority. But which practical effects have priorities on threads execution? Priorities impact on the scheduling algorithm, allowing to alter the execution order in case a thread with a higher priority turns in *ready* state. Priorities are a fundamental aspect of RTOSes and provide the foundation blocks to achieve short responses to deadlines. **It is important to underline that thread priority is not related to the priority of IRQs**.

Imagine you are designing the control board of a machine that could potentially cause injuries to workers in critical situations. Usually, this type of machines has an emergency stop button. That button could be connected to one pin of the MCU, and the corresponding interrupt may resume a blocked thread waiting for this event. This thread may be designed to shutdown an engine, or something like that, and to place the machine in a safe state.

Once the IRQ fires, the task running at that moment is formally *running* but it is not effectively running on the CPU, which is servicing the ISR. By invoking proper OS routines, that we will see later, the OS places our emergency thread in *ready* mode, but we have to be sure that it will be the first thread to be executed. Priorities allow to programmers to distinguish deferrable activities from not-deferrable ones.

FreeRTOS has a user-defined priority system, which gives a great degree of flexibility in defining priorities. The lowest priority (which means that threads with this priority will always be passed over by higher priority threads, if *ready* to be executed) is equal to zero. The user can then assign increasing priorities to more important threads, up to the maximum value defined by the symbolic constant `configMAX_PRIORITIES` defined in the **Core/Inc/FreeRTOSConfig.h** file.

Table 23.1: The fixed priorities defined in the CMSIS-RTOS2 specification

Priority level	Value	Description
`osPriorityNone`	0	No priority (not initialized).
`osPriorityIdle`	1	Priority: *idle* - Reserved for Idle thread.
`osPriorityLow`	8	Priority: *low*
`osPriorityLow[1..7]`	8+[1..7]	Priority: low + 1..7
`osPriorityBelowNormal`	16	Priority: *below normal*
`osPriorityBelowNormal[1..7]`	16+1	Priority: below normal + 1..7
`osPriorityNormal`	24	Priority: *normal*
`osPriorityNormal[1..7]`	24+1	Priority: normal + 1..7

Table 23.1: The fixed priorities defined in the CMSIS-RTOS2 specification

Priority level	Value	Description
osPriorityAboveNormal	32	Priority: *above normal*
osPriorityAboveNormal[1..7]	32+1	Priority: above normal + 1
osPriorityHigh	40	Priority: *high*
osPriorityHigh[1..7]	40+1	Priority: high + 1
osPriorityRealtime	48	Priority: *realtime*
osPriorityRealtime[1..7]	48+1	Priority: realtime + 1
osPriorityISR	56	Priority: *ISR* - Reserved for ISR deferred thread
osPriorityError	-1	System cannot determine priority or illegal priority
osPriorityReserved	0x7FFFFFFF	Prevents enum down-size compiler optimization

CMSIS-RTOS2, instead, has a well-defined priority scheme, made of eight main levels and - for each main level - seven fine-grain sub-levels (reported in **Table 23.1**), which are mapped on the FreeRTOS priorities. For example, the `osPriorityLow` level has a FreeRTOS priority value equal to 8, and of its sub-level is `osPriorityLow3` corresponding to a FreeRTOS priority value equal to 8+3=11.
The function

```
osStatus_t osThreadSetPriority(osThreadId_t thread_id, osPriority_t priority);
```

allows to change the priority of an existing thread, while the function

```
osPriority_t osThreadGetPriority(osThreadId_t thread_id);
```

allows to retrieve the priority of an existing thread.

It is quite meaningless to talk about thread priorities without knowing the exact scheduling policy adopted by the RTOS. For single-core architectures[19], like the majority of STM32 MCUs, FreeRTOS provides three different scheduling algorithms, which are selected by the right combination of the symbolic constants `configUSE_PREEMPTION` and `configUSE_TIME_SLICING`, both defined in the **Core/Inc/FreeRTOSConfig.h** file[20]. **Table 23.2** shows the combination of these two macros to select the wanted scheduling algorithm.

Table 23.2: How to select the wanted scheduling policy in FreeRTOS

configUSE_PREEMPTION	configUSE_TIME_SLICING	Scheduling algorithm
1	1 or undefined	*Prioritized preemptive scheduling with time slicing*
1	0	*Prioritized preemptive scheduling without time slicing*
0	any value	*Cooperative scheduling*

[19]FreeRTOS provides dedicated scheduling policies for multi-core embedded architectures, which are becoming more and more popular in recent years. This topic will be covered in a future chapter.

[20]By default, the **Core/Inc/FreeRTOSConfig.h** file does not contain the definition of the `configUSE_TIME_SLICING` macro. It is so automatically defined and set to 1 by FreeRTOS in the **Middlewares/Third_Party/FreeRTOS/Source/include/FreeRTOS**. file.

Let us give a quick introduction to these algorithms.

- **Prioritized preemptive scheduling with time slicing**: this is the most common algorithm implemented by all RTOSes and this the default scheduling policy chosen by CubeMX, and it works in this way. Every thread has a fixed priority, which is assigned during its creation. The scheduler will never change this priority, but the programmer is free to reassign a different priority by calling the `osThreadSetPriority()` function. In this mode, the scheduler will immediately preempt a *running* thread if one with a higher priority becomes *ready* to be executed. Being preempted means being involuntary (without explicitly yielding or blocking) moved out of the *running* state into the *ready* state to allow the higher priority thread to become *running*. The *time slicing* (also known as *quantum time*) is used to share CPU processing time between threads **with the same priority**, even when they leave the control by explicitly yielding or blocking. When a thread "consumes" its time slice, the scheduler will select the next running thread in the scheduling list (if available) by assigning it the same slice time. If there are no available *ready* threads, the scheduler will mark as *running* a special thread named *idle*, which we will describe next. The slice time corresponds to the tick time of the RTOS, which by default is equal 1kHz, that is 1ms. This can be changed by configuring the macro `configTICK_RATE_HZ`, and rearranging the UEV frequency of the timer used as timebase generator. Tuning this value it is up to the specific application, and it also depends on how fast the MCU runs. The slower the MCU runs, the slower the tick frequency should be. Usually, a value ranging from 100Hz up to 1000Hz is suitable for a lot of applications.
- **Prioritized preemptive scheduling without time slicing**: this algorithm is almost equal to the previous one, except for the fact that once a thread enters in *running* state, it will leave the CPU only on a voluntary basis (by blocking, stopping or yielding) or if a higher priority thread enters in *ready* state. This algorithm minimizes a lot the impact of the *context switch* on the overall performances, since the number of switches is dramatically reduced. However, a bad designed thread may monopolize the CPU, causing unpredictable behavior of the whole device.
- **Cooperative scheduling**: when this algorithm is used, a thread will leave the CPU only on a voluntary basis (by blocking, stopping or yielding). Even if a higher priority thread becomes *ready*, the OS will never preempt the current thread, and it will reschedule it again in case of an external interrupt. This form of scheduling gives all the responsibility to the programmer, which must carefully design the threads as if he is designing a firmware without using an RTOS.

Special care must be placed when assigning priorities to threads, even if we are using a prioritized preemptive scheduling with time slicing. Let us consider this example.

Filename: `Core/Src/main-ex2.c`

```
21 #include "nucleo_hal_bsp.h"
22 #include "cmsis_os.h"
23 #include <string.h>
24
25 /* Private variables ---------------------------------------------------------*/
26 extern UART_HandleTypeDef huart2;
27
28 /* Private function prototypes -----------------------------------------------*/
29 void blinkThread(void *argument);
30 void UARTThread(void  *argument);
31
32 /* Definitions for blinkThread and UARTThread */
33 osThreadId_t blinkThreadID;
34 const osThreadAttr_t blinkThread_attr = {
35   .name = "blinkThread",
36   .stack_size = 128 * 4, /* In bytes */
37   .priority = (osPriority_t) osPriorityNormal,
38 };
39
40 osThreadId_t UARTThreadID;
41 const osThreadAttr_t UARTThread_attr = {
42   .name = "UARTThread",
43   .stack_size = 128 * 4, /* In bytes */
44   .priority = (osPriority_t) osPriorityAboveNormal,
45 };
46
47 int main(void) {
48   HAL_Init();
49
50   Nucleo_BSP_Init();
51
52   /* Init scheduler */
53   osKernelInitialize();
54
55   /* Creation of blinkThread */
56   blinkThreadID = osThreadNew(blinkThread, NULL, &blinkThread_attr);
57   /* Creation of UARTThread */
58   UARTThreadID = osThreadNew(UARTThread, NULL, &UARTThread_attr);
59
60   /* Start scheduler */
61   osKernelStart();
62
63   /* We should never get here as control is now taken by the scheduler */
64   while (1);
65 }
66
```

```
67  void blinkThread(void *argument) {
68    while(1) {
69      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
70      osDelay(500);
71    }
72  }
73
74  void UARTThread(void *argument) {
75    while(1) {
76      HAL_UART_Transmit(&huart2, (uint8_t *)"UARTThread\r\n",
77                        strlen("UARTThread\r\n"), HAL_MAX_DELAY);
78    }
79  }
```

This time we have two threads, one that blinks the LD2 LED and one that constantly prints on the UART2 a message. The `UARTThread()` is created with a priority higher than the `blinkThread()` one. Running this example, you can see that the LD2 LED never blinks. This happens because `UARTThread()` is designed to continuously do something and when its *slice time* expires, it is still in *ready* state and, having a higher priority, it is rescheduled for execution. This clearly proves that priorities must be used carefully to prevent other processes from *starving*[21].

23.3.3 Voluntary Release of the Control

A *running* thread can release the control (it is said to *yield* the control), if the programmer knows that it is useless to consume CPU cycles, by calling the function

```
osStatus_t osThreadYield(void);
```

This causes a *context switch*, and the next *ready* thread in the scheduling list is placed in *running* state. The `osThreadYield()` has a relevant role if the *cooperative scheduling* is the scheduler policy.

23.3.4 The *idle* Thread

A CPU never stops, unless we enter one of the low-power modes offered by STM32 microcontrollers. This means that, if all threads in a system are *blocked* or *suspended* waiting for external events, then we need a way to "do something" while waiting for other threads becoming active again. For this reason, all Operative Systems provide a special task named *idle*, which is scheduled during system inactive states, and its priority is defined as the lowest possible. For this reason, it is common to say that the lowest priority corresponds to the *idle priority*.

[21] In concurrent programming, the *starvation* happens when a thread is perpetually denied necessary resources to process its work. *Starvation* usually, it is caused by a bad synchronization among threads, but even by a wrong priority allocation scheme. The *starvation* is an unwanted condition that no programmer would want never to reach, and sometimes to identify its origin can be a nightmare.

In FreeRTOS versions prior to version 9, whenever a thread was deleted, the memory allocated by FreeRTOS to the thread was freed by the *idle* thread. In FreeRTOS version 10, if one thread deletes another thread, then the memory allocated by FreeRTOS to the deleted thread is freed immediately. However, if a thread deletes itself, then the memory allocated by FreeRTOS to the thread is freed by the *idle* thread. Note that, in all cases, it is only the stack and *Thread Control Block* (TCB) allocated to the thread by the RTOS that get freed automatically. Any dynamic buffer allocated by a thread need to be properly managed by the thread itself.

The *idle* thread also plays an important role in *low-power* designs, as we will discover later in this chapter.



A Word About Concurrent Programming

You will be astonished by fantastic numbers presented to you by designers of Real Time Operating Systems. They will say to you that their OS is able to fork hundreds of thousands of threads per second, showing stunning *context switch* performances.

Know that, from a practical point of view, this has the same utility of pub talks.



Figure 23.8: What usually happens when the number of thread increases too much

In the past, I reviewed projects sent to me from readers of this book (but sometimes I have seen projects, having the same bad approach, made by professionals - whether you believe it or not) where you can see tens of threads spawn around in the code that do nothing relevant. Sometimes you can also find threads that do nothing more than forking another thread after a comparison.

Theorists of concurrent programming will teach you that the more concurrent streams you have the more issue you will probably have. Governing threads may be hard, and often the cost involved in synchronizing them overtakes the advantage in using them. Moreover, the same operation of spawning a new thread has a non-negligible cost. And the same applies to the *context switch*.

Multithreaded programming must always be handled with care, especially on embedded systems, where the SRAM is often limited. Remember: **keep it simple**.

23.4 Memory Allocation and Management

In the two previous examples we started using FreeRTOS without dealing too much with the memory allocation for threads and the other structures used by the OS. The only exception is represented by the `osThreadAttr_t.stack_size` attribute that is used by the `osThreadNew()` routine to allocate memory for the thread's stack. FreeRTOS, however, not only needs sufficient memory for threads allocation, but it also uses additional SRAM portions for the allocation of its internal structures (list of TCBs, and so on). The same applies to other synchronization primitives we will study later, such as semaphores and mutexes. Where is this memory exactly taken from?

Traditionally, FreeRTOS implemented a dynamic allocation model until the 8.x release. This constituted an important limitation, because in some application domains the dynamic memory allocation is strongly discouraged or even expressly prohibited. Even if, as we will see soon, one of five dynamic allocators implemented by FreeRTOS answers to the majority of requirements about memory allocation in these application domains, unfortunately this FreeRTOS characteristic prevented its usage when this limitation applies. Starting from the latest 9.x release, FreeRTOS implements two memory allocation models: a full static and a dynamic one.

Two macros are used to enable the memory allocation model: `configSUPPORT_STATIC_ALLOCATION` and `configSUPPORT_DYNAMIC_ALLOCATION`. Both can assume the values `0` or `1` to disable/enable the corresponding memory model. It is important to underline that the two memory models are not mutually exclusive: it is possible to use both simultaneously according to the user need. As we will see later, the two memory models force the usage of separated APIs.

23.4.1 Dynamic Memory Allocation Model

FreeRTOS implements a dynamic memory allocation model, which uses regions of the SRAM to allocate all OS internal structures, including TCBs. When compared to static allocation model, the dynamic one has some non-negligible advantages:

- The memory allocation occurs automatically, within the RTOS API functions.
- Developers do not need to concern with allocating memory themselves.
- The RAM used by an RTOS object can be re-used if the object is deleted, potentially reducing the application's maximum RAM footprint.
- RTOS API functions are provided to return information on heap usage, allowing the heap size to be optimized.
- FreeRTOS provides five dynamic memory allocation schemes, and they can be chosen to best satisfy the application requirements.
- Fewer function parameters are required when an object is created.

FreeRTOS does not make use of the classical `malloc()` and `free()` functions provided by the C *run-time* library[22], because:

[22]With one notable exception represented by the `heap_3.c` allocator, as we will see soon.

1. they use a lot of code space, increasing the size of the firmware;
2. they are not designed to be thread safe (more about this later);
3. they are not deterministic (the amount of time taken to execute the function will differ from call to call).

So, FreeRTOS provides its own dynamic allocation scheme to handle the memory it needs, but since there are several ways to do it, each one with its benefits and tradeoffs, FreeRTOS is designed so that this part is abstracted from the rest of the core OS, and it provides five different allocation schemes the user can choose from, according to his specific needs. The `pvPortMalloc()` and `vPortFree()` are the most important functions implemented in each scheme, and their name clearly says what they do.

This five schemes are not part of the FreeRTOS core, but they are part of the *port layer*, and they are implemented inside five C source files, named `heap_1.c`..`heap_5.c`, contained inside the **portable/MemMang** folder. By compiling one of these files together with the rest of FreeRTOS code, we automatically choose that allocation scheme for our application. Moreover, we can eventually provide our allocation model by implementing this API layer (we essentially need to implement 5 functions, in the worst case) according to our specific needs. CubeMX allows easily to select the wanted memory scheme by choosing the corresponding setting, as shown in **Figure 23.9**.

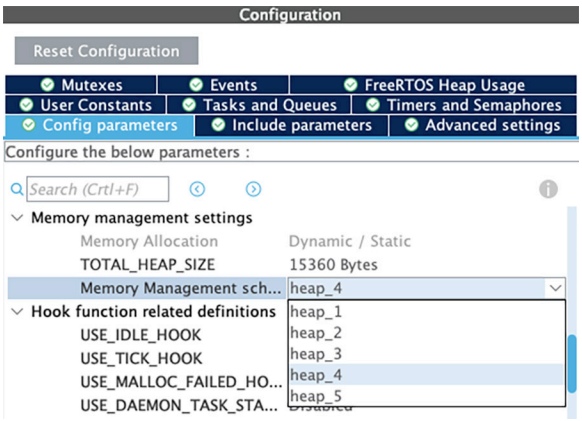


Figure 23.9: How CubeMX allows to select the wanted FreeRTOS memory scheme

23.4.1.1 `heap_1.c`

A lot of embedded applications use an RTOS to logically divide the firmware in blocks. Each block has its own features, and often it runs independently from other blocks. For example, suppose that you are developing a device with a TFT display (maybe the controller of a modern washing machine). Usually, the firmware is partitioned in few threads, where one is the responsible of the graphical interaction (it updates the display by printing information and showing stunning graphical widgets) and other threads are responsible of managing the washing program (and so the handling of sensors, motors, pumps and so on). These applications usually have a `main()` that spawns the threads (as we have done in the past examples), and almost nothing more is initialized by the OS once it starts

spinning. This means that the memory allocator does not have to consider any of the more complex allocation issues, such as determinism and fragmentation, and it can be simplified.

`heap_1.c` allocator implements a very basic version of the `pvPortMalloc()`, and does not provide `vPortFree()`. Applications that never delete a thread, or other kernel objects like queues, semaphores, etc, are suitable to use this memory allocation scheme. Those application domains, where the use of dynamically allocated memory is discouraged, may benefit from this allocation scheme, since it offers a deterministic approach to the memory management, avoiding fragmentation (because the memory is never deallocated).

`heap_1.c` allocator subdivides a statically allocated array in small chunks as calls to `pvPortMalloc()` are made. This is indeed the FreeRTOS heap. The total size of this array (expressed in bytes) is defined by the macro `configTOTAL_HEAP_SIZE` in the `FreeRTOSConfig.h` file. The only tradeoff with this allocation scheme is that, being the whole array allocated at compile time, the application will consume al lot of SRAM even if it does not entirely use it. This means that programmers must carefully choose the right value for `configTOTAL_HEAP_SIZE` size.



It is fundamental to remark an important thing. The memory of C programs is traditionally partitioned in two relevant regions: *stack* and *heap*. The heap is said to grow dynamically at runtime, and it grows in the opposite direction of the stack. As you can see, however, `heap_1.c` allocator has nothing related to heap of the whole application, since it uses an uninitialized array declared as `static`, which is allocated in `.bss` section as we have learned in Chapter 13, to store the objects it needs dynamically. It is a form of dynamic allocation for sure, but not connected with the use of `malloc()` and `free()` functions. This means that we can safely use them in our application, even if their usage is not encouraged in embedded applications.

23.4.1.2 heap_2.c

`heap_2.c` also works by subdividing a statically allocated array, which is dimensioned by the `configTOTAL_HEAP_SIZE` macro. It uses a best-fit algorithm to allocate the memory and, unlike the `Heap_1.c` allocation scheme, it allows memory to be freed. This algorithm is considered deprecated and not suitable for new designs. The `Heap_4.c` is the better alternative to this allocator. For this reason, we will not go into details of how it works. If interested, you can consult the official FreeRTOS documentation[23].

23.4.1.3 heap_3.c

`heap_3.c` uses the conventional C `malloc()` and `free()` functions to perform memory allocation. This means that the `configTOTAL_HEAP_SIZE` parameter has no effects on the memory management, since the `malloc()` is designed to manage the heap by itself. This means that we need to configure our linker script accordingly, as shown in Chapter 13. Moreover, consider that the `malloc()`

[23] https://bit.ly/1PMSPRM

implementation changes from the one provided by the `newlib-nano` and the regular `newlib`. However, the more versatile implementation provided by the `newlib` library requires a lot of more flash space.

`heap_3.c` makes `malloc()` and `free()` thread-safe by temporarily suspending FreeRTOS scheduler. We will deepen the usage of `malloc()` in conjunction with FreeRTOS later in this chapter.

23.4.1.4 `heap_4.c`

`heap_4.c` is the allocator selected by default by CubeMX and it works in the same way of `heap_1.c` and `heap_2.c`. That is, it uses a statically allocated array, dimensioned by the value of the `configTOTAL_HEAP_SIZE` macro, to store the objects allocated at run-time. However, it has a different approach during the allocation of memory. In fact, it uses a *first fit* algorithm, which combines adjacent free blocks into a single large block, reducing the risk of memory fragmentation. This technique, commonly used by the garbage collector in languages with dynamic and automatic memory allocation, is also called as *coalescing*.

Unfortunately, this behaviour of the `heap_4.c` allocator causes that it is non-deterministic: the allocation/deallocation of many small objects, together with the creation/destroy of threads, could cause a lot of fragmentation, which requires more computing processing to pack the memory. Moreover, there is no guarantee that the algorithm avoids memory leaks at all. However, it is usually faster than the most standard implementation of `malloc()` and `free()`, especially the ones provided by the `newlib-nano` lib.

Explaining in detail the `heap_4.c` algorithm is outside the scope of this book. For more information refer to the FreeRTOS documentation[24].

23.4.1.5 `heap_5.c`

`heap_5.c` uses the same algorithm of the `heap_4.c` allocator, but it allows to split the memory pool among different non-contiguous memory regions. It is especially useful for STM32 MCUs providing the FSMC controller, which allows to transparently use external SDRAMs to increase the whole RAM. Programmers may decide to allocate some heavy used thread in the internal SRAM memory (or the CCM memory, if available) and then use the external SDRAM for less relevant objects like semaphores and mutexes.

By defining a custom linker script, it is possible to allocate two pools in two memory regions, and then use the `vPortDefineHeapRegions()` function from FreeRTOS to define them as memory pools. However, this is an advanced usage of the OS that we will not detail here. If interested, you can refer to the excellent book *Mastering the FreeRTOS Real Time Kernel*[25] by Richard Barry, creator of FreeRTOS.

[24] https://bit.ly/1TqxX9S
[25] https://bit.ly/3A1avK9

23.4.1.6 FreeRTOS Heap Definition

Starting from FreeRTOS 9.x it is also possible to control the definition of the heap in addition to its size. By setting the `configAPPLICATION_ALLOCATED_HEAP` macro to `1` we can declare the FreeRTOS heap, and we can decide to place it in specific memory regions (for example, in a faster memory like the CCM one) with a custom linker script. When configuring the `configAPPLICATION_ALLOCATED_HEAP` to `1` we must provide a `uint8_t` array with the exact name and dimension as shown below.

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

23.4.2 Static Memory Allocation Model

Starting from FreeRTOS 9.x, it is possible to enable a full-static memory allocation model. This means that we are totally responsible of the correct allocation of memory pools needed by the OS to carry out its activities. Statically allocated RAM provides developers some important advantages:

- OS structures can be placed at specific memory locations. This constitutes an important advantage for those STM32 microcontrollers having CCM memory or other cached SRAM memories.
- The maximum RAM footprint can be determined at link time, rather than at run-time.
- Developers do not need to concern themselves with graceful handling of memory allocation failures.
- It allows the OS to be used in applications that simply do not allow any dynamic memory allocation (although FreeRTOS includes allocation schemes that can overcome most objections, as we will see later).

The static memory allocation model is enabled by setting the `configSUPPORT_STATIC_ALLOCATION` macro to `1` and it affects all the APIs used to define FreeRTOS objects. Both `configSUPPORT_DYNAMIC_ALLOCATION` and `configSUPPORT_STATIC_ALLOCATION` macro can be set to `1`, allowing to mix dynamic and static allocation. When using static allocation, we must provide to FreeRTOS the memory areas where to store both the stack and the TCB objects by pre-allocating them. This can be performed by setting the `osThreadAttr_t struct` accordingly:

```
...
uint32_t threadStack[128];
StaticTask_t threadTCB;
const osThreadAttr_t thread_attr = {
  .name = "Thread",
  .priority = (osPriority_t) osPriorityNormal,
  .stack_size = 128 * 4, /* In bytes */
  .stack_mem = threadStack,
  .cb_mem = &threadTCB,
  .cb_size = sizeof(StaticTask_t)
};
```



Read Carefully

Take note that FreeRTOS allows to selectively allocate thread's TCB and stack both statically and dynamically. For example, it is perfectly possible to allocate the stack statically and letting FreeRTOS to allocate the TCB dynamically. Instead, the current implementation of the `osThreadNew()` made by ST works so that both stack and TCB must be allocated statically simultaneously (and so passed their references with the `osThreadAttr_t struct`), otherwise the thread will be automatically allocated dynamically. This is a limitation that have to be considered when deciding the allocation of a thread.

23.4.2.1 *idle* Thread Allocation with Static Memory Allocation Model

With the dynamic allocation, FreeRTOS completely takes care of the memory allocation for the *idle* thread (including its stack and TCB). Instead, when using static allocation, we are responsible of the proper memory allocation for the *idle* thread like any other thread.

Being the *idle* thread automatically allocated by the kernel during its initialization, FreeRTOS provides a generic way to supply the necessary memory room for the *idle* thread. The function:

```
void vApplicationGetIdleTaskMemory(StaticTask_t **ppxIdleTaskTCBBuffer, StackType_t **ppxIdleT\
askStackBuffer, uint32_t *pulIdleTaskStackSize);
```

is invoked by FreeRTOS before starting the *idle* thread. That routine must be used to allocate the *idle* TCB and stack and to pass the reference to those memory areas to the kernel. When using CubeMX to generate a project with FreeRTOS, and the static memory allocation model is chosen, the file **Middlewares/Third_Party/FreeRTOS/Source/CMSIS_RTOS_V2/cmsis_os2.c** already contains an implementation for the `vApplicationGetIdleTaskMemory()` function (look at the end of the file).

23.4.3 FreeRTOS and the C stdlib

As programmers we are used to taking some things for granted. And this statement is becoming more and more true day-by-day due to the increasing complexity of development environments.

But, as embedded developers, we must be sure that the usage of functions like `sprintf()` - or even an "innocent" `dtoa()` - cannot hurt a worker operating to a dangerous machine or stalling the autopilot system of a modern Boeing 737.

For this reason, we have to take same precaution before using FreeRTOS (or any other concurrent system - including simple ISRs) with the C stdlib.

23.4.3.1 How to Configure `newlib` to Handle Concurrency with FreeRTOS

`newlib`, and the reduced (compact) version `newlib-nano`, is the C run-time library adopted by STM for its GCC-based development environment. It is important to clarify that `newlib` it is not the only possible option around, but since STM offers `newlib` as the sole alternative will focus just on it.

Every time we use a library (or any given piece of code) in a multi-threaded environment we must ask our-self if that library is reentrant. Reentrancy is a characteristic of functions which allows multiple execution flows to use the same function with assurance that the values stored in those execution contexts will remain constant between calls. Let us consider this simple example:

```
#define ENOERR     0
#define EDIVBYZERO 1
int errno;

int divide(int x, int y) {
    if (y != 0) {
        errno = ENOERR;
        return x/y;
    } else {
        errno = EDIVBYZERO;
        return 0;
    }
}

void funcA(void *argument) {
    int x, y, ret;
    //Long manipulation of x and y
    //Here, both x and y are !=0
    ret = divide(x,y);
    ...
}

void funcB(void *argument) {
    int x, y, ret;
    //Long manipulation of x and y
    //Here, y is == 0
    ret = divide(x, 0);
    if(errno == ENOERR && ret == 0) {
        /* In a multithreaded environment this is possible */
```

```
30          ...
31  }
32  ....
```

Now, let us analyze the `divide()` function storage, considering the diagram in **Figure 23.10**. The function makes use of three memory locations: two variables will be stored on the function's stack (`x` and `y`), while the `errno` one will be stored in the `.bss` section (if this matter sounds new to you, please go back to Chapter 20).
Let us assume that the `divide()` function is called by two different routines running in two separated threads with the same priority. Since every thread has its own TCB and so its stack, the access to local variables is not an issue. Instead, the access to the `errno` global variable is subject to race conditions. The thread running `funcB()` could be preempted just before the `return` instruction at line 11, and the control could be passed to the thread running `funcA()`, which will set the `errno` variable to `ENOERR`: in this case, when the control will go back again to `funcB()`, the result will be `0` and the caller (that is the `funcB()`) will assume that the result of division is zero (we are in the `integer` space, here).

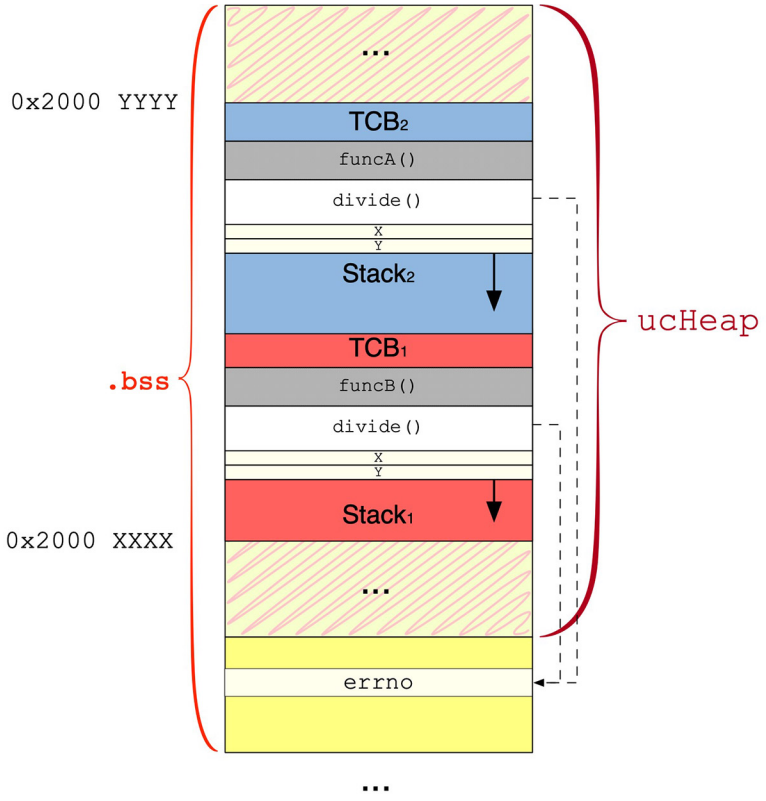


Figure 23.10: The memory layout of the `divide()` function in a multi-threaded environment

The choice of citing the `errno` variable is not accidental here. Unfortunately, even if the majority of the `newlib` functions are designed to be reentrant, there exists a given amount of C stdlib functions that - at least in theory - cannot be reentrant at all. This happens mostly for the global `errno` variable but also for other global or statically allocated variables.

To address this issue, the `newlib` is designed so that all globally defined variables are packed in a

struct called `struct _reent`. This struct contains a relevant list of fields[26] (including `errno` but even other unsuspectable fields like pointers to `stdin/stdout/stderr`) and there is a global instance of this `struct _reent` for the whole application. For all non-reentrant functions[27] there is a corresponding wrapper function (prefixed with _ and ending with `_r` - for example, `_dtoa_r()` to wrap the `dtoa()`) which are designed to access this globally allocated `_reent struct` through a global pointer named "`_impure_ptr`" (just wondering the name where comes from...). FreeRTOS will change the reference of the `_inpure_ptr` at every context switch, updating the reference to an instance of the `_reent struct` inside the TCB (see **Figure 23.11**). Clearly, this will cause that the TCB dimension will grow-up a lot: around 100 bytes depending on the compile options (this size varies a lot if using `newlib` or `newlib-nano`).

By setting to `1` the global FreeRTOS macro `configUSE_NEWLIB_REENTRANT` inside the **Core/Inc/FreeRTOSConfig.h** file, FreeRTOS will allocate a `struct _reent` in every TCB, so that the `_impure_ptr` will point to a "global memory area" inside the current TCB's stack. This will guarantee the reentrancy of the `newlib`: however, keep in mind that reentrancy is not a synonymous for "thread safe". As we are going to see in the next paragraph.

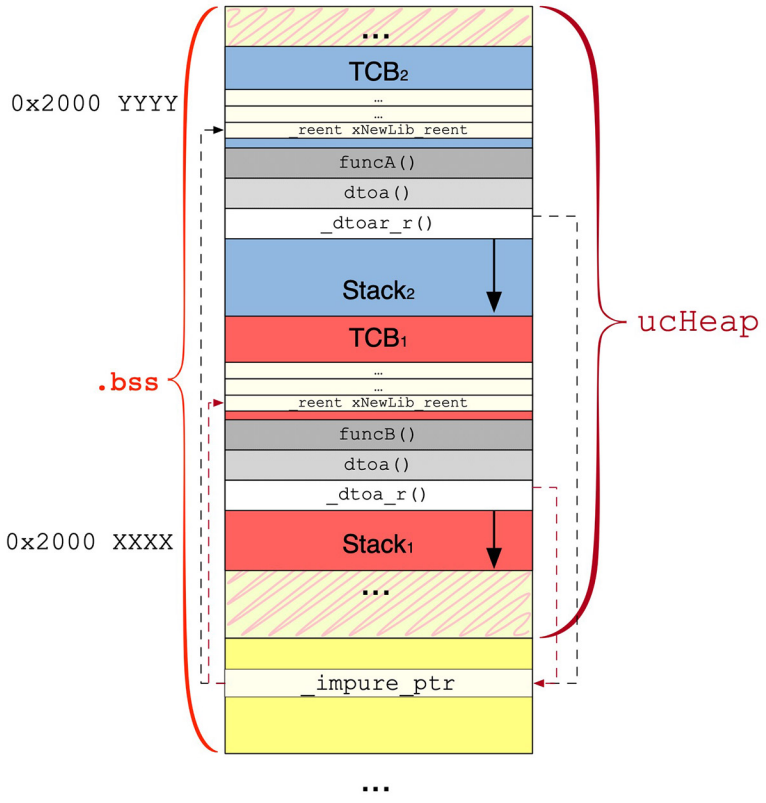


Figure 23.11: How FreeRTOS handles the access to the `_reent struct` by individual threads

[26] You can give a look to the `_reent struct` here.
[27] A complete list of non-reentrant functions is available here.

23.4.3.2 How to Use `malloc()` and `malloc()`-dependant `newlib` Functions With FreeRTOS

As said before, except for the `heap_3.c` allocation scheme, FreeRTOS does not make use of the stdlib C heap memory to allocate threads and other objects. But what we have to do if one of the libraries we are going to use in our project uses `malloc()` and `free()`?

This question cannot be easily ignored and needs a careful check of the whole firmware. And this for one simple reason: the same `newlib` library has some functions that use `malloc()` to allocate memory dynamically. For example, `printf()` and `sprintf()` are two stdlib functions that dynamically allocate buffers at needs. But they are not the only ones. Dave Nadler extensively wrote[28] about this topic even if, according to this author, he mixes two separated aspects (reentrancy and memory allocation policies) causing that it is quite easy to get lost. Moreover, the information regarding STM implementation of the `_sbrk()` routine are no longer updated, since STM fixed the way maximum stack boundaries are computed.

To easily understand how to handle this topic, we have to take a step back to the Chapter 20, when we discussed about the memory layout of an STM32 application made with the STM32CubeIDE toolchain. The **Figure 23.12** shows the *out-of-the-box*[29] SRAM memory layout of an STM32 application running FreeRTOS. Let us start from the SRAM ORIGIN address (this matter should be familiar to you, but it is better too much than not enough).

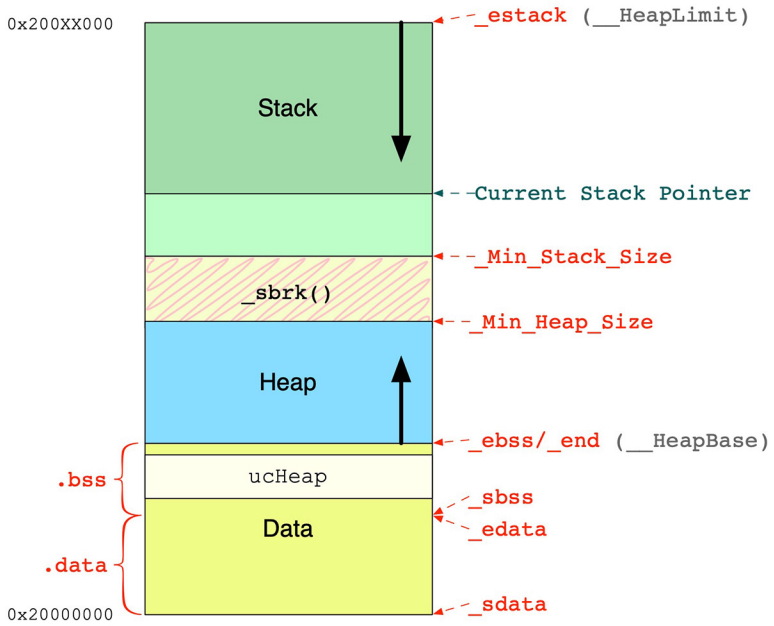


Figure 23.12: The out-of-the-box SRAM memory layout of an STM32CubeIDE application running FreeRTOS

- `_sdata` is the linker symbol pointing to the SRAM ORIGIN address (that is `0x2000 0000`) and `_edata` corresponds to the end of the initialized data section.

[28] https://nadler.com/embedded/newlibAndFreeRTOS.html

[29] The term *out-of-the-box* means the memory layout of a standard application as generated by CubeMX in the STM32CubeIDE 1.8.0. Things may change if you have a newer version of the IDE. So, please check before taking the information in this paragraph as pure gold.

- `_sbss` points to the start of the uninitialized data while `_ebss` (or the linker symbol `_end`) points to the end of this region, which also contains the FreeRTOS heap if using `heap_4.c` scheme. `_ebss` corresponds to the begininning of the C Heap region, if using the `_sbrk()` implementation in **Core/Src/sysmem.c**.
- `_Min_Heap_Size` is the "logical" limit of the Heap region. This limit is established by the programmer (by default, CubeMX sets this to `0x200`). and it should be carefully checked to avoid any corruption of the Stack region, which grows in the opposite direction.
- `_Min_Stack_Size` is the logical limit of the Stack region. Again, this limit is established by the programmer (by default, CubeMX sets this to `0x400`).
- `Current Stack Pointer` is the content of the Cortex-M core register `sp`, and it corresponds to the current base stack pointer.
- `_estack` by default represent the end of the SRAM memory and it is the first location of the stack that contains the MSP.

To safely use the `malloc()`/`free()` routines in our application, we have the following options.

OPTION 1: Not Use Them at All
Well, this may seem drastic, but it is an option to consider. If your code, and all dependant libraries, do not use `malloc()`/`free()`, then it is ok to do not care about them. However, never trust others' code: you have to carefully check it at run-time, and not by simply looking at the code or the documentation. You can easily perform the test by instructing the linker accordingly. LD allows to "wrap" a routine by using the command-line option `-Xlinker --wrap` (which can be set using Project properties as shown in **Figure 23.13**). Dave Nadler provides[30] an excellent implementation of the wrapper to keep track of `malloc()` usage, which is reported below. You can place a breakpoint to see if the wrappers are called or you can inspect the value of `MallocCallCnt` variable (you could print it by using the ITM, if supported by the MCU).

```
size_t TotalMallocdBytes;
int MallocCallCnt;
static bool inside_malloc;

void *__wrap_malloc(size_t nbytes) {
    extern void * __real_malloc(size_t nbytes);
    MallocCallCnt++;
    TotalMallocdBytes += nbytes;
    inside_malloc = true;
    void *p = __real_malloc(nbytes); // will call malloc_r...
    inside_malloc = false;
    return p;
};
void *__wrap__malloc_r(void *reent, size_t nbytes) {
    (void)(reent);
    extern void * __real__malloc_r(size_t nbytes);
```

[30] https://nadler.com/embedded/newlibAndFreeRTOS.html

```
    if(!inside_malloc) {
      MallocCallCnt++;
      TotalMallocdBytes += nbytes;
    };
    void *p = __real__malloc_r(nbytes);
    return p;
};
```

Pro:

- Simplified memory layout.
- Reduced risk of memory overflow.
- Reduced risk of memory fragmentation (especially if using `newlib-nano`).

Cons:

- It could force you from re-inventing the wheel implementing existing functions provided by `newlib` and other libraries.
- You may be forced to study code of other people.

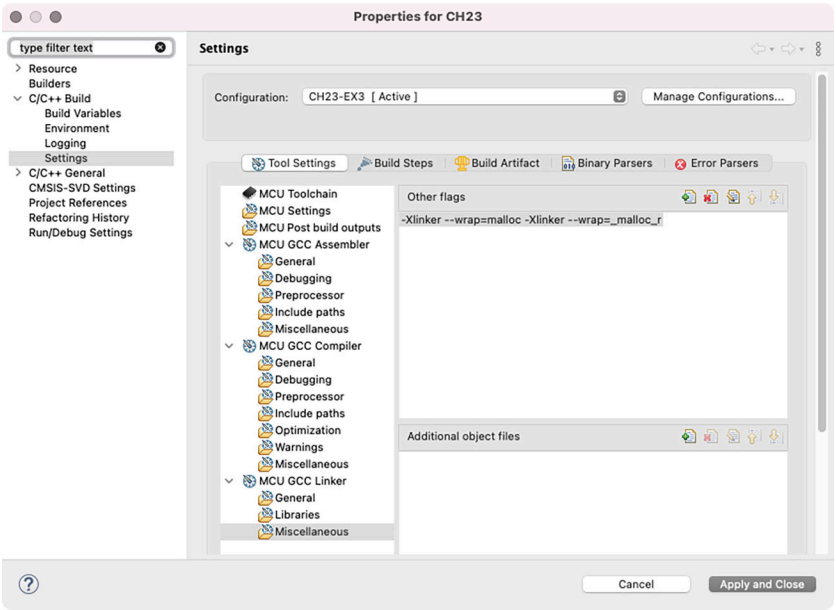


Figure 23.13:

OPTION 2: Letting FreeRTOS to Handle the Dynamic Memory Allocation

This is another drastic solution but that could work well if you know what you are doing. Both the `malloc()` and `free()` could be easily rewritten so that they make use of FreeRTOS allocation routines:

```
void *malloc (size_t size) {
    return pvPortMalloc(size);
}
void free (void *ptr) {
    vPortFree(ptr);
}
```

This works because, in `newlib` both the functions are declared as `__weak`. With this option it is important to size both `_Min_Stack_Size` and the dimensions of the global FreeRTOS heap `ucHeap` so that no SRAM space is wasted (just to clarify, the area named `_sbrk()` in **Figure 23.12** should be reduced at minimum). I strongly suggest to place the `ucHeap` inside a dedicated section right after the `.bss` one.

Pro:

- Simplified memory layout.
- Freedom of choosing the best FreeRTOS dynamic allocator.

Cons:

- You may be forced to study code of other people.

OPTION 3: Letting `newlib` to Handle the Dynamic Memory Allocation
This solution is the exact opposite of the previous one, and it is the solution proposed by Dave Nadler. Dave provides a new memory allocator[31], which is an advanced version of the standard `heap_3.c` provided with FreeRTOS. The file `heap_useNewlib_ST.c` contains a lot of code, which may be confusing for novice users. The file does the following things:

- First, it properly defines the boundaries of stack and heap so that the area to dedicate to `_sbrk()` is used at best.
- Next, it implements a proper error management scheme, just in case we run out of free memory.
- Finally (and this is the most relevant thing) it properly handles the concurrent access to `malloc()`/`free()` routines.

[31] https://bit.ly/3nIOtqC

Filename: `Middlewares/Third_Party/FreeRTOS/Source/portable/MemMang/heap_useNewlib_ST.c`

```
extern char end, _estack, _Min_Stack_Size;  // symbols from linker LD command file

#ifdef MALLOCS_INSIDE_ISRs
    // If we plan to use malloc() in a ISR context - not that good practice
    #define DRN_ENTER_CRITICAL_SECTION(_usis) { _usis = taskENTER_CRITICAL_FROM_ISR(); }
    #define DRN_EXIT_CRITICAL_SECTION(_usis)  { taskEXIT_CRITICAL_FROM_ISR(_usis);     }
#else
    #define DRN_ENTER_CRITICAL_SECTION(_usis) vTaskSuspendAll();
    #define DRN_EXIT_CRITICAL_SECTION(_usis)  xTaskResumeAll();
#endif

//! _sbrk_r version supporting reentrant newlib
void * _sbrk_r(struct _reent *pReent, int incr) {
    #ifdef MALLOCS_INSIDE_ISRs //block interrupts during free-storage use
      UBaseType_t usis; //saved interrupt status
    #endif
    static char *currentHeapEnd = &end;
    if (currentHeapEnd + incr > &_Min_Stack_Size) {
        // Ooops, no more memory available...
            vApplicationMallocFailedHook();
            pReent->_errno = ENOMEM; // newlib's thread-specific errno
        return (char *)-1; // the malloc-family routine that called sbrk will return 0
    }
    // 'incr' of memory is available: update accounting and return it.
    char *previousHeapEnd = currentHeapEnd;
    currentHeapEnd += incr;
    heapBytesRemaining -= incr;
    return (char *) previousHeapEnd;
}

//! non-reentrant sbrk uses is actually reentrant by using current context
// ... because the current _reent structure is pointed to by global _impure_ptr
char * sbrk(int incr) { return _sbrk_r(_impure_ptr, incr); }
//! _sbrk is a synonym for sbrk.
char * _sbrk(int incr) { return sbrk(incr); };

#ifdef MALLOCS_INSIDE_ISRs // block interrupts during free-storage use
  static UBaseType_t malLock_uxSavedInterruptStatus;
#endif
void __malloc_lock(struct _reent *r)   {
  (void)(r);
  #if defined(MALLOCS_INSIDE_ISRs)
    DRN_ENTER_CRITICAL_SECTION(malLock_uxSavedInterruptStatus);
  #else
    bool insideAnISR = xPortIsInsideInterrupt();
    configASSERT( !insideAnISR ); // Make damn sure no more mallocs inside ISRs!!
```

```
47      vTaskSuspendAll();
48    #endif
49  };
50  void __malloc_unlock(struct _reent *r) {
51    (void)(r);
52    #if defined(MALLOCS_INSIDE_ISRs)
53      DRN_EXIT_CRITICAL_SECTION(malLock_uxSavedInterruptStatus);
54    #else
55      (void)xTaskResumeAll();
56    #endif
57  };
58
59  // newlib also requires implementing locks for the application's environment memory space,
60  // accessed by newlib's setenv() and getenv() functions.
61  // As these are trivial functions, momentarily suspend task switching (rather than semaphore).
62  // Not required (and trimmed by linker) in applications not using environment variables.
63  // ToDo: Move __env_lock/unlock to a separate newlib helper file.
64  void __env_lock()    {       vTaskSuspendAll(); };
65  void __env_unlock()  { (void)xTaskResumeAll();  };
66
67  // ===============================================================
68  // Implement FreeRTOS's memory API using newlib-provided malloc family.
69  // ===============================================================
70
71  void *pvPortMalloc( size_t xSize ) PRIVILEGED_FUNCTION {
72      void *p = malloc(xSize);
73      return p;
74  }
75  void vPortFree( void *pv ) PRIVILEGED_FUNCTION {
76      free(pv);
77  };
```

Lines [3:10] define two macros to suspend/resume context switching so that we avoid any race condition during the allocation of new memory in the heap. Here we have two ways to perform this operation. If we do not plan to use `malloc()` in an ISR context (which should be the correct way in general), then it is ok to use FreeRTOS `vTaskSuspendAll()`/`xTaskResumeAll()` routines. Otherwise, we have to use the macros `taskENTER_CRITICAL_FROM_ISR()`/`taskEXIT_CRITICAL_FROM_ISR()`[32]. Lines [13:29] define the `_sbrk_r()`: as you can see, there is no difference from the current `_sbrk()` implementation provided by STM in the tool-chain. Finally, lines [40:57] defines the `__malloc_lock()`/`__malloc_unlock()` routines: these hook functions are called automatically by the `newlib` library to avoid race conditions during memory allocation.

Pro:

[32] Dave provides the possibility to use `malloc()` in an ISR context because some ST libraries (notably the USB Device Stack) use `malloc()` in code called in an ISR. The reason why this happens is because, as we will see in the chapter dedicated to USB peripheral, the majority of USB operations are handled by the stack in a context of a ISR.

- Simplified memory layout.
- The only available option if using FreeRTOS with static memory allocation (but if so, why dealing with `malloc()`?)

Cons:

- If your code spawns and kills many tasks during the firmware lifecycle, then it is best to let FreeRTOS to handle memory allocation.

OPTION 4: Mixing `newlib` and FreeRTOS to Handle Dynamic Memory Allocation
This is the "default" CubeMX solution, that needs to be tuned-up to avoid issues. FreeRTOS will have its memory space defined by the global macro `configTOTAL_HEAP_SIZE` and `newlib malloc()` will allocate the dynamic memory inside the free space ranging from `_Min_Stack_Size` and `_ebss`. The STM's `_sbrk()` implementation is not that different from the one provided by Dave Nadler, and we do not need to change it. Instead, we need to ensure thread safety of `malloc()` routine by providing `__malloc_lock()`/`__malloc_unlock()` hooks. The following one is a possible and more complete implementation for the `_sbrk()` routine.



Read Carefully

Owners of Cortex-M0/0+ based MCUs will find a different implementation of the `__malloc_lock()` routine. The function `xPortIsInsideInterrupt()` is not defined in the Cortex-M0/0+ ports. In this case, the function `__get_IPSR()` is used to detect if we are running in the context of an ISR.

Filename: `Core/Src/sysmem.c`

```
30 #ifdef MALLOCS_INSIDE_ISRs
31     // If we plan to use malloc() in a ISR context - not that good practice
32     UBaseType_t usis; //saved interrupt status
33     #define DRN_ENTER_CRITICAL_SECTION(_usis) { _usis = taskENTER_CRITICAL_FROM_ISR(); }
34     #define DRN_EXIT_CRITICAL_SECTION(_usis)  { taskEXIT_CRITICAL_FROM_ISR(_usis);     }
35 #else
36     #define DRN_ENTER_CRITICAL_SECTION(_usis) vTaskSuspendAll();
37     #define DRN_EXIT_CRITICAL_SECTION(_usis)  xTaskResumeAll();
38 #endif
39
40 /**
41  * Pointer to the current high watermark of the heap usage
42  */
43 void * _sbrk_r(struct _reent *pReent, int incr) {
44   extern uint8_t _end; /* Symbol defined in the linker script */
45   extern uint8_t _estack; /* Symbol defined in the linker script */
46   extern uint32_t _Min_Stack_Size; /* Symbol defined in the linker script */
47   const uint32_t stack_limit = (uint32_t)&_estack - (uint32_t)&_Min_Stack_Size;
```

```
48    const uint8_t *max_heap = (uint8_t *)stack_limit;
49    static uint8_t *__sbrk_heap_end = NULL;
50
51    uint8_t *prev_heap_end;
52
53    /* Initialize heap end at first call */
54    if (NULL == __sbrk_heap_end) {
55      __sbrk_heap_end = &_end;
56    }
57
58    /* Protect heap from growing into the reserved MSP stack */
59    if (__sbrk_heap_end + incr > max_heap) {
60      errno = ENOMEM;
61      return (void *)-1;
62    }
63
64    prev_heap_end = __sbrk_heap_end;
65    __sbrk_heap_end += incr;
66
67    return (void *)prev_heap_end;
68  }
69
70  char * sbrk(int incr) {
71    return _sbrk_r(_impure_ptr, incr);
72  }
73
74  #ifdef MALLOCS_INSIDE_ISRs // block interrupts during free-storage use
75    static UBaseType_t malLock_uxSavedInterruptStatus;
76  #endif
77  void __malloc_lock(struct _reent *r)   {
78    (void)(r);
79    #if defined(MALLOCS_INSIDE_ISRs)
80      DRN_ENTER_CRITICAL_SECTION(malLock_uxSavedInterruptStatus);
81    #else
82      BaseType_t insideAnISR = xPortIsInsideInterrupt();
83      configASSERT( !insideAnISR ); // Make sure no malloc() inside ISRs
84      vTaskSuspendAll();
85    #endif
86  };
87
88  void __malloc_unlock(struct _reent *r) {
89    (void)(r);
90    #if defined(MALLOCS_INSIDE_ISRs)
91      DRN_EXIT_CRITICAL_SECTION(malLock_uxSavedInterruptStatus);
92    #else
93      (void)xTaskResumeAll();
94    #endif
```

```
95 };
```

Pro:

- Everyone makes his job.
- You do not need to be worried too much by the usage of `malloc()` in other libraries.

Cons:

- You need to carefully define memory regions size.

23.4.3.3 STM32CubeMX Approach to Thread-Safety

Starting from STM32CubeIDE 1.7 and STM32CubeMX 6.3, STM addressed the handling of thread-safe access to the regular newlib C functions. In the **Project Manager** section in CubeMX it is possible to enable multi-thread support by choosing among five different locking strategies, as shown in **Figure 23.14**. The strategies are divided into generic strategies and RTOS-dependent strategies.

- Generic strategies:
  - **Strategy #1**: user-defined solution for handling thread safety.
  - **Strategy #2**: allows lock usage from interrupts. This implementation ensures thread safety by disabling all interrupts during, for instance, calls to `malloc()`.
  - **Strategy #3**: denies lock usage from interrupts. This implementation assumes single-thread execution and denies any attempt to take a lock from ISR context.
- FreeRTOS-based strategies:
  - **Strategy #4**: allows lock usage from interrupts. Implemented using FreeRTOS locks. This implementation ensures thread safety by entering RTOS ISR capable critical sections during, for instance, calls to `malloc()`. This implies that thread safety is achieved by disabling low-priority interrupts and task switching. High-priority interrupts are however not safe.
  - **Strategy #5**: denies lock usage from interrupts. Implemented using FreeRTOS locks. This implementation ensures thread safety by suspending all tasks during, for instance, calls to `malloc()`.

For more information about the complete technical solution, refer to the AN5731[33].

[33] https://bit.ly/3B4GGJi

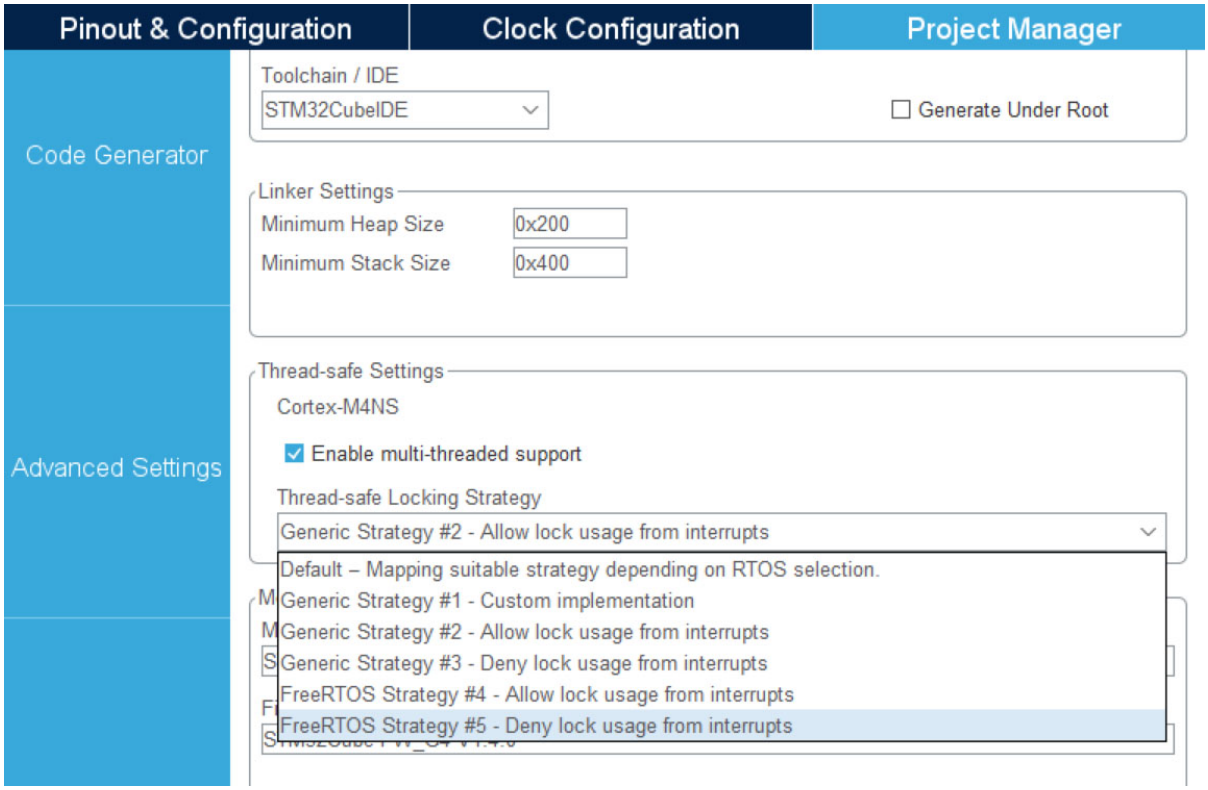


Figure 23.14: How to select Thread-Safe Locking Strategy in CubeMX

23.4.4 Memory Pools

The CMSIS-RTOS2 specification provides the notion of memory pools, and the layer developed by ST on the top of the FreeRTOS OS implements them, even if FreeRTOS does not provide this data structure natively. *Memory pools* are fixed-size blocks (which can have an arbitrary dimension) of dynamic-allocated memory, implemented so that they are thread-safe. This allows them to be accessed from threads and ISRs alike. Memory pools are implemented by ST using the `pvPortMalloc()`/`vPortFree()` routines, and hence the effective memory allocation is demanded to one of the `heap_x.c` allocators. Instead, a counting semaphore is used to discipline the access to the memory blocks in a pool. We will discuss about counting semaphores later. This implies that memory pools are available only if the macro `configUSE_COUNTING_SEMAPHORES` in the **Core/Inc/FreeRTOSConfig.h** file is defined and set to `1`.

A memory pool is created and initialized by using the function:

```
osMemoryPoolId_t osMemoryPoolNew(uint32_t block_count,
                                 uint32_t block_size, const osMemoryPoolAttr_t *attr);
```

where `block_count` defines the number of blocks and `block_size` is the dimension (in bytes) of every block in the pool. The memory pool is defined by the following `struct`:

```
typedef struct {
  const char     *name;      /* Memory pool name */
  uint32_t       attr_bits;  /* Attribute bits: reserved for future use (set to '0') */
  void           *cb_mem;    /* Control block to hold MP's data (default: NULL).
                                Used only for static allocation */
  uint32_t       cb_size;    /* Size of provided memory for control block (default: 0) */
  void           *mp_mem;    /* Pointer to the memory holding all MP's blocks  (default: NULL).
                                Used only for static allocation */
  uint32_t       mp_size;    /* Size of provided memory for blocks storage */
} osMemoryPoolAttr_t;
```

The `attr` parameter of the function `osMemoryPoolNew()` can be set to `NULL`.
The CMSIS-RTOS2 specifications defines the function:

```
void *osMemoryPoolAlloc(osMemoryPoolId_t mp_id, uint32_t timeout);
```

to retrieve a single block of memory from the pool, whose size is equal to the `block_size` parameter passed to the `osMemoryPoolNew()`. If no more space is available in the pool, the function returns `0`. The parameter `timeout` specifies how long the system waits to allocate the memory. According to the CMSIS-RTOS2 specification, while the system waits the thread that calling this function is put into the *blocked* state. The thread will become *ready* as soon as at least one block of memory gets available. The parameter timeout can have the following values:

- when timeout is 0, the function returns instantly (either if the block is available or not);
- when timeout is set to `osWaitForever` the function will wait for an infinite time until the memory is allocated;
- all other values specify a time in kernel ticks for a timeout (for example, a vale equal to `100` corresponds to 100 ticks, which by default corresponds to 100ms).

However, the current STM implementation just handle the case the `timeout` parameter is equal to `0`. In all other cases, the function `osMemoryPoolAlloc()` will return `NULL`.

To free a block in the pool, we use the function:

```
osStatus_t osMemoryPoolFree(osMemoryPoolId_t mp_id, void *block);
```

A pool is destroyed, and all its memory freed, by calling the function:

```
osStatus_t osMemoryPoolDelete        (osMemoryPoolId_t mp_id);
```

The CMSIS-RTOS2 specifications also defines several utility functions to retrieve relevant information about the status of the pool. For example, the function:

```
uint32_t osMemoryPoolGetSpace(osMemoryPoolId_t mp_id);
```

which returns the number of free blocks in a pool. For the complete list of functions to work with memory pools refer to the official CMSIS-RTOS2 specification[34].

The following pseudo-code shows how to easily use memory pools.

```
#include "cmsis_os2.h"                          // CMSIS RTOS header file

#define MEMPOOL_OBJECTS 16                      // Number of Memory Pool Objects

typedef struct {                                // Object data type
  uint8_t Buf[32];
  uint8_t Idx;
} MEM_BLOCK_t;

osMemoryPoolId_t mpid_MemPool;                  // Memory pool id
osThreadId_t tid_Thread_MemPool;                // Thread id

void Thread_MemPool (void *argument);           // Thread function

int Init_MemPool (void) {
  mpid_MemPool = osMemoryPoolNew(MEMPOOL_OBJECTS, sizeof(MEM_BLOCK_t), NULL);
  if (mpid_MemPool == NULL) {
    ; // MemPool object not created, handle failure
  }

  tid_Thread_MemPool = osThreadNew(Thread_MemPool, NULL, NULL);
  if (tid_Thread_MemPool == NULL) {
    return(-1);
  }
  return(0);
}

void Thread_MemPool (void *argument) {
  MEM_BLOCK_t *pMem;
  osStatus_t status;

  while (1) {
    pMem = (MEM_BLOCK_t *)osMemoryPoolAlloc(mpid_MemPool, 0U);  // Get Mem Block
    if (pMem != NULL) {                                         // Mem Block was available
      pMem->Buf[0] = 0x55U;                                     // Do some work...
      pMem->Idx    = 0U;

      status = osMemoryPoolFree(mpid_MemPool, pMem);            // Free mem block
    }
```

[34] https://bit.ly/3tND6Se

```
40      osThreadYield();                                    // Suspend thread
41    }
42  }
```

At line 16 a new pool is created so that it contains sixteen elements each one with a size equal to `sizeof(MEM_BLOCK_t)`. Then the pool is accessed by the thread `Thread_MemPool`: a block is retrieved from the pool at line 33 and it is manipulated.

23.4.5 Stack Overflow Detection

Before we talk about the features offered by FreeRTOS to detect stack overflows, we should spend some words about how to compute the right amount of memory a thread needs.

Unfortunately, it is not easy to give a definite answer, because it depends on a quite long list of aspects to keep in mind. First of all, stack size is affected by how deep the call stack is, that is by the number of functions called by our thread, and by the room occupied by each one of them. This space is essentially composed by local variables and passed parameters. Another relevant factor is the processor architecture, the compiler used and the optimization level chosen.

Usually, the stack size of a thread is computed experimentally, and FreeRTOS offers a way to try to detect stack overflows. Read my leaps: **to try** to detect. Because stack overflow detection is one of the hardest aspect of debugging, as well as static analysis of program code.

FreeRTOS offers two ways to detect stack overflows. The first one consists in using the function:

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

which returns the number of "unused" words of the thread stack. For example, assume a thread defined with a stack of 100 words (that is, 400 bytes on an STM32). Suppose that, in the worst scenario, the thread uses 90 words of its stack. Then the `uxTaskGetStackHighWaterMark()` returns the value 10.

The `TaskHandle_t` type of the parameter `xTask` is nothing more than the `osThreadId` returned by the `osThreadCreate()` function, and if we call the `uxTaskGetStackHighWaterMark()` from the same thread we can pass `NULL`.

This function is available only if:

- the `configCHECK_FOR_STACK_OVERFLOW` macro is defined with a value higher than 0, or
- the `configUSE_TRACE_FACILITY` is defined with a value higher than 0, or
- the `INCLUDE_uxTaskGetStackHighWaterMark` is defined with a value higher than 1.

All of them must be obviously defined in the `FreeRTOSConfig.h` file.

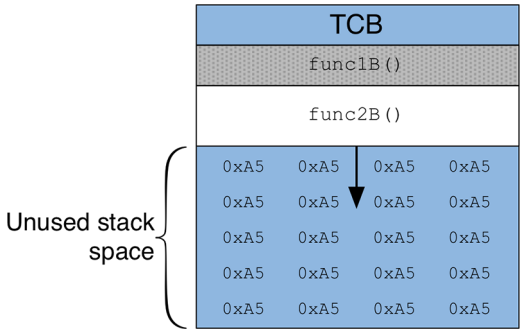


Figure 23.15: How FreeRTOS fills the stack with a fixed value (0xA5) to detect stack overflows



How does the `uxTaskGetStackHighWaterMark()` know how much stack has been used? There is nothing magic performed by that function. When one of the above macros is defined, FreeRTOS fills the stack of a thread with a "magic" number (defined by the macro `tskSTACK_FILL_BYTE` inside the `task.c` file), as shown in **Figure 23.15**. This is a "watermark" used to derive the number of free memory locations (that is the number of locations through the end of the thread stack still containing that value). This is one of the most efficient techniques used to detect buffer overflows.

The `uxTaskGetStackHighWaterMark()` function can be also used to verity the effective usage of the thread stack, and hence reduce its size if too much space is wasted.

FreeRTOS offers two additional methods to detect at run-time a stack overflow. Both consist in setting the `configCHECK_FOR_STACK_OVERFLOW` macro in the `FreeRTOSConfig.h` file. If we set it to 1, then every time a thread runs out, FreeRTOS check for the value of the current stack pointer: if it is higher than the top of the thread stack, then it is likely that a stack overflow is happened. In this case, the callback function:

```
void vApplicationStackOverflowHook(xTaskHandle *pxTask, signed portCHAR *pcTaskName);
```

is automatically called. By defining this function in our application, we can detect the stack overflow and debug it. For example, during a debug session we could place a software breakpoint in it:

```
void vApplicationStackOverflowHook(xTaskHandle *pxTask, signed portCHAR *pcTaskName) {
        asm("BKPT #0"); /* If a stack overflow is detected then, the debugger stop
                           the firmware execution here */
}
```

This method is fast, but it could miss stack overflows that happen in the middle of a *context switch*. So, by configuring the macro `configCHECK_FOR_STACK_OVERFLOW` to 2, FreeRTOS will apply the same method of the function `uxTaskGetStackHighWaterMark()`, that is it will fill the stack with a watermark value and it will call the `vApplicationStackOverflowHook` in case the latest 20 bytes of the stack have changed from their expected value. Since FreeRTOS performs this check at every *context switch*, this mode impacts on overall performances, and it should be used only during the firmware development (especially for high tick frequencies).

23.5 Synchronization Primitives

In a multi-threaded application, soon or later threads need a way to synchronize themselves, both while accessing to shared resources and when transmitting data between several execution streams. The literature about concurrent programming is full of algorithms and data structures best suited as synchronization primitives. The CMSIS-RTOS2 API, and the underlying FreeRTOS OS, defines those primitives that are common to all Operating Systems and threading libraries. This paragraph briefly introduces the most relevant ones.

23.5.1 Message Queues

A *queue*[35](see **Figure 23.16**) is a *First-In-First-Out* (FIFO) collection, which is implemented in FreeRTOS with a linear data structure where the first added element will be the first to be removed. When an element is added to the queue is said to be *enqueued*, while when it is removed is said to be *dequeued*.

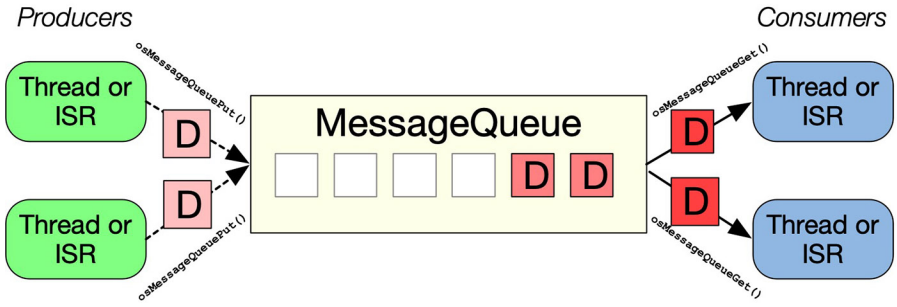


Figure 23.16:

Queues are widely used in concurrent programming, especially when data need to be exchanged between several threads that have different response time to events. For example, often we have two threads, one acting as *producer* and one as *consumer*, sharing a common buffer. The producer's job is to generate a piece of data, put it into the buffer and keep generating. At the same time, the consumer job consists in removing it from the buffer one piece at a time. The problem is to make sure that the producer will not try to add data into the buffer if it is full and that the consumer will not try to remove data from an empty buffer. In an RTOS, queues are designed so that if a thread tries to add data in full queue, it can be placed in blocked mode until at least one element is removed from the queue. At the same time, the OS kernel places the consumer in blocking mode if no data is available in the queue). Being handled from the OS, queues are designed so that no race conditions can occur between different threads (unless the programmer introduces evident errors in its code).

A queue is created and initialized by using the function:

[35]The CMSIS-RTOS2 use the term *message queues* to indicate what usually are simply called *queues*. As we will see in a while, this also impacts on the API (all structures and functions have the prefix `osMessage`). However, in the remaining part of this chapter, we will simply refer to them as *queues*.

```
osMessageQueueId_t osMessageQueueNew(uint32_t msg_count, uint32_t msg_size,
                                     const osMessageQueueAttr_t *attr);
```

where `msg_count` is the number of items in the queue and `msg_size` is the size of the individual item. The `struct osMessageQueueAttr_t` a structure similar to the `struct osMemoryPoolAttr_t` seen before, and we will not detail it here.

To enqueue a new element in the queue we use the function

```
osStatus_t osMessageQueuePut(osMessageQueueId_t mq_id, const void *msg_ptr,
                             uint8_t msg_prio, uint32_t timeout);
```

where `mq_id` is the id of the queue returned by the function `osMessageQueueNew`, `msg_ptr` is the pointer to the data to enqueue, `msg_prio` is used to sort messages in the queue according to a given priority (this is completely ignored by the CMSIS-RTOS2 layer developed by STM), `timeout` indicates the amount of ticks we are willing to wait if the queue is full: if sufficient room is not made available before the timeout period, then the `osMessageQueuePut()` function returns the value `osErrorTimeout`[36]. Passing `osWaitForever` will cause `osMessagePut()` to wait indefinitely.

To dequeue a data from the queue we use the function

```
osStatus_t osMessageQueueGet(osMessageQueueId_t mq_id, void *msg_ptr,
                             uint8_t *msg_prio, uint32_t timeout);
```

A queue needs to be explicitly deleted by using the function:

```
osStatus_t osMessageQueueDelete(osMessageQueueId_t mq_id);
```

Instead, its content can be reset to the initial state (the empty state) by using the function:

```
osStatus_t osMessageQueueReset(osMessageQueueId_t mq_id);
```



Take in account that FreeRTOS provides two separated APIs to manipulate queues from a thread or from an ISR. For example, the `xQueueReceive()` function is used to dequeue an element from a thread, while the `xQueueReceiveFromISR()` is used to safely dequeue elements from an ISR. The CMSIS-RTOS2 layer developed by ST is designed to abstract this aspect, and it automatically checks if we are performing the call from a thread or from an ISR. As usual, at the expense of speed.

The following example shows how a queue can be used to exchange data between two threads, one acting as *producer* (`UARTThread()`) and one as *consumer* (`blinkThread()`), which can run slow if a large timeout is specified.

[36] The `osMessageQueuePut()` and `osMessageQueueGet()` can return other status codes, according to if they are called from a thread or an ISR. For more information, consult the official CMSIS-RTOS2 specification (https://bit.ly/3tND6Se).

Filename: `Core/Src/main-ex3.c`

```
51  osMessageQueueId_t msgQueueID;
52
53  int main(void) {
54    HAL_Init();
55
56    Nucleo_BSP_Init();
57    RetargetInit(&huart2);
58
59    /* Init scheduler */
60    osKernelInitialize();
61
62    /* Creation of msgQueue */
63    msgQueueID = osMessageQueueNew(5, sizeof(uint16_t), NULL);
64    /* Creation of blinkThread */
65    blinkThreadID = osThreadNew(blinkThread, NULL, &blinkThread_attr);
66    /* Creation of UARTThread */
67    UARTThreadID = osThreadNew(UARTThread, NULL, &UARTThread_attr);
68
69    /* Start scheduler */
70    osKernelStart();
71
72    /* We should never get here as control is now taken by the scheduler */
73    while (1);
74  }
75
76  void blinkThread(void *argument) {
77    uint16_t delay = 500; /* Default delay */
78    uint16_t msg = 0;
79    osStatus_t status;
80
81    while(1) {
82      status = osMessageQueueGet(msgQueueID, &msg, 0, 10);
83      if(status == osOK)
84        delay = msg;
85
86      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
87      osDelay(delay);
88    }
89  }
90
91  void UARTThread(void *argument) {
92    uint16_t delay = 0;
93
94    while(1) {
95      printf("Specify the LD2 LED blink period: ");
96      fflush(stdout);
```

```
97      scanf("%hu", &delay);
98      printf("\r\nSpecified period: %hu\n\r", delay);
99      osMessageQueuePut(msgQueueID, &delay, 0, osWaitForever);
100   }
101 }
```

The `UARTThread`, defined at lines [91:101] uses the I/O retargeting technique seen in Chapter 5, allowing us to use the classical `printf()`/`scanf()` routines of the C standard library. The thread reads an `uint16_t` value from the UART and places it inside the queue `msgQueueID`. The `blinkThread()`, defined at lines [76:89] takes these values from the queue and uses them as delay values for the `osDelay()` function. This simple application allows us to pass the wanted LD2 LED blinking frequency from a terminal emulator.

If you specify a large delay value, you can easily see how queues can be used when a *producer* thread runs faster than a *consumer* one. By passing a delay equal to 10000, we can then immediately put another delay value equal to 50 inside the queue (because the queue has sufficient room to store another value). As you will see, we need about 10 seconds before the LED starts blinking at a rate of 20Hz, since `blinkThread()` is blocked by the `osDelay()` function.

23.5.2 Semaphores

In concurrent programming, a *semaphore* is a datatype used to control the access, by multiple execution streams, to a common resource. A bare-bone form of semaphore is represented by a Boolean variable: the state of the variable is used as a condition to control the access to a resource. For example, if the variable is equal to `False`, then a thread is placed in the blocked state until that variable becomes `True` again. A semaphore is said *to be taken* from the thread that acquires it, that is the thread that firstly finds the semaphore equal to `True`. This is indeed a *binary semaphore*, since it can assume only two states, and in FreeRTOS is implemented as a queue with just one element. If the queue is empty, then the first thread that tries to acquire it places a "flag" value in the queue, and it continues its execution; other threads will not be able to add other "flags" until the thread that has acquired the semaphore does not dequeue its flag.

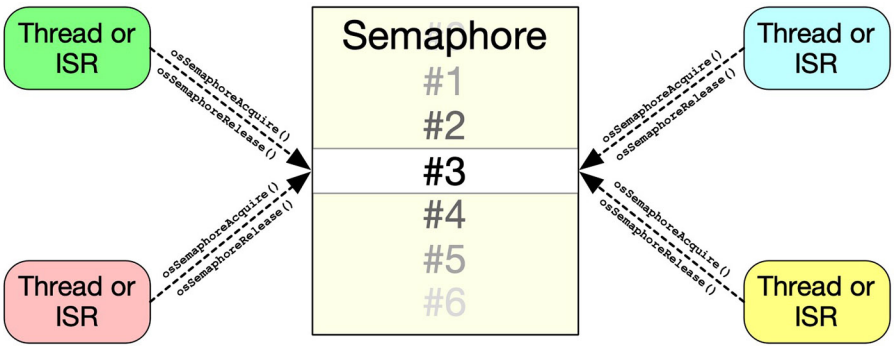


Figure 23.17:

A more general form of semaphore is the *counting semaphore* (see **Figure 23.17**), which allows more than one thread to acquire it. Just as binary semaphores are implemented as queues that have a length of one, a counting semaphore can be thought as queues that have a length more than one. A counting semaphore usually has an initial value, which is decremented every time a thread acquires it. While binary semaphores are usually used to discipline the concurrent access to just one resource, a counting semaphore can be used to:

- **discipline the access to pools of common resources**: in this case the count value indicates the number of available resources. For example, STM uses a counting semaphore with a maximum number of available tokens (the maximum counter value) equal to `block_count` to rule the access to a memory pool.
- **count the number of recurring events**: in this case an execution stream (for simplicity assume that it is an ISR) will release a semaphore (causing that its counter increases) to signal to another thread that a given event is occurred (e.g., a data coming from the UART is ready to be processed); this thread can then take the semaphore and start performing its activities; if another "event" takes place (new data arrived), then the ISR will increase again the semaphore by releasing it; in this way the processing thread will be able to take again the semaphore and perform its activities.

However, a simple variable cannot be used as a semaphore, since there is no guarantee that the operation of "taking" a semaphore is carried out in an atomic manner. So, to acquire a semaphore we need the intervention of a "third party", that is the OS kernel, which suspends the execution of other threads during the acquisition process.

FreeRTOS provides two distinct APIs to manage binary and counting semaphores, while the CMSIS-RTOS2 specifies that semaphores are implemented as counting semaphore (leaving to the mutexes the role of binary semaphores). However, the usage of counting semaphores increases the FreeRTOS codebase, which may have an important impact on microcontrollers with small amount of flash memory. For this reason, FreeRTOS provides them only if the macro `configUSE_COUNTING_SEMAPHORES` in the `FreeRTOSConfig.h` file is defined and equal to `1`.

In the CMSIS-RTOS2 API a semaphore is created by using the function:

```
osSemaphoreId_t osSemaphoreNew(uint32_t max_count, uint32_t initial_count,
                               const osSemaphoreAttr_t *attr);
```

where `max_count` specifies the maximum number of available tokens[37] (a `max_count` value of `1` creates a binary semaphore) and `initial_count` sets the initial number of available tokens.
To acquire a token in a semaphore we use the function:

[37] In the CMSIS-RTOS2 terminology, a resource accessed by a counting semaphore is called *token*. For this reason, we say that a semaphore allows to acquire/release tokens.

```
osStatus_t osSemaphoreAcquire(osSemaphoreId_t semaphore_id, uint32_t timeout);
```

which accepts the semaphore id and the timeout expressed in ticks. If the semaphore counter is higher than zero, the thread acquires it (reducing the counter) and it can continue. Otherwise, it is placed in blocked state for a period equal to the timeout value, until the counter increases again. A thread can wait indefinitely by specifying the `osWaitForever` value. The `osSemaphoreAcquire()` returns `osOK` if the thread has successfully acquired the semaphore. To release a token in a counting semaphore we use the function

```
osStatus_t osSemaphoreRelease(osSemaphoreId_t semaphore_id);
```

To see how many free tokens are available in a counting semaphore (that is, how many times we can still acquire a semaphore before the counter goes to zero), we can use the function:

```
uint32_t osSemaphoreGetCount(osSemaphoreId_t semaphore_id);
```

A semaphore must be explicitly destroyed by using the function:

```
osStatus_t osSemaphoreDelete(osSemaphoreId_t semaphore_id);
```



As seen for the APIs related to queues manipulation, FreeRTOS provides two separated APIs to manipulate semaphores from a thread or from an ISR. For example, the `xSemaphoreTake()` function is used to acquire a semaphore from a thread, while the `xSemaphoreTakeFromISR()` is used to perform this operation from an ISR. The CMSIS-RTOS2 layer developed by ST is designed to abstract this aspect.

The following example shows how to use a semaphore as notification primitive. This is again the classical blinking application, but this time the delay of the `blinkThread()` is established by another thread, `delayThread()`, which "unlock" the blinking thread by releasing a binary semaphore.

Filename: `Core/Src/main-ex4.c`

```
51 osSemaphoreId_t semID;
52
53 int main(void) {
54   HAL_Init();
55
56   Nucleo_BSP_Init();
57   RetargetInit(&huart2);
58
59   /* Init scheduler */
60   osKernelInitialize();
61
62   /* Creation of binary semaphore */
```

```
63    semID = osSemaphoreNew(1, 1, NULL);
64    osSemaphoreAcquire(semID, osWaitForever);
65    /* Creation of blinkThread */
66    blinkThreadID = osThreadNew(blinkThread, NULL, NULL);
67    /* Creation of UARTThread */
68    delayThreadID = osThreadNew(delayThread, NULL, NULL);
69
70    /* Start scheduler */
71    osKernelStart();
72
73    /* We should never get here as control is now taken by the scheduler */
74    while (1);
75  }
76
77  void blinkThread(void *argument) {
78    while(1) {
79      osSemaphoreAcquire(semID, osWaitForever);
80      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
81    }
82  }
83
84  void delayThread(void *argument) {
85    while(1) {
86      osDelay(500);
87      osSemaphoreRelease(semID);
88    }
89  }
```

Line 63 defines and creates a binary semaphore with ID `semID`: the semaphore is immediately acquired, causing its counter to become equal to zero. `blinkThread()` and `delayThread()` are scheduled, but the first one is placed in blocked state as soon as it reaches the `osSemaphoreAcquire()` call: being the semaphore already "acquired", the thread will be swapped out until the semaphore is released by the `delayThread()` thread, which performs this operation every 500ms. This will cause the LD2 LED to blink at a 2Hz rate.

23.5.3 Event and Thread Flags

The example 4 could be rearranged to use a feature more suitable for this kind of applications: the *event flags*. Event flags are used to trigger execution states between threads or between ISRs and threads. Unlike queues and semaphores:

- Event flags allow a task to wait in the *blocked* state for a combination of one of more events to occur.
- An event flags can be accessed by multiple tasks.

- Event flags unblock *all the tasks* that were waiting for the same event, or combination of events, when the event occurs.

These unique properties of event flags make them useful for synchronizing multiple tasks, broadcasting events to more than one task, allowing a task to wait in the *blocked* state for any one of a set of events to occur, and allowing a task to wait in the Blocked state for multiple actions to complete. Event flags also provide the opportunity to reduce the SRAM used by an application, as often it is possible to replace many binary semaphores with a single event flag. The event flags management functions in CMSIS-RTOS2 allow you to control or wait for signal flags. Event flags are built on top of *event groups*, a similar synchronization primitive provided by FreeRTOS. In the CMSIS-RTOS2 specification, each event flag has up to 31 assigned signal flags, corresponding to individual bits in a `uint32_t` datatype. However, in FreeRTOS an event flag can have up to 24 assigned signal flags, as shown in **Figure 23.18**.

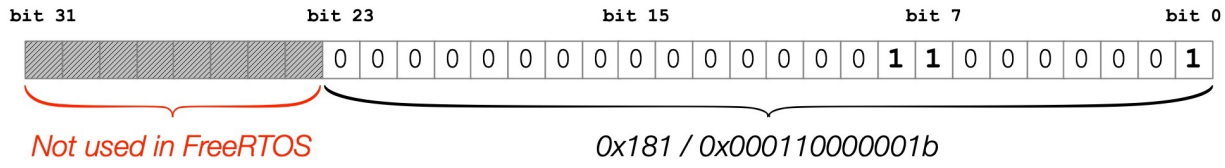


Figure 23.18: How bits in a event flag are interpreted

An event flag is created by using the function:

```
osEventFlagsId_t osEventFlagsNew(const osEventFlagsAttr_t *attr);
```

A thread can wait for event flags to be set by using the function:

```
uint32_t        osEventFlagsWait(osEventFlagsId_t ef_id, uint32_t flags,
                        uint32_t options, uint32_t timeout);
```

where `flags` is the bitwise OR of flags to wait for, `timeout` is the maximum timeout value (or no timeout if equal to `0`), `options` specifies the wait condition and can assume the following values:

- **`osFlagsWaitAny`**: wait for any flag (default).
- **`osFlagsWaitAll`**: wait for all flags.
- **`osFlagsNoClear`**: when a thread wakes up and resumes execution, its signal flags are automatically cleared unless the option `osFlagsNoClear` is specified; in this case the function `osEventFlagsClear()` can be used to clear flags manually.

If the wanted flags are not set, the calling thread enter in *blocked* state.
It is possible to set one or more flags by using the function:

```
uint32_t osEventFlagsSet(osEventFlagsId_t ef_id, uint32_t flags);
```

Instead, one or more flags can be cleared by using:

```
uint32_t osEventFlagsClear(osEventFlagsId_t ef_id, uint32_t flags);
```

To retrieve the status of a flag in a event flag object you can use the function:

```
uint32_t osEventFlagsGet(osEventFlagsId_t ef_id);
```

The following example shows a possible usage of event flags.

Filename: `Core/Src/main-ex5.c`

```
51  #define FLAG_LED_BLINK        (uint32_t)0xb00000001
52  #define FLAG_CHANGE_FREQUENCY (uint32_t)0xb00000010
53
54  osEventFlagsId_t evtID;
55
56  int main(void) {
57    HAL_Init();
58
59    Nucleo_BSP_Init();
60    RetargetInit(&huart2);
61
62    /* Init scheduler */
63    osKernelInitialize();
64
65    /* Creation of a event flag */
66    evtID = osEventFlagsNew(NULL);
67    /* Creation of blinkThread */
68    blinkThreadID = osThreadNew(blinkThread, NULL, NULL);
69    /* Creation of UARTThread */
70    delayThreadID = osThreadNew(delayThread, NULL, NULL);
71
72    /* Start scheduler */
73    osKernelStart();
74
75    /* We should never get here as control is now taken by the scheduler */
76    while (1);
77  }
78
79  void blinkThread(void *argument) {
80    while(1) {
81      osEventFlagsWait(evtID, FLAG_LED_BLINK, osFlagsWaitAll, osWaitForever);
82      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
83    }
```

```
}

void delayThread(void *argument) {
  uint8_t step = 100;
  uint16_t delay = 500;

  while(1) {
    osEventFlagsSet(evtID, FLAG_LED_BLINK);

    if(osEventFlagsWait(evtID, FLAG_CHANGE_FREQUENCY, osFlagsWaitAll, delay)
       != osFlagsErrorTimeout ) {
      delay -= step;
      switch(delay) {
      case 100:
        step = 50;
        break;
      case 50:
        step = 25;
        break;
      case 0:
        step = 100;
        delay = 500;
        break;
      }
    }
  }
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  if(GPIO_Pin == GPIO_PIN_13)
    osEventFlagsSet(evtID, FLAG_CHANGE_FREQUENCY);
}
```

The thread `blinkThread()` is placed in blocked state until the flag `FLAG_LED_BLINK` is set by the thread `delayThread()`. By default, `delayThread()` set the flag every 500ms, causing the LD2 LED to blink at 2Hz rate. When the USER BUTTON is pressed, the corresponding EXTI callback sets the flag `FLAG_CHANGE_FREQUENCY`: this causes that the `delay variable is decremented`, increasing the blinking frequency.

While an event flag can be accessed by an arbitrary number of threads and ISRs, *thread flags* are a more specialized version of the event flags, which are only related to a given specific thread. Every thread instance can receive thread flags without any additional allocation of a thread flags object. However, the underlying implementation of a thread flag is totally different from event flags: instead of using a FreeRTOS event group, the wrapper uses a specific mechanism offered by FreeRTOS called *task notifications*.

There is no need to create a new thread flag, but a given thread's flag can be directly set by using

the function:

```
uint32_t osThreadFlagsSet(osThreadId_t thread_id, uint32_t flags);
```

This function is the sole thread flag function that can be called from an ISR context. Once set, a flag can be just cleared by the running thread by calling the function:

```
uint32_t osThreadFlagsClear(uint32_t flags);
```

23.6 Resources Management and Mutual Exclusion

In embedded applications it is quite frequent to access to hardware resources. For example, assume that we use the UART peripheral to write debug messages to the console, and assume that our application is made of several threads that can print messages using the `HAL_UART_Trasmit()` routine. If you remember, in Chapter 8 we have seen that when we use the UART in polling mode, the bytes contained in the message we are going to transmit are transferred one-by-one in the UART *Data Register* (DR). This is a quite "slow procedure", compared to the number of activities an RTOS may performs in a unit of time. This means that, if two threads call the `HAL_UART_Trasmit()` they are likely to overwrite the content of the buffer register.



If you remember, always in that chapter we have seen that the HAL tries to protect concurrent accesses to peripherals by using the `__HAL_LOCK()` macro. However, there is no guarantee that in a multithreaded environment that macro will prevent race conditions since the locking operation is not performed atomically.

While semaphores are best suited to synchronize thread activities, mutexes and critical sections are a way to protect shared resources in concurrent programming. FreeRTOS provides us both the primitives, while the CMSIS-RTOS2 layer only defines the notion of mutex. However, critical sections come in handy in several situations, and sometimes they represent a better solution to problems that would require more programming effort from the developer to avoid subtle conditions, like the *priority inversion*.

23.6.1 Mutexes

Mutex is acronym for *MUTual EXclusion*, and they are a sort of binary semaphores used to control the access to shared resources. From a conceptual point of view, mutexes differentiate from semaphore for two reasons:

- a mutex must be always taken and then released to signal that the protected resource is now available again, while a semaphore can even be released to wake up a blocking thread (we have seen this mode in the example 4); moreover, usually a mutex is taken and released by the same thread[38];
- a mutex implement the *priority inheritance*, a feature we will analyze later used to minimize the *priority inversion* problem.

To use mutexes, we need to define the macro `configUSE_MUTEXES` inside the `FreeRTOSConfig.h` file and set it to 1. A mutex is defined using the function:

```
osMutexId_t osMutexNew(const osMutexAttr_t *attr);
```

where the `osMutexAttr_t.attr_bits` field can assume the following values:

- **`osMutexRecursive`**: a thread can consume the mutex multiple times without locking itself. More about recursive mutexes in a while.
- **`osMutexPrioInherit`**: the owner thread inherits the priority of a (higher priority) waiting thread. This option is currently ignored by the `osMutexNew()`, since this is the default FreeRTOS implementation policy. More about priority inversion problem next.
- **`osMutexRobust`**: the mutex is automatically released when owner thread is terminated. This option is currently ignored by the `osMutexNew()`, since this is the default FreeRTOS implementation policy.

Similarly to semaphores, to acquire a mutex we use the function

```
osStatus_t osMutexAcquire(osMutexId_t mutex_id, uint32_t timeout);
```

and to release it we use the function:

```
osStatus_t osMutexRelease(osMutexId_t mutex_id);
```

Finally, to destroy a mutex we must explicitly call the function

```
osStatus_t osMutexDelete(osMutexId_t mutex_id);
```

[38]However, different from other Operating Systems, FreeRTOS is not implemented to check that only the thread that has acquired the mutex can release it.

23.6.1.1 The Priority Inversion Problem

Mutexes may introduce an unwanted subtle problem, well known in literature as the *priority inversion* problem. Let us consider this scenario with the help of the **Figure 23.19**.

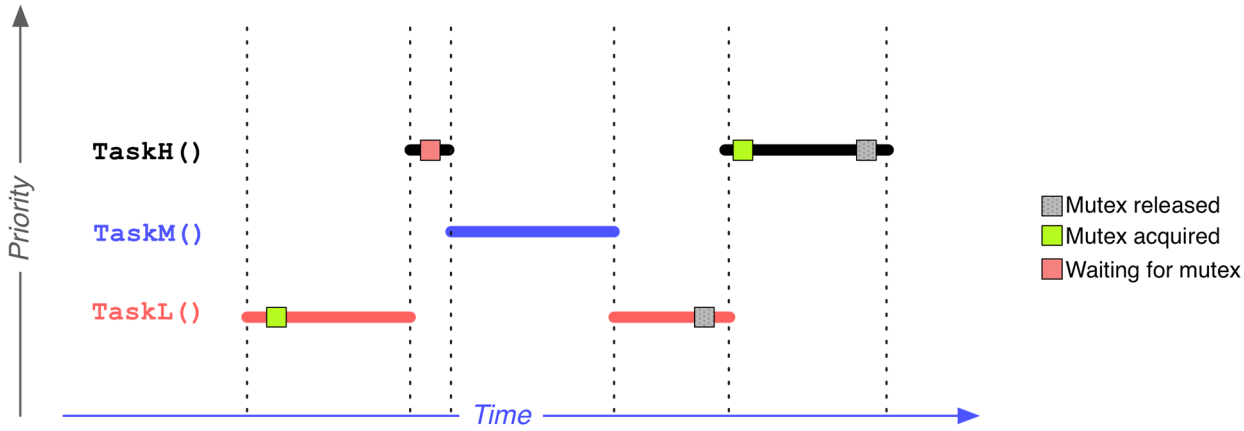


Figure 23.19: The diagram schematizes the *priority inversion* problem

`ThreadL()`, `ThreadM()` and `ThreadH()` are three threads with an increasing priority (`L` stands for low, `M` for medium and `H` for high). `ThreadL()` starts its execution and it acquires a mutex used to protect a shared resource. During its execution, `ThreadH()` returns in *ready* mode and it is scheduled for execution having a higher priority. However, it also needs to acquire the same mutex and it goes back in *blocked* state. Suddenly, the medium-priority thread `ThreadM()` goes available, and it is scheduled for execution having a priority higher than `ThreadL()`. This cannot so finish its job and the mutex remain locked, preventing `ThreadH()` from being executed. In this case, we have the practical effect that the priority between `ThreadL()` and `ThreadH()` is inverted, since `ThreadH()` cannot be executed until `ThreadL()` releases the mutex.

The priority inversion problem should be avoided at all by rearranging application in a different manner. However, FreeRTOS tries to minimize the impact of this issue by temporarily increasing the priority of the mutex holder (in our case `ThreadL()`) to the priority of the highest priority thread that is attempting to acquire the same mutex.

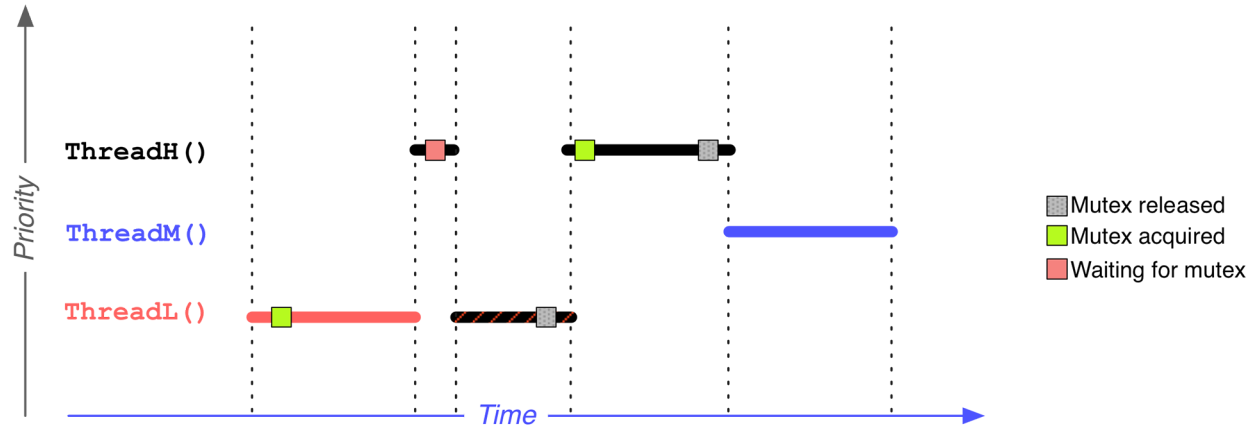


Figure 23.20: How the *priority inversion* problem is addressed by temporary increasing the priority of `ThreadL`

The **Figure 23.20** clearly shows this process. `ThreadL()` starts its execution and it acquires a mutex. During its execution, `ThreadH()` returns in *ready* mode and it is scheduled for execution having a higher priority. However, it also needs to acquire the same mutex and it goes back in *blocked* state. This time, the priority of the `ThreadL()` is increased to the same of `ThreadH()`, preventing the `ThreadM()` from being executed. `ThreadL()` is scheduled again and it can release the mutex, allowing `ThreadH()` to run. Finally, `ThreadM()` can execute, since the priority of `ThreadL()` is decreased to its original priority when it releases the mutex.

23.6.1.2 Recursive Mutexes

Sometimes it happens that, especially when our application is fragmented in several APIs, a thread accidentally acquires a mutex more than once. Since a mutex can be acquired only once, any subsequent attempt from the same thread to acquire the same mutex will cause a *deadlock* (because a successive call to the `osMutexWait()` will place the thread in *blocking* state, but it is the only thread designed to release the mutex).

To prevent this unwanted behaviour, FreeRTOS introduces the notion of *recursive mutexes*, that is mutexes than can be acquired more than once. Clearly, a recursive mutex needs to be released the same number of times it has been acquired. To configure a mutex as recursive with the CMSIS-RTOS2 API it is necessary to specify the attribute `osMutexRecursive` to the `osMutexNew()` function.

23.6.2 Critical Sections

Sometimes, especially when we need to perform a quick operation on a shared resource, it is best to avoid using synchronization primitives at all. As seen before, it is easy to introduce weird behavior in our application unless we handle with special care synchronization constructs offered by the RTOS.

Critical sections are a way to protect the access to shared resources. A critical section is a region of code that is executed after all interrupts have been disabled. Since the preemption of tasks occurs inside an ISR (the ISR of the the timer chosen as timebase generator), by disabling all ISRs we are sure that no other code will preempt the execution of the code inside the critical section.

```
...
__disable_irq();
//All IRQs are disabled and we are sure that the next code will not be preempted
...
//Critical code here
...
__enable_irq();
//All IRQs are now enabled again, and normal behaviour of the RTOS is restored
```

Implementing a critical section using CMSIS APIs is not a trivial task, because we need to take care of special hardware situations may occur. However, FreeRTOS provide us four routines that we can use to define critical sections in our application.

The `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()` functions allow to define a critical section inside a thread. Those routines are designed to keep tracking of the nesting, that is each time the `taskENTER_CRITICAL()` is called a counter is incremented, and it is decremented on a subsequent call to the `taskEXIT_CRITICAL()` function. This means that we have to be sure to respect the calling order.

```
taskENTER_CRITICAL(); //Internal counter increased to 1
...
        taskENTER_CRITICAL(); //Internal counter increased to 2
        ...
        taskEXIT_CRITICAL(); //Internal counter decreased to 1
...
taskEXIT_CRITICAL(); //Internal counter decreased to 0
```

Critical sections works well only if they are used to protect few lines of code, that perform their activities in short time. Otherwise, the whole application can be impacted by their usage.

The `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()` functions should never called from an ISR: the corresponding The `taskENTER_CRITICAL_FROM_ISR()` and `taskEXIT_CRITICAL_FROM_ISR()` functions are suited for this application. For more information consult the FreeRTOS documentation.

23.6.3 Interrupt Management With an RTOS

The general rule of thumb of interrupt service routines is that they need to be fast. A slow ISR may cause the loss of other events, both generated from the same peripheral or from other sources if this ISR has a higher priority.

Some features of an RTOS can simplify the interrupt management by deferring the effective interrupt handling to a thread. A *deferred execution*, or simply a *deferred*, consists in delegating to another execution stream, not working at the same "low-level" of interrupt routines, the effective interrupt handling. For example, in Chapter 8 we have seen that the `USARTx_IRQn` interrupt is generated when a new data is ready to be transferred from the UART *Data Register*: the ISR effectively takes this

bytes from the register and places it inside a buffer. However, we have also seen that the `UART_IRQ_Hanler()` performs a lot of other operations, that slow down the ISR execution.

In this scenario, we could have a dedicated thread for each ISR. This thread would spend a lot of time in blocking mode waiting for a given signal. When the IRQ fires, we could trigger that signal, causing that the blocked thread is resumed to carry out the job that would be performed by the corresponding ISR. By assigning different priorities to threads, we may establish an execution order in case of concurrent ISRs. Another approach is to use a queue to transfer the data coming to the peripheral to a worker thread, which will process it later. This is especially useful when the consumer thread is slower than the peripheral ISR, which acts as a consumer thread in this case.

FreeRTOS provides another convenient way to defer the ISR execution to another execution stream. This is called *centralized deferred interrupt processing* and it consists in deferring the execution of a routine in the FreeRTOS *daemon* task[39]. This method uses the `xTimerPendFunctionCallFromISR()` which is documented in the FreeRTOS manual[40].

However, take in mind that either deferring the execution to another thread or using a queue to exchange data implies that several operations are performed by the CPU, and this may impact on the reliability of ISR management. If your peripheral runs fast, it is better to use other ways to transfer data, for example using the DMA. Always considering the example of the UART transfer, if our application exchanges fixed-length messages over the UART we could setup the DMA to transfer a message and then use the DMA IRQ to move the whole message inside a queue. This would certainly minimize the overhead connected with the transfer of individual bytes.

23.6.3.1 FreeRTOS API and Interrupt Priorities

So far, we have seen that FreeRTOS provides some APIs that are expressly designed to be called within ISRs. For a given FreeRTOS function, there exists a corresponding ISR-safe routine ending with `FromISR()` (for example, the `xQueueReceiveFromISR()` for the `xQueueReceive()` routine). These routines are designed so that interrupts are masked (by entering and then exiting a critical section), preventing the execution of other interrupts that could generate race conditions by calling other FreeRTOS functions.

The interrupts masking is required because interrupts are a source of multiprogramming handled by the hardware. While threads are different program flows handled by the RTOS, which avoids race conditions by simply suspending the execution of the scheduler, ISR are generated by the hardware and there is little we can do to avoid race conditions unless we mask their execution or define a strict priority-based execution order. Moreover, the nesting mechanism offered by Cortex-M cores increases the risk of race conditions in our code. For example, an ISR which is acquiring a semaphore may be preempted by another ISR with higher priority performing the same operation. This will have a catastrophic effect for sure.

Even if the CMSIS-RTOS2 layer is designed to abstract this dual API system, we must place special care when calling FreeRTOS APIs from ISR routines in Cortex-M3/4/7 based microcontrollers. This

[39] The FreeRTOS *daemon* task is also called the *timer service* task because it is the thread that handles the execution of timer's callback routines, which we will analyze later.

[40] http://www.freertos.org/xTimerPendFunctionCallFromISR.html

happens because these cores allow to selectively mask interrupts on a priority level basis. In Chapter 7 we have seen that the BASEPRI register allows to disable selectively ISRs execution by masking all those IRQs having a priority lower than a given value. FreeRTOS uses this mechanism to allow the execution of higher priority interrupts, which are assumed to be non-interruptible, while suspending lower ones. This means that it is not safe to call FreeRTOS APIs from all ISRs, but it is only safe to call FreeRTOS functions from those ISRs having a given (or lower) priority level.

We can set this maximum priority level by defining the macro configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY[41] in the FreeRTOSConfig.h file. CubeMX automatically performs this operation for us, and usually the maximum priority level is set to 5. Special care must be placed when we enable IRQs using CubeMX: even if recent releases of CubeMX seem to handle this aspect correctly, always ensure that an ISR that calls FreeRTOS functions is configured with a priority equal to configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY or lower.

Despite to the fact that this macro is also defined in projects generated by CubeMX for STM32F0/L0 MCUs, this has no practical effects since the FreeRTOS port for those families uses the PRIMASK register to mask all interrupts (Cortex-M0/0+ cores do not offer a way to selectively disable IRQs). So, that macro is simply ignored.

Finally, it is important to remember that FreeRTOS is designed so that the *tick* interrupt (that is the IRQ associated to the timer that acts as timebase generator for the kernel) must be set to the lowest possible interrupt, which is equal to 7 in STM32F0/L0 families and to 15 for all other MCUs. The macro configLIBRARY_LOWEST_INTERRUPT_PRIORITY in FreeRTOSConfig.h file sets this, and it is strongly suggested to leave it as is.

23.7 Software Timers

Software *timers* are the way an RTOS provides to schedule the execution of routines on a time-basis. Software timers are implemented by, and under the control of, the FreeRTOS kernel. They do not require specific hardware support (except for the timer used as *tick* generator for the OS) and they have nothing related to hardware timers. Moreover, they are not able to provide the same accuracy of hardware timers and they should never be used to perform activities related with the hardware (for example, to trigger a DMA event).

Software timers are an optional feature in FreeRTOS, and they need to be enabled by setting the macro config_USE_TIMERS to 1 in the FreeRTOSConfig.h file. When we enable timers, FreeRTOS also requires that we define the macros configTIMER_TASK_PRIORITY, configTIMER_QUEUE_LENGTH, configTIMER_TASK_STACK_DEPTH. We will see the role of this macro in a while.

In the CMSIS-RTOS2 layer, a software timer is created by using the function:

[41] If you read the official FreeRTOS documentation, you can see that the macro used to setup the maximum interruptible priority level is configMAX_SYSCALL_INTERRUPT_PRIORITY. However, being FreeRTOS portable among several silicon vendors, the priority level specified with that macro is the exact value of the IPR register, that accepts only the upper 4 bits in STM32 MCUs (for example, a priority equal to 0x2 must be specified as 0x20). ST engineers have defined the macro configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY so that we can specify the priority level according to the HAL convention (in LSB form), while the configMAX_SYSCALL_INTERRUPT_PRIORITY is defined in the following way: `#define configMAX_SYSCALL_INTERRUPT_PRIORITY ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )`

```
osTimerId_t osTimerNew(osTimerFunc_t func, osTimerType_t type,
                       void *argument, const osTimerAttr_t *attr);
```

where `func` is the pointer to the function to call when the timer expires, and `argument` is an optional argument to pass to the callback routine. The CMSIS-RTOS2 API provides two kinds of software timers, configured with the parameter `type`:

- **`osTimerOnce`**: *one-shot* timers, that is timers that execute the callback only once.
- **`osTimerPeriodic`**: *periodic* timers, which act like hardware STM32 timers that restarts counting again after they overflow.

To start a timer, we use the function

```
osStatus_t osTimerStart(osTimerId_t timer_id, uint32_t ticks);
```

where the `ticks` parameter represents the period of the timer expressed in ticks. To stop the timer, we use the function

```
osStatus_t osTimerStop(osTimerId_t timer_id);
```

Instead, to check if a timer is running, we use the function:

```
uint32_t osTimerIsRunning(osTimerId_t timer_id);
```

Finally, a timer is dynamically allocated by the OS and needs to be destroyed when no longer needed by using the function

```
osStatus osTimerDelete(osTimerId timer_id);
```

The following example shows our omnipresent blinking application made with a software timer.

Filename: `Core/Src/main-ex6.c`

```
1  void blinkFunc(void *argument);
2
3  int main(void) {
4    osTimerId_t timID;
5
6    HAL_Init();
7
8    Nucleo_BSP_Init();
9
10   /* Init scheduler */
11   osKernelInitialize();
```

```
12
13    /* Creation of blinkThread */
14    timID = osTimerNew(blinkFunc, osTimerPeriodic, NULL, NULL);
15    osTimerStart(timID, 500);
16
17    /* Start scheduler */
18    osKernelStart();
19
20    /* We should never get here as control is now taken by the scheduler */
21    while (1);
22  }
23
24  void blinkFunc(void *argument) {
25    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
26  }
```

23.7.1 How FreeRTOS Manages Timers

As you can see in the previous example, our application does not use threads. So, who takes care of timers? FreeRTOS uses a centralized thread, named RTOS *daemon* (or also *timer service* thread), which automatically calls the callback routines when a timer expires. This thread is a regular thread, which has a priority defined by the macro `configTIMER_TASK_PRIORITY` and a stack with a size defined by the macro `configTIMER_TASK_STACK_DEPTH`. Moreover, it has an internal pool of timer objects, whose size is defined by the macro `configTIMER_QUEUE_LENGTH`.

Another important aspect to stress is the way FreeRTOS computes the time internally. FreeRTOS measure the time in function of the *tick* frequency, which is in turn defined by the overflow frequency of the timer chosen as timebase generator. This means that, if we use the *SysTick* timer configured to overflow ever 1ms, then internal software timers have a resolution of 1ms (which corresponds to 1 tick). The `ticks` value passed to the `osTimerStart()` routine is so bound to the global *tick* frequency.

23.8 A Case Study: Low-Power Management With an RTOS



This is a really advanced topic, that requires the knowledge of many concepts underlying an RTOS. Moreover, a decent knowledge of the concepts illustrated in Chapter 20 is required. **Un-experienced users can safely skip this part**.

In Chapter 19 we have analyzed the low-power features offered by STM32 microcontrollers. We have seen that, especially for MCUs belonging to the STM32L-series, they offer several power modes

useful to reduce the energy consumption of the MCU when there is not too much active work to do. We have also seen that the MCU enters in one of its low-power modes on a voluntary basis, by calling one of the two dedicated assembly instructions: `WFI` or `WFE`. If we know that the firmware has nothing important to do for a "long" period , we can enter in low-power mode waiting for an external interrupt or event.

When we use an RTOS, it is harder to say "when there is not too much work to do". So far, we have seen that the RTOS schedules a particular thread when all other threads are in *blocked* or *suspended* state: the *idle*. This means that an RTOS always must find a way to do something (simply because the CPU never stops), unless we enter in a low-power mode halting the MCU core.

An RTOS is so a source of "power leaks" if we do not find a solution to suspend its execution. There are essentially two ways to place the MCU in a low-power mode when we use an RTOS: one is suitable "to take a nap", another one to longer and deeper sleep modes. Let us analyze both.

23.8.1 The *idle* Thread Hook

So far, we have seen that the ISR associated to the timer used as timebase generator for the RTOS (usually, the *SysTick* timer) rules the RTOS activities. Every 1ms the *SysTick* timer underflows, and its ISR passes the control to the OS scheduler, which establishes the next thread to be executed[42]. If no thread is in *ready* state, then the OS execute the *idle* thread, until another thread becomes ready. This means that, when the *idle* thread is scheduled, it is likely to be the right time to place the MCU in *sleep* mode to reduce power consumption.

For this reason, FreeRTOS gives to the user the ability to define an *idle hook*, that is a callback function invoked within the *idle* thread. To enable the hook, we have to define the macro `configUSE_IDLE_HOOK` inside the `FreeRTOSConfig.h` file and set it to 1. Next, we can define the function `vApplicationIdleHook(void)` somewhere in our source code.
For example, to place the MCU in *sleep* mode every time the *idle* thread is scheduled, we can define that function in this way:

```
void vApplicationIdleHook( void ) {
  //Assume __HAL_RCC_PWR_CLK_ENABLE() is called elsewhere
  HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFE);
}
```

[42]This is behavior is enabled when the scheduling policy is the *prioritized preemptive scheduling with time slicing*, according to **Table 23.2**.



Which Sleep Instruction to Use?

Cortex-M based MCUs offer two assembly instructions to enter in low-power modes: `WFI` and `WFE`. But which one is more suitable to be called from the *idle* hook? The `WFI` instruction will keep the MCU core OFF until an interrupt is raised. This could be either the interrupt of the *SysTick* timer or of another peripheral. The `WFE` instruction, instead, is conditional: it does not enter in *sleep* mode if the event register is set (the `WFI` always enter and then re-exit, if an interrupt is pending, wasting several CPU cycles). Moreover, it allows to wake up the processor if we are using events associated to a given peripheral instead of interrupts, while it is still able to wake up in case of interrupts. For these reasons, the `WFE` instruction is always preferred to the `WFI` one in *idle* loops.

The power saving that can be achieved by this simple method is limited by the necessity to periodically exit and then re-enter the low-power mode to process *tick* interrupts (which are related to the underflow frequency of the *SysTick* timer), as shown in **Figure 23.21**. Moreover, if the frequency of the *tick* interrupt is too high, the energy and time consumed entering and then exiting a low-power mode for every tick will outweigh any potential power saving gain for all but the lightest power saving modes.

For these reasons, it is completely impracticable to enter deeper sleep modes, like the *stop* one. Moreover, the overhead connected with the entering and exiting from low-power mode affects the reliability of the *tick* counter, causing shifts that impact on software timers and timeout delays.

23.8.2 The Tickless Mode in FreeRTOS

To address these issues, FreeRTOS offers a working mode named *tickless idle* mode (or simply *tickless* mode), which stops the periodic *tick* interrupt during idle periods. The duration of these *periods* is arbitrary: it can be several milliseconds, some seconds, minutes or even days. When the MCU exits from low-power mode, FreeRTOS makes a correcting adjustment to the *tick* count value when the tick interrupt is restarted, if needed (more about this soon). This means that FreeRTOS does not stop the timer at all: it just configures the timer so that it reaches its maximum update period before overflowing. When the MCU wakes up again, the kernel reads the counter value of the timer and computes the number of elapsed *ticks* during the sleep time.

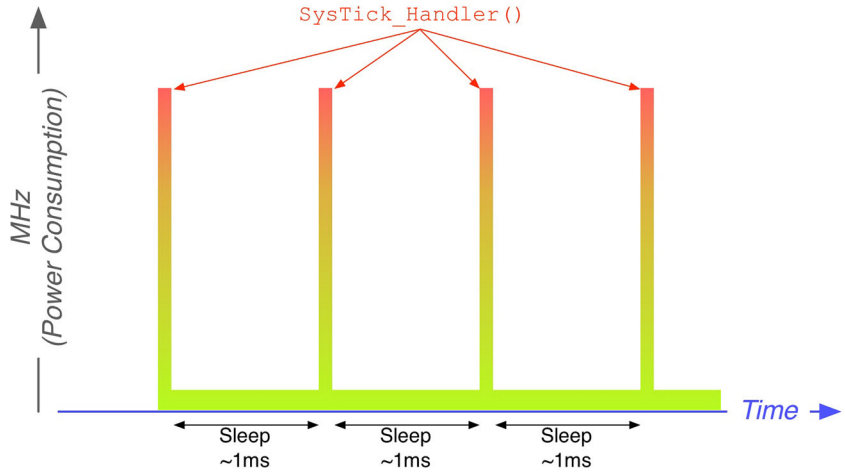


Figure 23.21: The effects of *SysTick* interrupts on the power consumption

For example, assume a 16-bit timer clocked at the core SYSCLK frequency of 48MHz. The maximum values for the `Period` and `Prescaler` registers are equal to 0xFFFF. So instead of configuring the timer so that it overflows ever 1ms, we can configure it to overflow after:

$$UpdateEvent = \frac{48.000.000}{0xFFFF \times 0xFFFF} \approx 90s$$

FreeRTOS provides a built-in *tickless* functionality, which is enabled by defining the macro `configUSE_TICKLESS_IDLE` as 1 in `FreeRTOSConfig.h`. The built-in *tickless* mode is platform dependent: for this reason, it is implemented inside the `port.c` file. The built-in *tickless* is available for all Cortex-M cores, but it has one relevant limitation: it relies on the *SysTick* timer, because it is the only timer available in all MCUs based on this architecture.

What's wrong with it? The *SysTick* timer is a 24-bit down-counter timer, clocked at the same core clock frequency. Unfortunately, it cannot be easily prescaled like regular STM32 timers (it has just one prescaler value, equal to 8, in all STM32 MCUs). For example, for an STM32F030 running at 48MHz we have that, applying the equation [1] from Chapter 11, the *SysTick* timer will overflow every:

$$UpdateEvent = \frac{48.000.000}{8 \times 0xFFFFFF} \approx 0.350Hz \approx 2.8s$$

Since we cannot lose the overflow event at all, otherwise the global *tick* count would be compromised[43], we have to wake up again even if we have nothing relevant to do. For most low-power applications this is a short time between two consecutive sleep periods.

A solution may be represented by lowering the HCLK speed to further increase the overflow period, but we have to pay attention to lowering the core frequency too much, because when the MCU

[43] As we will discover later, under certain circumstances we can safely stop incrementing the global *tick* counter. This can be done when we are not going to use software timers and timeouts: if all threads are blocked or suspended indefinitely, then it is safe to completely turn OFF the timebase generator.

exits from low-power mode to service an interrupt a low HCLK speed could compromise the system reliability. And to increase the clock speed from an ISR is not a smart thing.



Why *tick* Count Accuracy Is So Relevant?

The accuracy of the global *tick* count is important for two main reasons: to guarantee the same *quantum* time to all *ready* threads with the same priority (if preemption is enabled) and to ensure precise timeout delays. In fact, several blocking OS routines allow to specify a maximum delay we are willing to wait before the operation is performed. Timeouts are specified in *ticks*, knowing that a *tick* usually lasts 1ms for Cortex-M FreeRTOS port. If we specify a timeout smaller than `osWaitForever`, then it is important that the *tick* count is the most accurate one. The global *tick* count is also used by FreeRTOS to implement software timers.

Another limitation in using the *SysTick* timer arises from the fact that it cannot be used in *stop* modes, because the HCLK clock source is turned off. This is one of the typical applications of the low-power timers (LPTIM) provided by the most of STM32L microcontrollers. LPTIM timers, in fact, can run independently from the system clock: this allows to use them even in *stop* modes.

For all these reasons, we are now going to provide a custom implementation of the *tickless idle* functionality, which can be provided for any FreeRTOS port (including those that provide a built-in implementation) by defining `configUSE_TICKLESS_IDLE` to 2 in `FreeRTOSConfig.h`. When this configuration is chosen, we can override two FreeRTOS functions: `void prvSetupTimerInterrupt()`[44] and `void vPortSuppressTicksAndSleep()`. The former is used by the kernel to setup the timer used as *tick* generator. The latter is automatically called by the kernel when some conditions (that we will see later) are satisfied, and we can enter in low-power modes delaying or suspending at all the periodic timer interrupt.

23.8.2.1 A Schema for the *tickless* Mode

Before we dive into the real source code needed to implement those two routines, it is best to look at the underlying logic without struggling with implementation details.

```
1  /* Override the default definition of vPortSetupTimerInterrupt() with a version
2   that configures another STM32 timer to generate the tick interrupt. */
3  void vPortSetupTimerInterrupt(void) {
4    /* Scale the clock so longer tickless periods can be achieved by dividing
5     the HCLK frequency for the wanted tick frequency (usuallu 1ms).  */
6
7    htimx.Instance = TIMx;
8    htimx.Init.Prescaler = PRESCALER_VALUE;
9    htimx.Init.Period = PERIOD_VALUE
10   HAL_TIM_Base_Init(&htimx);
11
```

[44] In Cortex-M3/4 ports this function is called `vPortSetupTimerInterrupt()`.

```
12    /* Enable the TIMx interrupt.  This must execute at the lowest interrupt priority. */
13    HAL_NVIC_SetPriority(TIMx_IRQn, configLIBRARY_LOWEST_INTERRUPT_PRIORITY, 0);
14    HAL_NVIC_EnableIRQ(TIMx_IRQn);
15
16    /* Start the timer */
17    HAL_TIM_Base_Start_IT(&htimx);
18  }
```

The first routine we are going to override is the `vPortSetupTimerInterrupt()` one. It simply uses one of the available STM32 timers as timebase generator, configuring the right `Period` and `Prescaler` values to achieve a *tick* interrupt with a frequency equal to 1kHz. The timer ISR (shown later) will have the responsibility to increment the global *tick* counter.



Read Carefully

In Chapter 10 we have seen that the HAL is designed to automatically invoke the `SystemCoreClockUpdate()` when we change the HCLK frequency. This ensures us that the *SysTick* interrupt is generated every 1ms even if the core clock changes. If, instead, we use another timer for the RTOS *tick* counter, then it is up to us to carefully ensure that the timer is reconfigured accordingly when the APB bus clock, where the timer belongs to, speed changes.



Read Carefully

In the port file for the Cortex-M0/0+ architecture (**Middlewares/Third_Party/FreeRTOS/Source/portable/GCC/ARM_CM0/port.c**) the function to setup the timer is called `prvSetupTimerInterrupt()` and it is not defined as `__attribute__((weak))`. Change the attribute of the function from `static` to `__attribute__((weak))` so that the one inside the **Core/Src/tickless-mode.c** is called. Refer to the book samples for the complete code.

The next lines of code show a possible implementation for the `vPortSuppressTicksAndSleep()`, which is called when the following two conditions are both true:

1. The *idle* thread is the only thread able to run because all the application threads are either in the *blocked* or in the *suspended* state.
2. At least *n* further complete *tick* periods will pass before the kernel moves an application thread out of the *blocked* state, where *n* is set by the `configEXPECTED_IDLE_TIME_BEFORE_SLEEP` macro in `FreeRTOS.h` file[45].

If the above conditions are satisfied, then the scheduler is suspended and the `vPortSuppressTicksAndSleep()` function is called, allowing us to temporarily suppress the *tick* interrupt or to delay its execution.

[45]This is a user-defined parameter that represents a further delay before to start the *tick* suppression procedure. Since this procedure is computationally intensive, and it may introduce minor shifts in the global *tick* count, we can programmatically decide to wait at least *n* consecutive ticks before starting the procedure.

```
20 /* Override the default definition of vPortSuppressTicksAndSleep() with a version
21  that uses another STM32 timer to derive how long the micro is remained in sleep state */
22 void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime) {
23   unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;
24   eSleepModeStatus eSleepStatus;
25
26   /* Read the current time from the timer configured by the
27    vPortSetupTimerInterrupt() function */
28   ulLowPowerTimeBeforeSleep = __HAL_TIM_GET_COUNTER(TIMx);
29
30   /* Stop the timer that is generating the tick interrupt. */
31   HAL_TIM_Base_Stop_IT(TIMx);
32
33   /* Enter a critical section that will not affect interrupts bringing the MCU
34    out of sleep mode. */
35   __disable_irq();
36
37   /* Ensure it is still ok to enter the sleep mode. */
38   eSleepStatus = eTaskConfirmSleepModeStatus();
39
40   if (eSleepStatus == eAbortSleep) {
41     /* A task has been moved out of the Blocked state since this macro was
42      executed, or a context switch is being held pending.  Do not enter a
43      sleep state.  Restart the tick and exit the critical section. */
44     HAL_TIM_Base_Start_IT (TIMx)
45     __enable_irq();
46   } else {
47     if (eSleepStatus == eNoTasksWaitingTimeout) {
48       /* There are no running state tasks and no tasks that are blocked with a
49        time out.  Assuming the application does not care if the tick time slips
50        with respect to calendar time then enter a deep sleep that can only be
51        woken by another interrupt. */
52       StopMode();
53     } else {
54       /* Configure an interrupt to bring the microcontroller out of its low
55        power state at the time the kernel next needs to execute.  The
56        interrupt must be generated from a source that remains operational
57        when the microcontroller is in a low power state. */
58       vSetWakeTimeInterrupt(xExpectedIdleTime);
59
60       /* Enter the low power state. */
61       SleepMode();
62
63       /* Determine how long the microcontroller was actually in a low power
64        state for, which will be less than xExpectedIdleTime if the
65        microcontroller was brought out of low power mode by an interrupt
66        other than that configured by the vSetWakeTimeInterrupt() call.
```

```
67          Note that the scheduler is suspended before
68          vPortSuppressTicksAndSleep() is called, and resumed when it returns.
69          Therefore no other tasks will execute until this function completes. */
70        ulLowPowerTimeAfterSleep = __HAL_TIM_GET_COUNTER(TIMx);
71
72        /* Correct the kernels tick count to account for the time the
73          microcontroller spent in its low power state. */
74      vTaskStepTick( ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep );
75    }
76
77    /* Exit the critical section - it might be possible to do this immediately
78     after the prvSleep() calls. */
79    __disable_irq();
80
81    /* Restart the timer that is generating the tick interrupt. */
82    HAL_TIM_Base_Stop_IT(TIMx);
83  }
```

The routine starts by saving the current counter value of the timer before it is stopped. All interrupts are disabled to prevent race conditions, entering in a critical section by calling the CMSIS function `__disable_irq()`. As said before, `()` is called when the scheduler is suspended, but an interrupt firing before we enter the critical section at line 35 may ask to the kernel to resume the execution of another thread in blocked state[46]. By calling the `eTaskConfirmSleepModeStatus()` we can know if we need to abort the *tick* suppression procedure, resuming the timer. If the function returns the value `eAbortSleep`, then we restart the *tick* generator timer and we immediately exit from the critical section by re-enabling all interrupts (line 45). If, instead, the function returns the value `eNoTasksWaitingTimeout`, it means that there are no *running* threads, no software timers[47] or other threads blocked with a definite timeout. Since there is no need to preserve the *tick* count accuracy in this case (no timers, no running threads, no timeouts), we can so enter in *stop* mode, which will cause that the timer clock is gated. The MCU will exit from the `StopMode()` routine when an external interrupt wakes up the MCU.

If, instead, the `eTaskConfirmSleepModeStatus()` function returns the value `eStandardSleep`, the `else` at line 53 matches and we can *sleep* for a time equal to the `xExpectedIdleTime` parameter, which corresponds to the total number of tick periods before a thread is moved back into the *ready* state. The parameter value is therefore the time the microcontroller can safely remain in a low-power state, with the tick interrupt temporarily suppressed. The timer ISR will wake up the MCU, exiting from the `SleepMode()` routine and the global *tick* count is adjusted at line 74.

23.8.2.2 A Custom *tickless* Mode Policy

The above pseudo-code represents a schema that all programmers can use to implement their custom *tickless* mode. For example, if we know that our software does not make use of software timers and

[46] This happens because this routine is called within an IRQ with the lowest possible priority, as seen before. So, a more privileged IRQ may resume the execution of another blocked task.

[47] Please, take note that it is not sufficient we do not use timers in our code. The macro `configUSE_TIMERS` in `FreeRTOSConfig.h` must be set to 0, otherwise the `eTaskConfirmSleepModeStatus()` never return the `eNoTasksWaitingTimeout` value.

non-indefinite timeouts, then we can safely handle only the *deep sleep* mode case.

Now we are going to implement a custom *tickless* mode policy, analyzing real code made to work on an STM32F030 MCU. Refer to the book example for other STM32 MCUs, even if the implementation is almost the same.

Filename: `Core/Src/tickless-mode.c`

```
12  #define ulPeriodValueForOneTick   ((1000000U / 1000U) - 1U)
13
14  /* Holds the maximum number of ticks that can be suppressed - which is
15   basically how far into the future an interrupt can be generated without
16   loosing the overflow event at all. It is set during initialization. */
17  static TickType_t xMaximumPossibleSuppressedTicks = 0;
18
19  /* Flag set from the tick interrupt to allow the sleep processing to know if
20   sleep mode was exited because of an tick interrupt or a different interrupt. */
21  static volatile uint8_t ucTickFlag = pdFALSE;
22
23  /* The HAL handler of the TIM2 timer */
24  TIM_HandleTypeDef htim2;
25
26  void xPortSysTickHandler( void );
27
28  /* The callback function called by the HAL when TIM2 overflows. */
29  void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
30    if (htim->Instance == TIM2) {
31      xPortSysTickHandler();
32
33      /* In case this is the first tick since the MCU left a low power mode.
34       The period is so configured by vPortSuppressTicksAndSleep(). Here
35       the reload value is reset to its default. */
36      __HAL_TIM_SET_AUTORELOAD(htim, ulPeriodValueForOneTick);
37      __HAL_TIM_SET_COUNTER(htim, 0);
38
39      /* The CPU woke because of a tick. */
40      ucTickFlag = pdTRUE;
41    } else if (htim->Instance == TIM3) {
42        HAL_IncTick();
43    }
44  }
45  /*-----------------------------------------------------------*/
46
47  /* Override the default definition of vPortSetupTimerInterrupt() that is weakly
48   defined in the FreeRTOS Cortex-M0 port layer with a version that configures TIM2
49   to generate the tick interrupt. */
50  void vPortSetupTimerInterrupt(void) {
51    uint32_t              uwTimclock = 0;
52    uint32_t              uwPrescalerValue = 0;
```

```
53
54    /* Enable the TIM2 clock. */
55    __HAL_RCC_TIM2_CLK_ENABLE();
56
57    /* Ensure clock stops in debug mode. */
58    __HAL_DBGMCU_FREEZE_TIM2();
59
60    /* Compute TIM2 clock */
61    uwTimclock = 2*HAL_RCC_GetPCLK1Freq();
62    /* Compute the prescaler value to have TIM2 counter clock equal to 1MHz */
63    uwPrescalerValue = (uint32_t) ((uwTimclock / 1000000U) - 1U);
64
65    /* Configure the TIM2 timer */
66    htim2.Instance = TIM2;
67    htim2.Init.Period = ulPeriodValueForOneTick;
68    htim2.Init.Prescaler = uwPrescalerValue;
69    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
70    HAL_TIM_Base_Init(&htim2);
71
72    /* Enable the TIM2 interrupt. This must execute at the lowest interrupt priority. */
73    HAL_NVIC_SetPriority(TIM2_IRQn, configLIBRARY_LOWEST_INTERRUPT_PRIORITY, 0);
74    HAL_NVIC_EnableIRQ(TIM2_IRQn);
75
76    HAL_TIM_Base_Start_IT(&htim2);
77    /* See the comments where xMaximumPossibleSuppressedTicks is declared. */
78    xMaximumPossibleSuppressedTicks = ((unsigned long) USHRT_MAX)
79        / ulPeriodValueForOneTick;
80  }
```

The first two functions we are going to analyze are related to the setup of the timer used as *tick* generator and the handling of the related overflow interrupt. The `prvSetupTimerInterrupt()` function is automatically invoked by FreeRTOS when the `osKernelStart()` routine is called. It configures the TIM2 timer so that it expires every 1ms. The corresponding interrupt is enabled, and the ISR priority is set to the lowest one (remember that, unless different needed, it is always important to setup the timer ISR with the lowest priority). The `HAL_TIM_PeriodElapsedCallback()` callback simply increases the global tick count by 1. Don't care about the instructions at lines [36:37] because they will be clear later. The same callback is responsible of the increment of the HAL tick counter (line 42)

Now we are going to analyze the most complex part: the `vPortSuppressTicksAndSleep()` function. We will divide it in blocks, so that it is simpler to analyze its code. It is strongly suggested to keep the real code in the IDE at your hands.

Filename: `Core/Src/tickless-mode.c`

```
89  void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime) {
90    uint32_t ulCounterValue, ulCompleteTickPeriods;
91    eSleepModeStatus eSleepAction;
92    TickType_t xModifiableIdleTime;
93    const TickType_t xRegulatorOffIdleTime = 50;
94
95    /* Make sure the TIM2 reload value does not overflow the counter. */
96    if (xExpectedIdleTime > xMaximumPossibleSuppressedTicks) {
97      xExpectedIdleTime = xMaximumPossibleSuppressedTicks;
98    }
99
100   /* Calculate the reload value required to wait xExpectedIdleTime tick
101    periods. */
102   ulCounterValue = ulPeriodValueForOneTick * xExpectedIdleTime;
103
104   /* To avoid race conditions, enter a critical section.  */
105   __disable_irq();
106
107   /* If a context switch is pending then abandon the low power entry as
108    the context switch might have been pended by an external interrupt that
109    requires processing. */
110   eSleepAction = eTaskConfirmSleepModeStatus();
111   if (eSleepAction == eAbortSleep) {
112     /* Re-enable interrupts. */
113     __enable_irq();
114     return;
115   } else if (eSleepAction == eNoTasksWaitingTimeout) {
116     /* Stop TIM2 */
117     HAL_TIM_Base_Stop_IT(&htim2);
118
119     /* A user definable macro that allows application code to be inserted
120      here.  Such application code can be used to minimize power consumption
121      further by turning off IO, peripheral clocks, the Flash, etc. */
122     configPRE_STOP_PROCESSING();
123
124
125     /* There are no running state tasks and no tasks that are blocked with a
126      time out.  Assuming the application does not care if the tick time slips
127      with respect to calendar time then enter a deep sleep that can only be
128      woken by (in this demo case) the user button being pushed on the
129      STM32L discovery board.  If the application does require the tick time
130      to keep better track of the calendar time then the RTC peripheral can be
131      used to make rough adjustments. */
132     HAL_PWR_EnterSTOPMode(PWR_MAINREGULATOR_ON, PWR_STOPENTRY_WFI);
133
134     /* A user definable macro that allows application code to be inserted
```

```
135       here.  Such application code can be used to reverse any actions taken
136       by the configPRE_STOP_PROCESSING().  In this demo
137       configPOST_STOP_PROCESSING() is used to re-initialize the clocks that
138       were turned off when STOP mode was entered. */
139       configPOST_STOP_PROCESSING();
140
141       /* Restart tick. */
142       HAL_TIM_Base_Start_IT(&htim2);
143
144       /* Re-enable interrupts. */
145       __enable_irq();
```

The function starts checking if the expected idle time, that is the time window within we can safely stop the *tick* generation, is less than the `xMaximumPossibleSuppressedTicks`: this value is computed inside the `prvSetupTimerInterrupt()` routine according to the given `Prescaler` and `Period` values. Then, at line 102, it computes the `Period` value to use so that the timer will overflow after the `xExpectedIdleTime` time. To avoid race conditions, we then enter in a critical section (line 105) and we invoke the `eTaskConfirmSleepModeStatus()` to decide how to proceed in the tick suppression procedure. If the function returns `eNoTasksWaitingTimeout`, then we can stop the TIM2 timer at all, and we can enter in *stop* mode until the MCU is woken up by an event or an interrupt.

Filename: `Core/Src/tickless-mode.c`

```
147     else {
148       /* Stop TIM2 momentarily.  The time TIM2 is stopped for is not accounted for
149        in this implementation (as it is in the generic implementation) because the
150        clock is so slow it is unlikely to be stopped for a complete count period
151        anyway. */
152       HAL_TIM_Base_Stop_IT(&htim2);
153
154       /* The tick flag is set to false before sleeping.  If it is true when sleep
155        mode is exited then sleep mode was probably exited because the tick was
156        suppressed for the entire xExpectedIdleTime period. */
157       ucTickFlag = pdFALSE;
158
159       /* Trap underflow before the next calculation. */
160       configASSERT(ulCounterValue >= __HAL_TIM_GET_COUNTER(&htim2));
161
162       /* Adjust the TIM2 value to take into account that the current time
163        slice is already partially complete. */
164       ulCounterValue -= (uint32_t) __HAL_TIM_GET_COUNTER(&htim2);
165
166       /* Trap overflow/underflow before the calculated value is written to TIM2. */
167       configASSERT(ulCounterValue < ( uint32_t ) USHRT_MAX);
168       configASSERT(ulCounterValue != 0);
169
170       /* Update to use the calculated overflow value. */
```

```
171     __HAL_TIM_SET_AUTORELOAD(&htim2, ulCounterValue);
172     __HAL_TIM_SET_COUNTER(&htim2, 0);
173
174     /* Restart the TIM2. */
175     HAL_TIM_Base_Start_IT(&htim2);
176
177     /* Allow the application to define some pre-sleep processing.  This is
178      the standard configPRE_SLEEP_PROCESSING() macro as described on the
179      FreeRTOS.org website. */
180     xModifiableIdleTime = xExpectedIdleTime;
181     configPRE_SLEEP_PROCESSING( xModifiableIdleTime );
182
183     /* xExpectedIdleTime being set to 0 by configPRE_SLEEP_PROCESSING()
184      means the application defined code has already executed the wait/sleep
185      instruction. */
186     if (xModifiableIdleTime > 0) {
187       /* The sleep mode used is dependent on the expected idle time
188        as the deeper the sleep the longer the wake up time.  See the
189        comments at the top of main_low_power.c.  Note xRegulatorOffIdleTime
190        is set purely for convenience of demonstration and is not intended
191        to be an optimized value. */
192     if (xModifiableIdleTime > xRegulatorOffIdleTime) {
193         /* A slightly lower power sleep mode with a longer wake up time. */
194         HAL_PWR_EnterSLEEPMode(PWR_LOWPOWERREGULATOR_ON, PWR_SLEEPENTRY_WFI);
195     } else {
196         /* A slightly higher power sleep mode with a faster wake up time. */
197       HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);
198      }
199   }
```

If the `eTaskConfirmSleepModeStatus()` returns `eStandardSleep`, then we can enter in *sleep* mode. The timer is stopped, and its `Period` is set (at line 171) to the value computed before (at line 164). The `configPRE_SLEEP_PROCESSING()` is a macro we can implement to perform operations preliminary to the sleep mode (for example, in some STM32 MCUs it is required to lower the clock speed, or we could use this macro to turn OFF unneeded peripherals). We can so enter in *sleep* mode or in *low-power sleep* mode, according to the computed sleep time (in some STM32 MCUs exiting from *low-power sleep* requires more time that would waste a lot of power uselessly if the sleeping period is too short).

Filename: `Core/Src/tickless-mode.c`

```
201     /* Allow the application to define some post sleep processing.  This is
202      the standard configPOST_SLEEP_PROCESSING() macro, as described on the
203      FreeRTOS.org website. */
204     configPOST_SLEEP_PROCESSING( xModifiableIdleTime );
205
206     /* Re-enable interrupts. If the timer has overflowed during this period
207      then this will cause that the TIM2_IRQHandler() is called. So the
208      global tick counter is incremented by 1 and the ulTickFlag variable
209      is set to pdTRUE.
210      Take note that in the STM32L example in the official FreeRTOS
211      distribution interrupts are re-enabled after the TIM2 is stopped.
212      This is wrong, because it causes that the IRQ is leaved pending,
213      even if has been set. So we must first re-enable interrupts - this
214      causes that a pending TIM2 IRQ fires - and then stop the timer. */
215     __enable_irq();
216
217     /* Stop TIM2.  Again, the time the clock is stopped for in not accounted
218      for here (as it would normally be) because the clock is so slow it is
219      unlikely it will be stopped for a complete count period anyway. */
220     HAL_TIM_Base_Stop_IT(&htim2);
221
222     if (ucTickFlag != pdFALSE) {
223       /* The MCU has been woken up by the TIM2. So we trap overflows
224        before the next calculation. */
225       configASSERT(
226           ulPeriodValueForOneTick >= (uint32_t ) __HAL_TIM_GET_COUNTER(&htim2));
227
228       /* The tick interrupt has already executed, although because this
229        function is called with the scheduler suspended the actual tick
230        processing will not occur until after this function has exited.
231        Reset the reload value with whatever remains of this tick period. */
232       ulCounterValue = ulPeriodValueForOneTick
233           - (uint32_t) __HAL_TIM_GET_COUNTER(&htim2);
234
235       /* Trap under/overflows before the calculated value is used. */
236       configASSERT(ulCounterValue <= ( uint32_t ) USHRT_MAX);
237       configASSERT(ulCounterValue != 0);
238
239       /* Use the calculated reload value. */
240       __HAL_TIM_SET_AUTORELOAD(&htim2, ulCounterValue);
241       __HAL_TIM_SET_COUNTER(&htim2, 0);
242
243       /* The tick interrupt handler will already have pended the tick
244        processing in the kernel.  As the pending tick will be processed as
245        soon as this function exits, the tick value  maintained by the tick
246        is stepped forward by one less than the  time spent sleeping.  The
```

```
247         actual stepping of the tick appears later in this function. */
248       ulCompleteTickPeriods = xExpectedIdleTime - 1UL;
249     } else {
250       /* Something other than the tick interrupt ended the sleep.  How
251        many complete tick periods passed while the processor was
252        sleeping? */
253       ulCompleteTickPeriods = ((uint32_t) __HAL_TIM_GET_COUNTER(&htim2))
254           / ulPeriodValueForOneTick;
255
256       /* Check for over/under flows before the following calculation. */
257       configASSERT(
258           ((uint32_t ) __HAL_TIM_GET_COUNTER(&htim2)) >= (ulCompleteTickPeriods * ulPeriodValu\
259 eForOneTick));
260
261       /* The reload value is set to whatever fraction of a single tick
262        period remains. */
263       ulCounterValue = ((uint32_t) __HAL_TIM_GET_COUNTER(&htim2))
264           - (ulCompleteTickPeriods * ulPeriodValueForOneTick);
265       configASSERT(ulCounterValue <= ( uint32_t ) USHRT_MAX);
266       if (ulCounterValue == 0) {
267         /* There is no fraction remaining. */
268         ulCounterValue = ulPeriodValueForOneTick;
269         ulCompleteTickPeriods++;
270       }
271       __HAL_TIM_SET_AUTORELOAD(&htim2, ulCounterValue);
272       __HAL_TIM_SET_COUNTER(&htim2, 0);
273     }
274
275     /* Restart TIM2 so it runs up to the reload value.  The reload value
276      will get set to the value required to generate exactly one tick period
277      the next time the TIM2 interrupt executes. */
278     HAL_TIM_Base_Start_IT(&htim2);
279
280     /* Wind the tick forward by the number of tick periods that the CPU
281      remained in a low power state. */
282     vTaskStepTick(ulCompleteTickPeriods);
283   }
284 }
```

When the MCU exists from the *sleep* mode, either because the timer has overflowed or another interrupt has been generated, the `configPOST_SLEEP_PROCESSING()` macro allows us to perform needed operations, such as restoring some peripherals or increasing the clock speed. Now the tricky part takes place, and we need to careful explain the operation involved.

After the MCU ha exited from low-power mode, ISRs are unmasked by exiting critical section (line 215). This will cause that the `TIM2_IRQHandler()` ISR is called **if we have exited from the *sleep* mode due to a timer overflow**. When this happens the `HAL_TIM_PeriodElapsedCallback()`

function is called: this causes that the `ucTickFlag` is set to `TRUE` and the timer `Period` needs to be set accordingly. If, instead, the MCU has exited from the low-power mode for another reason (for example, it has been awakened by an EXTI interrupt), the `ucTickFlag` is equal to `FALSE`.

The code checks the status of the `ucTickFlag` at line 222. If it is equal to `TRUE`, then the global *tick* counter is increased for a value equal to `xExpectedIdleTime` minus one, because the *tick* counter has been already incremented by the `HAL_TIM_PeriodElapsedCallback()` routine by one (the ISR is called as soon as we leave the critical section at line 215). If, instead, it is equal to `FALSE`, then we compute how long the MCU has spent in *sleep* mode, and we increase the *tick* counter accordingly.

This policy could be adapted according to your actual needs. For example, if you are working on an STM32L platform you may consider using a LPTIM timer during the *stop* mode, so that you can know how many ticks are elapsed during the *stop* mode (a regular STM32 timer do not work in *stop* mode).



A Note About LPTIM Timers

I have spent a lot of time trying to use a LPTIM timer as timebase generator. While it works well as a regular timer, I reached to the conclusion that LPTIM timers are not suitable to be used with the *tickless* mode, because they are implemented so that reading the value of the counter register (`LPTIM->CNT`) is not reliable, especially when the timer exits from deeper low-power modes. This is clearly stated in the official STM32 documentation, and it constitutes a severe limit of this peripheral, according to this author.

23.9 Debugging Features

The debugging of a firmware built using an RTOS could not be trivial. *Context switches* can make complicated to perform step-by-step debugging. FreeRTOS offers some debugging features, and some of them are useful especially when your design uses a lot of threads spawned dynamically.

23.9.1 `configASSERT()` Macro

FreeRTOS source code is full of calls to the macro `configASSERT()`. This is an empty macro that developers can define inside the `FreeRTOSConfig.h`, and it plays the same role of the C `assert()` function. CubeMX automatically defines it for us in the following way:

```
#define configASSERT( x ) if ((x) == 0) {taskDISABLE_INTERRUPTS(); for( ;; );}
```

The macro works so that if the assert condition is false then all interrupts are disabled (by setting the `PRIMASK` register on Cortex-M0/0+ cores and rising the `BASEPRI` value in other STM32 MCUs) and an infinite loop takes place. While this behaviour is ok during a debug session, it can be a source of a lot of headaches if our device is not running under a debugger, because it is hard to say why the firmware stopped working. So, this author prefers to define the macro in this other way:

```
void __configASSERT(uint8_t x) {
  if ((x) == 0) {
      taskDISABLE_INTERRUPTS();
      if((CoreDebug->DHCSR & 0x1) == 0x1) { /* If under debug */
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        HAL_Delay(1000);
        asm("BKTP #0");
      } else {
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        HAL_Delay(100);
      }
  }
}

#define configASSERT( x ) __configASSERT(x)
```

The `__configASSERT()` function uses the Cortex-M *CoreDebug interface* to check if the MCU is under debug: debuggers set the first bit of the *Debug Halting Control and Status Register* (DHCSR) when the MCU is under debugging. If so, a software breakpoint is placed when the assert condition is false. However, this function has two relevant limitations:

- it works only on Cortex-M3/4/7 based microcontrollers;
- the DHCSR register is not reset to zero when a system reset occurs, neither it is possible to clear the first bit within the firmware; this means that we need to completely power OFF the device, otherwise the firmware will stick if the assert condition is false.

23.9.2 Run-Time Statistics and Thread State Information

When threads are spawned dynamically, it is hard to keep track of their lifecycle. FreeRTOS provides a way to retrieve both the complete list of live threads and some relevant information regarding their status.

The `uxTaskGetNumberOfTasks()` function returns the number of live threads. With the term *live threads* we mean all threads effectively allocated by the kernel, even those ones marked as *deleted*[48]. The function

```
UBaseType_t uxTaskGetSystemState(TaskStatus_t * const pxTaskStatusArray,
            const UBaseType_t uxArraySize, unsigned long * const pulTotalRunTime );
```

returns the status information of every thread in the system, by populating an instance of the `TaskStatus_t` structure for each thread. The `TaskStatus_t struct` is defined in the following way:

[48]Deleted threads usually persist in memory for a short time. When a thread is marked for deletion, it is effectively moved out from the system by the *idle* thread.

```
typedef struct xTASK_STATUS {
  TaskHandle_t xHandle;          /* The handle of the thread to which the rest of the
                                    information in the structure relates */
  const char *pcTaskName;        /* A pointer to the thread's name */
  UBaseType_t xTaskNumber;       /* Corresponds to Thread ID */
  eTaskState eCurrentState;      /* The state in which the thread existed when the
                                    structure was populated */
  UBaseType_t uxCurrentPriority; /* The priority at which the thread was running */
  UBaseType_t uxBasePriority;    /* The priority to which the thread will return
                                    if the thread's current priority has been inherited
                                    to avoid unbounded priority inversion when obtaining
                                    a mutex. Only valid if configUSE_MUTEXES is defined
                                    as 1 in FreeRTOSConfig.h. */
  uint32_t ulRunTimeCounter;     /* The total run time allocated to the thread so far,
                                    as defined by the run time stats clock. */
  uint16_t usStackHighWaterMark; /* The minimum amount of stack space that has remained
                                    for the thread since the thread was created */
} TaskStatus_t;
```

The `uxTaskGetSystemState()` accepts a pre-allocated array containing the instances of `TaskHandle_t` structures for each thread, the maximum number of elements that the array can hold (`uxArraySize`) and a pointer to a variable (`pulTotalRunTime`) that will contain the total run-time since the kernel started. FreeRTOS, in fact, can optionally collect information on the amount of processing time that has been used by each thread. The run-time statistics must be explicitly enabled by defining the `configGENERATE_RUN_TIME_STATS` macro in the `FreeRTOSConfig.h`. Moreover, this feature requires that we use another timer different from the one used to feed the *tick* counter. This because the run-time statistics timebase needs to have a higher resolution than the tick interrupt, otherwise the statistics may be too inaccurate to be truly useful.

If thread functions are well designed, and they do not make use of busy loops, usually a thread lasts for less than the *tick* time, which is equal to 1ms, and it represents the maximum slice time dedicated to a thread. However, the run-time statistics work so that before the thread is moved in *running* state the current value of the timer used for statistics is saved. When a thread exits from the *running* state (either because it yields the control, or its quantum time is over) a comparison is performed between the previous saved time and the current one. If the *tick* timer is used for this operation, this difference is always equal to zero. For this reason, it is recommended to configure the timebase generator for statistics between 10 and 100 times faster than the *tick* interrupt. The faster the timebase the more accurate the statistics will be - but also the sooner the timer value will overflow.

When the `configGENERATE_RUN_TIME_STATS` macro is set to 1, we have to provide two additional macros. The first one, `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()`, is used to setup the timer needed for run-time statistics. The second one, `portGET_RUN_TIME_COUNTER_VALUE()`, is used by FreeRTOS to retrieve the cumulative value of the timer counter. Since this timer needs to run fast, it is not suggested to setup its ISR and to increase a global variable when it expires: this would affect the overall system performance. In STM32 MCUs providing a 32-bit timer it is sufficient to use one

of these, setting the `Period` to the maximum value (`0xFFFFFFFF`). Another alternative, on Cortex-M3/4/7 consists in using the DWT cycle counter, as seen in Chapter 11. The following code shows a possible implementation for the two macros:

```
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()  \
  do {                                            \
    DWT->CTRL |= 1 ; /* enable the counter */     \
    DWT->CYCCNT = 0;                              \
  }while(0)

#define portGET_RUN_TIME_COUNTER_VALUE() DWT->CYCCNT
```

We are now going to analyze a complete tracing implementation, which consists in having a dedicated thread that prints on the UART2 interface statistic information when the Nucleo USER button is pressed.

Filename: `Core/Src/main-ex8.c`

```
48 void threadsDumpThread(void *argument) {
49  TaskStatus_t *pxTaskStatusArray = NULL;
50  char *pcBuf = NULL;
51  char *pcStatus;
52  uint32_t ulTotalRuntime;
53
54   while(1) {
55     if(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET) {
56       /* Allocate the message buffer. */
57       pcBuf = pvPortMalloc(100 * sizeof(char));
58
59       /* Allocate an array index for each task. */
60       pxTaskStatusArray = pvPortMalloc( uxTaskGetNumberOfTasks() * sizeof( TaskStatus_t ) );
61
62       if( pcBuf != NULL && pxTaskStatusArray != NULL ) {
63         /* Generate the (binary) data. */
64         uxTaskGetSystemState( pxTaskStatusArray, uxTaskGetNumberOfTasks(), &ulTotalRuntime );
65
66         sprintf(pcBuf, "         LIST OF RUNNING THREADS(%lu)         \r\n--------------------\
67 --------------------\r\n", uxTaskGetNumberOfTasks());
68         HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
69
70         for(uint16_t i = 0; i < uxTaskGetNumberOfTasks(); i++ )
71         {
72           sprintf(pcBuf, "Thread: %s\r\n", pxTaskStatusArray[i].pcTaskName);
73           HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
74
75           sprintf(pcBuf, "Thread ID: %lu\r\n", pxTaskStatusArray[i].xTaskNumber);
76           HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
```

```
77
78            switch (pxTaskStatusArray[i].eCurrentState) {
79            case eRunning:
80              pcStatus = "RUNNING";
81              break;
82            case eReady:
83              pcStatus = "READY";
84              break;
85            case eBlocked:
86              pcStatus = "BLOCKED";
87              break;
88            case eSuspended:
89              pcStatus = "SUSPENDED";
90              break;
91            case eDeleted:
92              pcStatus = "DELETED";
93              break;
94
95            default: /* Should not get here, but it is included
96                        to prevent static checking errors. */
97              pcStatus = "UNKNOWN";
98              break;
99            }
100
101           sprintf(pcBuf, "\tStatus: %s\r\n", pcStatus);
102           HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
103
104           sprintf(pcBuf, "\tStack watermark number: %d\r\n", pxTaskStatusArray[i].usStackHighW\
105 aterMark);
106           HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
107
108           sprintf(pcBuf, "\tPriority: %lu\r\n", pxTaskStatusArray[i].uxCurrentPriority);
109           HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
110
111           sprintf(pcBuf, "\tRun-time time: %lu\r\n", pxTaskStatusArray[i].ulRunTimeCounter);
112           HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
113
114           float data = (float)(((float)pxTaskStatusArray[i].ulRunTimeCounter)/ulTotalRuntime)*\
115 100;
116           sprintf(pcBuf, "\tRun-time time in percentage: %lu%%\r\n", (uint32_t)data);
117           HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
118         }
119
120         vPortFree(pcBuf);
121         vPortFree(pxTaskStatusArray);
122       }
123     }
```

```
124     osDelay(50);
125   }
126 }
```

The code should be easy to understand. When the USER button is pressed, this thread allocates a buffer (`pxTaskStatusArray`) that will contain the `TaskStatus_t` structures for each thread in the system. The `uxTaskGetSystemState()` at line 64 populates this array, and for each thread contained in it some statistics are printed on the Nucleo VCP.

Whereas `uxTaskGetSystemState()` populates a `TaskStatus_t` structure for each thread in the system, `vTaskGetInfo()` populates a `TaskStatus_t` structures for just a single task, and it can be useful if we want retrieve information about a specific thread.

Finally, FreeRTOS provides some convenient routines to automatically format the raw data statistics into a human readable (ASCII) format. For example, the `vTaskGetRunTimeStats()` formats the raw data generated by `uxTaskGetSystemState()` into a human readable (ASCII) table that shows the amount of time each task has spent in the *running* state (how much CPU time each task has consumed). For more information, refer to this page[49] of the on-line FreeRTOS documentation.

23.9.3 FreeRTOS Debugging in STM32CubeIDE

The STM32CubeIDE provides some advanced debugging features allowing to debug with dedicated views the following FreeRTOS objects:

- Tasks
- Semaphores
- Timers
- Queues

To enable the additional debug views you can click, from the debug perspective, on the menu **Window->Show View->FreeRTOS**, as shown in **Figure 23.22**.

To collect information from the running firmware, the views need some support from FreeRTOS.

- The `configUSE_TRACE_FACILITY` macro needs to be defined and set to `1`. This allows the **FreeRTOS Task List** view to collect additional information regarding the status of the thread's stack, as shown in **Figure 23.23**, and to show the complete list of defined semaphores.
- The firmware needs to call the FreeRTOS routine `vQueueAddToRegistry(QueueHandle_t xQueue, const char *pcQueueName )` function to make the **FreeRTOS Queues** and **FreeRTOS Semaphores** views able to display objects. The function adds an object to the FreeRTOS Queue registry and takes two parameters, the first is the handle of the queue, and the second is a description of the queue, which is presented in FreeRTOS-related views.

[49] http://www.freertos.org/rtos-run-time-stats.html

- To get valid RTOS run-time statistics, the application must set up a run-time statistics time base. The time-base clock is recommended to run at least 10 times faster than the frequency of the clock used to handle the RTOS tick interrupt. To enable the FreeRTOS collection of run-time you need to set the macros `configGENERATE_RUN_TIME_STATS`, `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()` and `portGET_RUN_TIME_COUNTER_VALUE()`. Refer to the previous paragraph to know how to setup these macros.

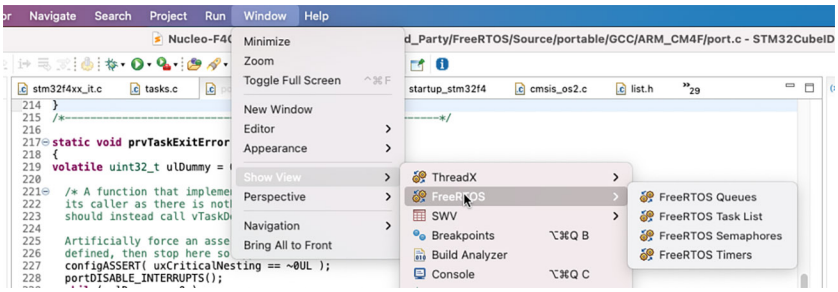


Figure 23.22: How to enable FreeRTOS specific debug views

Name	Description
(No title)	A green arrow symbol indicates the task currently running.
Name	The name assigned to the task with `osThreadAttr_t.name`.
Priority (Base/Actual)	The task base priority and actual priority. The base priority is the priority assigned to the task with `osThreadAttr_t.priority` or the `osThreadSetPriority()` function. The actual priority is a temporary priority assigned to the task due to the priority inheritance mechanism.
Start of Stack	The address of the stack region assigned to the task.
Top of Stack	The address of the saved task stack pointer.
State	The current state of the task.
Event Object	The name of the resource that has caused the task to be blocked.
Min Free Stack	The stack "high watermark" (requires the FreeRTOS option `configUSE_TRACE_FACILITY(1)`). Displays the minimum number of bytes left on the stack for a task. A value of 0 (most likely) indicates that a stack overflow has occurred. *Note: This feature must be enabled in the "View" toolbar.*
Run Time (%)	The run time statistics provide information on the percentage of time the task has been used. This can be used for profiling the system during development.

Table 23.3: Description of the columns in the FreeRTOS Task List view

FreeRTOS Task List View

The **FreeRTOS Task List** view displays detailed information regarding all available tasks in the target system. The task list is updated automatically each time the target execution is paused. There is one column for each type of task parameter, and one row for each task. If the value of any parameter for a task has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow, as shown in the example in **Figure 23.23**. Due to performance reasons, stack analysis (the **Min Free Stack** column) is disabled by default. To enable stack analysis click on the **Toggle Stack Checking** button, circled in red in **Figure 23.23**. When FreeRTOS is configured with `configRECORD_STACK_HIGH_ADDRESS` set to `1`, the column **Min Free Stack** is changed to **Stack Usage**: in this case it displays the stack usage in detailed format as **Used/Total(%Used)**.
The column information in the **FreeRTOS Task List** view is described in **Table 23.3**.

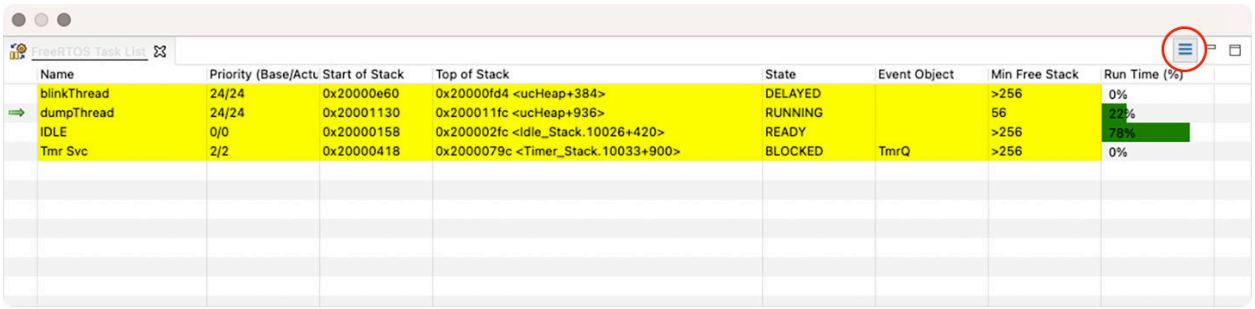


Figure 23.23: The FreeRTOS Task List view

FreeRTOS Timers view

The **FreeRTOS Timers** view displays detailed information regarding all available software timers in the target system. The view is updated automatically each time the target execution is paused. There is one column for each type of timer parameter, and one row for each timer. If the value of any parameter for a timer has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.
The column information in the FreeRTOS Timers view is described in **Table 23.4**.

Name	Description
Name	The name assigned to the timer with `osTimerAttr_t.name`.
Active	The active status information.
Period	The time (in ticks) between timer start and the execution of the callback function.
Type	The type of timer. Auto-reload timers are automatically reactivated after expiration. One-shot timers expire only once.
Id	The timer identifier.
Callback	The address and name of the callback function executed when the timer expires.

Table 23.4: Description of the columns in the FreeRTOS Timers view

FreeRTOS Semaphores view

The **FreeRTOS Semaphores** view displays detailed information regarding all available synchronization objects in the target system, including:

- Mutexes
- Counting semaphores
- Binary semaphores
- Recursive semaphores

The view is updated automatically each time the target execution is paused. There is one column for each type of semaphore parameter, and one row for each semaphore. If the value of any parameter for a semaphore has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.
The column information in the FreeRTOS Semaphores view is described in **Table 23.5**.

Name	Description
Name	The name assigned to the semaphore with `osSemaphoreAttr_t.name`.
Address	The address of the object.
Type	The type of the object.
Size	The maximum number of owning tasks.
Free	The number of free slots currently available.
#Blocked tasks	The number of tasks currently blocked waiting for the object.

Table 23.5: Description of the columns in the FreeRTOS Semaphores view

FreeRTOS Queues view

The **FreeRTOS Queues** view displays detailed information regarding all available queues in the target system. The view is updated automatically each time the target execution is paused. There is one column for each type of queue parameter, and one row for each queue. If the value of any parameter for a queue has changed since the last time the debugger was suspended, the corresponding row is highlighted in yellow.
The column information in the FreeRTOS Queues view is described in **Table 23.6**.

Name	Description
Name	The name assigned to the queue in the queue registry by using the function with `vQueueAddToRegistry()`.
Address	The address of the queue.
Max Length	The maximum number of items that the queue can hold.
Item Size	The size in bytes of each queue item.
Current Length	The number of items currently in the queue.
#Waiting Tx	The number of tasks currently blocked waiting to be sent to the queue.
#Waiting Rx	The number of tasks currently blocked waiting to be received from the queue.

Table 23.6: Description of the columns in the FreeRTOS Queues view

23.9.4 FreeRTOS Kernel-Aware Debugging in STM32CubeIDE

The FreeRTOS kernel-aware debugging is an important feature provided in the STM32CubeIDE useful when the firmware architecture requires several threads dynamically spawned upon specific hardware and fast events hard to track. When FreeRTOS-kernel-aware debugging is enabled and a debug session is started, all threads are listed in the Debug view, as shown in **Figure 23.24**. By selecting a thread in the Debug view, the thread current context is visualized in views. For instance, the Variables, Registers, Editor views reflect the active stack frame.

The Kernel-Aware debugging requires a dedicated support from the debugger, with a special proxy daemon spawned automatically by the STM32CubeIDE. To enable the FreeRTOS-kernel-aware debugging the Debugger tab in the Debug Configurations dialog contains settings to enable RTOS proxy, as shown in **Figure 23.25**. Once the **Enable RTOS Proxy** flag is checked you have to select the driver for FreeRTOS and the Cortex-M core corresponding the target MCU. It is strongly suggested to leave the port number to `60000`.

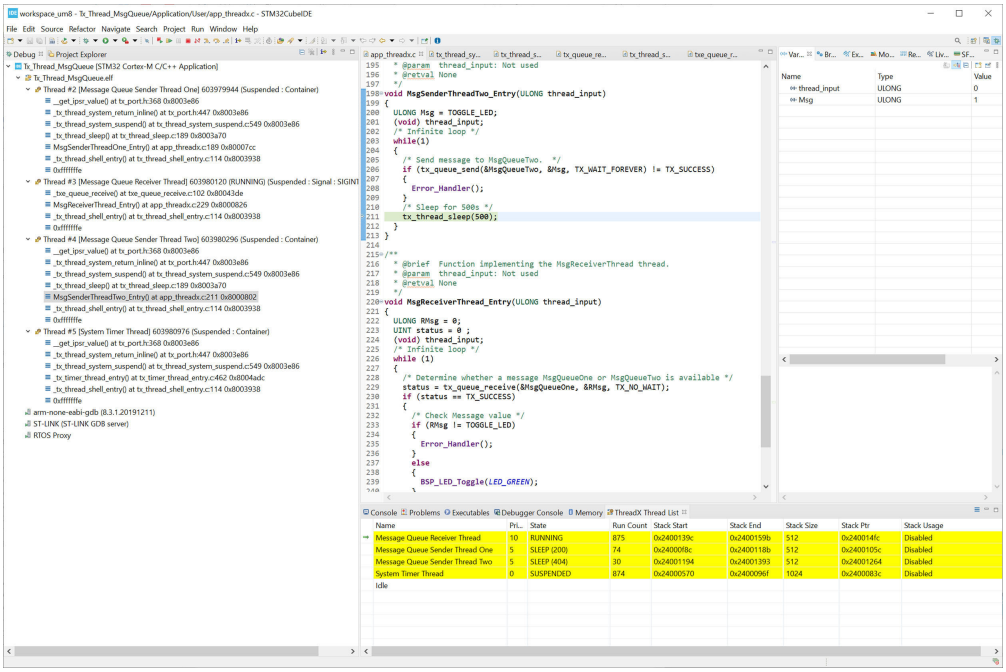


Figure 23.24:

At the time of writing this chapter (January 2022) there are some important limitations to take in account:

- Live expressions must be disabled when used with the ST-LINK GDB server (see **Figure 23.25**).
- The Registers view content for swapped out threads is intermixed with active CPU context for some registers (all registers are not saved by the context switcher).
- The floating point registers in the Registers view are not updated correctly.

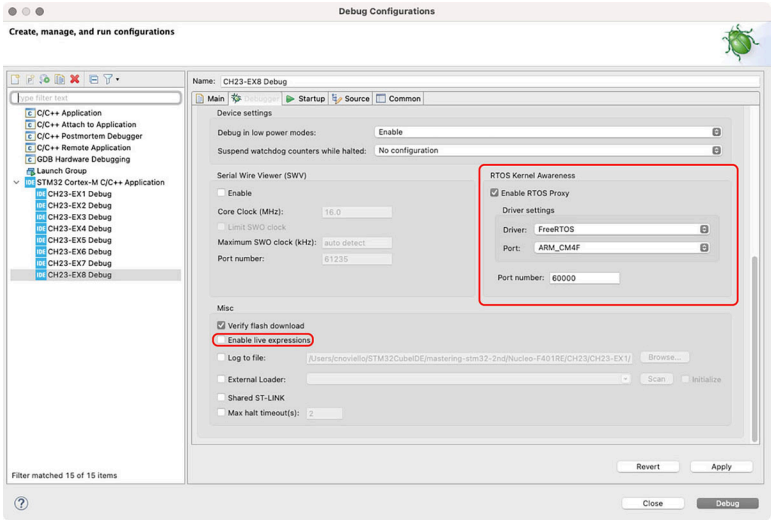


Figure 23.25:



FPU Support in Cortex-M4F and Cortex-M7 Cores

Cortex-M4F or Cortex-M7 architectures provide a dedicate *Floating Point Unit* (FPU), which allows to process floating point operations directly in hardware, without the need of dedicated, and necessarily slow, functions provided by the C *run-time* library. Processors equipped with an FPU unit implement additional hardware registers that need to be saved during a *context switch* operation. For this reason, the FreeRTOS GCC port for M4F/7 architectures expects that the FPU is enabled, which by default is disabled.

To enable it, go in the **Project Settings->C/C++ Build->Settings->MCU Settings** section and select the entry **FPv4-SP-D16** in the **Floating-point unit** field (or **FPv5-SP-D16**[50] if you have a Cortex-M7 based microcontroller), and for the **Hardware implementation** in the **Float-point ABI** field. In case you are working on more recent STM32F76xx/STM32H7xx MCUs, which provide a double precision FPU unit, then you have to select the **FPv5-D16** entry.
Now you have to rebuild the whole source tree.

However, support to the FPU needs to be explicitly enabled in FreeRTOS by settings the macro `configENABLE_FPU` in the **Core/Src/FreeRTOSConfig.h** file to `1`.

23.10 Alternatives to FreeRTOS

As stated in the introduction to this book, there are several good alternatives to FreeRTOS on the market. Here you will find some words about other good RTOS available for the STM32 platform.

23.10.1 AzureRTOS

23.10.2 ChibiOS

If you are not new to the STM32 platform, probably you already know about ChibiOS[51]. ChibiOS is an independent and open source project started by an STMicroelectronics engineer, Giovanni Di Sirio, who works at the ST site in Naples (Italy). ChibiOS is quite popular in the STM32 community, since Giovanni has a deep knowledge of the STM32 platform, and this has allowed to him to create probably one of the most optimized solutions for STM32 MCUs. However, ChibiOS is designed to run on any MCU architecture apart from STM32.

ChibiOS is essentially composed by two layers: the kernels (named ChibiOS/RT or ChibiOS/NIL) and a complete HAL (named Chibios/HAL), which allows to abstract from the underlying hardware peculiarities. While it is perfectly possible to mix the official ST CubeHAL with the ChibiOS/RT/NIL kernels, probably the ChibiOS/HAL is a simpler solution to program STM32 devices, at least for the supported peripherals. Even if this author does not have a direct experience with it, ChibiOS has a

[50] **FPv4-SP-D16** means that the MCU implements a floating-point unit conforming to the *VFPv4-D16* architecture, single precision (*SP*), while **FPv5-SP-D16** refers to the *VFPv5-D16* architecture, single precision (*SP*).
[51] http://www.chibios.org/

good reputation among a lot of people he knows and some readers of this book. Moreover, you can find several projects and good tutorials around in the web[52] based on this RTOS and its related HAL. Chibios uses a full static memory allocation model, allowing to use it in those application domains where dynamic allocation is prohibited. Finally, Giovanni also provides a pre-configured version of Eclipse, named ChibiStudio, which ships all required tools (GCC tool-chain, OpenOCD, etc.) already pre-configured. It runs just in Windows and Linux OSes at the time of writing this chapter.

ChibiOS uses a mixed licensing model[53]. ChibiOS RT and NIL kernels are distributed under the GPL 3 license, HAL is distributed under the more permissive Apache 2.0 license. GPL 3 prevents the usage of the software if you sell electronic devices without releasing the firmware source code publicly. A "free commercial license" exists that removes GPL 3 for commercial users. This license requires a registration process, and it is valid for 500 MCU cores. The free license can be renewed indefinitely by just re-submitting the request form for extra 500 cores.

23.10.3 Contiki OS

Contiki[54] is another open source RTOS, which has a strong accent on wireless low-power sensors and IoT devices. It is a project started by Adam Dunkels in 2003, but it is currently supported by several large companies including Texas Instruments and Atmel. It is quite popular among CC2xxx devices from TI. It is based on a kernel scheduler and an independent TCP/IP stack designed for low-resources devices, which provides IPv4 networking, the *uIPv6* stack and the Rime stack, which is a set of custom lightweight networking protocols designed for low-power wireless networks. The IPv6 stack was contributed by Cisco and was, when released, the smallest IPv6 stack to receive the *IPv6 Ready* certification. The IPv6 stack also contains the Routing Protocol for Low power and Lossy Networks (RPL) routing protocol for low-power lossy IPv6 networks and the 6LoWPAN header compression and adaptation layer for IEEE 802.15.4 links.

ST provides an application note, the UM2000[55], which describes how to get started with the Contiki OS on its microcontrollers, in conjunction with the SPIRIT transceiver to develop sub-1GHz wireless devices.

Contiki is distributed with a BSD-style license, which allows to use its source code in commercial applications without any form of limitations.

23.10.4 OpenRTOS

OpenRTOS is the commercial edition of FreeRTOS, described in this chapter and officially supported by ST. OpenRTOS and FreeRTOS share the same code base. The additional value offered by OpenRTOS is a "commercial and legal wrapper" for FreeRTOS users.

Developers upgrade to an OpenRTOS license for two main reasons: the ability to sell their devices and/or to ship derived code without having to share source code publicly, and the dedicated support

[52] http://www.playembedded.org/
[53] https://bit.ly/3fT572c
[54] http://www.contiki-os.org/
[55] https://bit.ly/1URnLZc

in developing custom solutions based on OpenRTOS. For large companies the possibility to receive paid support is important.

24. Advanced Debugging Techniques

In Chapter 5 we started analyzing basic tools and techniques to debug the firmware running on a target microcontroller. We studied some important Eclipse features, like breakpoints and step-by-step debugging, useful to understand what's going wrong with our code. Those techniques, however, could be not sufficient to debug real-life applications. Things can go wrong in several ways, and it is quite common the need of dedicated, and often expensive, hardware tools to better debug our embedded applications.

This chapter aims to introduce the reader to some advanced debugging capabilities offered by Cortex-M based microcontrollers. The role of Cortex-M exceptions is finally presented, showing how to decode core registers that can provide useful information about the exception source. This chapter also provides a brief introduction to the *Serial Wire Viewer* feature implemented in Cortex-M3/4/7 MCUs, a distinctive ARM technology that allows to perform real-time tracing of the MCU activities using an external debugger tool.

This chapter is not limited to low-level debugging techniques. We will also see in action some other features offered by the STM32CubeIDE tool-chain, like **Live Expressions** and **SFRs** view, and we will analyze the features offered by the CubeHAL to improve error management and to optimize the debugging process.



In an ideal world, this chapter would come right after the Chapter 5. Information reported here is important to perform an efficient debug during the early experiences with the STM32 platform. Unfortunately, to master concepts illustrated in this chapter, you need to study several other topics before you can deeply understand the techniques and tools shown here. As a rule of thumb, this author suggests reading at least chapters 7, 20 and 21 before approaching this one.

24.1 Understanding Cortex-M Fault-Related Exceptions

At beginning of this long journey, we have seen that Cortex-M based microcontrollers implement a number of system-related exceptions. Some of them are fault-related, that is those exceptions are triggered when something wrong happens during the normal execution flow. By implementing proper handlers for those fault-related exceptions, we can get rid of the fault origin. This is extremely useful during debugging, because it helps us isolating the issue from the rest of the application. However, a correct fault-handling can be useful even in a "production" firmware: once a fault is detected, we may place the device in a safe state before trying to reset the board.

Cortex-M3/4/7 cores offer to programmers four fault-related exceptions (see **Table 7.1**):

- **Memory Management Fault**
- **Bus Fault**
- **Usage Fault**
- **Hard Fault**

The first three exceptions are triggered when specific faults take place, and they are available only in Cortex-M3/4/7 cores. The last one, the *Hard Fault* exception, is the only one available even in Cortex-M0/0+ cores. It is also called the *generic fault exception*, since it cannot be disabled, and it acts as a collector for specific fault conditions when the other fault-related exceptions are disabled.

When a fault exception is raised, we can try to derive the cause of fault by analyzing the content of some "system registers". Moreover, the simple analysis of the stack trace can bring us to the root of fault condition at least in the majority of fault causes[1].

What circumstances can generate a system fault? Answering to this obvious question is not trivial. The most frequent source of fault is a bug in the firmware, especially during the development stage. An access to an invalid memory location (quite often due to a broken pointer) is the most frequent source of fault conditions. An invalid or a not well-implemented *vector table* is another common source of faults. A stack overflow is another quite frequent fault condition, especially in low-cost STM32 MCUs when running an RTOS.

Sometimes, the origin of the fault is not related to the software, but it may be caused by external factors such as:

- Poor PCB design and layout (this is more common than you might think).
- Unstable or poor power supply (quite common in poor designs).
- Electrical noise (this is especially true for devices operating in rush environments).
- Electromagnetic interference (EMI) or electrostatic discharge (ESD).
- Extreme operating environment (e.g., temperature, humidity, etc.).
- Damage of some components (e.g., Flash/EEPROMs devices, crystal oscillators, electrolytic capacitors).
- Radiations.

To diagnose the above nasty fault-conditions is really hard. Those are conditions that no hardware developer would ever want to meet, and they are outside the scopes of this book. Here we will focus only on software-related faults and to the ways to identify them. However, before starting analyzing the causes that trigger the four fault-related exceptions, it is fundamental to analyze the way an exception is generated from the software point of view. This is important to identify, or at least to try to, the code that leads to a fault exception.

[1]Clearly, without the RTOS kernel-aware debugging the stack trace most likely will provide us misleading information if we are using an RTOS.

24.1.1 The Cortex-M Exception Entrance Sequence and the ARM Calling Convention

For high-level programmers[2], to invoke a routine seems an obvious thing. We just write down the name of the function we are going to call, passing to it a given number of parameters. And that's it. However, from the processor point-of-view, what happens under the hood needs to be specified down to the finest details and it must match both the processor architecture and the programming language semantics. For this reason, it is common to talk about *calling convention* when describing the process of placing a new routine on the stack.

The *ARM Architecture Procedure Call Standard* (AAPCS) precisely defines the calling convention for ARM based architectures. In Chapter 1 we have seen that Cortex-M based microcontrollers provide several *core registers*, which are shown again in **Figure 24.1** for your convenience. Not all those core registers are available in all Cortex-M cores: for example, FPU registers S0-S31 are only available in Cortex-M4F and Cortex-M7 cores, when the FPU unit is enabled and used.

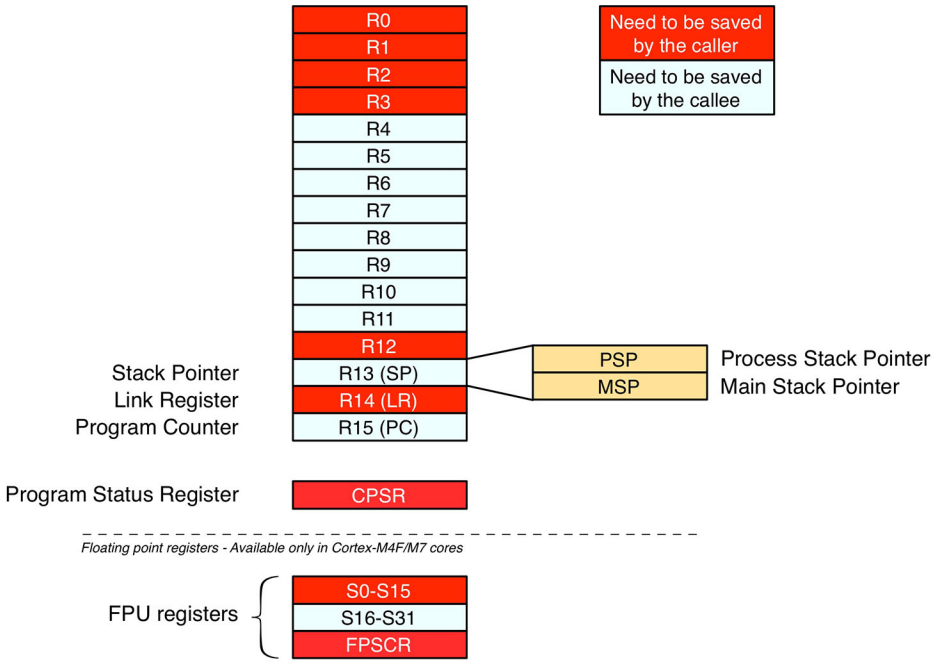


Figure 24.1: Cortex-M CPU core registers

Some core registers play a special role because they are used to carry out processor's activities. R13 is the *Stack Pointer* (SP), that is the pointer in SRAM (so something similar to `0x2000 XXXX` in an STM32) to the base of the most recent entry placed on the stack. This entry represents the local memory area of a given function and, in a full-descendent stack, SP coincides with the lowest address of the stack. R14 is the *Link Register* (LR), that is the address in FLASH[3] (so something similar to `0x0800X XXXX` in an STM32) of the instruction following the instruction that called the

[2] As C programmers, we are all "high level programmers", whether you believe it or not.

[3] This is not entirely true, because CPU could execute code placed in SRAM as well as in other external memories. But it is ok to consider it true here.

given function on the stack. R15 is the *Program Counter* (PC), that is the register that contains the address in FLASH memory of the current assembly instruction.

R0-R3 registers play another important role in the ARM calling convention. They are used to store the first four parameters to pass to the called function (from now on, we will use the term *callee* to indicate the called function, and *caller* to indicate the function that calls the other one). If the *callee* uses up to four parameters, then the first four general purpose registers contain the content of those parameters. Clearly, here we are assuming that arguments are word aligned (four bytes aligned). If, instead, our function accepts more than four parameters, or their total size exceeds sixteen bytes, then we need to allocate sufficient room on the *callee* stack to store the other parameters, before passing the control to the *callee*. This usage of R0-R3 registers allows to speedup calling process and to reduce the amount of used SRAM. Finally, R0-R1 registers are also used to store the function return value. So, a good rule would be to restrict the number of parameters to a maximum of four wherever possible. If that isn't possible, then you should try to place the most frequently accessed parameters in R0-R3 (that is, define them as the first four function parameters) so that stack accesses in the *callee* are minimized.

Since some of the general-purpose registers play specific roles, as a *callee* we cannot modify their content freely, but we must adhere to the following conventions:

- *Callee* can freely modify registers R0, R1, R2 and R3.
  - This implies that *caller* needs to save their content (if they are used to store relevant data for the *caller*) before passing the control to the *callee*.
- *Callee* cannot assume anything on the contents of R0, R1, R2 and R3 unless they are playing the role of parameters.
- *Callee* can freely modify LR register but the value upon entering the function will be needed when leaving the function (so this value needs to be stored in the *callee* stack frame).
- *Callee* can modify all the remaining registers as long as their values are restored upon leaving the function. This includes SP and registers R4-R11. This means that, after calling a function, we have to assume that (only) registers R0-R3, R12 and LR have been overwritten.
- A function should not make any assumption on the contents of the *Current Program Status Register* (CPSR).
- If FPU is enabled and used, *callee* can freely modify S0-S15 registers, which must be saved (together with the FPSCR register) by the *caller* before calling the *callee*. Instead, *callee* needs to save content of the S16-S31 registers before changing their content.
- R12 is a special "scratch register" used by linkers to perform dynamic linking. Not that useful in true-embedded microcontrollers like Cortex-M ones, but it is a register that must be saved by the *caller* according to AAPCS[4].

So, to recap, from the *caller* point-of-view, before invoking another routine we need to save the content of the following registers: R0-R3, R12, R14, CPSR (plus S0-S15 and FPSCR if FPU is enabled). These registers are highlighted in red in **Figure 24.1**.

[4]It is important to underline that the same ARM calling convention applies to Cortex-A based microprocessors, which have all the features to handle dynamic linking with high-level OSes like Linux and Windows.

As high-level programmers, we do not need to take care about these rules. It is a compiler task to ensure that AAPCS rules are respected. In Chapter 7 we saw that a distinctive feature of Cortex-M cores is the ability to use regular C functions as exception handlers. This means that exception handlers are "stacked" on the main stack as a regular C routine. But this implies that, in order to allow a C function to be used as an exception handler, the exception mechanism needs to adhere to the requirements of the AAPCS calling convention and so it needs to save automatically those "red" registers in **Figure 24.1** at exception entrance and restore them at exception exit under the control of the processors. In this way when returned to the interrupted program, all registers would have the same values as when the interrupt entry sequence started.

In addition, since an exception corresponds to an interruption of the main program flow, and since it can fire anytime, we need to save the content of the PC, otherwise we do not have a way to return to the main flow when the exception exits. In a regular function call, the value of the PC is stored inside the LR register by the branching instructions. Instead, when an exception fires the value of the return address (PC) is not stored in LR (the exception mechanism puts a special `EXC_RETURN` code in LR at exception entry, which is used in exception return - we will analyze it in a while), and the value of the return address also needs to be saved by the exception sequence.

So, in total eight registers need to be saved during the exception handling sequence on the Cortex-M based microcontrollers:

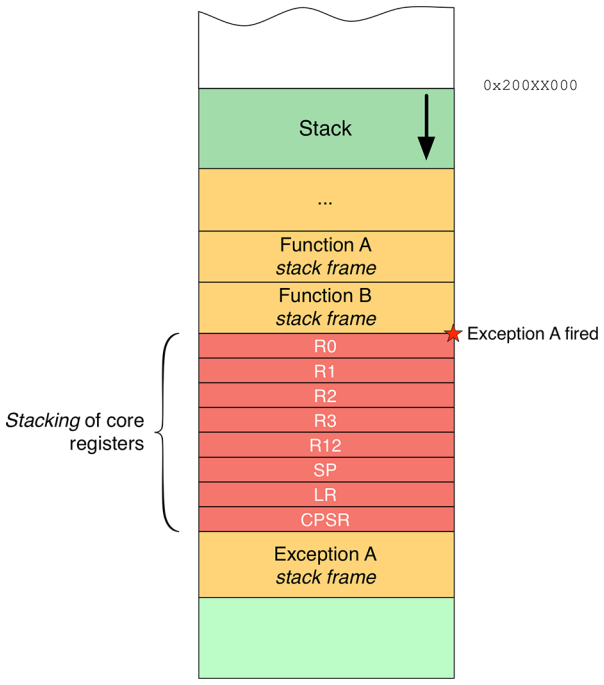


Figure 24.2: How core registers are stacked by the CPU on exception entrance

- R0-R3
- R12
- SP

- LR
- CPSR

In addition, S0-S15 and FPSCR register need to be saved if the FPU is used.

Where does the processor store these registers? Obviously, they are stored on the stack[5], right at the beginning of the exception handler's stack frame. This procedure is called *stacking* and **Figure 24.2** clearly shows the process. Please note that in **Figure 24.2** the color of core registers is lighter than the one used in **Figure 24.1**. This because it is important to underline that the processors stores in those locations the **content** of core registers before entering in the exception sequence. When the exception fires, the content of core registers are updated with the data related to the exception context (for example, the PC will point to the first instruction of the exception handler, or the SP will point to the top of MSP right after the stacked core registers).

The content of saved core registers can be useful in evaluating what did generate a fault exception. For example, if a fault exception triggers due to an access to an invalid memory location (maybe due to a broken pointer), by inspecting those registers we can try to understand the place where the illegal memory access is performed. So, the question is: as high-level programmers, do we have a way to access to those values? For sure! We only need a little bit of assembly programming.

Let us suppose we want to access to the content of stacked register when the `EXTI15_10_IRQHandler()` is invoked (this is the ISR called when the PC13 pin - connected with the Nucleo's blue button - is configured in interrupt mode on the majority of STM32 microcontrollers). We can define the ISR in the following way:

```
1  void EXTI15_10_IRQHandler(void) {
2          asm volatile(
3            " tst lr,#4       \n"
4            " ite eq          \n"
5            " mrseq r0,msp    \n"
6            " mrsne r0,psp    \n"
7            " mov r1,lr       \n"
8            " ldr r2,=EXTI15_10_IRQHandler_C \n"
9            " bx r2"
10         );
11 }
12
13 EXTI15_10_IRQHandler (uint32_t *core_registers,  uint32_t lr) {
14         /* core_registers points to the R0-R3, R13, SP and CPSR
15            registers, while the lr argument contains the content
16            of the LR register just before the exception entrance */
17         ....
18 }
```

[5]Here the story is a little bit more complex. Depending on the usage of an RTOS, there could be "multiple" stacks at the same time: a *Main Stack* or a stack specific for the single thread, called *Process Stack*. This topic is outside the scope of this book. For more information about it, refer to the excellent book by Joseph Yiu(http://amzn.to/1P5sZwq) about Cortex-M architectures.

The above assembly code may seem hard to understand, but instead is not that black magic art. The `tst` instruction performs a bitwise comparison between the content of the LR register (the **current** register, not the one saved on the stack) and the literal 4. If they match (that is, the third bit of LR register is set to 1), then the PSP stack was the one used at the time of exception entrance. Otherwise, the MSP was the current used stack. The reason why this check is performed will be clear soon. Take it as is here.

Instruction at line 7 does a simple thing (this is the tricky part): the content of the current LR register is placed in the R1 register, and the function `EXTI15_10_IRQHandler_C()` is called (note the final `_C`). This other function accepts two parameters: `core_registers` and `lr`. According to the AAPCS specification, `core_registers` will coincide with the register R0[6] while `lr` with the content of R1. When the exception handler is entered, R0 coincides with the starting address on the current stack (MSP or PSP) where the core registers have been stored.

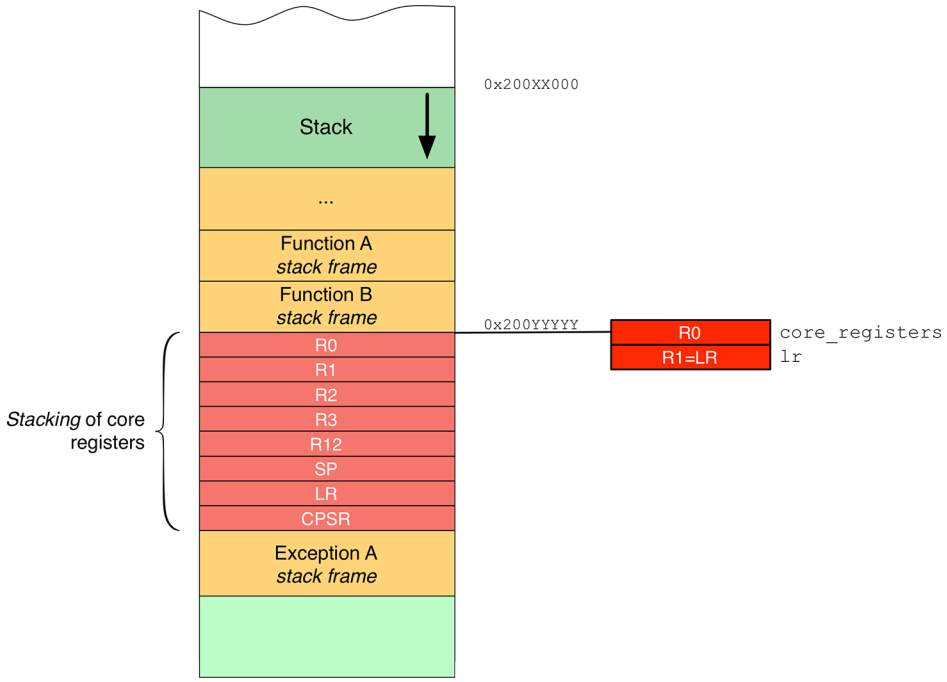


Figure 24.3: How current R0-R1 registers point to stacked register and actual LR register

Figure 24.3 clearly explains this. As you can see, `core_registers` corresponds to the R0 register, which holds the base address of stacked registers. `lr` corresponds to the R1 register, whose content has been filled with the one of the actual LR register by the assembly instruction at line 7. We can so access to stacked registers from the `EXTI15_10_IRQHandler_C()` routine, and perform analysis of their content, as we will see later.

[6]Please, take note that the `core_registers` parameter is a pointer, so the R0 register will contain the memory location (a 32-bit integer) where the core registers have been saved.

24.1.1.1 How to Interpret the Content of the LR Register on Exception Entrance

In Cortex-M based processors, the exception return mechanism is triggered using a special return address called `EXC_RETURN`. This value is generated at exception entrance, and it is stored in the *Link Register* (LR). When this value is written to the PC with one of the allowed function return instructions, it triggers the exception return sequence.

The `EXC_RETURN` address does not correspond to actual FLASH addresses. It can assume up to six values, which are listed in **Table 24.1**.

Table 24.1: `EXC_RETURN` possible values and their interpretation

`EXC_RETURN`	Return Mode	Return Stack	FPU Enabled	Description
`0xFFFF FFF1`	0 (Handler)	MSP	N	Returns to handler mode (using MSP)
`0xFFFF FFF9`	1 (Thread)	MSP	N	Returns to thread mode (using MSP)
`0xFFFF FFFD`	1 (Thread)	PSP	N	Returns to thread mode (using PSP)
`0xFFFF FFE1`	0 (Handler)	MSP	Y	Returns to handler mode (using MSP)
`0xFFFF FFE9`	1 (Thread)	MSP	Y	Returns to thread mode (using MSP)
`0xFFFF FFED`	1 (Thread)	PSP	Y	Returns to thread mode (using PSP)

For example, if the CPU was running "regular code" (that is, the CPU was in *Thread* mode) before entering the exception, if the stack used was the MSP and if the FPU unit was disabled, then the LR register contains the value `0xFFFF FFF9`. If, instead, the CPU was servicing another exception (maybe an interrupt) when the current exception entered (that is, the CPU was in *Handler* mode), then the content of the LR register is `0xFFFF FFF1`.

It is thanks to the `EXC_RETURN` mechanism that regular C functions can be used as exception handlers without writing any lines of assembly code. This differs from other microcontroller architectures, where additional work from the compiler (or from the developer) is needed to handle the stacking/unstacking of exception handlers.

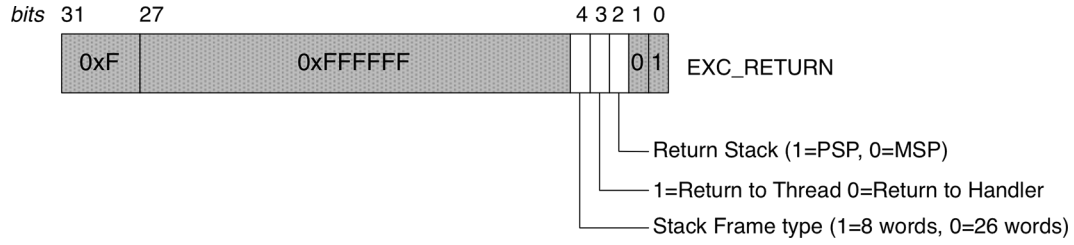


Figure 24.5: How the `EXC_RETURN` value is interpreted

Figure 24.5 shows the complete structure of the `EXC_RETURN` value. As you can see, the third bit indicates which stack was used at the time the fault condition triggers. This clearly explains the usage of the `tst` instruction in the previous assembly code to detect the used stack.

24.1.2 Fault Exceptions and Faults Analysis

The fault exception mechanism provided by Cortex-M CPU is useful to detect sources of faults. During the development lifecycle, it is common to have fault conditions, especially if you are new to the STM32 platform or the embedded programming.

This paragraph shows a brief overview of the analysis of fault conditions. It does not aim to replace the official ARM documentation or the excellent work from Joseph Yiu[7](http://amzn.to/1P5sZwq). Its main goal is to provide the necessary tools and concepts to understanding what's going wrong when one of the four fault exceptions is raised. Moreover, as we will see later in the chapter, STM32CubeIDE provides a dedicated tool to easily inspect fault-related exceptions and registers to find for the possible fault origin.

Cortex-M3/4/7 cores provide several registers that are used for fault analysis. They may be used by the fault handler code, but in most cases, they are used during a debug session. **Table 24.2** lists the available registers useful to fault analysis.

Table 24.2: Registers for fault status and address information

CMSIS Symbol	Register name	Description
`SCB->CFSR`	Configurable Fault Status Register	Provides status information about configurable exceptions (*MemFault*, *BusFault*, *UsageFault*)
`SCB->HFSR`	Status for HardFault	Provides status information for the *HardFault* exception
`SCB->DFSR`	Debug Fault Status Register	Provides status information for the *Debug Monitor* exception
`SCB->MMFAR`	MemManage Fault Address Register	If available, shows the address that triggered the *MemManage* fault
`SCB->BFAR`	BusFault Address Register	If available, shows the address that triggered the *BusFault* fault

`SCB->CFSR` is the *Configurable Fault Status Register* and it provides information for those exceptions that can be optionally enabled (*MemFault*, *BusFault*, *UsageFault*). It is in turn dived in three sub-registers, as shown in **Figure 24.6**. We are going to provide a complete description of them in the related sub-paragraphs.

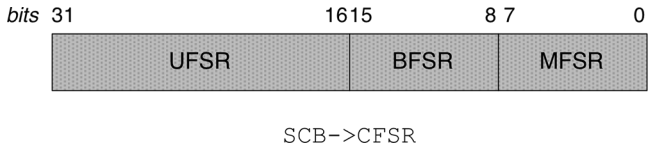


Figure 24.6: How the `SCB->CFSR` is further divided in three sub-registers

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

24.1.2.1 *Memory Management* Exception

This exception can be triggered due to a violation of access rules defined by the MPU configuration. For example, it is triggered when trying to access in write mode to a region defined as read only. This exception is available only in Cortex-M3/4/7 cores and it must be enabled. Once enabled, individual bits of the `SCB->MFSR` register (which corresponds to the first byte of the `SCB->CFSR` register) can assume the values reported in **Table 24.3**. The `SCB->MFSR` register is set to 0x0 upon reset, and its values stay high until a value of 1 is written to the register. By inspecting individual bit values, we can derive more information about the fault cause. For example, if the `DACCVIOL` bit is set, then an access to a protected memory location caused the exception. In this case the `MMARVALID` bit is set, the register `SCB->MMFAR` contains the destination memory location that generated the fault. To see this exception at work, try to execute the example provided in the paragraph about the MPU unit.

Table 24.3: ***MemManage Fault Status Register*** **(`SCB->MFSR`)**

Bit	Name	Description
7	`MMARVALID`	Indicates that the content of `SCB->MMFAR` register is valid
6	*RESERVED*	*RESERVED*
5	`MLSPERR`	Floating point lazy stacking error (available on Cortex-M4F cores only)
4	`MSTKERR`	Stacking error
3	`MUNSTKERR`	Unstacking error
2	*RESERVED*	*RESERVED*
1	`DACCVIOL`	Data access violation
0	`IACCVIOL`	Instruction access violation

24.1.2.2 *Bus Fault* Exception

This exception is mostly raised due to wrong access either to SRAM memory or program memory. The two more frequent sources of *Bus Fault* exception are a wrong pointer to an illegal SRAM memory region and a bad function pointer. In addition, the bus fault can also occur during stacking and unstacking of the exception handling sequence:

- If the bus error occurred during stack pushing in the exception entrance sequence, it is called a *stacking error*.
- If the bus error occurred during stack popping in the exception exit sequence, it is called an *unstacking error*.

Usually, a *stacking error* indicates a stack overflow: the stack runs out of space and this causes *Bus Fault* due to an access to an invalid SRAM location. The exception system triggers the fault exception, but the CPU cannot push saved core register on the full stack. This causes a *stacking error*, which in turn triggers a *Hard Fault*. By accessing to the `SCB->BFSR` we can see that both bits 15 and 12 are set. The content of the `SCB->BFAR` is so valid, and we can see that it contains something equal to `0x1fff bff8`. This is an invalid SRAM location in STM32 MCU, and so we can easily derive that a stack overflow happened.

Table 24.4 shows the meaning of individual bits in the `SCB->BFSR` register.

Table 24.4: ***Bus Fault Status Register*** (`SCB->BFSR`)

Bit	Name	Description
15	`BFARVALID`	Indicates that the content of `SCB->BFAR` register is valid
14	*RESERVED*	*RESERVED*
13	`LSPERR`	Floating point lazy stacking error (available on Cortex-M4F cores only)
12	`STKERR`	Stacking error
11	`UNSTKERR`	Unstacking error
10	`IMPRECISERR`	Imprecise data access error
9	`PRECISERR`	Precise data access error
8	`IBUSERR`	Instruction access error

Bus faults can be classified as:

- **Precise bus faults**: the fault exceptions happened immediately when the memory access instruction is executed.
- **Imprecise bus faults**: the fault exceptions happened sometime after the memory access instruction is executed.

The reason for a bus fault to become imprecise is due to the presence of write buffers in the processor bus interface. When the processor writes data to a bufferable address, the processor can proceed to execute the next instruction even if the transfer takes several clock cycles to complete. When an imprecise data access error takes place, the `SCB->BFAR` register is invalid. To derive the source of fault we need to disassemble the C source code and to identify the assembly instruction that logically precedes the one pointed by the stacked PC.

24.1.2.3 *Usage Fault* Exception

This exception can be raised by a wide range of factors. The most common ones, while developing STM32 applications, are:

- Execution of an undefined instruction (including trying to execute floating point instructions when the floating point unit is disabled). This often happens when we have an invalid function pointer, that points to a valid memory location (often it happens when we have some functions in SRAM), but the content of the pointed location does not correspond to an ARM assembly instruction.
- Invalid `EXC_RETURN` code during exception-return sequence. For example, trying to return to *Thread Mode* with exceptions still active (apart from the current serving exception).
- Unaligned memory access with multiple load or multiple store instructions (including *load double* and *store double* instructions).

- Execution of SVC instruction when the priority level of the SVC is the same or lower than current level. This may happen when something nasty has occurred with the FreeRTOS configuration of system exceptions (usually the *SysTick* IRQ does not have the lowest priority).

It is also possible, once the corresponding configuration is set, to generate usage faults for the following conditions:

- Divide by zero.
- All unaligned memory accesses.

Table 24.5 shows the meaning of individual bits in the `SCB->UFSR` register.

Table 24.5: ***Usage Fault Status Register*** **(`SCB->UFSR`)**

Bit	Name	Description
31-26	*RESERVED*	*RESERVED*
25	`DIVBYZERO`	Indicates divide by zero fault (can be set only if enabled)
24	`UNALIGNED`	Indicates that an unaligned access fault has taken place
23-20	*RESERVED*	*RESERVED*
19	`NOCP`	Attempt to execute a floating point instruction when the Cortex-M4F floating point unit is not available or when the floating point unit has not been enabled.
18	`INVPC`	Attempts to do an exception with a bad value in the `EXC_RETURN` number
17	`INVSTATE`	Attempts to switch to an invalid state (e.g., from ARM to Thumb)
16	`UNDEFINSTR`	Attempts to execute an undefined instruction

By default, Cortex-M based MCUs return the value `0` when dividing a number by zero. If, instead, you need to catch a divide by zero error, then you can enable this fault condition by setting `DIV_0_TRP` bit in the `SCB->CCR` register:

```
SCB->CCR |= SCB_CCR_DIV_0_TRP_Msk;
```

The same applies to unaligned memory accesses:

```
SCB->CCR |= SCB_CCR_UNALIGN_TRP_Msk;
```

24.1.2.4 *Hard Fault* Exception

This exception is usually raised by an escalation of the previous configurable exceptions, if not enabled. In addition, the HardFault can be triggered by:

- Bus error received during a *vector table* fetch. This happens because the *vector table* is invalid (most of the times we forgot to include the assembly file provided by STM).
- Execution of breakpoint instruction (`asm("BKPT #0");`) with a debugger attached.

Table 24.6 shows the meaning of individual bits in the `SCB->HFSR` register.

Table 24.6: *Hard Fault Status Register* (`SCB->HFSR`)

Bit	Name	Description
31	`DEBUGEVT`	Indicates that the *Hard Fault* is triggered by a debug event
30	`FORCED`	Indicates *that Hard Fault* is generated by an escalation of configurable fault exceptions while they are disabled. In this case we need to inspect the content of `SCB->MFSR`, `SCB->BFSR` and `SCB->UFSR` register to derive the fault cause.
29-2	*RESERVED*	*RESERVED*
1	`VECTBL`	Indicates that the *Hard Fault* is caused by failed *vector table* fetch
0	*RESERVED*	*RESERVED*

24.1.2.5 *Secure Fault* Exception

The *SecureFault* exception is available only in Cortex-M33 cores. This exception is triggered by the various security checks that are performed. It is triggered, for example, when jumping from *Non-secure* code to an address in *Secure* code that is not marked as a valid entry point. Usually, the firmware treats a *SecureFault* as a terminal condition that either halts or restarts the system. Any other handling of the *SecureFault* must be checked carefully to make sure that it does not inadvertently introduce a security vulnerability. Secure faults always target the Secure state, which is the default core state upon a system reset. This means that the *SecureFault* exception is enabled by default.

The *Secure Fault Status Register* (SFSR) provides information about any security related faults. Since this book does not cover Cortex-M33 cores we will not deepen this topic here. For more information, refer to the official STM documentation[8].

24.1.2.6 Enabling Optional Fault Handlers

Memory Fault, *Bus Fault* and *Usage Fault* are disabled by default. Neither the `HAL_NVIC_EnableIRQ()` nor the `NVIC_EnableIRQ()` can turn ON those exceptions, which are enabled by setting bits 16, 17 and 18 of the `SCB->SHCSR` register. To enable the *Memory Fault* exception, we use the following instruction:

```
SCB->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk; //Set bit 16
```

To enable the *Bus Fault* exception, we use the following instruction:

```
SCB->SHCSR |= SCB_SHCSR_BUSFAULTENA_Msk; //Set bit 17
```

To enable the *Usage Fault* exception, we use the following instruction:

[8] https://bit.ly/3o8mkJw

```
SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk; //Set bit 18
```

Once one of those exception is enabled, we can configure its priority using the `HAL_NVIC_SetPriority()`, like any other configurable exception.

24.1.2.7 Fault Analysis in Cortex-M0/0+ Based Processors

Cortex-M0/0+ cores do not provide *Memory Fault*, *Bus Fault* and *Usage Fault* exception. Moreover, the corresponding status registers are not available. This means that we do not have the same diagnostic features offered by Cortex-M3/4/7 cores.

The analysis of the stacked registers is the sole relevant technique we can use to diagnose fault reasons. This answer[9] by Joseph Yiu on the official ARM forum provides additional useful details. Other techniques, such as filling the SRAM with a sentinel value to detect a stack overflow, may help you finding the source of fault in your code.

24.2 STM32CubeIDE Advanced Debugging Features

In Chapter 5 we have started analyzing the debugging functionalities offered by STM32CubeIDE tool-chain. We have familiarized with the most basic features like breakpoints insertion and step-by-step debugging. Now it is the right time to see the other debugging functionalities integrated in the official STM development environment.

All the features shown here are available through the *Debugging* perspective.

24.2.1 Expressions and Live Expressions

The **Expressions** view is a powerful feature that allows to access to the content of memory addresses, variables and other data structures during debugging. Moreover, it is also able to perform function calls, so that you can evaluate the result of a given routine. The **Expressions** view must be explicitly enabled going to **Window->Show View->Expressions**.

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

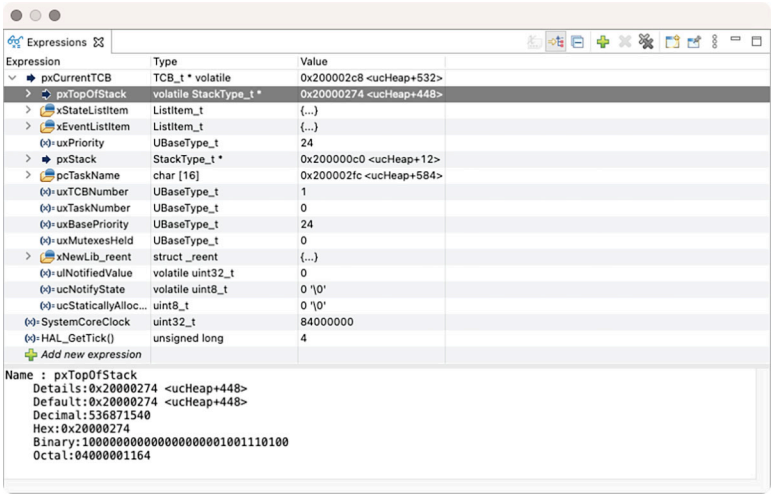


Figure 24.7: The Expressions view in the debug perspective

The **Figure 24.7** shows several expression examples. `pxCurrentTCB` is the current *Thread Control Block* (TCB) of a task ready to be stopped in FreeRTOS. As you can see from **Figure 24.7**, by simply writing down the variable name in the expression view we can access to its content wherever it is defined in the code. We can also show a C pointer as an array, using the expression `(variable@len)`, where `variable` is the pointer name and `len` is the amount of data stored in the array.

In **Figure 24.7** also shows that it is possible to call a function (the `HAL_GetTick()` in our case) and to obtain its result[10]. An expression can also contain arithmetic operations. Finally, the **Expressions** view is also able to access to the content of individual memory locations, and to cast their content to a given datatype (by right-clicking on the expression row you can cast a variable to a different datatype).

Expressions view in recent Eclipse CDT releases accepts *enhanced expressions*. An enhanced expression is a way of easily writing an expression pattern that will automatically expand to a larger subset of children expressions. Four types of enhanced expressions can be used:

- Pattern-matched local variables
- Pattern-matched registers
- Pattern-matched array elements
- Expression groups

For example, the pattern "=*" allows to show all local variable in the current stack frame, while the pattern "=$*" shows core registers. For more information about *enhanced expressions* refer to the Eclipse CDT documentation[11].

Live Expressions view is an STM32CubeIDE distinctive view that works very much like the Expression view, with the exception that all the expressions are sampled live during debug execution. The sampling speed is determined by the number of expressions being sampled. An increased number of expressions being sampled results in a slower sample rate.

[10] Clearly, that function must be included in the binary image, that is it must be a function used in the firmware code.
[11] https://bit.ly/2cRC6ra

24.2.1.1 Memory Monitors

STM32CubeIDE allows to access to the content of the whole 4GB address space. You can access to the content of a memory location by using **Memory** view. To show the view, go to **Window->Show View->Memory**. The **Memory** view lets you monitor and modify the MCU memory. The address space is presented as a list of so-called memory monitors. Each monitor represents a section of memory specified by its location called base address.

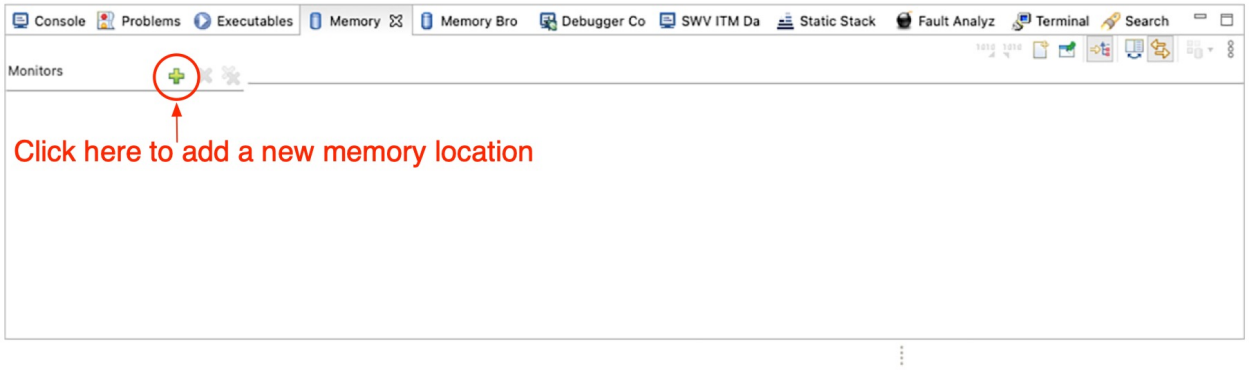


Figure 24.8: The *Memory Monitors* view

Once the view is shown, you can add a new memory location to the monitor view by clicking on the green cross shown in **Figure 24.8**. The next step consists in selecting a "renderer", that is a way to show the content of the memory location. You can choose between:

- Floating Point
- Traditional
- Hexadecimal
- ASCII
- Signed and unsigned integer

You can also add more renderers for the same memory location. Finally, you can configure several options of the memory view (cell size, endianness, memory format, etc.) by right-clicking on a memory cell.

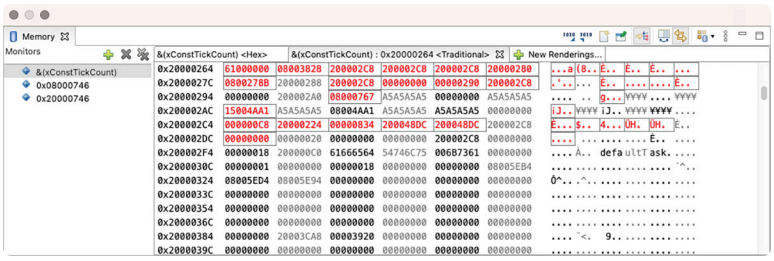


Figure 24.9: A memory location shown using Hexadecimal and Traditional renderers

24.2.2 Watchpoints

Every Cortex-M based processor provides a given number of breakpoints and watchpoints (see **Table 24.7**). While breakpoints are used to break execution at a given instruction, watchpoints are used to break execution when a data location is accessed. Any data or peripheral address can be marked as a watched variable, and an access to this address causes a debug event to be generated, which halts program execution. Watchpoint can also be used to halt execution only when a given expression matches.

Table 24.7: Available breakpoints/watchpoints in Cortex-M cores

Cortex-M	Breakpoints	Watchpoints
M0/0+	4	2
M3/4/7/33	6	4

There are several ways to add a watchpoint in the Eclipse CDT tool-chain. For example, you can right-click on a variable in the **Variables** view and the select the entry **Add Watchpoint(C/C++)**. The same can be performed from the **Expressions** view and the **Memory monitors** view while right-clicking on a memory location.

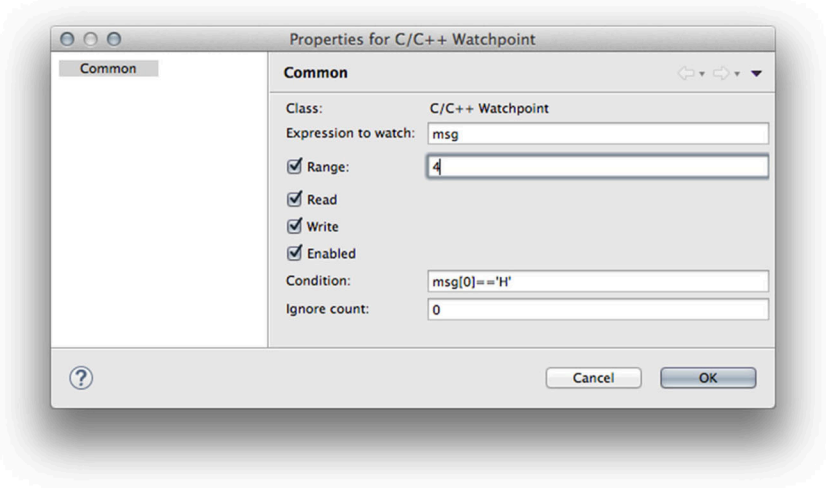


Figure 24.10: The watchpoint configuration view

Once clicked on the **Add Watchpoint(C/C++)** entry the watchpoint configuration view appears, as shown in **Figure 24.10**. Here we can setup the amount of memory to watch starting from the first word (**Range** field). Moreover, we can specify if we want to halt execution when that memory location is accessed in **Read** or **Write** mode. The **Enable** field allows to enable/disable the watchpoint. Finally, the **Condition** field allows to specify a condition. Watchpoints are listed inside the *Breakpoints* view.

24.2.3 Instruction Stepping Mode

The *Instruction Stepping Mode* is a debugging mode that allows to perform step-by-step debugging of ARM assembly instructions "underlying" a given C instruction.

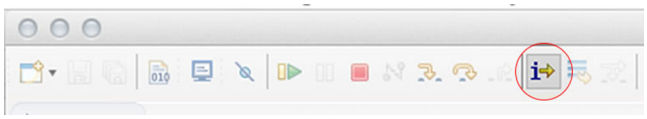


Figure 24.11: The *Instruction Stepping Mode* icon on the Eclipse toolbar

Instruction Stepping Mode is enabled by clicking on the related icon on the Eclipse main toolbar, as shown in **Figure 24.11**. Once enabled, the *Disassembly* appears, as shown in **Figure 24.12**. Eclipse will automatically show the ARM assembly instructions corresponding to the current C instruction.



Read Carefully

The *Instruction Stepping Mode* dramatically slows down the debugging process, because the CPU halts at every assembly instruction. If you cannot understand why the debugging is so slow, then you probably forgot the *Disassembly* view active.

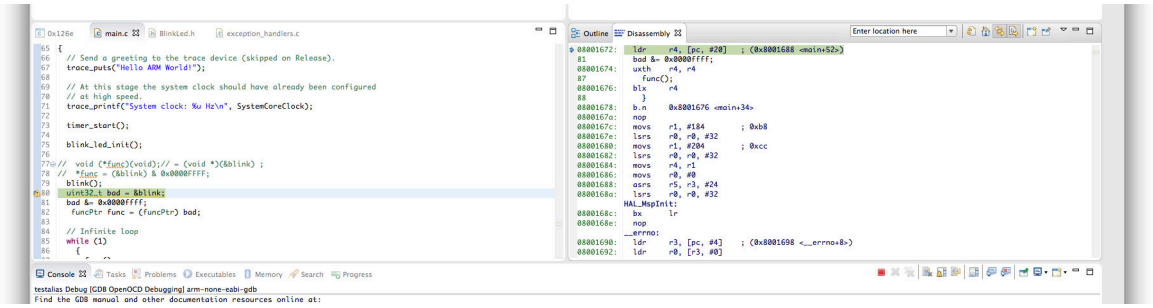


Figure 24.12: The disassembly view

24.2.4 SFRs View

The *Special Function Registers* (SFRs) view, shown in **Figure 24.13**, allows to access to Cortex-M core registers, plus all specific register of a given STM32 device. The registers' content can be eventually modified by double-clicking on the register value or on the convenient bit field representation. When debugging the project, the registers and bit fields are populated with the values read from the target.

The top of the **SFRs** view contains a search field to filter visible nodes, such as peripherals, registers, bit fields. Upon text entry in the search field, only the nodes containing this text are displayed.
The information at the bottom of the **SFRs** view displays detailed information about the selected line. For registers and bit fields, this includes [Access permission] and [Read action] information. The [Access permission] contains the following details:

- **RO** (Read-Only)
- **WO** (Write-Only)
- **RW** (Read-Write)
- **W1** WriteOnce
- **RW1** Read-WriteOnce

The toolbar buttons are located at the top-right corner of the **SFRs** view. The [**RD**] button in the toolbar is used to force a read of the selected register. It causes a read of the register even if the register, or some of the bit fields in the register, contains a ReadAction attribute set in the SVD file. When the register is read by pressing the [**RD**] button, all the other registers visible in the view are read again also to reflect all register updates. The program must be stopped to read registers. The base format buttons ([**X16**], [**X10**], [**X2**]) are used to change the registers display base.

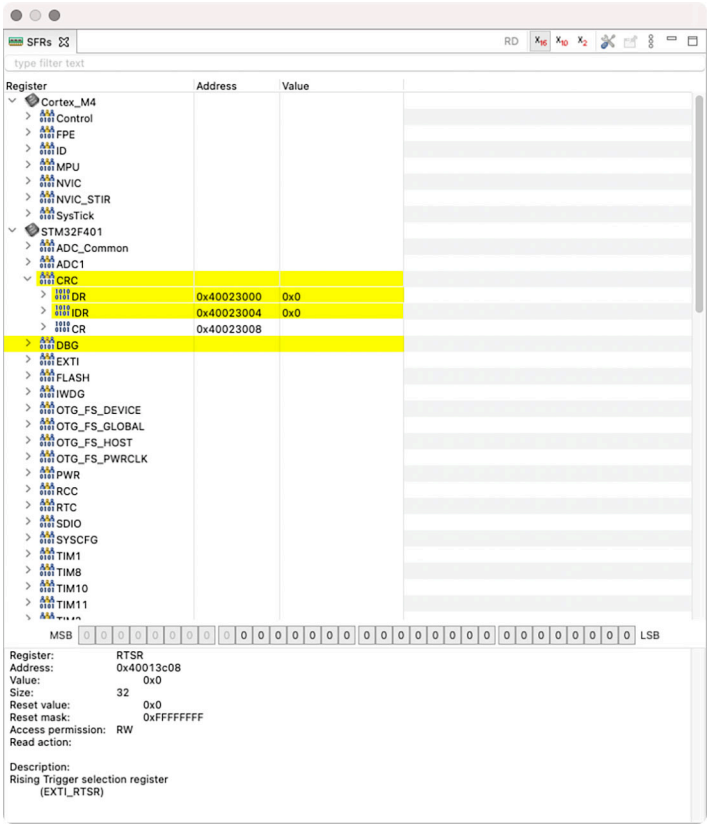


Figure 24.13: The *Registers* view in the debug perspective

24.2.5 Fault Analyzer

STM32CubeIDE provides a useful tool to simplify the debugging process: the **Fault Analyzer** (see **Figure 24.14**). **Fault Analyzer** tool interprets information extracted from the Cortex-M fault-related exceptions and registers to identify the conditions that caused a fault. Upon fault occurrence, the

code line where the fault occurred is displayed in the debugger. The view displays the reasons for the error condition. Faults are categorized into *Hard Fault*, *Bus Fault*, *Usage Fault* and *Memory Fault*.

To further assist fault analysis, an exception stack frame visualization option provides a snapshot of the MCU register values at the time of the crash. If not shown, the **Fault Analyzer** view can be enabled in the Debugger perspective by going to **Window->Show View->Fault Analyzer**.

The **Fault Analyzer** view contains in the top right part a toolbar (see red box in **Figure 24.14**):

- The first toolbar button (left) opens the *Editor* on the fault location return address by using the information in the PC and LR registers in the stack and the symbol information in the debugged ELF file.
- The second toolbar button (middle) opens the *Disassembly* view on the fault location return address by using the information in PC and LR registers in the stack and the symbol information in the debugged ELF file.
- The third toolbar button (right) selects if the PC or LR register is used when opening the *Editor* or *Disassembly* view on error location.

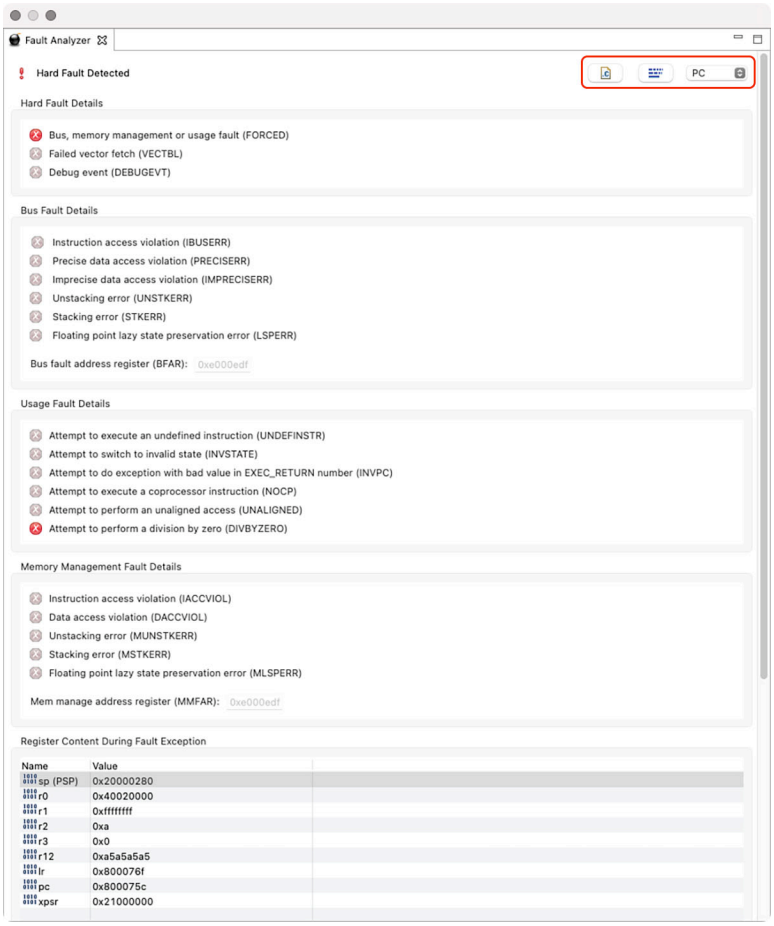


Figure 24.14: The Fault Analyzer view

24.2.5.1 Tracing Fault-Related Registers Without the IDE Support

Often, it happens that we need to trace a fault happened when the target device is not in a debug session and the debugger probe is not attached to the device. In this unenviable situation it could help a lot to know the status of fault and stacked registers.

We can define the fault-related exceptions (*Hard Fault*, *Memory Fault*, *Bus Fault* and *Usage Fault*) so that the stacked registers are easily printed through the preferred output method (UART, USB, ITM, etc.) in the way described next. Since the fault-related registers differ from Cortex-M0/0+ (ARMv6-M architecture) and Cortex-M3/4/7 (ARMv7-M and ARMv7E-M for Cortex-M7), we are going to describe them separately.

Cortex-M0/0+ - ARMv6-M
The dumping procedure uses the same technique described before when we have discussed about the *stacking* procedure performed by the Cortex-M core when an exception raises. By redefining the system call `int _write(int file, char *ptr, int len)` we can redirect the output to the physical medium we want.

```
typedef struct {
  uint32_t r0;
  uint32_t r1;
  uint32_t r2;
  uint32_t r3;
  uint32_t r12;
  uint32_t lr;
  uint32_t pc;
  uint32_t psr;
} ExceptionStackFrame;

void dumpExceptionStack (ExceptionStackFrame* frame, uint32_t lr) {
  printf ("Stack frame:\r\n");
  printf (" R0 =  %08X\r\n", frame->r0);
  printf (" R1 =  %08X\r\n", frame->r1);
  printf (" R2 =  %08X\r\n", frame->r2);
  printf (" R3 =  %08X\r\n", frame->r3);
  printf (" R12 = %08X\r\n", frame->r12);
  printf (" LR =  %08X\r\n", frame->lr);
  printf (" PC =  %08X\r\n", frame->pc);
  printf (" PSR = %08X\r\n", frame->psr);
  printf ("Misc\r\n");
  printf (" LR/EXC_RETURN= %08X\r\n", lr);
}

void HardFault_Handler (void) {
  asm volatile(
      " movs r0,#4      \r\n"
      " mov r1,lr       \r\n"
```

```
      " tst r0,r1       \r\n"
      " beq 1f          \r\n"
      " mrs r0,psp      \r\n"
      " b   2f          \r\n"
      "1:               \r\n"
      " mrs r0,msp      \r\n"
      "2:"
      " mov r1,lr       \r\n"
      " ldr r2,=HardFault_Handler_C \r\n"
      " bx r2"

      : /* Outputs */
      : /* Inputs */
      : /* Clobbers */
  );
}

void HardFault_Handler_C (ExceptionStackFrame* frame __attribute__((unused)),
                          uint32_t lr __attribute__((unused))) {
  printf ("[HardFault]\r\n");
  dumpExceptionStack (frame, lr);

#if defined(DEBUG)
  __DEBUG_BKPT();
#endif
  while (1);
}
```

Cortex-M3/4/7 - ARMv7-M/ARMv7E-M

The dumping procedure uses the same technique described before when we have discussed about the *stacking* procedure performed by the Cortex-M core when an exception raises. By redefining the system call `int _write(int file, char *ptr, int len)` we can redirect the output to the physical medium we want.

```
typedef struct {
  uint32_t r0;
  uint32_t r1;
  uint32_t r2;
  uint32_t r3;
  uint32_t r12;
  uint32_t lr;
  uint32_t pc;
  uint32_t psr;
#if  defined(__ARM_ARCH_7EM__)
  uint32_t s[16];
#endif
} ExceptionStackFrame;
```

```
14
15  void dumpExceptionStack (ExceptionStackFrame* frame, uint32_t cfsr,
16                           uint32_t mmfar, uint32_t bfar, uint32_t lr) {
17    printf ("Stack frame:\r\n");
18    printf (" R0 =  %08X\r\n", frame->r0);
19    printf (" R1 =  %08X\r\n", frame->r1);
20    printf (" R2 =  %08X\r\n", frame->r2);
21    printf (" R3 =  %08X\r\n", frame->r3);
22    printf (" R12 = %08X\r\n", frame->r12);
23    printf (" LR =  %08X\r\n", frame->lr);
24    printf (" PC =  %08X\r\n", frame->pc);
25    printf (" PSR = %08X\r\n", frame->psr);
26    printf ("FSR/FAR:\r\n");
27    printf (" CFSR =  %08X\r\n", cfsr);
28    printf (" HFSR =  %08X\r\n", SCB->HFSR);
29    printf (" DFSR =  %08X\r\n", SCB->DFSR);
30    printf (" AFSR =  %08X\r\n", SCB->AFSR);
31
32    if (cfsr & (1UL << 7)) {
33            printf (" MMFAR = %08X\r\n", mmfar);
34    }
35    if (cfsr & (1UL << 15)) {
36        printf (" BFAR =  %08X\r\n", bfar);
37    }
38    printf ("Misc\r\n");
39    printf (" LR/EXC_RETURN= %08X\r\n", lr);
40  }
41
42  void HardFault_Handler (void) {
43    asm volatile(
44        " tst lr,#4       \r\n"
45        " ite eq          \r\n"
46        " mrseq r0,msp    \r\n"
47        " mrsne r0,psp    \r\n"
48        " mov r1,lr       \r\n"
49        " ldr r2,=HardFault_Handler_C \r\n"
50        " bx r2"
51
52        : /* Outputs */
53        : /* Inputs */
54        : /* Clobbers */
55    );
56  }
57
58  void HardFault_Handler_C (ExceptionStackFrame* frame __attribute__((unused)),
59                            uint32_t lr __attribute__((unused))) {
60    uint32_t mmfar = SCB->MMFAR; // MemManage Fault Address
```

```
61    uint32_t bfar = SCB->BFAR; // Bus Fault Address
62    uint32_t cfsr = SCB->CFSR; // Configurable Fault Status Registers
63
64    printf ("[HardFault]\r\n");
65    dumpExceptionStack (frame, cfsr, mmfar, bfar, lr);
66
67  #if defined(DEBUG)
68    __DEBUG_BKPT();
69  #endif
70    while (1);
71  }
```

24.2.6 Build Analyzer

The **Build Analyzer** feature interprets program information from the binary ELF file in detail and presents the information in a dedicated view available in **Window->Show view->Build Analyzer**. If a `.map` file, with the same name of the ELF file, is found in the same folder as the ELF file the information from the `.map` file is also used, and even more information can be presented.

The Build Analyzer view is useful to optimize or simplify a program. The view contains two tabs, the **Memory Regions** and **Memory Details** tabs:

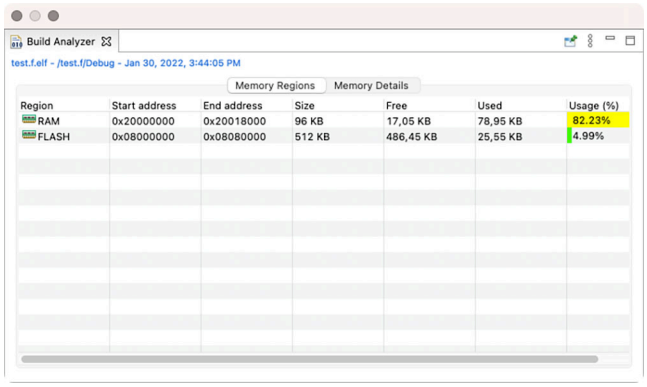


Figure 24.15: The Memory Regions tab in the Build Analyzer view

- The Memory Regions tab is populated with data if the ELF file contains a corresponding `.map` file. When the `.map` file is available, this tab can be seen as a summary of the memory regions with information about the region name, start address and size. The size information also comprises the total size, free and used part of the region, and usage percentage.
- The Memory Details tab (see **Figure 24.16**) contains detailed program information based on the ELF file. The different linker section names are presented with address and size information. Each section can be expanded and collapsed. When a section is expanded, functions/data in this section is listed. Each presented function/data contains address and size information.

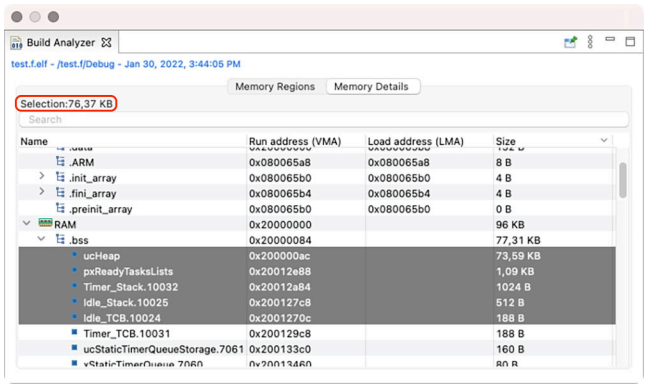


Figure 24.16: The Memory Details tab in the Build Analyzer view

The sort order of a **Memory Details** tab column can be changed by clicking on the column name. The information in the **Memory Details** tab can be filtered by entering a string in the search field. The view provides a convenient feature. The sum of the sizes of several lines in the **Memory Details** tab can be calculated by selecting these lines in the view. The sum of the selection is presented above the Name column in the view, as shown in **Figure 24.16**. The data in the **Memory Details** tab can be copied to other applications in CSV format by selecting the rows to copy and typing Ctrl+C (or CMD+C in Mac).

24.2.7 Static Stack Analyzer

The **Static Stack Analyzer** feature calculates the stack usage based on the built program. It analyzes every `.su` file, generated by GCC from the given `.c` file, and the ELF file in detail, and presents the resulting information in the view. The view can be enabled by going to **Window->Show view->Static Stack Analyzer**. The view contains two tabs, the **List** and **Call Graph** tabs.

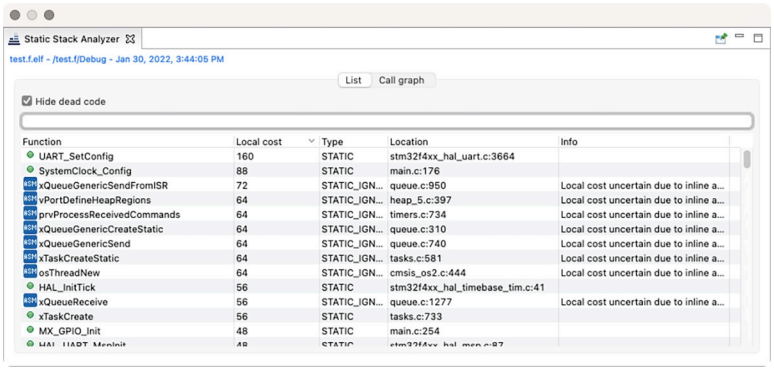


Figure 24.17: The List tab in the Static Stack Analyzer view

- The **List** tab (see **Figure 24.17**) contains a list of all functions included in the selected program. Use the **Hide dead code** selection to enable or disable the listing of dead code functions (that is functions discarded by the linker because not used at run-time). If used, the **Filter** field restricts the display to functions matching the characters it contains.

- The **Call Graph** tab (see **Figure 24.18**) contains detailed program information in a tree view. Each function included in the program but not called by any other function is presented at the top level. It is possible to expand the tree to see called functions. Only functions available in the ELF file can be visible in the tab. When used, the **Search**... button triggers the display of the functions matching the characters in the search field. The search can be made case sensitive or not depending on the selection in checkbox **Case sensitive**. The small icon left of the function name in column Function column indicates the following:
  - **Green dot**: the function uses STATIC stack allocation (fixed stack).
  - **Blue square**: the function uses DYNAMIC stack allocation (run-time dependent).
  - **010 icon**: used if the stack information is not known. This can be the case for library functions or assembler functions.
  - **Three arrows in a circle**: used in the Call Graph tab when the function makes recursive calls.

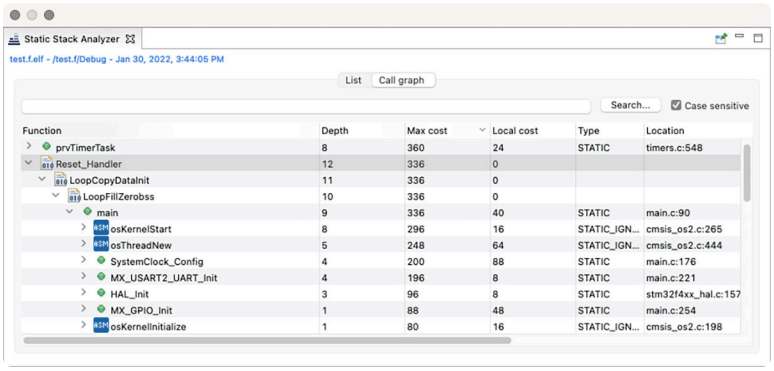


Figure 24.18: The Call Graph tab in the Static Stack Analyzer view

24.3 Serial Wire Viewer Tracing

Cortex-M based microcontrollers integrate several debugging and tracing technologies in the same die. JTAG and SWD are two complimentary specifications that allow to connect an external debugger to the target MCU. The same interfaces are used to implement tracing capabilities. Tracing allows to export internal activities performed by the CPU in real-time. It is a sort of live-hardware debugging, and it is carried out using the 5 signals of the JTAG port. Tracing is carried out due to the presence of a technology named *Embedded Trace Macrocell* (ETM), but it requires faster and more advanced debuggers. ETM tracing is a sort of "sniffing" technology, and it does not impact on the MCU performances.



Differences Between JTAG and SWD Interfaces

Novice users tend to be confused by these two debugging standards, which are both supported by STM32 microcontrollers. The *Joint Test Action Group* (JTAG) is a standard that defines both signaling characteristics and data protocol specification. It is based on five signals, plus two additional wires used to detect target VDD voltage and GND. JTAG allows to connect external debug probes to microcontrollers. It is a widely adopted standard in the electronics industry.

The *Serial Wire Debug* (SWD) is an alternative ARM proprietary 2-pin electrical interface that uses the same JTAG protocol. SWD enables the debugger to become another AMBA bus master for access to system memory and peripherals or debug registers. Data rate is up to 4 Mbytes/sec at 50 MHz. SWD also has built-in error detection. On JTAG devices with SWD capability, the TMS and TCK are used as `SWDIO` and `SWCLK` signals, providing for dual-mode programmers. An additional and optional signal, named *Serial Wire Output* (SWO), is used to exchange data and messages with the host application with a little impact on the MCU performances. We will analyze this functionality in a while.

The *Instrumentation Trace Macrocell* (ITM) is a less demanding tracing technology that allows sending software-generated debug messages through the SWD, using a specific signal I/O named *Serial Wire Output* (SWO). The protocol used by the SWO pin to exchange data with the debugger probe is called *Serial Wire Viewer* (SWV). The SWV support is not available in Cortex-M0/0+ based microcontrollers.

Compared to other "debugging-alike" peripherals like UART or to other technologies like the ARM semihosting, SWV is much faster. Its communication speed is proportional to the MCU speed, and this allows to limit the impact of the exchanged data on firmware performances. Clearly, the more fast runs the SWO I/O, the faster needs to be the debugger.

SWV provides the following types of target information:

- Event notification on data reading and writing.
- Event notification on exception entry and exit.
- Event counters.
- Timestamp and CPU cycle information, which can be used for program statistical profiling.

The CMSIS-Core package for Cortex-M3/4/7 cores provide necessary glue to handle SWV protocol. For example, the `ITM_SendChar()` routines allows to send a character using the SWO pin. This allows us to redefine the `_write()` system call so that the output of the `printf()` routine can be redirected to the SWV console. SWV protocol defines 32 different stimulus ports: a port is a "tag" on the SWV message used to enable/disable messages selectively. Writing different types of data to different SWV channels allows the debugger to interpret or visualize the data on various channels differently. The default stimulus port is 0.

The STM32CubeIDE provides an extensive set of tools and views to exploit all debugging possibility offered by the SWV technology. The usage of this debugging weaponry is clearly optional, but for large and complex projects its usage is a must.

24.3.1 Enabling SWV Debugging

To debug and use the Serial Wire Viewer (SWV) in STM32CubeIDE, both the debugger probe and the GDB server must support SWV. In the typical development environment used in this text, this is always true since both integrated ST-LINK interface (both V2.1 or V3) and ST-LINK GDB Server support SWV. Clearly, when using SWV, the target MCU shall be configured so that the SWD is properly enabled, as shown in **Figure 24.19**[12].

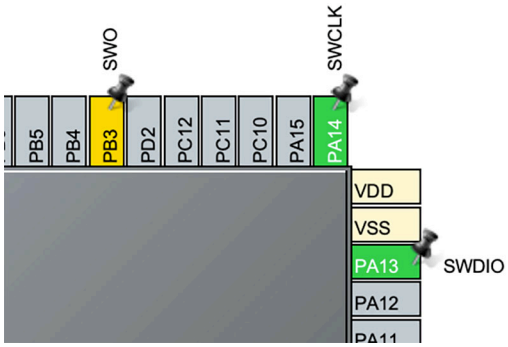


Figure 24.19: MCU PINs configuration to support SWD

To use the SWV, it is required to configure the *Debug Configuration* accordingly, as shown in **Figure 24.20**. To properly decode the bytes sent over the SWO port, the host debugger needs to know the frequencies of CPU core. This value is specified through the **Core Clock (MHz)** field. By default, STM32CubeIDE sets this value to the default clock configuration for the selected STM32 MCU. For example, by default CubeMX sets the default clock speed for an STM32F401RE MCU at 84MHz. If in doubt about the effective running frequency, you can inspect the content of the `SystemCoreClock` global variable by using the **Expressions** view during a debug session, as shown in **Figure 24.21**.



Please, take note that if your firmware changes the core clock firmware dynamically at run-time, then it is important to know the current core clock frequency when you need to use the SWD interface. This is a quite frequent source of confusion the first times the SWV is used.

Another important configuration is the SWO clock frequency. By default, the STM32CubeIDE sets it at the maximum switching frequency given the configured core clock. This allows to reduce the overhead introduced by the SWV interface: bytes sent over the SWD port need to be exchanged at a given frequency, and this could impact on the overall firmware performances. However, the ST-LINK V2.1 provides a maximum SWD frequency up to 4MHz, while the more recent ST-LINK V3 runs up to 24MHz. If you are using a debug adapter running a lower SWD frequency, then you can limit the SWO clock by specifying the maximum value for your probe.

[12]By default, CubeMX shows the PA14 pin, which corresponds to the *Serial Wire Debug Clock*, as **TCK** and the PA13 pin, which corresponds to the *Serial Wire Debug I/O*, as **TMS**. These acronyms come from the JTAG specification that, as said before, uses 4+1 pins to establish a debug connection with the target MCU. These pins are: *Test Data In* (TDI), *Test Data Out* (TDO), *Test Clock* (TCK), *Test Mode Select* (TMS) and *Test Reset* (TRST) - optional.

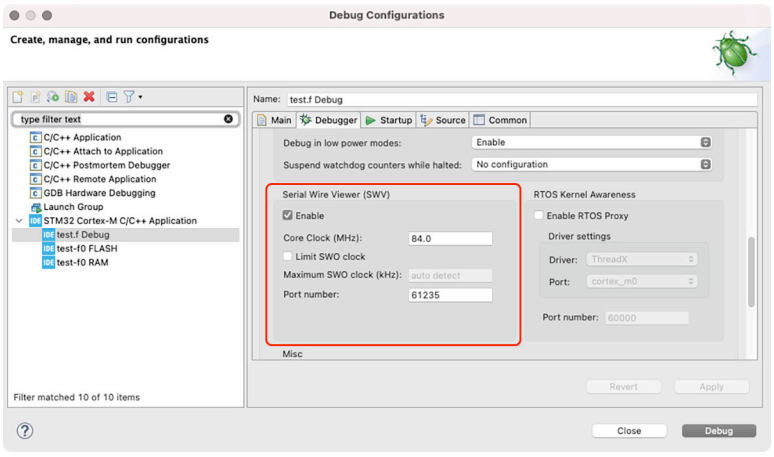


Figure 24.20: How to enable SWD in STM32CubeIDE

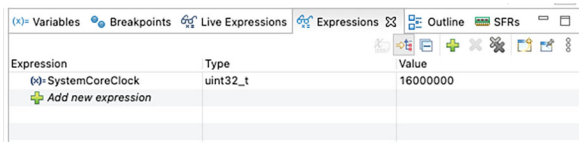


Figure 24.21: How to derive the running core clock frequency by using *Debug Expressions*

24.3.2 Configuring SWV

Once the SWV is enabled in the *Debug configuration*, we have to configure some relevant global settings that define what kind of information we are going to trace. It is possible to access to SWV settings from any of the SWV views available in **Window->Show View->SWV**, as shown in **Figure 24.22**. For example, the **Figure 24.23** shows the **SWV Trace Log** view: as you can see, in the top-left part of the view there is the main SWV toolbar; by clicking on the **Settings** icon, you can access to SWV settings, as shown in **Figure 24.24**. Let us describe the most relevant setting fields.

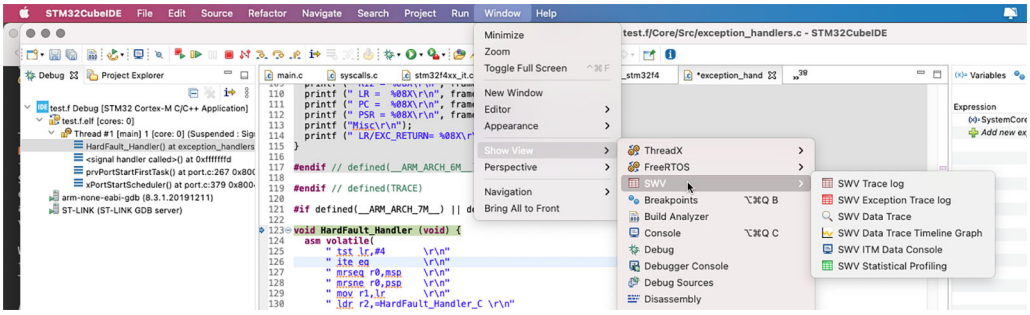


Figure 24.22: How to enable one of the SWV views

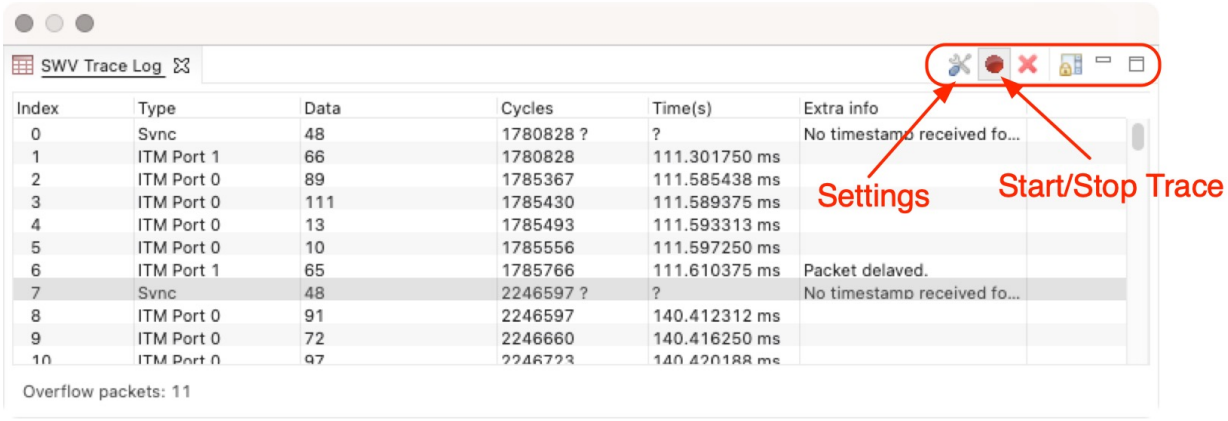


Figure 24.23: How to access to SWV settings

- **Clock Settings**: these fields are disabled and only present the values used and configured in the Debug Configurations for the debug session. If these values need to be changed, close the debug session and open the Debug Configurations to modify them.
- **Trace Events**: the following events can be traced.
  - **CPI (*Cycles per instruction*)**: for each cycle beyond the first one that an instruction uses, an internal counter is increased with one. The counter (DWT CPI count) can count up to 256 and is then set to 0. Each time that happens, one of these packets are sent. This is one aspect of the processors performance and used to calculate instructions per seconds. The lower the value, the better the performance.
  - **SLEEP (*Sleep cycles*)**: the number of cycles the CPU is in sleep mode. Counted in DWT Sleep count register. Each time the CPU has been in sleep mode for 256 cycles, one of these packets is sent. This is used when debugging for power consumption or waiting for external devices.
  - **FOLD (*Folded instructions*)**: a counter for how many instructions are folded (removed). Every 256 folded instructions (taken zero cycles) will receive one of these events. Counted in DWT Fold count register. Branch folding is a technique where, on the prediction of most branches, the branch instruction is completely removed from the instruction stream presented to the execution pipeline. Branch folding can significantly improve the performance of branches, taking the CPI for branches below 1.
  - **EXC (*Exception overhead*)**: the DWT Exception count register keeps track of the number of CPU cycles spent in exception overhead. This includes stack operations and returns but not the time spent processing the exception code. When the timer overflows, one of these events is sent. Used to calculate the actual exception handling cost to the program.
  - **LSU (*Load Store Unit Cycles*)**: the DWT LSU count register counts the total number of cycles the processor is processing an LSU operation beyond the first cycle. When the timer overflows, one of these events is sent. With this measurement, it is possible to track the amount of time spent in memory operations.
  - **EXETRC (*Trace Exceptions*)**: whenever an exception occurs, exception entry, exception exit and exception return events are sent. These events can be monitored in the **SWV**

Exception Trace Log view. From this view, it is possible to jump to the exception handler code for that exception.

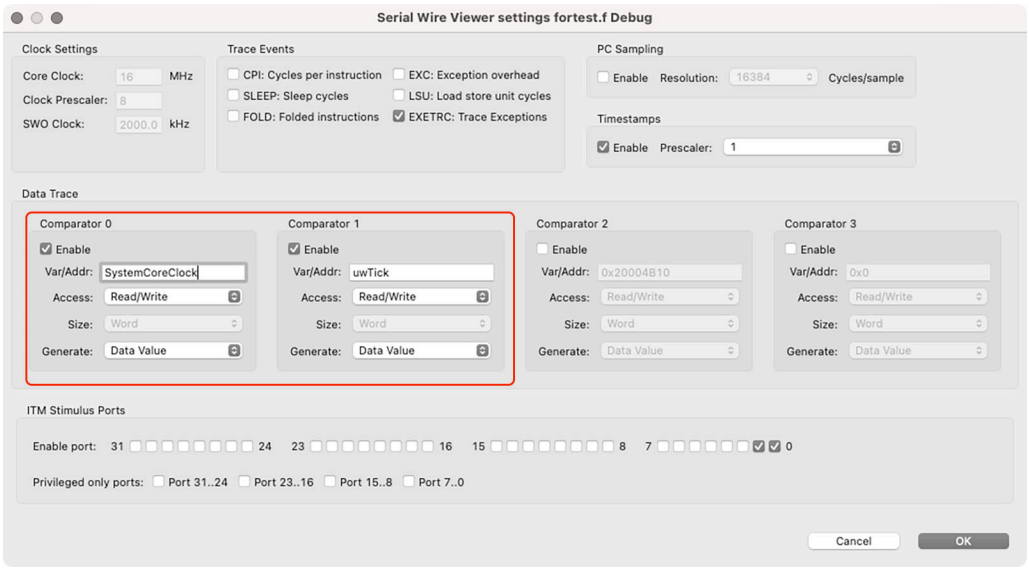


Figure 24.24: The SWV settings dialog

- **PC Sampling**: enabling this starts sampling the Program Counter at some cycle interval. Since the SWO pin has a limited bandwidth, it is not advised to sample to fast. Experiment with the **Resolution** (cycles/ sample setting) to be able to sample often enough. The results from the sampling are used, among other things, for the SWV Statistical Profiling view.
- **Timestamps**: must be enabled to know when an event occurred. The **Prescaler** should only be changed as a last effort to reduce overflow packets. The **Timestamp** setting is used with the **SWV Data Trace Timeline Graph**, as we will see later.
- **Data Trace**: it is possible to trace up to four different C variable symbols, or fixed numeric areas of the memory. To do that, enable one comparator and enter the name of the variable or the memory-address to trace. The value of the traced variables can be displayed both in the Data Trace and Data Trace Timeline Graph views. More about this later.
- **ITM Stimulus Ports**: there are 32 ITM ports available (tags for messages), which can be used by the application. For instance, the CMSIS function `ITM_SendChar()` can be used to send characters to a given port. The packets from the ITM ports are displayed in the **SWV ITM Data Console view**. More about this later.

24.3.3 SWV Views

STM32CubeIDE provides six dedicated views to exploit the SWV features. We are now going to explain each one of them. The enable SWV tracing in any view, the **Start/Stop Trace** button (red one) on the view toolbar need to be toggled, as shown in **Figure 24.23**. **Please, take note that any change to the SWV settings dialog will affect the SWV tracing once the SWV is stopped and then re-started**.

24.3.3.1 SWV Trace Log

The **SWV Trace Log** view (shown in **Figure 24.23**) lists all incoming SWV packets in a spreadsheet. The data in this view can be copied to other applications in CSV format by selecting the rows to copy and type Ctrl+C (or CMD+C in Mac). The column information in the SWV Trace Log view is described in **Table 24.8**.

Name	Description
Index	The packet ID.
Type	The type of packet (example PC sample, data PC value (comp 1), exceptions, overflow).
Data	The packet data information.
Cycles	The timestamp of the packet in CPU cycles.
Time(s)	The timestamp of the packet in seconds.
Extra info	Optional extra packet information.

Table 24.8: Columns' description of SWV Trace Log view

24.3.3.2 SWV Exception Trace Log

The **SWV Exception Trace Log** view (shown in **Figure 24.25**) is composed of two tabs.

Data Tab

The first tab is similar to the **SWV Trace Log** view but is restricted to exception events. It also provides additional information about the type of event. The data can be copied and pasted into other applications. Each row is linked to the code for the corresponding exception handler. Double-click on the event to open the corresponding interrupt hander source code in the Editor view. The column information in the **SWV Exception Trace Log – Data tab** is described in **Table 24.9**.

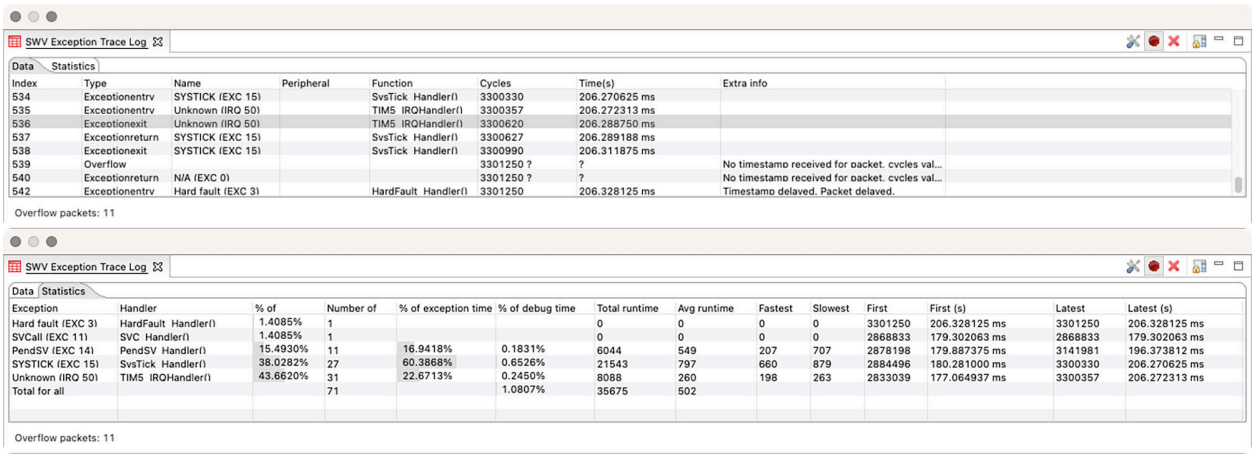


Figure 24.25: The SWV Exception Trace Log view

Name	Description
Index	The packet ID.
Type	Each exception generates three packets: *Exception entry*, *Exception exit* and then an *Exception return* packet.
Name	The name of the exception. Also the exception or interrupt number.
Peripheral	The peripheral for the exception.
Function	The name of the interrupt handler function for this interrupt. Updated when debug is paused. Is cached during the whole debug session. By double clicking the function, the editor will open that function in the source code.
Cycles	The timestamp for the exception in CPU cycles.
Time(s)	The timestamp of the packet in seconds.
Extra info	Optional extra packet information.

Table 24.9: Columns' description of SWV Exception Trace Log Data Tab

Statistics Tab

The second tab displays statistical information about exception events. This information may be of great value when optimizing the code. Hypertext links to exception handler source code in the editor is included. The column information in the **SWV Exception Trace Log – Statistics tab** is described in **Table 24.10**.

Name	Description
Index	The packet ID.
Type	Each exception generates three packets: *Exception entry*, *Exception exit* and then an *Exception return* packet.
Name	The name of the exception. Also the exception or interrupt number.
Peripheral	The peripheral for the exception.
Function	The name of the interrupt handler function for this interrupt. Updated when debug is paused. Is cached during the whole debug session. By double clicking the function, the editor will open that function in the source code.
Cycles	The timestamp for the exception in CPU cycles.
Time(s)	The timestamp of the packet in seconds.
Extra info	Optional extra packet information.

Table 24.10: Columns' description of SWV Exception Trace Log Statistics Tab

24.3.3.3 SWV Data Trace

The **SWV Data Trace** view (see **Figure 24.26**) tracks up to four different symbols or areas in the memory. For example, global variables can be referenced by their name. The data can be traced on Read, Write and Read/Write. Four comparators are available in the SWV settings dialog. For example, in **Figure 24.24** two global variables `SystemCoreClock` and `uwTick` (the global HAL tick counter) in the program are traced on Read and Write access. The generated output can contain both just the data value and the PC value tracking the instruction accessed in reading/writing the memory location. The **SWV Data Trace** is able to trace just integer values.

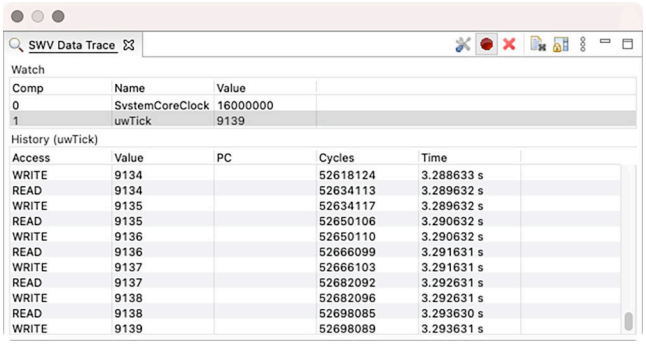


Figure 24.26: The SWV Data Trace view

24.3.3.4 SWV Data Trace Timeline Graph

The **SWV Data Trace Timeline Graph** view contains a graphical display that shows the distribution of variable values over time. It applies to the variables or memory areas in the **SWV Data Trace**. The **Figure 24.27** shows the plot of `sin()/cos()` functions. Like the **SWV Data Trace**, the timeline graph can plot just integer values.

The SWV Data Trace Timeline Graph has the following features:

- The graph can be saved as a JPEG image file by clicking on the camera toolbar button.
- The graph shows the time in seconds by default but can be changed to cycles by clicking on the clock toolbar button.
- Y-axis can be adjusted to best fit by clicking on the y-axis toolbar button.
- Zoom in and out by clicking on the [+] and [–] toolbar buttons.
- The zoom range is limited while debug is running. Zoom details are available when debug is paused.

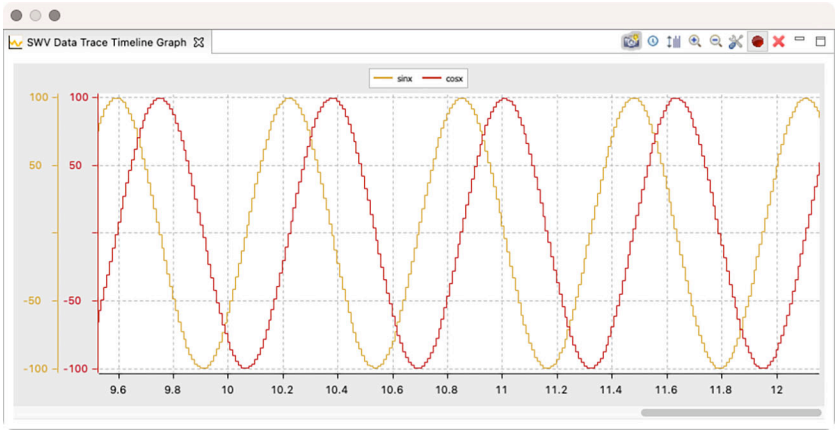


Figure 24.27: The SWV Data Trace Timeline Graph view

24.3.3.5 SWV ITM Data Console

The **SWV ITM Data Console** (see **Figure 24.28**) is the most useful feature in the SWV toolset, especially if your target device cannot have a dedicated UART port to print messages. The **SWV ITM Data Console** prints readable text output from the target application. Typically, this is done via `printf()` with output redirected to ITM channel 0. Other ITM channels can get their own console views.

To use the **SWV ITM Data Console** view, first enable one or more of the 32 ITM ports in the Serial Wire Viewer settings dialog (see **Figure 24.24**). The packets from the ITM ports are displayed in the **SWV ITM Data Console** view. The CMSIS function `ITM_SendChar()` can be used by the application to send characters to the port 0, and the `printf()` function can be redirected to use the `ITM_SendChar()` function by properly overriding the `_write()` system call in the **Core/Src/syscalls.c**, as shown below:

```
#include "stm32fXXxx.h"
...
int _write(int file, char *ptr, int len) {
        for (int DataIdx = 0; DataIdx < len; DataIdx++)
                ITM_SendChar(*ptr++);
        return len;
}
```

The CMSIS API does not provide a convenient function to send character to other stimulus ports. However, this can be easily performed by defining the following function:

```
#include "stm32fXXxx.h"
...

__STATIC_INLINE uint32_t ITM_SendCharToPort (uint32_t ch, uint8_t port)
{
  if (((ITM->TCR & ITM_TCR_ITMENA_Msk) != 0UL) &&      /* ITM enabled */
      ((ITM->TER & (1UL << port)     ) != 0UL))       /* ITM Port #n enabled */
  {
    while (ITM->PORT[port].u32 == 0UL) {
      __NOP();
    }
    ITM->PORT[port].u8 = (uint8_t)ch;
  }
  return (ch);
}
```

It is possible to open new port tabs in the **SWV ITM Data Console** (for the other stimulus ports) by pressing the green [+] button on the toolbar.

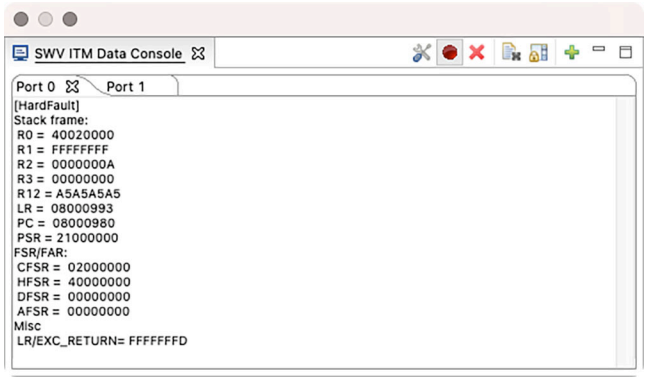


Figure 24.28: The SWV ITM Data Console view

24.3.3.6 SWV Statistical Profiling

The **SWV Statistical Profiling** (see **Figure 24.29**) view displays statistics based on Program Counter (PC) sampling. It shows the amount of execution time spent within various functions. This is useful when optimizing code. The data can be copied and pasted into other applications. The view is updated when debugging is suspended. To enable profiling, follow these steps:

1. Configure SWV to send Program Counter samples, as shown in **Figure 24.24**, by enabling **PC Sampling** flags and **Timestamps**. With the given **Resolution Cycles/sample**, SWV reports the Program Counter values to STM32CubeIDE. Set the **Resolution Cycles/sample** to a high value to avoid interface overflow.
2. Open the **SWV Statistical Profiling** view by selecting **Window->Show View->SWV Statistical Profiling**. The view is empty since no data is collected yet.
3. Press the red **Start/Stop Trace** button on the SWV view toolbar to send the configuration to the board.
4. Resume program debugging. STM32CubeIDE starts collecting statistics about function usage via SWV when the code is executing in the target system.
5. Suspend (Pause) the debugging. The view displays the collected data. The longer the debugging session, the more statistics are collected.

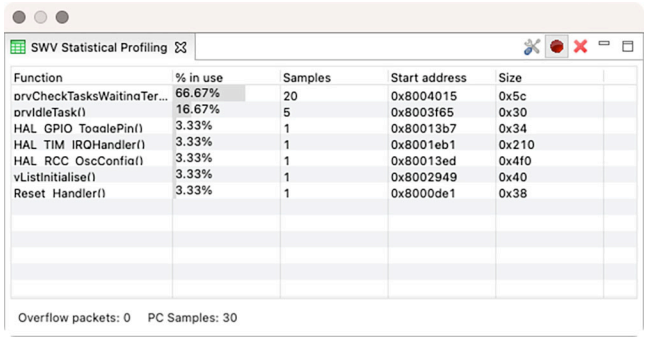


Figure 24.29: The SWV Statistical Profiling view

Name	Description
Function	The name of the function which is calculated by comparing address information in SWV packets with the program ELF file symbol information.
% in use	The calculated percentage of time the function is used.
Samples	The number of samples received from the function.
Start address	The start address for the function.
Size	The size of the function.

Table 24.11: Columns' description of SWV Statistical Profiling view

24.4 Debugging Aids from the CubeHAL

The CubeHAL implements run-time failure detection by checking the input values of all HAL API. The run-time checking is achieved by using an `assert_param()` macro. This macro is used in all CubeHAL functions having an input parameter. It allows verifying that the input value lies within the parameter allowed values.

To enable run-time checking you need to define the `USE_FULL_ASSERT` macro at project level (both in the project settings or by uncommenting the macro definition in the `stm32XXxx_hal_conf.h` file). CubeMX generates a function named `assert_failed()` in the `main.c` file. The function is defined in the following way:

```
void assert_failed(uint8_t* file, uint32_t line);
```

The function is automatically invoked by the `assert_param()` macro if an assertion is not satisfied. The macro will automatically pass to the function the filename and the exact lines of code where the assert condition is not satisfied.

The implementation of the `assert_failed()` function is left to the user. A simple implementation consists in placing a software breakpoint by invoking the `bkpt` ARM instruction:

```
void assert_failed(uint8_t* file, uint32_t line) {
        asm("BKPT #0");
}
```

Enabling the `USE_FULL_ASSERT` macro during the development stage can provide an important help to understand what's going wrong with the CubeHAL, especially if you are new to the CubeHAL.

24.5 External Debuggers

Serious projects demand serious tools. And this is dramatically true in electronics design. If you reached this part of the book without skipping any fundamental chapter, then you already know the limits of the ST-LINK debugging interface.



Figure 24.30: A SEGGER J-Link Ultra+ debug probe

SEGGER is a German company specialized in designing external debug probes for the ARM Cortex portfolio (including Cortex-M/R/A microprocessors and other modern MCUs like PIC32 and Renesas RX series). SEGGER J-Links (see **Figure 24.30**) are the most widely used line of debug probes available today, and they are often sold as OEM version for other vendors (IAR and Keil debug probes are nothing more than a J-Link).

The most relevant features offered by J-Link debuggers are:

- Up to 3 MByte/s download speed.
- Compatible with all popular tool-chains including the STM32CubeIDE.
- Supports an unlimited number of software breakpoints in flash memory.
- Allows setting breakpoints in external flash memory of Cortex-M systems through FMC controller.
- Cross platform support (Microsoft Windows, Linux, Mac OS X).
- Supports concurrent access to CPU by multiple applications.
- Support for multi core debugging.
- Remote Server included. Allows using J-Link remotely via TCP/IP.
- Software comes with free GDB server, allowing usage of J-Link with all GDB-based debug solutions.
- Production flash programming software (J-Flash) available.
- Debugger independent flash download (internal flash, CFI flash, SPIFI flash).
- Supports CPU/MCU internal trace buffer (ETB, MTB, etc.).

- Supports ETM tracing (J-Trace Cortex-M, J-Trace ARM).
- Wide target voltage range: 1.2V - 3.3V, 5V tolerant.
- Supports multiple target interfaces (JTAG, SWD, FINE, SPD, etc.).

J-Link probes ranges from the EDU edition, which costs about 60$, up to the J-Trace PRO edition that costs about $1100. If you are a student or a low-budget hobbyist, the EDU edition worth spending since it supports all relevant features provided by professional J-Link probes. If you are a professional, then the Ultra+ is a good deal according to this author.

However, for owners of STM development boards (Nucleo, Discovery, Eval) there is a good and totally free alternative: in April 2016 SEGGER has released a firmware upgrade for the ST-LINK V2/V2.1[13] interface that transforms it in a J-Link compatible debug probe. By downloading[14] a dedicated software tool[15], your ST-LINK is transformed in a J-Link OB compatible interface, and you can use the most important software tools by SEGGER[16]. Moreover, you can easily revert the interface to an ST-LINK if you want. **In case you experience issue in switching the ST-LINK V2 debugger to J-Link OB, please follow the instructions in this blog post[17].**

When debugging with SEGGER debug probe, there is no need to use ST-LINK GDB Server, because SEGGER provides its own compatible GDB server, named `JLinkGDBServer`. This is one of the fundamental reasons to choose these tools, because the `JLinkGDBServer` is a faster and more reliable alternative to ST-LINK being cross-platform at the same time. The instructions to upgrade the ST-LINK interface to a J-Link compatible one are clearly reported on the SEGGER website. We will not repeat them here. Instead, we are now going to analyze how to use a J-Link debug probe with the GNU MCU Eclipse tool-chain.

You are not forced to install SEGGER software tools to start using SEGGER debug probes, since the `JLinkGDBServer` is already integrated in the STM32CubeIDE. To use the J-LINK with the STM32CubeIDE you need to select the corresponding debug probe in the *Debug Configurations*.

24.6 Debugging two Nucleo Boards Simultaneously

We may need to debug two STM32 based devices simultaneously. This is not uncommon, especially when dealing with communication protocols. STM32CubeIDE allows us to debug two or more boards on the same computer.

To launch two instances of the STLINK GDB Server we need to create two separated *Debug Configurations*, one for each board. In every configuration we need to configure two fundamental parameters: the S/N of the ST-LINK interface and the GDB Server port number (see **Figure 24.31**). Regarding the GSB Server port this is up to us to choose two or more separated port number (61234

[13] At the time of writing this chapter February 2022, SEGGER does not provide a tool to convert an ST-LINK V3 debug adapter to a SEGGER J-LINK.

[14] https://www.segger.com/jlink-st-link.html

[15] Unfortunately, at the time of writing this chapter, the upgrade tool is only available for the Windows OS.

[16] Please, take note that the license of this "free" upgrade to the ST-LINK interface prevents you from using it to debug custom and commercial devices. Look at the SEGGER website for the complete list of limitations.

[17] https://bit.ly/3ostIQ6

and 61235 is fine). Instead, to derive the ST-LINK S/N it is sufficient to click on **Scan** button close to the **ST-LINK S/N** field. STM32CubeIDE will list you the serial number of every board attached to the PC.

Let us suppose that two different boards/microcontrollers are used: HW_A and HW_B. When the debug configuration has been configured for both projects so that each board is associated to a specific probe, it is time to test and debug each board individually first. When it is confirmed that this is working, the debug of both targets at the same time can be started as follow:

1. Start to debug HW_A.
2. The perspective switches automatically to the Debug perspective in STM32CubeIDE when a debug session for HW_A is started.
3. Switch to the C/C++ perspective.
4. Select the project for HW_B and start debugging it. The Debug perspective opens again.
5. There are two application stacks/nodes in the Debug view, one for each project. When changing the selected node in the Debug view, the related editor, the Variable view and others views are updated to present information associated to the selected project.

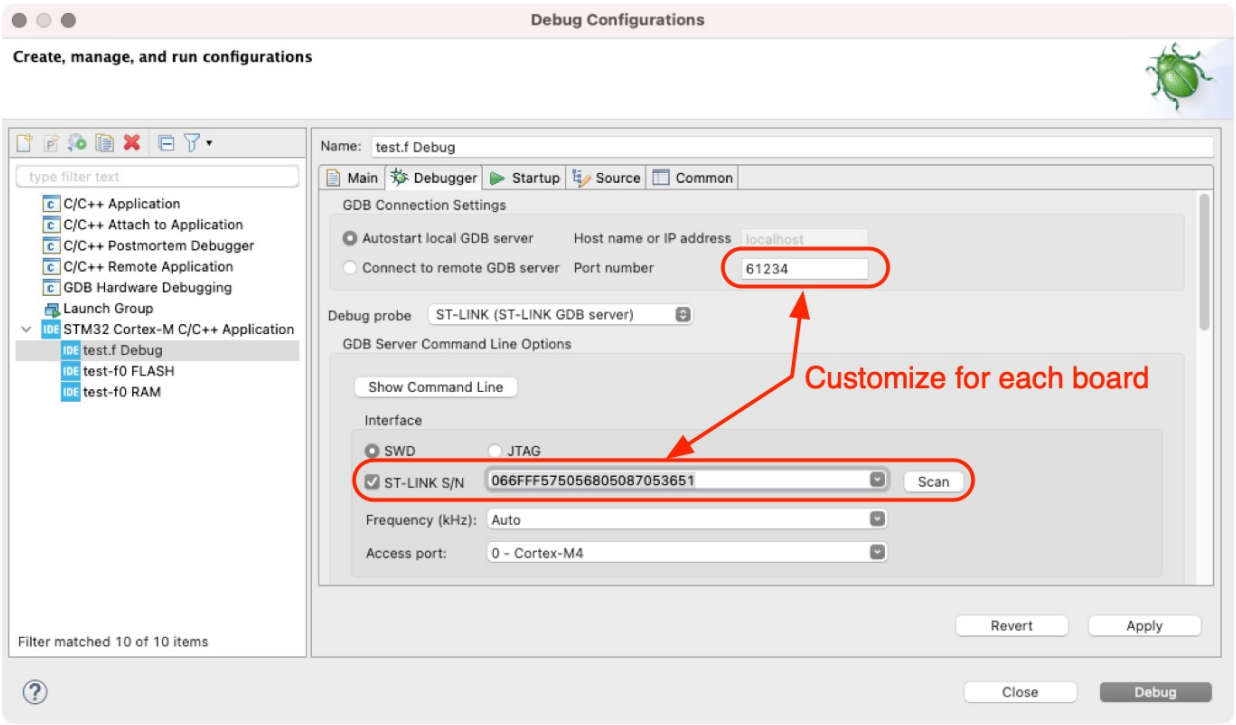


Figure 24.31: How to condigure Debug Configurations fields when using two ST-LINK simultaneously

24.7 ARM Semihosting

ARM semihosting is a distinctive feature of the Cortex-M platform, and it is extremely useful for testing and debug purpose. It is a mechanism that allows target boards to "exchange messages" from

the embedded firmware to a host computer running a debugger. In Cortex-M0/0+ core, where the ITM interface is not available, this mechanism enables `printf()` and `scanf()` functions if there is no way to use one of the integrated UARTs. However, the usage of the ARM semihosting is not limited to the message printout. Semihosting allows to access to other host PC I/O capabilities, such as terminals and files. This last feature is extremely useful when you have to transfer large amount of data between the target board and the host PC.

Semihosting requires additional runtime library code, and it can be implemented in several ways on Cortex-M architecture. However, the preferred one is using the `bktp` ARM assembly instruction, as we will see later. The next paragraph will give a quick explanation of how to configure our STM32CubeIDE project to use semihosting in our code. This will allow us to print messages on the OpenOCD console.

24.7.1 Enable Semihosting on a Project

To use semihosting with STM32CubeIDE some updates are needed to be done in the project. A debugger supporting semihosting is also required. This article guides on how to enable semihosting when using OpenOCD, ST-LINK and STM32 devices. Please note that ST-LINK GDB server does not support semihosting, and for this reason we will configure the *Debug configuration* to run OpenOCD.

To enable semihosting follow these steps:

1. Exclude from compilation (or delete it completely) the file **Core/Src/syscalls.c**. You can easily perform this task by right-clicking on the file in the **Project Explorer** pane and then selecting **Resource Configuration->Exclude from Build**....
2. Add the following two lines in beginning of **Core/Src/main.c** to include `<stdio.h>` and to use `initialise_monitor_handles()` prototype and add a call to `initialise_monitor_handles()` function in the beginning of `main()` function.

```
#include <stdio.h>
extern void initialise_monitor_handles(void);
...
int main(void) {
  /* USER CODE BEGIN 1 */
  initialise_monitor_handles();
```

3. Update GCC Linker project configuration adding `rdimon` library to the linked libraries, as shown in **Figure 24.32**.

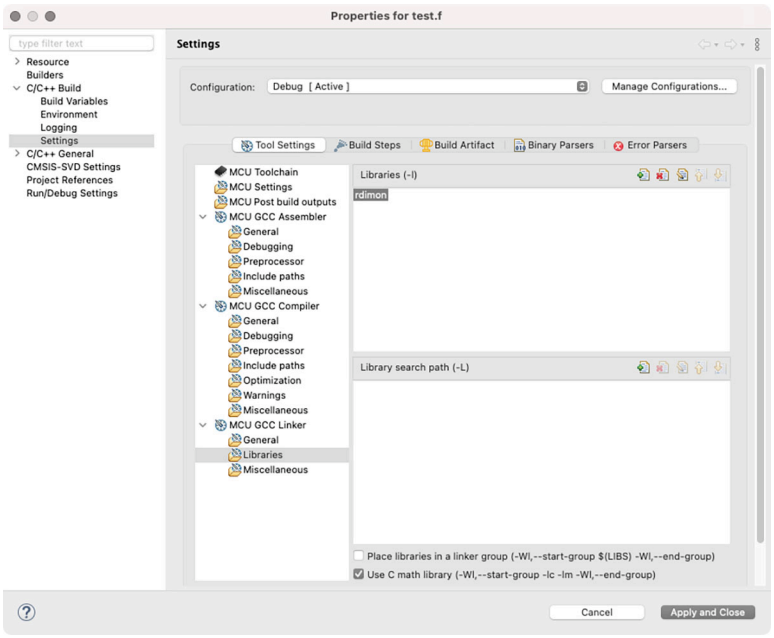


Figure 24.32: Configure the project to link rdimon library

4. Update GCC Linker project configuration adding "`-specs=rdimon.specs`" flag In Miscellaneous as shown in **Figure 24.33**.

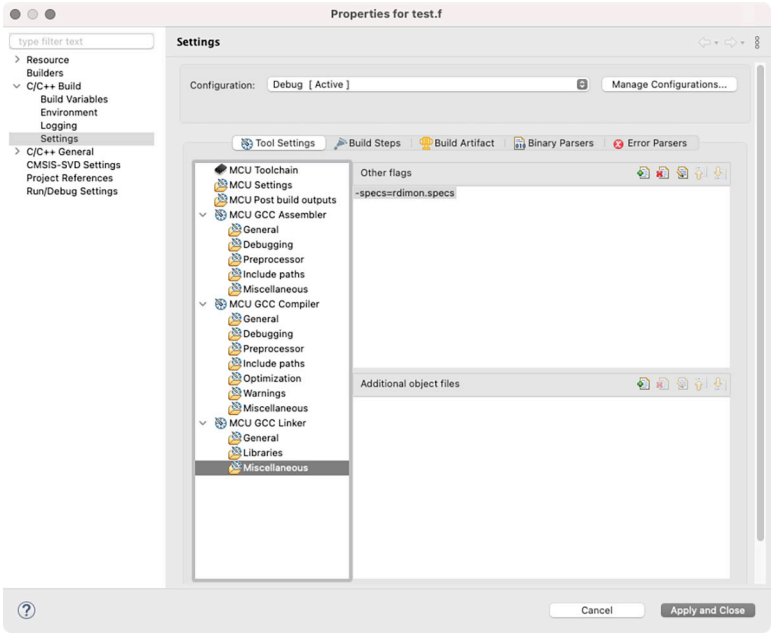


Figure 24.33: Configure the project to link add `rdimon.specs` flag

5. Configure the current *Debug configuration* by selecting **ST-LINK (OpenOCD)** in the **Debug probe** field and add "`monitor arm semihosting enable`" initialization command in *Debug Configurations - Startup tab*, as shown in **Figure 24.34**. The **Debug Console** view is used for semihosting

input/output and when debugging, as shown in **Figure 24.35**.

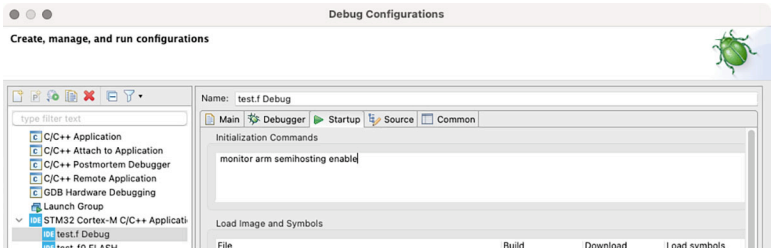


Figure 24.34: Configure the *Debug Configuration* to enable semihosting with OpenOCD

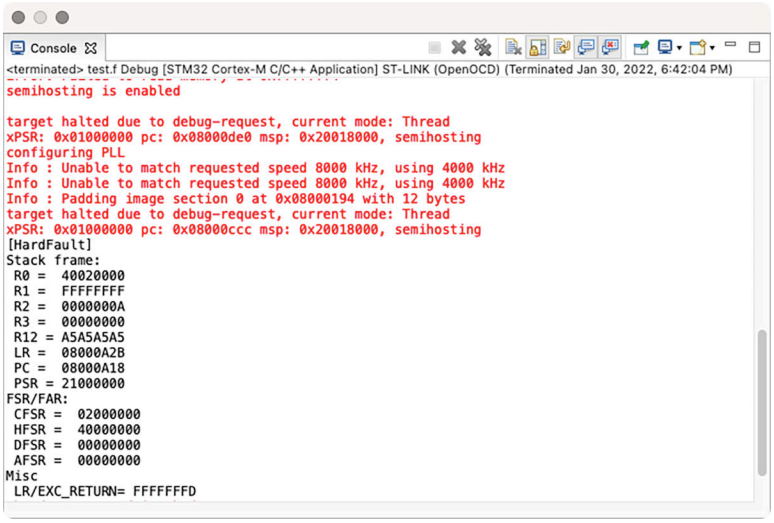


Figure 24.35: The output messages by using semihosting in the Debug Console view



Read Carefully

Semihosting implementation in OpenOCD is designed so that every string must be terminated with the newline character (`\n`) before the string appears on the OpenOCD console. This is a quite common error, and it leads to a lot of frustration the first times programmers start using it. Never forget to terminate every string passed to `printf()` routine with the (`\n`).

24.7.2 Semihosting Drawbacks

Semihosting is an excellent feature, but it has also several drawbacks. First of all, it works only during a debug session, and it completely hangs the firmware if not running under the GDB control. For example, upload am example using semihosting on your Nucleo board and terminate the debug session. If you reset your board pressing the RESET button, you will see that the firmware hangs. This happens because the firmware is stuck in the `printf()` routine (more about why this happens in the next paragraph). This is a quite frequent issue that every novice encounters every time it starts working with the STM32 platform.

Another important aspect to keep in mind is that semihosting has a great impact on the firmware performance. Every semihosting call costs several CPU cycles, and it impacts on the overall performance. Moreover, this cost in unpredictable, because it involves activities that happens outside the MCU execution streams (more about this in the next paragraph).

24.7.3 Understanding How Semihosting Works

There are several ways to implement semihosting capabilities. One of this is using software breakpoints. ARM Cortex-M offers two types of breakpoints: Hardware (HBP) and Software (SBP) breakpoints.

HBP is set by programming the *Break Point Unit* (a hardware unit inside every Cortex-M core) to monitor the core buses for an instruction fetch from a specific memory location. HBP can be set on any location in RAM or ROM using an external physical programmer connected to the Debug Interface. In case of the Nucleo board, the integrated ST-LINK programmer is connected to the MCU *Debug Interface*, and it is in turn managed by OpenOCD. **Figure 24.36** shows the relation between the external debugger and the internal MCU debug unit.

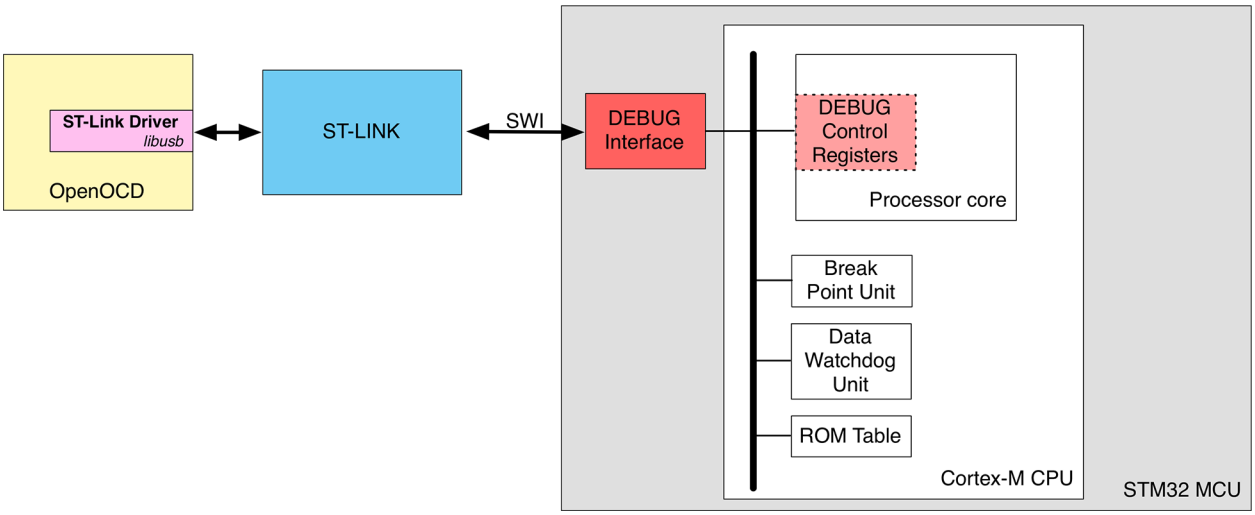


Figure 24.36: Debug components in Cortex-M microcontrollers

SBP are implemented by adding a special `bkpt` instruction immediately before the code we want to inspect. When the core executes the breakpoint instruction, it will be forced into debug state. The debugger sees that the MCU is halted and start debugging the MCU. The `bkpt` instruction accepts an immediate 8-bit opcode, which can be used to specify particular break conditions. If you want to halt the execution of your firmware and transfer the control to the debugger, then the instruction:

```
asm("bkpt #0")
```

is what you need. This technique is used to implement software conditional breakpoint. For example, suppose you have a strange fault condition that you need to inspect. You could have a piece of code like the following one:

```
if(cond == 0) {
   ...
} else if(cond > 0) {
   ...
} else { /* Abnormal state, let us debug it */
        asm("bkpt #0");
}
```

In the above code, if `cond` variable assumes a negative value, the MCU is halted and the control is transferred to GDB, which allows us to inspect the call stack and the current stack frame.

Semihosting is implemented using the special immediate opcode `0xAB`. That is, the instruction:

```
asm("bkpt #0xAB")
```

causes the MCU to stop, but this time OpenOCD sees the special opcode and interprets it as semihosting operation. By convention, the `r0` register contains the type of operation (`_write()`, `_read()`, etc) and the `r1` register contains the pointer to the region of memory containing the function parameters. For example, if we want to write a null terminated string on the host PC console, then we can write the following assembly instructions:

```
char msg[] = "Hello World!\r\n";
asm (
   "mov r0, 0x4 \n"      /* OPCODE for WRITE0 */
   "mov r1, %[addr] \n"  /* Address of string to transfer to OpenOCD */
   "bkpt #0xAB"
   :
   : [addr] "r" (msg)    /* Pass the reference to the msg string */
);
```

Here we use the capabilities of GCC `asm()` function to pass the pointer of the `msg` buffer containing the string "`Hello World!\r\n`".

Table 24.12 summarizes the supported semihosting operations. Please, take note that OpenOCD currently does not support all of them.

Now you can understand why semihosting causes the MCU to become stuck if the debugger is not active. The `bkpt` instruction halts the MCU execution, and there is no way to restore it without using an external debugger (or doing a hardware reset). Moreover, every time the `bkpt` instruction is issued, the internal MCU activities are suspended until the control passes to the debugger. During this time, important asynchronous events (like interrupts generated by peripherals) could be lost. This interval time is totally unpredictable, and it depends on many factors (speed of the hardware interface, current Host PC load, speed of ST-LINK firmware, etc., etc.).

Table 24.12: Summary of semihosting operations

Semihosting operation	immediate opcode	Description
EnterSVC	0x17	Sets the processor to Supervisor mode and disables all interrupts by setting both interrupt mask bits in the new CPSR.
ReportException	0x18	This SVC can be called by an application to report an exception to the debugger directly. The most common use is to report that execution has completed, using ADP_Stopped_ApplicationExit.
SYS_CLOSE	0x02	Closes a file on the host system. The handle must reference a file that was opened with SYS_OPEN.
SYS_CLOCK	0x10	Returns the number of centiseconds since the execution started.
SYS_ELAPSED	0x30	Returns the number of elapsed target ticks since execution started. Use SYS_TICKFREQ to determine the tick frequency.
SYS_ERRNO	0x13	Returns the value of the C library errno variable associated with the host implementation of the semihosting SVCs.
SYS_FLEN	0x0C	Returns the length of a specified file.
SYS_GET_CMDLINE	0x15	Returns the command line used to call the executable, that is, argc and argv.
SYS_HEAPINFO	0x16	Returns the system stack and heap parameters. The values returned are typically those used by the C library during initialization.
SYS_ISERROR	0x08	Determines whether the return code from another semihosting call is an error status or not. This call is passed a parameter block containing the error code to examine.
SYS_ISTTY	0x09	Checks whether a file is connected to an interactive device.
SYS_OPEN	0x01	Opens a file on the host system. The file path is specified either as relative to the current directory of the host process, or absolute, using the path conventions of the host operating system.
SYS_READ	0x06	Reads the contents of a file into a buffer.
SYS_READC	0x07	Reads a byte from the console.
SYS_REMOVE	0x0E	Deletes a specified file on the host filing system.
SYS_RENAME	0x0F	Renames a specified file.
SYS_SEEK	0x0A	Seeks to a specified position in a file using an offset specified from the start of the file. The file is assumed to be a byte array and the offset is given in bytes.
SYS_SYSTEM	0x12	Passes a command to the host command-line interpreter. This enables you to execute a system command such as `dir`, `ls`, or `pwd`. The terminal I/O is on the host, and is not visible to the target.
SYS_TICKFREQ	0x31	Returns the tick frequency.

Table 24.12: Summary of semihosting operations

Semihosting operation	immediate opcode	Description
SYS_TIME	0x11	Returns the number of seconds since 00:00 January 1, 1970.
SYS_TMPNAM	0x0D	Returns a temporary name for a file identified by a system file identifier.
SYS_WRITE	0x05	Writes the contents of a buffer to a specified file at the current file position. Current OpenOCD implementation expects that the buffer is terminated with the newline character (\n).
SYS_WRITEC	0x03	Writes a character byte, pointed to by R1, to the debug channel. When executed under an ARM debugger, the character appears on the host debugger console.
SYS_WRITE0	0x04	Writes a null-terminated string to the debug channel. When executed under an ARM debugger, the characters appear on the host debugger console.

25. FAT Filesystem

Electronic embedded devices are increasingly complex and nowadays it is common to have devices that need to read and store structured data. For example, consider an Internet-enabled device, which needs to serve HTTP requests and to transfer HTML files. Unless HTML pages are really simple, this device will need a way to handle several and separated HTML files, as well as CSS stylesheets and JavaScript files. For this reason, a lot of embedded developers need a way to handle structured filesystems in their applications.

ST has integrated in its CubeHAL a well-known library to manipulate FAT filesystems (FAT12, FAT16 and FAT32): the FatFs library by Chan[1]. This is a library expressly designed for embedded system, with limited SRAM and flash memories. It is popular and it is proven to be robust.

This chapter provides a quick introduction to this middleware library. It describes how to use CubeMX to generate a project that integrates it and how to develop applications based on this useful library. Moreover, we will see how to interface SD cards using SPI, which represents the most widespread way to use memory cards with low-cost embedded microcontrollers.

25.1 Introduction to FatFs Library

The *File Allocation Table* (FAT) is a filesystem architecture designed by Microsoft in the early '80 and used as official filesystem for the MS-DOS and Windows Operating Systems until the Windows NT 3.1 release. FAT filesystem was superseded by the more advanced NTFS, which offers improved support for metadata, and the use of advanced data structures to improve performance, reliability, and disk space utilization, plus additional extensions, such as security access control lists (that is, file permissions) and file system journaling.

Thanks to its simplicity and robustness, the FAT file system is still commonly found on USB-based memories, flash and other solid-state memory cards and modules like SD cards, and many portable and embedded devices. Technically, the term "FAT file system" refers to all three major variants of the file system: FAT12, FAT16 and FAT32. Those numbers essentially indicate how many bits are used to address filesystem *clusters*, a contiguous area of disk storage. The more clusters the filesystem can handle, the more bytes can be used. That's the reason why FAT32 is nowadays the most-used filesystem on large solid-state and removable memories. A disk as well as a solid-state memory initialized with the FAT filesystem can have an arbitrary number of partitions.

The FatFs is a space-optimized[2] library which provides the following features:

[1] https://bit.ly/2d6QUC5

[2] For the sake of completeness, we have to say that the same author made an even smaller version of the FatFs library, named Petit FatFs(https://bit.ly/2drLLAa), which is best suitable for 8-bit MCUs. It essentially implements a subset of the main FatFs library.

- Supports FAT12, FAT16, FAT32(r0.0) and exFAT(r1.0) filesystems.
- Allows un unlimited number of open files (the only limit is the available SRAM memory).
- Supports up to 10 volumes each one with a size up to 2 TiB at 512 bytes/sector.
- Every file can grow up to 4 GiB on FAT volume and virtually unlimited on exFAT volume.
- The cluster goes up to 128 sectors on FAT volume and up to 16 MiB on exFAT volume.
- Supports 4 different sector sizes: 512, 1024, 2048 and 4096 bytes.

The FatFs library provides up to 37 APIs, and they can be selectively disabled thanks to several configuration macros. In fact, to reduce the flash memory footprint, it is possible to disable unneeded functionalities. FatFs library is coded in pure ANSI C and it is totally abstracted from the underlying hardware. The official library does not provide any support to specific memory technology devices, and it is up to the user to implement the necessary glue to interface the hardware.

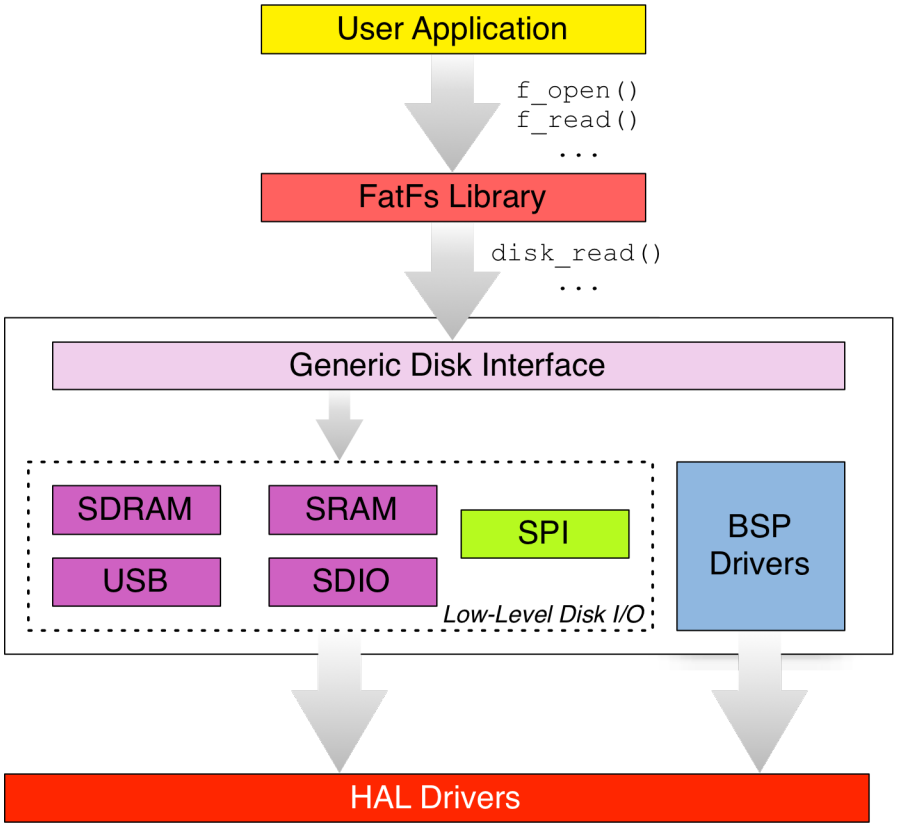


Figure 1: How the FatFs library interfaces the underlying hardware

ST engineers have integrated the FatFs library in the CubeHAL. They have developed necessary adapters to use FatFs library with the following devices:

- **SD memory cards using the SDIO peripheral**: the *Secure Digital Input Output* (SDIO) is an extension of the SD specification that covers I/O functions related to SD and MMC cards. More advanced STM32 microcontrollers, such as some STM32F4 ones (for example, the STM32F401RE) and STM32F7 microcontrollers, provide a this dedicated peripheral. The SDIO

interface can be configured to work in 1 bit mode (that is, data is transferred to the SD using just one output data port, named `DO`, plus two additional I/Os for the clock and commands transfer), or to work in 4-bit mode (that is data is transferred using 4 dedicated I/Os in addition to clock and command lines). This is the fastest way to use SD cards, and the maximum transfer rate of 50MHz can be reached in high-performing STM32 microcontrollers.
- **Static as well as Dynamic RAM memories**: two separated low-level drivers for SDRAM and SRAM memories allow to create filesystems in RAM. These two drivers work in combination with FMC and FSMC controllers. They allow to initialize RAM disks and this feature is especially useful when performances are critical for your application (SRAM are a lot faster than NVM memories).
- **USB-based disks**: a specific driver built upon the ST USB library allows to create USB Host devices supporting the *Mass Storage Class* (MSC) (that is a device that can interface USB disks).

Figure 1 shows the relation between the FatFs library and the CubeHAL. Unfortunately, ST engineers have not still developed a driver for SD cards when working in SPI mode. In fact, SD cards are designed to support, among the other protocols, commands exchanged through the SPI bus. However, I have arranged a complete SPI-compliant SD driver that I will introduce you later.

To integrate the FatFs library with a memory device we essentially need to implement the following six routines:

- `disk_initialize()`: this routine contains all the necessary code to initialize the hardware device. For example, for an SD card working in SPI mode this routine must contain all the necessary code to initialize the SPI interface and to place the SD in SPI mode (there exists a specific procedure to follow, as documented on Chan's website[3]).
- `disk_status()`: this function is used by the library to get information about the device status (for example, if it is initialized, etc.).
- `disk_read()`: this routine is used to retrieve a given number of sectors from the memory device, starting from a specified sector.
- `disk_write()`: as its name suggests, this function is used to store a given number of sector on the device.
- `disk_ioctl()`: this function reads and configures some specific device parameters, such as the size of sectors, the device power state, and so on.
- `get_fattime()`: returns the current time so that files can have a valid timestamp. If the MCU does not provide an RTC unit, then this function can return `0`.

Moreover, the last three routines are needed only if the FatFs library is compiled with the option `_FS_READONLY == 0`. That is, we can avoid providing a valid implementation for those functions if we use the FatFs read only mode.

[3] http://bit.ly/2dtWpWS

25.1.1 Adding FatFs Library in Your Projects

As said before, FatFs library is a component of the CubeHAL framework, and CubeMX supports it. However, the way CubeMX handles this library is a little bit counterintuitive, at least for beginners.

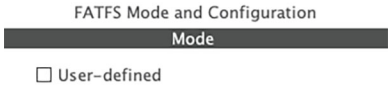


Figure 2: What CubeMX shows for those STM32 MCUs that does not provide a compatible adapter

Many of you will notice that CubeMX shows just one option related to the FatFs middleware library, as shown in **Figure 2**. The obscure **User-defined** entry will appear for the majority of STM32 microcontrollers. But what exactly that means? It simply means that your specific STM32 microcontroller provides no peripheral compatible with the adapters developed by ST developers (SRAM/SDRAM, USB, and SDIO), and you will need to provide your own implementation for the low-level I/O drivers.

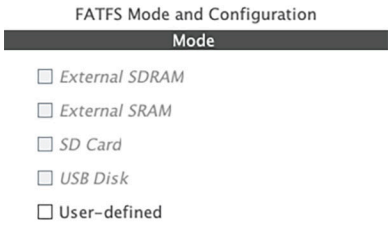


Figure 3: What CubeMX shows for an STM32F746VG MCU

Figure 3, instead, shows the options available if you use an STM32F779BI MCU, which provides the SDIO interface[4], the FMC controller and a USB-device interface. However, as you can see in **Figure 3**, the generation options appear grayed out. This happens because we need to enable the corresponding peripheral first, and then check the wanted FatFs configuration. For example, let us assume that we are working on an STM32F401RE MCU, which provides a SDIO peripheral. We first need to enable the wanted SDIO mode (1-bit, 4-bit, etc) in the *IP Tree* view, and then check the corresponding FatFs option.

The generated project has a structure similar to the one shown in **Figure 4**. The **Middlewares/Third_-Party/FatFs/src** folder contains the FatFs library, while the **FATFS/Target/sd_diskio.c** file contains I/O routines to handle SD cards through the specific interface (the SDIO). Those routines are abstracted from the specific board configurations, and they rely on APIs that are implemented inside the **src/bsp_driver_sd.c** file. The routines contained in that file use in turn CubeHAL functions (from the `HAL_SD` module for the SDIO).

[4] In STM32F7 MCUs the SDIO peripheral is called SDMMC.

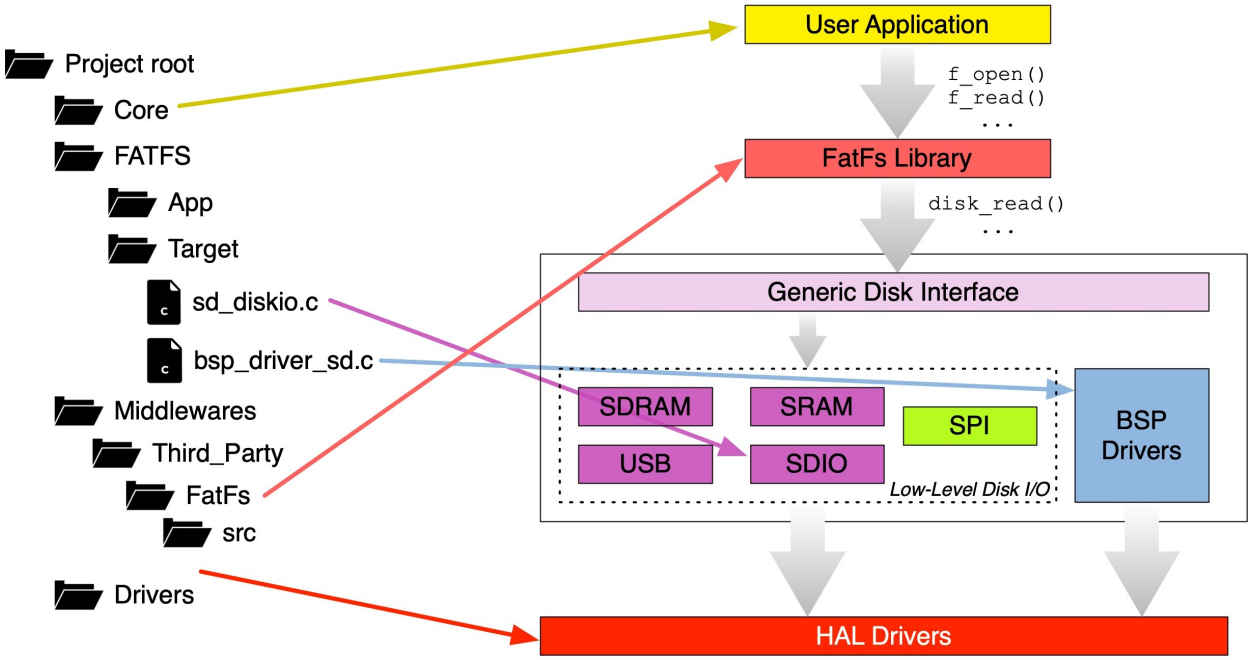


Figure 4: The structure of a generated project with the FatFs middleware

If, instead, you generate a project choosing the **User-defined** option, then you will find the file **FATFS/Target/user_diskio.c**, which contains the functions `USER_initialize()`, `USER_status()`, `USER_read()`, `USER_write()` and `USER_ioctl()`. Those routines are empty templates, and they need to be filled with the code to drive your specific memory device.

25.1.1.1 The *Generic Disk Interface* API

ST Engineers have developed another abstraction layer between the FatFs library and the low-level device drivers. This is called *Generic Disk Interface* layer and it is essentially an abstraction layer that allows to handle multiple disk-drivers in the same application. It resembles the *Virtual Filesystem* in the Linux Operating System.

Each device driver in this layer corresponds to an instance of the following C `struct`:

```
typedef struct {
  DSTATUS (*disk_initialize) (BYTE);
  DSTATUS (*disk_status)     (BYTE);
  DRESULT (*disk_read)       (BYTE, BYTE*, DWORD, UINT);
#if _USE_WRITE == 1
  DRESULT (*disk_write)      (BYTE, const BYTE*, DWORD, UINT);
#endif /* _USE_WRITE == 1 */
#if _USE_IOCTL == 1
  DRESULT (*disk_ioctl)      (BYTE, BYTE, void*);
#endif /* _USE_IOCTL == 1 */
} Diskio_drvTypeDef;
```

This is nothing more than a C `struct` containing five function pointers, which correspond to the implementation of those routines needed by the FatFs to handle the access to the specific memory device. The function:

```
uint8_t FATFS_LinkDriver(Diskio_drvTypeDef *drv, char *path);
```

is responsible of linking an instance of that `struct` to a given mount path (e.g. the path "0:/" to address the volume 0).

Thank to this little improvement by ST guys, we can use different filesystems using different devices (for example a USB-disk filesystem in conjunction with another filesystem stored in the SD card).

25.1.1.2 The Implementation of a Driver to Access SD Cards in SPI Mode

SD memory cards are more than simple flash memories. They also include a dedicated processor, which implements all the logic to exchange data through the SD interface (answering to several communication protocols) and to handle the proper access to the specific flash memory (NOR, NAND, etc.). Moreover, all SD cards implement wear-leveling techniques for lasting the service life of erasable flash memories.

One distinctive feature of SD cards is the ability to answer to commands and messages exchanged over an SPI bus[5]. This allows them to be used in conjunction with low-cost microcontrollers (even 8-bits ones are suitable for this operation), and this explains their popularity in embedded applications. I will not describe here the SPI protocol supported by SD cards. Chan provides[6] sufficient information to get started with this matter. To repeat it here is useless and counterproductive. Chan also provides several example projects that show how to interface SD card through the SPI interface.

The next chapter will show how to use SD cards in SPI mode to serve web pages stored on the SD cards in a web-based embedded application.

25.1.2 Relevant FatFs Structures and Functions

We are now going to analyze the most important structures and functions provided by the FatFs library to manipulate FAT drives[7].

25.1.2.1 Mounting a Filesystem

Before accessing any file or directory on the filesystem, we need to mount[8] it by using the function:

[5] However, the implementation of this functionality is not mandatory for SD manufacturers. There exist several SD cards on the market that do not implement the SPI specification or, at least, they do not implement it literally.

[6] https://bit.ly/2dtWpWS

[7] The full FatFS API is documented on the Chan's website

[8] The *mounting* of a disk is an operation performed by the filesystem driver that essentially consists in gathering all the logical information associated to a physical drive or a part of it (number of partitions, partition dimension, dimension of clusters, number of clusters and so on). It is impossible to use any filesystem primitive (directories, files, etc.) before mounting it.

```
FRESULT f_mount(FATFS *fs, const TCHAR *path, BYTE opt);
```

`fs` is an instance of the C `struct FATFS`, which holds information about the logical drive (partition); `path` is a pointer to the null-terminated string that specifies the logical drive (more about this in a while); `opt` can assume the value `0` to delay the filesystem mounting until to the first access to the volume (for example, opening a file or a directory) or it can assume the value `1` to immediately mount the logical volume. Application program must not modify any member the `FATFS` structure, otherwise the original logical/physical disk could me irremediably damaged.

The format of `path` parameter is similar to the drive name specification in Windows Operating System, at it can assume the form `N:/`, where N is a number starting from `0` which uniquely identifies a logical drive. By default, each physical drive can have just one logical drive (that is, a partition). This means that if our disk has more than one partition, then only the first partition in the partitions table will be mounted and assigned to a logical drive. Instead, by setting the macro `_MULTI_PARTITION=1` in the `ffconf.h` file, FatFs library will associate a logical drive to every partition in a physical disk. When drive number is omitted, the drive number is assumed as default drive (drive `0` or current drive). We can so pass to the `path` parameter slash or backslash chars (`\` or `/`) or even specify a `NULL` string. For example, the following code forces the mounting of the first partition on a physical drive:

```
FATFS fs;
f_mount(&fs, "/", 1);
```

If the logical disk is correctly mounted, then the `f_mount()` function returns the value `FR_OK`. Otherwise, a series of error conditions may be returned (`FR_INVALID_DRIVE`, `FR_DISK_ERR`, `FR_NOT_READY`, `FR_NO_FILESYSTEM`).

25.1.2.2 Opening a File

Once the drive is mounted, we can open a file by using the function:

```
FRESULT f_open(FIL* fp, const TCHAR* path, BYTE mode);
```

`fp` is an instance of the C `struct FIL`, which holds information about the open file (its name, size, starting cluster, and so on); `path` corresponds to the filesystem path to reach the file (more about this soon); `mode` specifies the type of access and open method for the file, and it can assume a value from **Table 1**.

Table 1: Lists of file open methods

Value	Description
FA_READ	Specifies read access to the object. Data can be read from the file.
FA_WRITE	Specifies write access to the object. Data can be written to the file. It can be combined (logically or-ed) with FA_READ for read-write access.
FA_OPEN_EXISTING	Opens the file. The function fails if the file is not existing. (Default)
FA_CREATE_NEW	Creates a new file. The function f_open() fails with FR_EXIST if the file is existing.
FA_CREATE_ALWAYS	Creates a new file. If the file is existing, it will be truncated and overwritten.
FA_OPEN_ALWAYS	Opens the file if it is existing. If not, a new file will be created.
FA_OPEN_APPEND	Same as FA_OPEN_ALWAYS except read/write pointer is set end of the file.

Regarding the file path, this corresponds to the full filesystem path to the file, including its name. For example, `0:\dir1\filename.txt` opens the file named `filename.txt` inside the directory named `dir1` on the first logical disk. If our application uses just one logical drive, then we can simply specify the path in this other form: `\dir1\filename.txt`. Please, take note that FatFs is able to process paths both in Windows and UNIX form. So, following the previous example, we can also specify the path in this equivalent form: `0:/dir1/filename.txt`.

If the file is correctly opened, then the `f_open()` function returns the value `FR_OK`. Otherwise, a series of error conditions may be returned (see documentation[9] for more about this).

25.1.2.3 Reading From/Writing into a File

Once the file is opened, we can read data from it or write new data from it according to the file opening `mode`. To do this, the FatFs library provides the following functions:

```
FRESULT f_read(FIL* fp, void* buff, UINT btr, UINT* br);
FRESULT f_write(FIL* fp, const void* buff, UINT btr, UINT* br);
```

`fp` corresponds to the file handler passed to the `f_open()` function; `buff` is the pointer to the buffer containing the data to read from the file or to store in it; `btw` specifies the number of bytes to read/write; `bw` corresponds to the amount of bytes effectively read/written.

The following example shows an application of the functions seen before. It is nothing more than a file copy procedure.

[9] https://bit.ly/3HqA8XO

```c
#define BUF_LEN 2048

FRESULT copy_file (char *srcPath, char *dstPath) {
    FATFS fs;             /* File system object corresponding to logical drive */
    FIL fsrc, fdst;       /* File objects */
    BYTE buffer[BUF_LEN]; /* File copy buffer */
    FRESULT fr;           /* FatFs function common result code */
    UINT br, bw;          /* File read/write count */

    /* Mount the filesystem */
    f_mount(&fs[0], "0:", 0);

    /* Open source file */
    fr = f_open(&fsrc, srcPath, FA_READ);
    if (fr) return (int)fr;

    /* Create destination file */
    fr = f_open(&fdst, dstPath, FA_WRITE | FA_CREATE_ALWAYS);
    if (fr) return (int)fr;

    /* Copy source to destination */
    while(1) {
        /* Read 'BUF_LEN' bytes from source file */
        fr = f_read(&fsrc, buffer, BUF_LEN, &br);
        if (fr != FR_OK || br == 0) break; /* Error condition or EOF */
        /* Write read data to the destination file */
        fr = f_write(&fdst, buffer, br, &bw);
        if (fr != FR_OK || bw < br) break; /* Error or disk full */
    }

    /* Close open files */
    f_close(&fsrc);
    f_close(&fdst);

    /* Unmount volume */
    f_mount(NULL, "0:", 0);

    return fr;
}
```

25.1.2.4 Creating and Opening a Directory

The FatFs library allows to easily manipulate files as well as directories. To create a new directory, we can use the function:

```
FRESULT f_mkdir(const TCHAR* path);
```

which accepts the full `path` to the directory to create. For example, if our filesystem already stores a directory named `dir1` in its root, then to create a subdirectory we can pass the string `"0:/dir1/subdir1"` to create a subdirectory within it.

To open an already existing directory, we can use the function:

```
FRESULT f_opendir(DIR* dp, const TCHAR* path);
```

`dp` is an instance of the C `struct DIR` that represent the handle to the opened directory; `path` is the full path to the directory we want to open. If a valid handle is returned by the `f_opendir()` function, then we can read its content by using the function:

```
FRESULT f_readdir(DIR* dp, FILINFO* fno);
```

`dp` is the instance corresponding to the directory opened using the `f_opendir()` function; `fno` is an instance of the `FILINFO struct` that holds information regarding the current item in the directory. The `f_readdir()` function works in this way. Once a directory is opened, we invoke `f_readdir()` until it returns a value different from `FR_OK` (in case of an error occurred) or the `fno.fname` entry is null. This last condition indicates that we reached the end of a directory and there are no more elements (files or directories) to retrieve.

The `FILINFO struct` is defined in the following way:

```
typedef struct {
        DWORD  fsize;     /* File size */
        WORD   fdate;     /* Last modified date */
        WORD   ftime;     /* Last modified time */
        BYTE   fattrib;   /* Attribute */
        TCHAR  fname[13]; /* Short file name (8.3 format) */
#if _USE_LFN
        TCHAR* lfname;    /* Pointer to the LFN buffer */
        UINT   lfsize;    /* Size of LFN buffer in TCHAR */
#endif
} FILINFO;
```

Let us analyze the fields of this struct.

- `fsize`: it stores the size of the file in bytes. It is meaningless if the object is a directory.
- `fdate`: indicates the date when the file was modified or the directory was created, and it has the following structure
  - bit [15:9]: year origin from 1980 (0..127)
  - bit [8:5]: month (1..12)

  - bit [4:0]: day (1..31)
- `ftime`: indicates the time when the file was modified or the directory was created, and it has the following structure
  - bit [15:11]: hour (0..23)
  - bit [10:5]: minute (0..59)
  - bit [4:0]: second / 2 (0..29)
- `fattrib`: corresponds to the file/directory attribute, which is a combination of the attributes listed in **Table 2**.
- `fname`: this null terminated string corresponds to the file/directory name in FAT 8:3 format. A NULL string is stored when there are no longer items to read, and it indicates that the structure is invalid.
- `lfname`: this null terminated string corresponds to the file/directory name when the support to long file names is enabled (`_USE_LFN` macro != 0). More about this later.

Table 2: Lists of file/directory attributes

File/Directory attribute	Description
`AM_RDO`	Read only
`AM_ARC`	Archive
`AM_SYS`	System
`AM_HID`	Hidden

The following example shows a routine that performs a depth-first traversal of the filesystem, printing the name of files and folders using the `trace_printf()` routine. The routine uses `f_opendir()` and `f_readdir()` functions to retrieve the content of each directory. The `_USE_LFN` macro allows to properly handle the support to long file names. Its usage will be explained in the next paragraph.

```
FRESULT scan_files (TCHAR* path) {
    FRESULT res;
    DIR dir;
    UINT i;
    static FILINFO fno;
    static TCHAR lfname[_MAX_LFN];
    TCHAR *fname;

    res = f_opendir(&dir, path); /* Open the directory */
    if (res == FR_OK) {
        while(1) {
#if _USE_LFN > 0
            fno.lfname = lfname;
            fno.lfsize = _MAX_LFN - 1;
#endif
            /* Read a directory item */
            res = f_readdir(&dir, &fno);
```

```
            /* Break on error or end of directory */
            if (res != FR_OK || fno.fname[0] == 0) break;
#if _USE_LFN > 0
            fname = *fno.lfname ? fno.lfname : fno.fname;
#endif
            if (fno.fattrib & AM_DIR) { /* It is a directory */
                i = strlen(path);
                sprintf(&path[i], "/%s", fname);
                /* Scan directory recursively */
                res = scan_files(path);
                if (res != FR_OK) break;
                path[i] = 0;
            } else { /* It is a file. */
                trace_printf("%s/%s\n", path, fname);
            }
        }
        f_closedir(&dir);
    }
    return res;
}
```

25.1.3 How to Configure the FatFs Library

The FatFs library is highly customizable. A set of configuration parameters (that is, configuration macros) allows to reduce the total footprint of the library and enable/disable some features at compile time.

All the configuration parameters are automatically exported by CubeMX inside the **FATFS/Target/ffconf.h** file. We are now going to analyze the most relevant ones. For a more complete treatment of this subject the reader should refer to the official documentation[10].

- `_FS_TINY`: this macro can assume the values 0 (default) and 1. This option allows to reduce the SRAM occupation of the FatFs library. When set to 0, every instance of the `FIL struct` has an its own temporary buffer used during file read/write. Instead, when set to 1, a global pool defined in `FATFS struct` is used. This slows down the read/write operations.
- `_FS_READONLY`: this macro can assume the values 0 (default) and 1. This macro skips from compilation those functionalities used to modify the filesystem, by removing writing API functions such as `f_write()`, `f_sync()`, `f_unlink()`, `f_mkdir()`, `f_chmod()`, `f_rename()`, `f_truncate()`, `f_getfree()` and optional writing functions as well.
- `_FS_MINIMIZE`: this macro defines the library minimization level, which consists in removing some API functions. If set to 0, all APIs are available. If set to 1, `f_stat()`, `f_getfree()`, `f_unlink()`, `f_mkdir()`, `f_chmod()`, `f_utime()`, `f_truncate()` and `f_rename()` functions are removed. If set to 2, `f_opendir()`, `f_readdir()` and `f_closedir()` routines are removed in

[10] https://bit.ly/2dJGh3E

addition to those ones when _FS_MINIMIZE is set to 1. If set to 3, the f_lseek() function is also removed.

- _CODE_PAGE: this option specifies the OEM code page to be used on the target system. Incorrect setting of the code page can cause a file open failure. For western countries, the CP1252 (aka Latin1) is the most widespread code page used in Windows OS.
- _USE_LFN: this option switches the support for *Long File Names* (LFN). When enabling the LFN mode, Unicode support functions contained in the option/unicode.c file needs to be added to the project. Moreover, when enabling LFN support, we need to provide a pre-allocated buffer to store the long file name. This requirement is not well documented in the FatFs library, and unexperienced people waste a lot of time trying to figure out how to retrieve LFN names. The scan_files() routine seen so far clearly shows how to perform this operation. The buffer lfname is statically allocated on the stack at line 6. Then the fno.lfname pointer is set to point to the lfname buffer at line 13. In the same way, the size of the buffer is specified at line 14. This allows to the FatFs library to properly retrieve the LFN name of a file or a directory. Otherwise, the fno.fname buffer contains the filename in the 8.3 format.
- _MAX_LFN: this macro specifies the maximum LFN length to handle (from 12 to 255).
- _LFN_UNICODE: this option switches character encoding on the API. (0:ANSI/OEM or 1:Unicode). To use Unicode strings for the path names, you have to enable LFN feature and set _LFN_UNICODE to 1. This option also affects behavior of string I/O functions. Please take note that to allow transparent usage of Unicode strings in FatFs library, the library itself defines the TCHAR datatype, which is automatically converted to a char if the Unicode support is disabled, or to an uint16_t if enabled. For more information about this topic, refer to the FatFs documentation[11].
- _VOLUMES: this macro sets the maximum number of logical drives per each physical disk. By default, this macro is set to 1. For more information about this topic, refer to the FatFs documentation[12].
- _FS_REENTRANT: this option switches the re-entrancy (thread safe) of the FatFs library itself. File/directory access to **different** volumes is always re-entrant and it can work simultaneously regardless of this option. Volume control functions (f_mount(), f_mkfs() and f_fdisk()) are always not re-entrant. File/directory access to the same volume is not reentrant, unless the _FS_REENTRANT macro is set to 1. To enable this feature, the user also needs to provide synchronization handlers routines, ff_req_grant(), ff_rel_grant(), ff_del_syncobj() and ff_cre_syncobj(). ST engineers have implemented these routines so that they use FreeRTOS semaphores. Look at the option/syscall.c file for more about this.
- _SYNC_t: when the _FS_REENTRANT macro is set to 1 it is required to also specify this macro, which specifies the "type" of synchronization structure. When using FreeRTOS in conjunction with the FatFs library, this macro is set to the osSemaphoreId type.

[11] https://bit.ly/2dJP9q1
[12] http://bit.ly/2dJPoBx

26. Develop IoT Applications

In a connected world, more and more devices are getting connected. Cars, home appliances such as washing machines and freezers, lights, blinds, thermostats, but also sensors for environmental monitoring are just few examples of devices that nowadays exchange messages through the Internet. Several observers agree by saying that this is the *Internet of Things* (IoT) era. Actually, it is hard to say if IoT will represent a new golden era for the electronics industry. But it is certain that many semiconductor companies are investing billions in this field.

IoT is a vague term. It does say nothing about communication standards, protocols, application layers and even system architectures. The world of IoT communication protocols and technologies is a jungle. There exist tens of standards, especially for wireless communication protocols. WiFi, Bluetooth, Zigbee, LoRaWAN, proprietary solutions like SimpliciTI from TI or MiWI from Microchip, 5G/4G mobile networks. But there exist even tens of communication mediums. For example, in wireless communications the 2.4GHz and 5.8GHz frequencies are worldwide standards, but there are several regional and alternative frequencies such as the 868MHz in EU (915MHz in US), 434MHz in several parts of EU and US, 169MHz in large parts of EU and Japan. Every one of these standards has its own rules about TX power, duty cycle, and so on. Everyone has its advantages and disadvantages.

The choice of communication medium and protocol also impacts on the application architecture. For example, a WiFi or Ethernet enabled device can connect to the Internet using just a router with integrated MODEM. A device that uses a proprietary protocol (for example, a Zigbee device) usually needs an intermediate device (a control unit) that collects the messages and sends them to a centralized server (or a *cloud server* as now are called centralized servers) through the Internet. For some "industrial" applications this is often an advantage (local devices can keep working even in absence of the Internet). For consumer applications usually this discourages users from adopting the solution.

Nowadays there are several silicon manufacturers that offer microcontrollers with integrated wired and wireless connectivity. Texas Instruments, after the acquisition of ChipCon, has developed several MCU with integrated radio front ends. For example, the CC2540 is an 8051 MCU with a 2.4GHz radio dedicated to Bluetooth applications. The CC3200 is a Cortex-M4 core with a 2.4GHz radio able to communicate according to the WiFi standard. Recently, another player appeared on the market. Espressif[1] is a Chinese company that introduced to the market a dual core Tensilica[2] LX6 microcontroller running at 240MHz with integrated 2.4GHz radio and MAC layer (ESP32[3]). These MCUs cost less than 5 USD at low volumes, and they are popular between makers.

This chapter provides a solution to owners of Nucleo-64 boards that do not implement an Ethernet

[1] https://espressif.com/

[2] Tensilica is a company that, like ARM, design IP cores that are later implemented by silicon manufacturers. It is currently owned by Cadence, the same company that designs the Allegro CAD.

[3] https://bit.ly/2dN52fx

controller. The proposed solution is based on the W5500 network processor from WIZnet[4], a Korean company specialized in designing this kind of devices. This company reached a great popularity thanks to the W5100 IC, which was used to develop the popular Arduino Ethernet Shield[5]. We will see how to develop an embedded application with an integrated web-server using the Nucleo. However, before entering in this stuff, I will provide a quick introduction to what ST offers to develop IoT applications with the CubeHAL initiative.

26.1 Solutions Offered by STM to Develop IoT Applications

As said before, several STM32 microcontrollers from F1/2/4/7-series provide an integrated Ethernet controller, which supports the *Media-Independent Interface* (MII) and its variant *Reduced-MII* (RMII). This is a communication standard that abstracts from the given physical medium and allows to connect the MAC layer of the ethernet controller to a physical controller chip, also called *phyther*. There exists several LAN phythers on the market, and the interconnection to the chip is all handled by the low-level MII protocol.

However, the existence of a dedicated hardware interface does not allow to build IoT applications quickly. A complete TCP/IP stack is also needed, otherwise to handle a so complex protocol stack like the TCP/IP is impossible for a single developer. ST does not provide a custom solution, but it adopted the *Lightweight IP* (LwIP) stack. LwIP is an Open Source framework started by Adam Dunkels and now maintained by a large community. Moreover, several other semiconductor companies, like Altera, Xilinx and Freescale, contribute to the development of this quite complex framework.

ST has integrated LwIP in CubeMX, which automatically adds to the project all the necessary files to work with this framework. Once you have enabled the Ethernet controller, the LwIP appears as a selectable middleware component. ST actively maintains and support it.

These are the most relevant features of LwIP:

- IP (Internet Protocol, IPv4 and IPv6) including packet forwarding over multiple network interfaces
- ICMP (Internet Control Message Protocol) for network maintenance and debugging
- IGMP (Internet Group Management Protocol) for multicast traffic management
- MLD (Multicast listener discovery for IPv6)
  - Aims to be compliant with RFC 2710. No support for MLDv2
- ND (Neighbor discovery and stateless address autoconfiguration for IPv6)
  - Aims to be compliant with RFC 4861 (Neighbor discovery) and RFC 4862 (Address autoconfiguration)
- UDP (User Datagram Protocol) including experimental UDP-lite extensions

[4] http://www.wiznet.co.kr/
[5] http://bit.ly/2dMXhGi

- TCP (Transmission Control Protocol) with congestion control, RTT estimation and fast recovery/fast retransmit
- Raw/native socket API for enhanced performance
- Optional Berkeley-like socket API
- DNS (Domain names resolver)

LwIP also provides complete implementation for the following application protocols:

- HTTP server with SSI and CGI
- SNMPv2c agent with MIB compiler (Simple Network Management Protocol)
- SNTP (Simple network time protocol)
- NetBIOS name service responder
- MDNS (Multicast DNS) responder
- iPerf server implementation

Due to the lack of a RMII interface in STM32 MCUs equipping the Nucleo-64 boards, I will not detail here the operations needed to setup LwIP in your applications. Refer to CubeHAL examples for more about this. Moreover, you can find some posts on my blog regarding this topic. Finally, the CubeMXImporter tool implements all the necessary logic to import the LwIP stack in a GNU MCU Eclipse project.

When it comes to IoT, the wireless protocols play an important role. And ST is aware of the key role of several wireless standards. For this reason, ST also entered in this market arena with two dedicated STM32 series: STM32WB and STM32WL. This first one is addressed to 2.4GHz radio application, and it support Bluetooth 5.2, Bluetooth Mesh as well as other protocols operating in that frequency band, such as the Zigbee 3.0. The second series, the STM32WL, is instead addressed to Sub-Ghz spectrum, ranging from 169MHz up to 915MHz. Moreover, ST has established a partnership with Semtech to develop custom solutions compatible with the *Long Range Alliance* (LoRa). Thanks to this partnership, the STM32WL series integrates a Semtech LoRa transceiver that supports the LoRaWAN™ standardized protocol. ST provides a set of dedicated Nucleo boards for the development with STM32WL and STM32WB series.

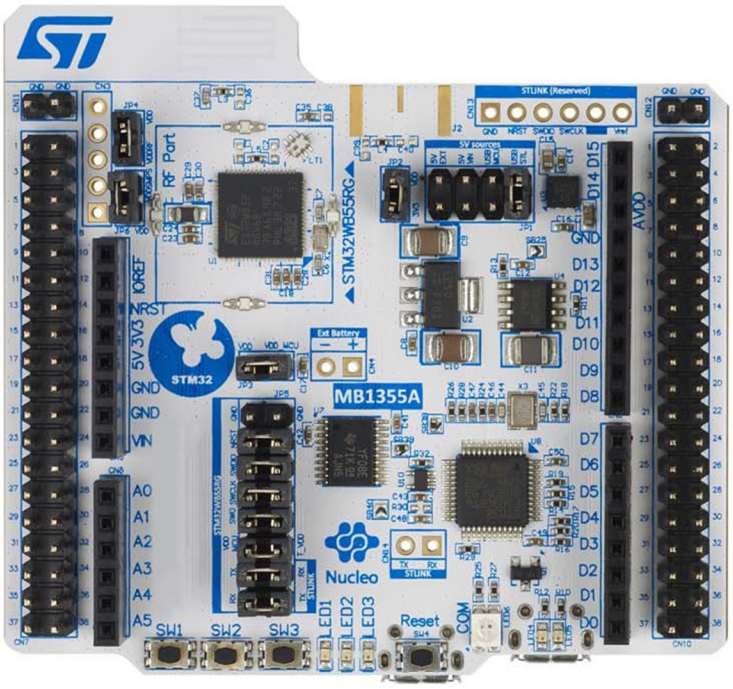


Figure 26.1: The Nucleo-WB55 development board

26.2 The W5500 Ethernet Controller

Apart from the more recent Nucleo-144 boards, none of the other Nucleo-64 and Nucleo-32 boards provide an STM32 MCU with the integrated Ethernet controller. This means that, if you want to develop IoT applications with your Nucleo-64, you need to use an external expansion board.

WIZnet is a Korean company that reached the popularity thanks to the Arduino board. In fact, its first ethernet controller, the W5100 IC, was the chip used to create the Arduino Ethernet Shield. Starting from the W5100 controller, WIZnet has iterated the development of other similar products. One of the best-in-class products is the W5500, which we will study in this chapter.

The W5500 is a monolithic Ethernet controller, with integrated LAN Phyther. Moreover, it is a complete network processor, with hardwired TCP/IP stack. The chip is designed to exchange data with a host microcontroller through a fast SPI interface (the interface is able to work up to 80MHz). These are the most important characteristic of this chip:

- Supports TCP, UDP, ICMP, IPv4, ARP, IGMP, PPPoE
- Supports 8 independent sockets simultaneously
- Supports Power down mode
- Supports Wake on LAN over UDP
- Supports High Speed Serial Peripheral Interface (SPI MODE 0, 3) up to 80MHz
- Internal 32Kbytes Memory for Tx/Rx Buffers

- 10BaseT/100BaseTX Ethernet PHY embedded
- Supports Auto Negotiation (Full and half duplex, 10 and 100-based)
- 3.3V operation with 5V I/O signal tolerance
- LED outputs (Full/Half duplex, Link, Speed, Active)
- 48 Pin LQFP Lead-Free Package (7x7mm, 0.5mm pitch)

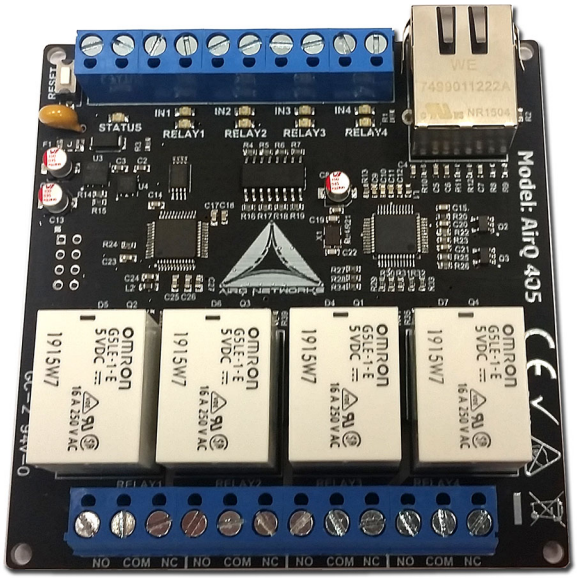


Figure 26.2: A custom device made with an STM32F0 MCU (on the left) and a W5500 IC (on the right)

The W5500 IC is easy to embed in a custom design. Just a crystal, few passives and a LAN transformer are required to make it work. The chip also integrates the charging pump needed to feed the LAN transformer. I have successfully used this chip in a couple of custom designs. Being the whole TCP/IP stack included inside the network processor, this chip can be used even in conjunction with low-cost STM32F0 microcontrollers. Moreover, for low-volume productions it is more convenient to use one of these chips with a low-cost STM32 MCU instead of a powerful one with the integrated Ethernet and the external dedicated LAN phyther. **Figure 26.2** shows a custom design made by this author where an STM32F030 MCU is used to drive the W5500 IC. Static webpages are stored inside an external SPI FLASH memory.

WIZnet developed an Arduino compatible shield (see **Figure 26.3**) that works out of the box even with Nucleo boards. This shield also integrates a MicroSD card reader, which is connected on the same SPI port of the W5500 IC. This allows to store webpages and other static contents (images, CSS, JavaScript files and so on) on an external SD card.

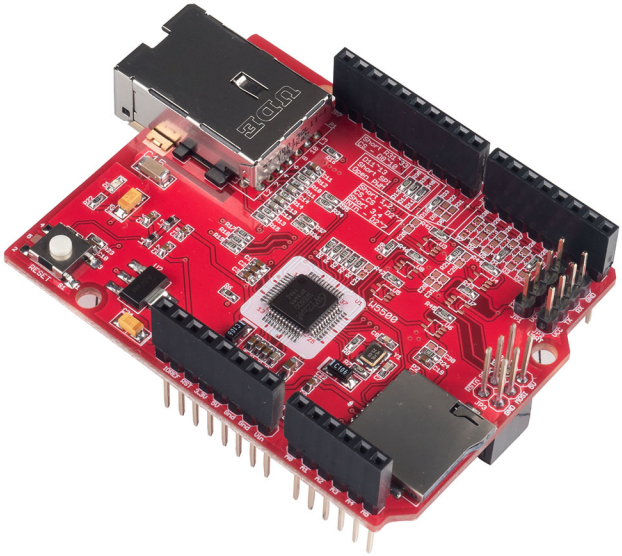


Figure 26.3: The W5500 Ethernet Shield by WIZnet

Network processors like the W5500 and similar work in a simple way. The chip offers up to eight *sockets*[6]. Every socket has a set of associated registers. By modifying the content of these registers, it is possible to drive the socket (open a connection, place it in listening mode, send/receive data and so on). To transfer data over a socket, the W5500 offers an internal 32KB buffer space, which can be freely subdivided between the eight sockets as we will see later. By writing into/reading from this buffer you can exchange data with the other endpoint. This means that, from the MCU point of view, driving these ICs is just a matter of bytes exchanged over the SPI interface. However, to handle all the internal states of a socket may be hard, especially for novices of these ICs. I have developed a library to drive the W5100, and I can confirm that this activity is time consuming. Moreover, unfortunately all W5X00 ICs (5100, 5200, 5300 and 5500) have some "nasty" and not well-documented bugs that it is hard to address.

[6]A socket in networking is an abstraction over the complex TCP/IP stack. It is just a handle that allows to send a stream of bytes from a machine to another, without dealing with the complex underlying protocol (unless you need to perform advanced operations).

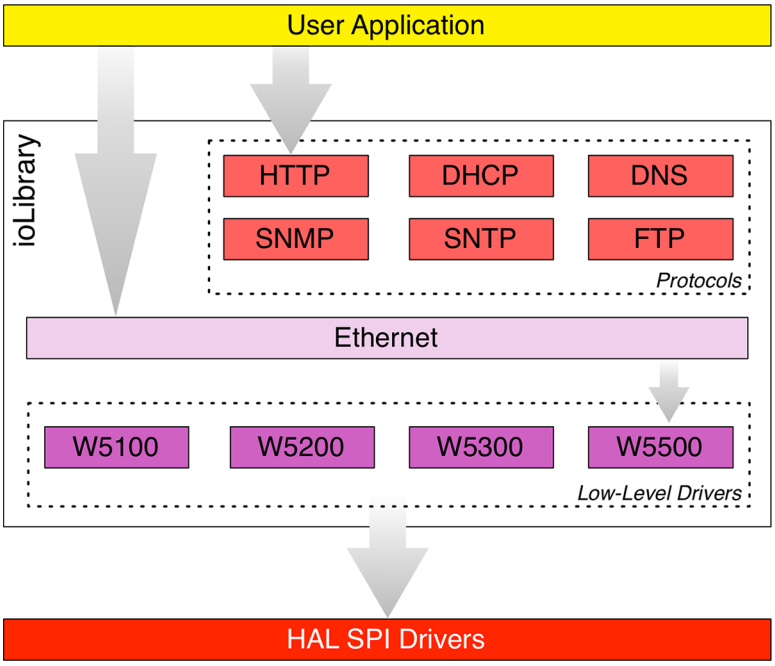


Figure 26.4: The architecture of the `ioLibrary_Driver` library

WIZnet released a dedicated library for this family of chips about seven years ago. It is named `ioLibrary_Driver` and it is available on GitHub[7]. The library architecture is shown in **Figure 26.4**. The library is essentially composed by two layers. One layer, called `Ethernet`, contains the primitives used to establish connections between peers. The file `Ethernet/socket.c` contains all routines related to socket management. The socket API is similar to the BSD socket API, even if it is not totally complaint with it. The same `Ethernet` layer contains low-level drivers for each WIZnet chip. For example, the file `Ethernet/w5500.c` contains all the necessary logic to drive the W5500 chip. Finally, the `Ethernet/wizchip_conf.h` file contains configuration macros used to setup the library (more about this later).

The `Internet` layer is built upon the `Ethernet` one and it is a collection of several Internet protocols and services:

- DHCP client
- DNS client
- FTP client and server
- SNMP agent/trap
- SNTP client
- TFTP client
- HTTP server

The user application can use one or more of the above protocols or access directly to the `Ethernet` layer to build a custom application.

[7] https://github.com/Wiznet/ioLibrary_Driver

26.2.1 How to Use the W5500 Shield and the `ioLibrary_Driver` Module

As said before, the W5500 mates seamlessly with all nine Nucleo boards used in this text. The **Figure 26.5**[8] shows the shield pinout. The SPI interface is routed to D13, D12 and D11 pins, and they correspond to the same pins of the SPI1 peripheral (apart for the Nucleo-F302R8 board where those pins correspond to SPI2 peripheral). The *Slave Select* (SS) pin for the W5500 pin corresponds to the Arduino D10 pin, while the SS pin for the SD card corresponds to the Arduino D4 pin.

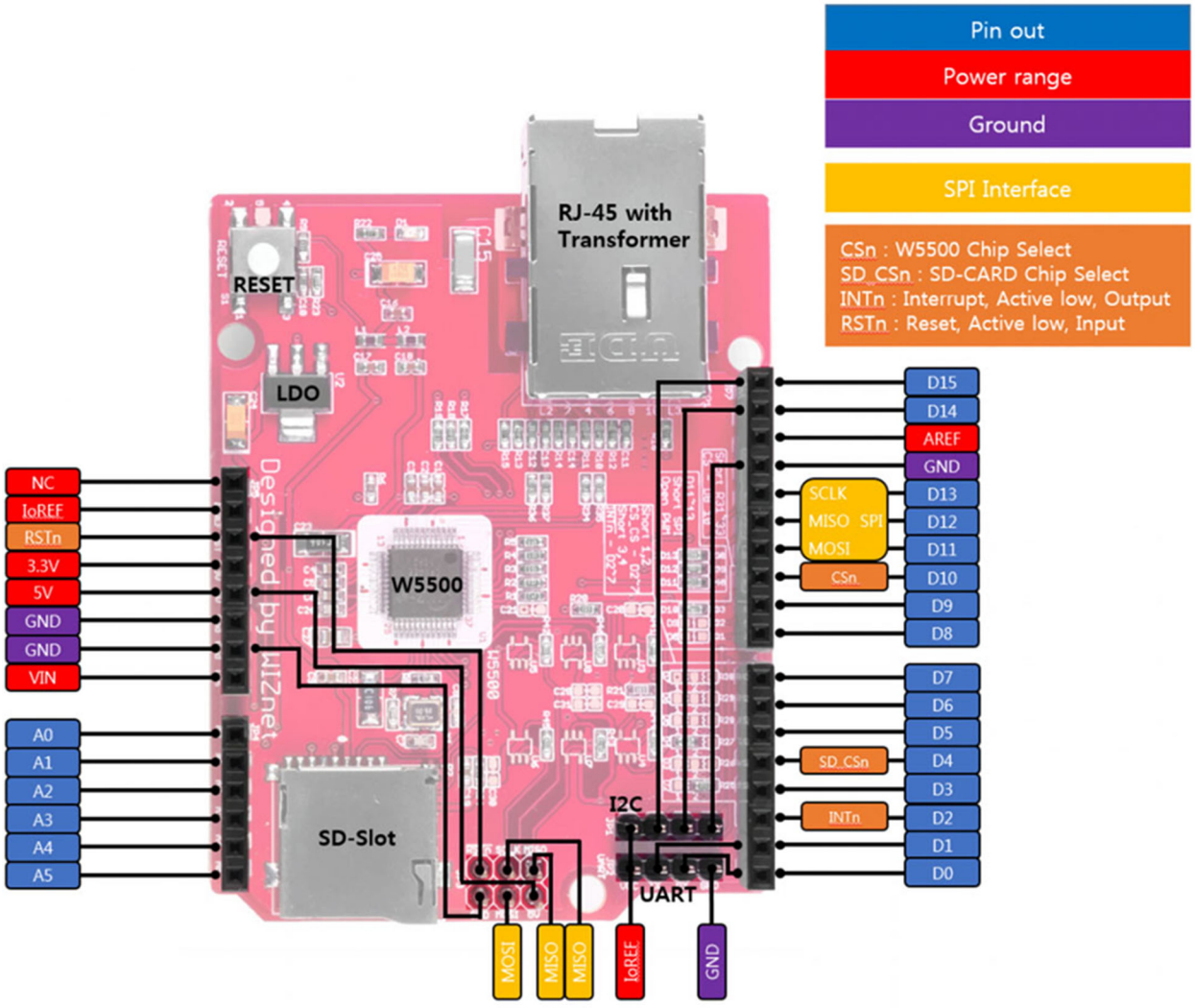


Figure 26.5: The pinout of the W5500 shield

[8]The figure is taken from the WIZnet website(https://bit.ly/2dxjblH).



Looking to **Figure 26.5**, you can see that the D2 pin plays an important role. The W5500 IC is designed to optionally assert low the INTn pin when several events related to the network interface (e.g., IP collision, etc.) or to the single socket (e.g., connection established, data received, etc.) happen. This feature allows to configure the corresponding MCU pin as `GPIO_MODE_IT_FALLING`, so that the corresponding IRQ fires when the IC asserts low the INTn pin. This allows to write down asynchronous applications, especially if you are using the `ioLibrary_Driver` in conjunction with an RTOS (for example, the ISR may use a semaphore to wake-up a sleeping thread that starts performing operation with a socket).

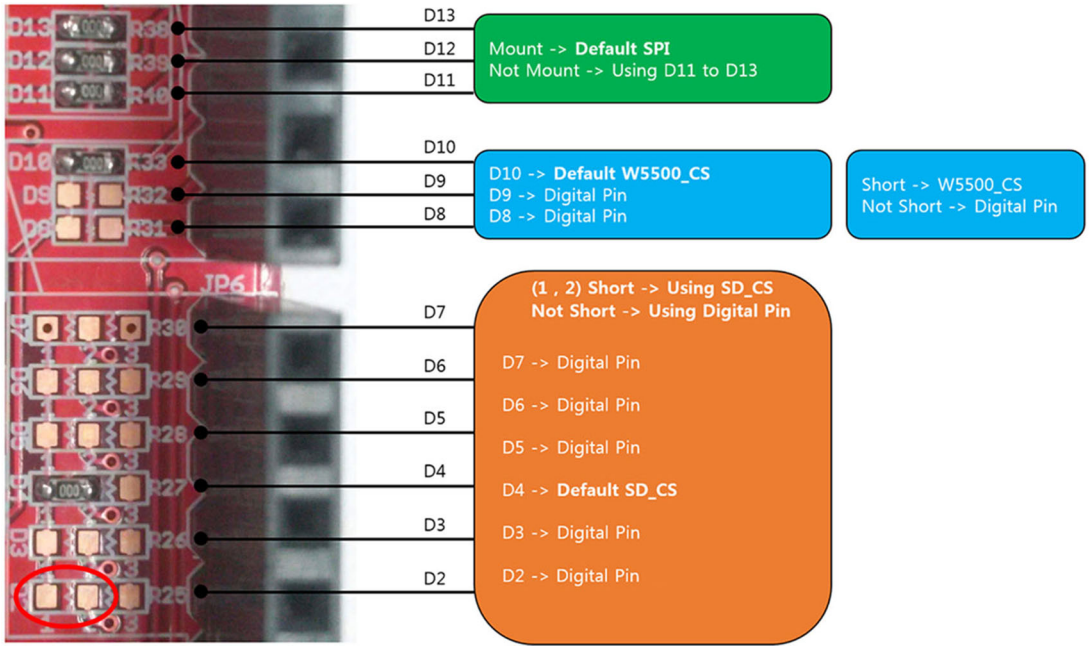


Figure 26.6: To enable INTn pin, it is required to short 1-2 pads

Take note that in the W5500 shield the D2 pin is not connected to the W5500 INTn pin. To enable it, you need to solder a 0603 0Ω resistor between pads 1-2, as shown in **Figure 26.6** (a joint with a solder drop is also sufficient).

Once the hardware connection is established, we can focus our attention to the software part. To import the `ioLibrary_Driver` module in an existing Eclipse project is easy. You first need to drag the whole library in the Eclipse project root dir. Then you need to add the following path to the list of Include paths inside the Project settings:

- `"../ioLibrary_Driver/Ethernet"`
- `"../ioLibrary_Driver/Internet"`

Finally, you need to specify the exact W5XXX chip type, by setting the macro `_WIZCHIP_` inside the **ioLibrary_Driver/Ethernet/wizchip_conf.h** file.

26.2.1.1 Configuring the SPI Interface

The `ioLibrary_Driver` module is designed to be abstracted from specific MCU and routines to manipulate the SPI interface. It can be used with an STM32, an AVR, a Microchip MCU and so on. So, we need a way to interface it with the `HAL_SPI` module needed to program the SPI peripheral.

As users of this library, we need to provide 6 callback routines that implement all the necessary logic to drive the SPI. These routines are the following ones:

- `void cs_sel()`: this callback is invoked by the library every time it needs to select (assert LOW) the pin latched to the W5500 SS pin.
- `void cs_desel()`: this callback is invoked by the library every time it needs to deselect (assert HIGH) the pin latched to the W5500 SS pin.
- `uint8_t spi_rb()`: this routine is invoked when one byte needs to be read from the SPI interface.
- `void spi_wb(uint8_t b)`: this routine is invoked when one byte need to be sent over the SPI interface.
- `void spi_rb_burst(uint8_t *buf, uint16_t len)`: this optional callback is invoked when more than three bytes need to be read over the SPI. This allows us to implement the callback handling the SPI in DMA mode, which speeds up the transfer speed (this mode is also called *burst* mode).
- `void spi_wb_burst(uint8_t *buf, uint16_t len)`: this optional callback is invoked when more than three bytes need to be sent over the SPI. This allows us to implement the callback handling the SPI in DMA mode.

Assuming that the SPI interface is configured accordingly, we can simply implement those callbacks in the following way:

```
void cs_sel() {
  HAL_GPIO_WritePin(W5500_CS_GPIO_Port, W5500_CS_Pin, GPIO_PIN_RESET); //CS LOW
}

void cs_desel() {
  HAL_GPIO_WritePin(W5500_CS_GPIO_Port, W5500_CS_Pin, GPIO_PIN_SET); //CS HIGH
}

uint8_t spi_rb(void) {
  uint8_t rbuf;
  HAL_SPI_Receive(&hspi1, &rbuf, 1, HAL_MAX_DELAY);
  return rbuf;
}

void spi_wb(uint8_t b) {
  HAL_SPI_Transmit(&hspi1, &b, 1, HAL_MAX_DELAY);
}
```

```
void spi_rb_burst(uint8_t *buf, uint16_t len) {
  HAL_SPI_Receive_DMA(&hspi1, buf, len);
  while(HAL_SPI_GetState(&hspi1) == HAL_SPI_STATE_BUSY_RX);
}

void spi_wb_burst(uint8_t *buf, uint16_t len) {
  HAL_SPI_Transmit_DMA(&hspi1, buf, len);
  while(HAL_SPI_GetState(&hspi1) == HAL_SPI_STATE_BUSY_TX);
}
```

Once we have defined the hardware related functions, we have to "pass" them to the `ioLibrary`. This job can be performed by using the following routines:

```
  ...
  reg_wizchip_cs_cbfunc(cs_sel, cs_desel);
  reg_wizchip_spi_cbfunc(spi_rb, spi_wb);
  reg_wizchip_spiburst_cbfunc(spi_rb_burst, spi_wb_burst);
```

Finished. With those few lines of code, we have successfully integrated the `ioLibrary_Driver` library with the CubeHAL.

26.2.1.2 Configuring the Socket Buffers and the Network Interface

Now we are finally ready to start using the W5500 IC. The first step consists in initializing the W5500 chip with a well-defined procedure. The first function to call, once the callbacks have been configured, is the following one:

```
int8_t wizchip_init(uint8_t* txsize, uint8_t* rxsize);
```

This routine does two things: it resets the W5500 IC (mandatory procedure) and configures the size of TX and RX buffers for each individual socket. All W5X00 ICs have two internal common memory areas dedicated to TX and RX buffers. In W5500 chip these areas are 16KB wide each. These two areas must be in turn subdivided among the sockets we want to use. The default configuration gives 2KB of these areas to each of the eight sockets. But if, for example, our application needs just four sockets, then we can subdivide RX and TX buffer by four. Or, if one socket needs more rooms to exchange data with the other peer, then we can give more space to just one socket and reduce the space to other ones.

We can specify the allocation of TX and RX buffer by passing to `wizchip_init()` routine two arrays, each one with eight values. The only requirement is that the sum of these values must not exceed 16. For example, if we want to use just two sockets in our application, we can define the configuration array in the following way:

```
uint8_t bufSize[] = {12, 4, 0, 0, 0, 0, 0, 0};
wizchip_init(bufSize, bufSize);
```

The above code simply allocates 12KB of TX and RX buffers to the first socket, and the remaining 4KB to the second socket. Clearly, we are free to arrange TX and RX buffer as long as we respect the total size of 16KB.

Once the chip is initialized, we can configure the network interface, by using the function:

```
void wizchip_setnetinfo(wiz_NetInfo* pnetinfo);
```

where the `wiz_NetInfo struct`, used to pass to the library the network configuration parameters, is defined in the following way:

```
typedef struct wiz_NetInfo_t {
   uint8_t mac[6];  /* Source Mac Address */
   uint8_t ip[4];   /* Source IP Address */
   uint8_t sn[4];   /* Subnet Mask */
   uint8_t gw[4];   /* Gateway IP Address (optional) */
   uint8_t dns[4];  /* DNS server IP Address (optional) */
   dhcp_mode dhcp;  /* 1 - Static, 2 - DHCP (optional) */
} wiz_NetInfo;
```

I will not detail those fields here because they are self-explaining. For example, to configure the W5500 so that it can connect to the 192.168.1.0/24 subnet, we can proceed in the following way:

```
  wiz_NetInfo netInfo = {.mac  = {0x00, 0x08, 0xdc, 0xab, 0xcd, 0xef}, // Mac address
                         .ip   = {192, 168, 1, 192},                   // IP address
                         .sn   = {255, 255, 255, 0},                   // Subnet mask
                         .gw   = {192, 168, 1, 1}};                    // Gateway address
  wizchip_setnetinfo(&netInfo);
```



Please, take note that here we are using an arbitrary MAC address in this example. This procedure is permitted for a private and test environment, but it is completely forbidden if you are planning to sell your W5500 based product. In this case, you have to buy a valid pool of MAC address from IEEE (pools starts from batches of 4096 addresses for about 800 USD[9]). Alternatively, Microchip sells pre-programmed ICs[10] with a valid IEEE EUI-48 and EUI-64 MAC addresses (they work like I²C EEPROM). For small volume productions they are a good alternative to buying custom MAC addresses.

[9] https://bit.ly/34HObty
[10] https://bit.ly/2dKLLhA

26.2.2 Socket APIs

The `ioLibrary_Driver` module provides an API to manipulate sockets that resembles the BSD socket API. Even if it is not compatible with it, if you have already worked with BSD API then it will be very easy for you to start working with it.

To initialize a new socket, we can use the function:

```
int8_t socket(uint8_t sn, uint8_t protocol, uint16_t port, uint8_t flag);
```

where:

- `sn`: corresponds to the socket number, and it can range from 0 up to 7 if you are using a W5500 IC that provides 8 sockets.
- `protocol`: this parameter defines the socket protocol type. The W5500 is able to handle three protocol types: TCP, UDP and RAW socket[11]. This parameter can so assume one of the values `Sn_MR_TCP`, `Sn_MR_UDP` and `Sn_MR_MACRAW`.
- `port`: specify the port number associated to the socket.
- `flag`: it is a combination (logical or) of additional configuration parameters listed in **Table 26.1**.

On success, the `socket()` function returns the `sn` value, otherwise it may return `SOCKERR_SOCKNUM` to indicate a wrong socket number, `SOCKERR_SOCKMODE` to indicate a wrong protocol or `SOCKERR_SOCKFLAG` to indicate an invalid `flag` parameter.

Table 26.1: Socket flag values

Mode	Description
`SF_IO_NONBLOCK`	Configures the socket in non-blocking mode
`SF_ETHER_OWN`	W5500 can only receive broadcasting packet or packet sent to itself. This parameter applies only to RAW socket
`SF_IGMP_VER2`	Enables IGMP version 2 when socket protocol is UDP and the `SF_MULTI_ENABLE` mode is also specified
`SF_MULTI_ENABLE`	Enables multicast mode when socket protocol is UDP
`SF_TCP_NODELAY`	Configures a TCP socket so that the ACK packet is sent as soon as a data packet is received from the remote peer
`SF_BROAD_BLOCK`	Prevents the socket from receiving broadcast packets when the socket protocol is UDP or RAW
`SF_MULTI_BLOCK`	Prevents the socket from receiving broadcast packets when the socket protocol is RAW
`SF_IPv6_BLOCK`	Prevents the socket from receiving IPv6 packets when the socket protocol is RAW
`SF_UNI_BLOCK`	Prevents the socket from receiving unicast packets when the socket protocol is UDP

To close a socket, and to deconfigure it, we use the function:

[11] A RAW socket is a socket that allows to exchange packets using directly the IP protocol and without any protocol-specific transport layer (that is, TCP or UDP).

```
int8_t close(int8_t sn);
```



Read Carefully

Please, take note that if you are using I/O functions from the C standard library, then this function will crash with the signature of the C stdlib `close()` function, which is designed to accept and return an `int` type. A lot of compiler errors will be generated. Unfortunately, WIZnet guys have chosen an unhappy name and signature. You can bypass this problem by modifying the library declaring the close() function in this other way:

```
int close(int sn);
```

A W5500 socket has a well-defined status, and it could be useful to retrieve it to better understate the connection state. The macro:

```
getSn_SR(sn);
```

automatically retrieves the socket status from its registers. The possible status values for a W5500 IC are listed in **Table 26.2**.

Table 26.2: Socket status values

Status	Description
SOCK_CLOSED	Closed
SOCK_INIT	In initialization state
SOCK_LISTEN	Listen
SOCK_ESTABLISHED	Connection established
SOCK_CLOSE_WAIT	Closing state
SOCK_UDP	UDP Socket
SOCK_SYNSENT	Connect-request packet (SYN) sent to remote peer
SOCK_SYNRECV	Connect-request packet (SYN) received from remote peer
SOCK_FIN_WAIT	Started socket closing procedure
SOCK_CLOSING	Closing the socker
SOCK_TIME_WAIT	Waiting for socket closing
SOCK_LAST_ACK	The socket is still open but the remote peer closed the connection

26.2.2.1 Handling Sockets in TCP Mode

Once the socket protocol and mode are configured, we can start a connection to a remote peer (client application) or place the socket in listen mode to accept connections from a remote peer (server application).

To start a connection with a remote peer, we use the function:

```
int8_t connect(uint8_t sn, uint8_t * addr, uint16_t port);
```

- `sn`: corresponds to the socket configured with the `socket()` function.
- `addr`: it is an array of four bytes corresponding to the IPv4 address of the remote peer.
- `port`: it is the port number of the remote peer.

On success, the `connect()` function returns `SOCK_OK` value. Otherwise, a list of possible error values exists. Take a look at the `socket.h` file.

When the connection with the remote peer is established, we can send a sequence of bytes by using the function:

```
int32_t send(uint8_t sn, uint8_t * buf, uint16_t len);
```

where `buf` is the array of bytes having the `len` length.

Instead, to receive an array of bytes from the remote peer, we use the function:

```
int32_t recv(uint8_t sn, uint8_t * buf, uint16_t len);
```

To disconnect from the remote peer, we can use the function:

```
int8_t disconnect(uint8_t sn);
```

If, instead, we are going to create a server application, once the socket has been configured by using the `socket()` function, we can place it in listen mode by using the function:

```
int8_t listen(uint8_t sn);
```

Once a connection is established, we can receive and send data using `recv()` and `send()` functions.

26.2.2.2 Handling Sockets in UDP Mode

UDP is a connection-less protocol, so we do not need to explicitly create connections to start exchanging bytes with the remote peer.

To send an array of bytes to a remote peer when the socket is in UDP mode, we use the function:

```
int32_t sendto(uint8_t sn, uint8_t * buf, uint16_t len, uint8_t * addr, uint16_t port);
```

while to receive bytes from the remote peer we use the function:

```
int32_t recvfrom(uint8_t sn, uint8_t * buf, uint16_t len, uint8_t * addr, uint16_t *port);
```

26.2.3 I/O Retargeting to a TCP/IP Socket

In Chapter 5 we have seen how to retarget C terminal I/O functions, like `printf()` and `scanf()`, to an UART interface. During the development of IoT applications, it comes really in handy to retarget I/O to a network socket, so that we can debug our device through a network connection. This is useful especially if the device is not under our direct control.

This operation is easy using a W5500 IC and the relative socket library. The `RetargetInit()` function can be rewritten in the following way:

Filename: `Core/Src/retarget-tcp.c`

```
13  #ifdef RETARGET_TCP
14
15  #define STDIN_FILENO  0
16  #define STDOUT_FILENO 1
17  #define STDERR_FILENO 2
18
19  #ifndef RETARGET_PORT
20  #define RETARGET_PORT 5000
21  #endif
22
23  int8_t gSock = -1;
24
25  uint8_t RetargetInit(int8_t sn) {
26    gSock = sn;
27
28    /* Disable I/O buffering for STDOUT stream, so that
29     * chars are sent out as soon as they are printed. */
30    setvbuf(stdout, NULL, _IONBF, 0);
31
32    /* Open 'sn' socket in TCP mode with a port number equal
33     * to the value of RETARGET_PORT macro */
34    if(socket(sn, Sn_MR_TCP, RETARGET_PORT, 0) == sn) {
35      if(listen(sn) == SOCK_OK)
36        return 1;
37    }
38    return 0;
39  }
```

The code is self-explaining. The `RetargetInit()` function accepts a socket number, which for a W5500 IC ranges from 0 to 7. The function so configures the socket and places it in listening mode. The `_write()` function can so be rearranged in the following way:

Filename: `Core/Src/retarget-tcp.c`

```
49  int _write(int fd, char* ptr, int len) {
50    int sentlen = 0;
51    int buflen = len;
52
53    if(getSn_SR(gSock) == SOCK_ESTABLISHED) {
54      if (fd == STDOUT_FILENO || fd == STDERR_FILENO) {
55        while(1) {
56          sentlen = send(gSock, (void*) ptr, buflen);
57          if (sentlen == buflen)
58            return len;
59          else if (sentlen > 0 && sentlen < buflen) {
60            buflen -= sentlen;
61            ptr += (len - buflen);
62          }
63          else if (sentlen < 0)
64            return EIO;
65        }
66      }
67    } else if(getSn_SR(gSock) != SOCK_LISTEN) {
68      /* Remote peer close the connection? */
69      close(gSock);
70      RetargetInit(gSock);
71    }
72
73    errno = EBADF;
74    return -1;
75  }
```

The function starts checking if the socket state is equal to `SOCK_ESTABLISHED`: this means that the remote peer has established a connection to our device. Instead, if the socket is not in listening mode (line 67), then it could be that the remote peer closed the connection: we so need to configure the socket again in listening mode by calling the `RetargetInit()` function. If the remote peer has established the connection, then we can start sending the `ptr` buffer over the TCP/IP connection.

The `_read()` function is almost identical to the `_write()` one. Refer to the book examples for the complete source code. To use this module, we simply need to define the macro `RETARGET_TCP` at project level, and to eventually remove the macro `OS_USE_SEMIHOSTING`.

To start a connection to the device, Linux and MacOS users can use the `telnet` command, while Windows users can use a terminal emulator program like `putty`.

26.2.4 Building up an HTTP Server

The `Internet/httpServer` module provides a complete HTTP server implementation built upon the `Ethernet` layer. This module allows you to setup an HTTP server in a few steps, especially if you need just to serve static content (that is, simple webpages that do not need to process data dynamically).

The function

```
void httpServer_init(uint8_t * tx_buf, uint8_t * rx_buf, uint8_t cnt, uint8_t * socklist);
```

is used to configure the HTTP module. It accepts two pointers, `tx_buf` and `rx_buf`, to two memory buffers used to store data exchanged by the HTTP server. These arrays need to have enough space to store HTTP headers. In fact, when accessing to a webpage, the browser needs to exchange with the webserver several "underlying" messages defined by the HTTP protocol. These messages consume several hundreds of bytes, and for this reason a minimum viable size for both `tx_buf` and `rx_buf` buffers is equal to 1024 bytes[12]. The `cnt` parameter says to the HTTP module how many W5500 sockets it can use, and the `socklist` parameter is used to pass the exact list of available sockets.

For example, the following code fragment initialize the HTTP server by passing two buffers, each one sized 1024 bytes, and an array containing the list of sockets to use to process HTTP requests:

```
#define DATA_BUF_SIZE 1024
#define MAX_HTTPSOCK 5

uint8_t RX_BUF[DATA_BUF_SIZE], TX_BUF[DATA_BUF_SIZE];
uint8_t socknumlist[] = {0, 1, 2, 3, 4};
...
httpServer_init(TX_BUF, RX_BUF, MAX_HTTPSOCK, socknumlist);
...
```

Once the HTTP server is configured from the network point of view, we need to make it aware of the content to serve (HTML pages, images, and so on). There are two ways to perform this: one is suitable for small and limited applications, and the other one for more complex and structured web applications.

By using the function:

```
void reg_httpServer_webContent(uint8_t * content_name, uint8_t * content);
```

we can associate to a given resource (for example the file `index.html`) an array of bytes to send over the socket to the browser. The `content_name` parameter corresponds to the resource name and the `content` one to the array containing the bytes forming the resource.

[12] It is possible to reduce the size of the two buffers, since the HTTP library is able to split the whole HTTP stream in smaller chunks, but this will increase the transfer time.



Why the HTTP server needs more than a free socket to accomplish its activities? This is a fundamental concept to keep in mind while developing web-based embedded applications, and so we will spend few words about it.

Modern web-based applications are really complex. Usually, a website is made of several resources:

- HTML pages that contains the actual content of the web application;
- images that adorn the HTML content and, in some cases, constitute the part of the webpage content.
- CSS and Javascript files that configure the page rendering and its functionalities.

When a web browser accesses to a website, it starts loading the main HTML page (which corresponds to the `index.html` file, if not specified). This page is parsed almost immediately (some browser can start parsing pages as long as they receive the very first bytes), and if it contains references to other web resources, the browser starts loading them almost in parallel. This simultaneous access to website resources implies that several sockets are opened by the browser towards the HTTP server (the HTTP protocol is stateless, and it defines that for each web resource a separated request must be performed to the server). For a true web-server, like Apache or NGIX, designed to run on a powerful machine, this does not constitute a problem. These server applications are designed to process even thousands concurrent connections. Moreover, the powerful underlying hardware allows to serve a content even in few milliseconds, depending on the connection speed. Sockets are so opened and closed in less than a second.

For a web-server running on a true-embedded platform the access to simultaneous resources is a thing that must be characterized carefully. Every socket eats several hardware resources and for a WIZnet device the maximum number of sockets is limited. This implies that we are not free to arrange the application as we want, and some modern framework (like Bootstrap, Angular JS, and so on) often need to be rearranged when used on an embedded device (sometimes we have to avoid them at all).

For example, to send a simple HTML page we can write the following code:

```
const char webpage[] = "<html>
<head>
<title>Simple Web Page</title>
</head>
<body>
        <h1>Hello World!</h1>
</body>";

reg_httpServer_webContent((uint8_t*)"index.html", webpage);
...
```

This approach has several pitfalls. First of all, the web page is embedded inside the firmware code. This means that the HTML content adds to the firmware itself, increasing the whole binary image.

Secondly, every time we change something to the web content, we need to recompile the whole binary image. For large and structured web applications this approach is impractical.

The second approach consists in instructing the HTTP module so that it finds and takes static content from a memory device. For example, we can modify its code so that it loads the web resources from a flash memory. This is what we will make in the next example, where we will use the FatFs library to retrieve web content stored on the external SD card.

When the HTTP server is properly configured, we can start servicing requests coming from remote peers. By default, this operation is performed in polling mode by calling the following function:

```
void httpServer_run(uint8_t seqnum);
```

This function accepts the index corresponding to the socket id stored inside the `socklist` parameter passed to the `httpServer_init()` function. For every registered socket, this function checks the status of the corresponding socket and executes the HTTP state machine according to the given socket status. For example, if the passed socket is opened and in listening mode, the `httpServer_init()` function checks if a remote peer has established a connection. The whole HTTP protocol and state machine is handled by the HTTP module, and there is no need to know the implementation details unless we need to do something advanced with it.

Finally, the HTTP module uses some internal delays which are based on a common timebase unit (tick). The function:

```
httpServer_time_handler();
```

must be called from a timer configured to expire every 1ms (the *Systick* ISR is a right place to call that function).

26.2.4.1 A Web-Based Oscilloscope



Due to limited hardware resources of the most of STM32 MCUs equipping the nine Nucleo boards, this example has been tested just on STM32F401RE, STM32F411RE and STM32F446RE MCUs.

We are now going to review a more complete example that shows how to use the W5500 IC and the `ioLibrary_Driver`[13] module to build complex and structured applications. In this example we will use several STM32 peripherals to build a sort of web-based oscilloscope, which is shown in **Figure 26.7**. By connecting a signal source to one of the inputs of the ADC peripheral, we can see the corresponding waveform by simply accessing to a web console using a common browser. A video that shows how the oscilloscope works can be seen here[14].

[13]The `ioLibrary_Driver` shipped with this chapter's example is not the official library provided by WIZnet. This author made several modifications to the HTTP module to improve its reliability, performances and flexibility. For example, this modified version is able to serve content stored on an SD card using the FatFs module or on the developer's PC using ARM semihosting.

[14]https://youtu.be/fjtLQJDJ_04

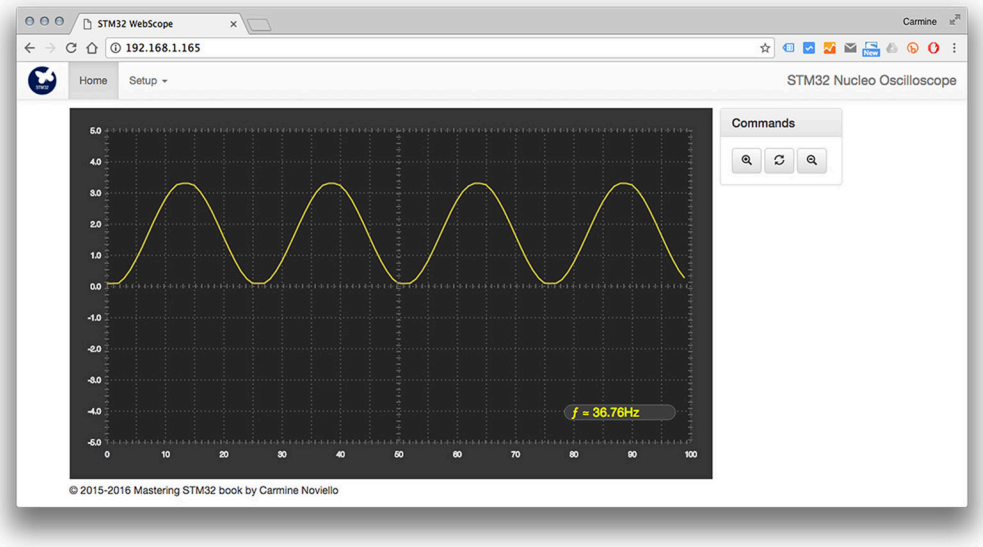


Figure 26.7: The interface of the web-based oscilloscope

The example uses several popular and modern web-frameworks to structure the user interface: Bootstrap[15], jQuery[16] and D3js[17]. The description of these frameworks is outside the scope of the book, and it will be assumed that the reader is familiar with the most common and modern web-development techniques.

The application is organized in two main parts: an "underlying" part responsible of the analog-to-digital conversion from the selected ADC input (IN0 by default) and a part that serves HTTP requests using the `ioLibrary_Driver` module. All web resources are assumed to be placed on an SD card, which is accessed using the FatFs module and the SPI-compatible driver developed by this author. However, to simplify the development process, the application is also able to serve content from the developer's PC using ARM semihosting calls and regular C stdlib functions.

The following code fragment is related to the `main()` function. To acquire a signal that varies over the time (for example, a 50Hz sinewave), we need to perform ADC conversions at regular intervals. We so use the TIM2 timer[18] to drive the ADC1 peripheral, which is configured to work in DMA circular mode: in this way the timer will trigger conversion continuously. The ADC is so started in DMA mode (line 145) and converted values are stored inside the `_adcConv` array. The `adcSem` semaphore instantiated at line 140 will be used to rule the access to the `_adcConv` array, which holds the ADC converted values. Its role will be better explained later.

[15] http://getbootstrap.com/
[16] https://jquery.com/
[17] https://d3js.org/
[18] The source code presented here is related to the Nucleo-F401RE board.

Filename: `src/ch25/main-ex2.c`

```
135 int main(void) {
136   HAL_Init();
137   Nucleo_BSP_Init();
138 
139   osSemaphoreDef(adcSem);
140   adcSemID = osSemaphoreCreate(osSemaphore(adcSem), 1);
141 
142   MX_ADC1_Init();
143   MX_TIM2_Init();
144   HAL_TIM_Base_Start(&htim2);
145   HAL_ADC_Start_DMA(&hadc1, (uint32_t*)_adcConv, 200);
146 
147   MX_SPI1_Init();
148 
149 #if defined(_USE_SDCARD_) && !defined(OS_USE_SEMIHOSTING)
150   SD_SPI_Configure(SD_CS_GPIO_Port, SD_CS_Pin, &hspi1);
151   MX_FATFS_Init();
152 
153   if(f_mount(&diskHandle, "0:", 1) != FR_OK) {
154 #ifdef DEBUG
155     asm("BKPT #0");
156 #else
157     while(1) {
158       HAL_Delay(500);
159       HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
160     }
161 #endif //#ifdef DEBUG
162   }
163 
164 #ifdef OS_USE_TRACE_ITM
165   /* Prints the SD content over the ITM port */
166   TCHAR buff[256];
167   strcpy(buff, (char*)L"/");
168   scan_files(buff);
169 #endif //#ifdef OS_USE_TRACE_ITM
170 
171 #endif //#if defined(_USE_SDCARD_) && !defined(OS_USE_SEMIHOSTING)
172 
173   osThreadDef(w5500, SetupW5500Thread, osPriorityNormal, 0, 512);
174   osThreadCreate(osThread(w5500), NULL);
175 
176   osKernelStart();
177   /* Never coming here, but just in case... */
178   while(1);
179 }
```

If the global macro `_USE_SDCARD_` is set and we are not using the ARM semihosting (so the global macro `OS_USE_SEMIHOSTING` is not set), then this means that web resources (HTML files, images and CSS/JS script files) are stored on a MicroSD card inserted in the W5500 shield card holder. The FatFs library is so initialized (lines [150:151]) and the first partition is mounted (line 153). Moreover, if we are using a debugger able to read ITM stimuluses, we print on the SWV console the content of the SD card by calling the `scan_files()` routine (which has been shown in Chapter 25).

Finally the `SetupW5500Thread()` thread is started, which is responsible to configure the W5500 IC and to handle the incoming HTTP requests. Its code is simple and it is shown next.

Filename: `src/ch25/main-ex2.c`

```
181 void SetupW5500Thread(void const *argument) {
182   UNUSED(argument);
183
184   /* Configure the W5500 module */
185   IO_LIBRARY_Init();
186
187   /* Configure the HTTP server */
188   httpServer_init(TX_BUF, RX_BUF, MAX_HTTPSOCK, socknumlist);
189   reg_httpServer_cbfunc(NVIC_SystemReset, NULL);
190
191   /* Start processing sockets */
192   while(1) {
193     for(uint8_t i = 0; i < MAX_HTTPSOCK; i++)
194       httpServer_run(i);
195     /* We just delay for 1ms so that other threads with the same
196      * or lower priority can be executed */
197     osDelay(1);
198   }
199 }
```

The thread starts configuring both the W5500 IC and the `ioLibrary_Driver` library by calling the function `IO_LIBRARY_Init()` (shown below). Next, the HTTP server is configured (line 188) and an infinite loop calls the `httpServer_run()` function for every socket allocated to the HTTP server.

Filename: `src/ch25/main-ex2.c`

```
79 void IO_LIBRARY_Init(void) {
80   uint8_t runApplication = 0, dhcpRetry = 0, phyLink = 0, bufSize[] = {2, 2, 2, 2, 2};
81   wiz_NetInfo netInfo;
82
83   reg_wizchip_cs_cbfunc(cs_sel, cs_desel);
84   reg_wizchip_spi_cbfunc(spi_rb, spi_wb);
85   reg_wizchip_spiburst_cbfunc(spi_rb_burst, spi_wb_burst);
86   reg_wizchip_cris_cbfunc(vPortEnterCritical, vPortExitCritical);
87
88   wizchip_init(bufSize, bufSize);
```

```
89
90    ReadNetCfgFromFile(&netInfo);
91
92    /* Wait until the ETH cable is plugged in */
93    do {
94      ctlwizchip(CW_GET_PHYLINK, (void*) &phyLink);
95      osDelay(10);
96    } while(phyLink == PHY_LINK_OFF);
97
98    if(netInfo.dhcp == NETINFO_DHCP) { /* DHCP Mode */
99      DHCP_init(DHCP_SOCK, RX_BUF);
100
101     while(!runApplication) {
102       switch(DHCP_run()) {
103       case DHCP_IP_LEASED:
104       case DHCP_IP_ASSIGN:
105       case DHCP_IP_CHANGED:
106         getIPfromDHCP(netInfo.ip);
107         getGWfromDHCP(netInfo.gw);
108         getSNfromDHCP(netInfo.sn);
109         getDNSfromDHCP(netInfo.dns);
110         runApplication = 1;
111         break;
112       case DHCP_FAILED:
113         dhcpRetry++;
114         if(dhcpRetry > MAX_DHCP_RETRY)
115         {
116           netInfo.dhcp = NETINFO_STATIC;
117           DHCP_stop();      // if restart, recall DHCP_init()
118 #ifdef _MAIN_DEBUG_
119           printf(">> DHCP %d Failed\r\n", my_dhcp_retry);
120           Net_Conf();
121           Display_Net_Conf();   // print out static netinfo to serial
122 #endif
123           dhcpRetry = 0;
124           asm("BKPT #0");
125         }
126         break;
127       default:
128         break;
129       }
130     }
131   }
132   wizchip_setnetinfo(&netInfo);
133 }
```

The `IO_LIBRARY_Init()` function is responsible of the proper configuration of the W5500 IC. It starts

by configuring the functions used to exchange data over the SPI bus (lines [83:88], as seen in the first example of this chapter). Next, at line 90, the function `ReadNetCfgFromFile()` is used to retrieve network configuration from a file stored inside the SD card. This file is named `net.cfg` and it must have the following structure:

```
1 NODHCP
2 0:11:22:33:44:55
3 192.168.1.165
4 255.255.255.0
5 192.168.1.1
6 8.8.8.8
```

The first line can assume the values (`NODHCP` and `DHCP`) and it indicates if the network IP is configured statically or dynamically. The second line corresponds to the MAC address, while the next four lines correspond to the device IP, the subnet mask, the network gateway, and the primary DNS. By reading the content of this file the network interface is automatically configured. The user can modify network parameter through a dedicated web page, as shown in **Figure 26.8**.

Once the network settings are retrieved from the configuration file, the `IO_LIBRARY_Init()` function enters in an infinite loop until the LAN cable is plugged inside the RJ45 port (lines [93:96]). When this happens, the function starts a DHCP discovery procedure if the network interface is configured in DHCP mode. Finally, at line 132 the network interface is configured using settings stored inside the `net.cfg` file or those ones retrieved by a DHCP server on the same network.

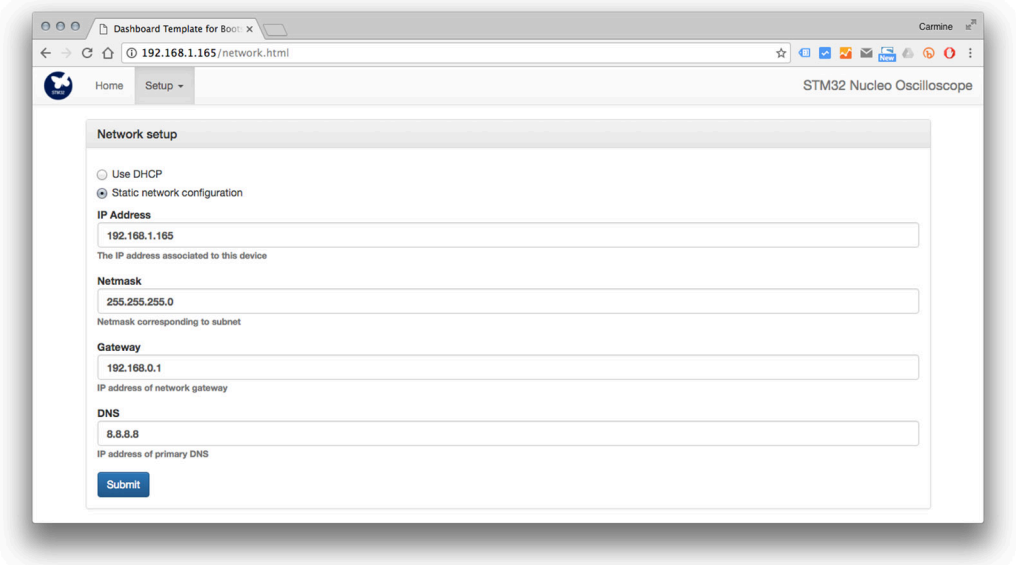


Figure 26.8: The web page used to setup network settings

The remaining of the application is essentially composed by the HTTP server. When a socket establishes a connection with a remote peer, the `httpServer_run()` routine makes a call to the `http_process_handler()` function, which is responsible to process the incoming the HTTP requests.

This function starts analyzing the request's HTTP method (`GET`, `POST`, `PUT` and so on). Here we are interested in the way the `GET` method is handled.

Filename: `Middlewares/ioLibrary_Driver/Internet/httpServer/httpServer.c`

```
531 case METHOD_GET :
532   get_http_uri_name(p_http_request->URI, uri_buf);
533   uri_name = uri_buf;
534 
535   // If URI is "/", respond by index.html
536   if (!strcmp((char *)uri_name, "/")) strcpy((char *)uri_name, INITIAL_WEBPAGE);
537   if (!strcmp((char *)uri_name, "m")) strcpy((char *)uri_name, M_INITIAL_WEBPAGE);
538   if (!strcmp((char *)uri_name, "mobile")) strcpy((char *)uri_name, MOBILE_INITIAL_WEBPAGE);
539   // Checking requested file types (HTML, TEXT, GIF, JPEG and Etc. are included)
540   find_http_uri_type(&p_http_request->TYPE, uri_name);
541 
542 #ifdef _HTTPSERVER_DEBUG_
543   printf("\r\n> HTTPSocket[%d] : HTTP Method GET\r\n", s);
544   printf("> HTTPSocket[%d] : Request Type = %d\r\n", s, p_http_request->TYPE);
545   printf("> HTTPSocket[%d] : Request URI = %s\r\n", s, uri_name);
546 #endif
547 
548   if(p_http_request->TYPE == PTYPE_CGI)
549   {
550     content_found = http_get_cgi_handler(uri_name, pHTTP_TX, &file_len);
551     if(content_found && (file_len <= (DATA_BUF_SIZE-(strlen(RES_CGIHEAD_OK)+8))))
552     {
553       send_http_response_cgi(s, http_response, pHTTP_TX, (uint16_t)file_len);
554     }
555     else
556     {
557       send_http_response_header(s, PTYPE_CGI, 0, STATUS_NOT_FOUND);
558     }
559   }
560   else
561   {
562     // Find the User registered index for web content
563     if(find_userReg_webContent(uri_buf, &content_num, &file_len))
564     {
565       content_found = 1; // Web content found in code flash memory
566       content_addr = (uint32_t)content_num;
567       HTTPSock_Status[get_seqnum].storage_type = CODEFLASH;
568     }
569     // Not CGI request, Web content in 'SD card' or 'Data flash' requested
570 #if defined(_USE_SDCARD_) && !defined(OS_USE_SEMIHOSTING)
571 #ifdef _HTTPSERVER_DEBUG_
572     printf("\r\n> HTTPSocket[%d] : Searching the requested content\r\n", s);
573 #endif
```

```
574     if((fr = f_open(&HTTPSock_Status[get_seqnum].fs, (const char *)uri_name, FA_READ)) == 0)
575     {
576       content_found = 1; // file open succeed
577 
578       file_len = f_size(&HTTPSock_Status[get_seqnum].fs);
579       HTTPSock_Status[get_seqnum].file_len = file_len;
580       strcpy(HTTPSock_Status[get_seqnum].file_name, uri_name);
581       HTTPSock_Status[get_seqnum].storage_type = SDCARD;
582     }
583 #elif defined(OS_USE_SEMIHOSTING)
584     // Not CGI request, Web content retrieved through ARM Semihosting
585     char *base_path = OS_BASE_FS_PATH;
586     char *path;
587 
588     path = malloc(sizeof(char)*strlen(base_path)+strlen(uri_name));
589     strcpy(path, base_path);
590     strcpy(path+strlen(base_path), uri_name);
591 
592     HTTPSock_Status[get_seqnum].fs = fopen((const char *)path,"r");
593     if(HTTPSock_Status[get_seqnum].fs != NULL) {
594       content_found = 1; // file open succeed
595 
596       fseek(HTTPSock_Status[get_seqnum].fs, 0L, SEEK_END);
597       file_len = ftell(HTTPSock_Status[get_seqnum].fs);
598       HTTPSock_Status[get_seqnum].file_len = file_len;
599       fseek(HTTPSock_Status[get_seqnum].fs, 0L, SEEK_SET);
600       strcpy(HTTPSock_Status[get_seqnum].file_name, uri_name);
601       HTTPSock_Status[get_seqnum].storage_type = SDCARD;
602     }
603   }
```

The `get_http_uri_name()` function at line 532 retrieve the URL requested by the client application. If this URL is only equal to `"/"`, then it means that the browser is requesting the default URL, which corresponds to the `index.html` file. The call to the function `find_http_uri_type()` at line 541 determines the *Content-Type* associated to the requested URL. The *Content-Type* is derived from the file extension. For example, the *Content-Type* of a file ending with `.gif` is set to `PTYPE_GIF`.

If the *Content-Type* is CGI[19] (line 549), then a call to the `http_get_cgi_handler()` function determines the generation of the dynamic content. We will analyze how this function is structured later. For all the other registered *Content-Types* (give a look at the `find_http_uri_type()` implementation for the complete list), the `http_process_handler()` function starts looking for the requested resource (line 561). First of all, the function checks if the content has been registered using the `reg_httpServer_webContent()` function. If so, the content is automatically retrieved from the flash

[19]The *Common Gateway Interface* (CGI) is a standardized protocol used to interface "server applications" that processes requests coming from clients dynamically. Historically, CGIs were introduced to generate web content dynamically. Nowadays, this form of server processing in web applications has been superseded by a multitude of web frameworks, built using dynamic and more powerful scripting languages like PHP, Python and Ruby.

memory and sent to the browser. If the content is not stored in the MCU flash memory and the _USE_SDCARD_ macro is set, then the function checks if the requested content is stored inside the MicroSD card (lines [575:584]). FatFs APIs are used to access the requested file. If, instead, the ARM *semihosting* is enabled, then standard C routines are used to retrieve the file from the developer's PC (lines [586:605]).

The `http_get_cgi_handler()` function has the responsibility to generate dynamic content of the web application (for example, data sampled by the ADC peripheral). The function is coded so that it requests access to the `/adc.cgi` and `/network.cgi` dynamic pages. Let us start from the second one.

Filename: `Middlewares/ioLibrary_Driver/Internet/httpServer/httpUtil.c`

```
22  extern ADC_HandleTypeDef hadc1;
23  extern uint16_t adcConv[100], _adcConv[200];
24  extern TIM_HandleTypeDef htim2;
25  extern osSemaphoreId adcSemID;
26
27  uint8_t http_get_cgi_handler(uint8_t * uri_name, uint8_t * buf, uint32_t * file_len)
28  {
29    uint8_t ret = HTTP_FAILED;
30    uint16_t len = 0;
31
32    if(strcmp((const char*)uri_name, "adc.cgi") == 0) {
33      char *pbuf = (char*)buf;
34
35      /* Compute the current TIM2 frequency */
36      uint32_t freq = HAL_RCC_GetPCLK2Freq() / (((htim2.Init.Prescaler + 1) *
37                                                 (htim2.Init.Period + 1)));
38      pbuf += sprintf(pbuf, "{\"f\":%lu,\"d\":[", freq);
39
40      /* Wait until the HAL_ADC_ConvCpltCallback() or
41         HAL_ADC_HalfConvCpltCallback() finish */
42      osSemaphoreWait(adcSemID, osWaitForever);
43      for(uint8_t i = 0; i < 100; i++)
44          pbuf += sprintf(pbuf, "%.2f,", adcConv[i]*0.805);
45      osSemaphoreRelease(adcSemID);
46
47      sprintf(--pbuf, "]}");
48      *file_len = strlen((char*)buf);
49
50      return HTTP_OK;
51
52    } else if(strcmp((const char*)uri_name, "network.cgi") == 0) {
53      wiz_NetInfo ni;
54      wizchip_getnetinfo(&ni);
55      sprintf((char*)buf, "{\"ip\":\"%d.%d.%d.%d\","
56                          "\"nm\":\"%d.%d.%d.%d\","
57                          "\"gw\":\"%d.%d.%d.%d\","
```

```
58                          "\"dns\":\"%d.%d.%d.%d\","
59                          "\"dhcp\":\"%d\"}", ni.ip[0], ni.ip[1], ni.ip[2], ni.ip[3],
60                                              ni.sn[0], ni.sn[1], ni.sn[2], ni.sn[3],
61                                              ni.gw[0], ni.gw[1], ni.gw[2], ni.gw[3],
62                                              ni.dns[0], ni.dns[1], ni.dns[2], ni.dns[3],
63                                              ni.dhcp);
64      *file_len = strlen((char*)buf);
65      return HTTP_OK;
66    }
67
68    if(ret) *file_len = len;
```

The `network.cgi` page does a simple thing: it returns the current network settings to the `/network.html` page, which in turn makes an AJAX call to the `/network.cgi` page. Just to be sure that all readers understand this matter, assuming that your Nucleo is reachable at the `192.168.1.165` IP address, then going to the URL http://192.168.1.165/network.cgi[20] in your web browser will give you the following result:

```
{"ip":"192.168.1.165","nm":"255.255.255.0","gw":"192.168.1.1","dns":"8.8.8.8","dhcp":"1"}
```

This corresponds to the network settings returned in JSON format. When the browser accesses to the `/adc.cgi` dynamic page, the application returns the current TIM2 frequency, and the ADC sampled data in JSON format. Lines [32:52] are responsible of this operation. The function `http_get_cgi_handler()` starts deriving the timer frequency at line 36. This information will be used by the web application to plot the data on the graph. Lines [42:45] represents the "tricky part" of the function.

The ADC conversion is performed in DMA circular mode. The conversion so goes on its own, and this operation is driven by the TIM2 timer. If the timer runs fast, the access to the `_adcConv[]` array could lead to a race condition: its content could be modified while the `http_get_cgi_handler()` converts it in string at line 44. The application is so organized in this way: half of the content of the `_adcConv[]` array is copied inside the `adcConv[]` array when `HAL_ADC_ConvHalfCpltCallback()` and `HAL_ADC_ConvCpltCallback()` routines are invoked, as shown below.

Filename: `src/ch25/main-ex2.c`

```
259 void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc) {
260   UNUSED(hadc);
261
262   if(osSemaphoreWait(adcSemID, 0) == osOK) {
263       memcpy(adcConv, _adcConv, sizeof(uint16_t)*100);
264       osSemaphoreRelease(adcSemID);
265   }
266 }
267
```

[20] http://192.168.1.165/network.cgi

```
268 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
269   UNUSED(hadc);
270
271   if(osSemaphoreWait(adcSemID, 0) == osOK) {
272       memcpy(adcConv, _adcConv+100, sizeof(uint16_t)*100);
273       osSemaphoreRelease(adcSemID);
274   }
275 }
```

When the `HAL_ADC_ConvHalfCpltCallback()` function is invoked, one hundred values have been stored inside the `_adcConv[]` array: we so copy the first half-part inside the `adcConv[]` array, which has a size of 100. When the other callback is called, we copy the second half. Before the two callback routines copy the content of the `_adcConv[]` array they try to acquire the semaphore `adcSem`. If available, they perform the copy, otherwise this means that the `http_get_cgi_handler()` has already acquired it and it is performing the conversion of the `adcConv[]` array. This solution prevents the generation of race conditions, even if it is not the fastest one.

The function `http_get_cgi_handler()` handles all GET requests to CGI scripts. Similarly, the function `http_post_cgi_handler()` handles all POST requests to CGI scripts.

Filename: `Middlewares/ioLibrary_Driver/Internet/httpServer/httpUtil.c`

```
72 uint8_t http_post_cgi_handler(uint8_t * uri_name, st_http_request * p_http_request, uint8_t * \
73 buf, uint32_t * file_len)
74 {
75   uint8_t ret = HTTP_OK;
76   uint16_t len = 0;
77   uint8_t *param = p_http_request->URI;
78
79   if(strcmp((const char *)uri_name, "sf.cgi") == 0) {
80     param = get_http_param_value((char*)p_http_request->URI, "f");
81     if(param != p_http_request->URI) {
82       /* User wants to change ADC sampling frequency. We so stop conversion */
83       HAL_ADC_Stop_DMA(&hadc1);
84       HAL_TIM_Base_Stop(&htim2);
85
86       /* Obtain the current TIM2 frequency */
87       uint32_t cfreq = HAL_RCC_GetPCLK2Freq() / (((htim2.Init.Prescaler + 1) *
88                                                  (htim2.Init.Period + 1))), nfreq = 0;
89
90       if(*param == '1')
91         nfreq = cfreq * 2;
92       else
93         nfreq = cfreq / 2;
94
95       htim2.Init.Prescaler = 0;
96       htim2.Init.Period = 1;
```

```
97        /* We cycle until we reach the wanted frequency. At frequencies below 30Hz,
98           this algorithm is largely inefficient */
99        while(1) {
100         cfreq = HAL_RCC_GetPCLK2Freq() / (((htim2.Init.Prescaler + 1) *
101                                           (htim2.Init.Period + 1)));
102         if (nfreq < cfreq) {
103           if(++htim2.Init.Period == 0) {
104             htim2.Init.Prescaler++;
105             htim2.Init.Period++;
106           }
107         } else {
108             break;
109         }
110       }
111       HAL_TIM_Base_Init(&htim2);
112       HAL_TIM_Base_Start(&htim2);
113       HAL_ADC_Start_DMA(&hadc1, (uint32_t*)_adcConv, 200);
114
115       sprintf((char*)buf, "OK");
116       len = strlen((char*)buf);
117     }
118
119   }
120   else if(strcmp((const char *)uri_name, "network.cgi") == 0) {
121     wiz_NetInfo netInfo;
122     wizchip_getnetinfo(&netInfo);
123
124     param = get_http_param_value((char*)p_http_request->URI, "dhcp");
125     if(param != 0) {
126       netInfo.dhcp = NETINFO_DHCP;
127     } else {
128       netInfo.dhcp = NETINFO_STATIC;
129
130       param = get_http_param_value((char*)p_http_request->URI, "ip");
131       if(param != 0)
132         inet_addr_((u_char*)param, netInfo.ip);
133       else
134         return HTTP_FAILED;
135
136       param = get_http_param_value((char*)p_http_request->URI, "sn");
137       if(param != 0)
138         inet_addr_((u_char*)param, netInfo.sn);
139       else
140         return HTTP_FAILED;
141
142       param = get_http_param_value((char*)p_http_request->URI, "gw");
143       if(param != 0)
```

```
144           inet_addr_((u_char*)param, netInfo.gw);
145         else
146           return HTTP_FAILED;
147
148         param = get_http_param_value((char*)p_http_request->URI, "dns");
149         if(param != 0)
150           inet_addr_((u_char*)param, netInfo.dns);
151         else
152           return HTTP_FAILED;
153       }
154       if(!WriteNetCfgInFile(&netInfo))
155         sprintf((char*)buf, "FAILED");
156       else
157         sprintf((char*)buf, "OK");
158
159       /* Change network parameters */
160       wizchip_setnetinfo(&netInfo);
161       len = strlen((char*)buf);
162     }
163
164   if(ret) *file_len = len;
```

This function is coded to serve two dynamic pages: `/sf.cgi` and `/network.cgi`. The second one handles the processing of the HTML form when the user changes network settings (look at the `network.html` file). The `/sf.cgi` page, instead, handles the change of the TIM2 frequency. When the user presses on the "zoom in/zoom out" icons, the browser performs a request to the `/sf.cgi` page, by passing the value `1` to increase the frequency, `0` to decrease it.

The remaining of our example application is all about HTML, CSS and JavaScript. The files `index.html` and `network.html` contain all the necessary code to plot the graph using D3js library and to show and change the network settings.

To use this example, you can simply copy the content of the **src/ch25/webpages** subdirectory on an SD card. Alternatively, you can use the ARM semihosting once the macro `OS_BASE_FS_PATH` has been set with the full path to the **src/ch25/webpages** on your PC filesystem. For example, if you are on Windows, the `OS_BASE_FS_PATH` could be set to the `C:/STM32Toolchain/projects/nucleo-f401RE/src/ch25/webpages` path.

27. Universal Serial Bus

In the nineties a group of seven companies[1] decided to join their energies to define and standardize a new hardware and communication standard to interconnect peripherals to host computers. The goal of this consortium was to unify the hardware mediums used to exchange data between peripherals and PCs, while standardizing the communication protocol at the same time. Moreover, with the advent of faster and powerful peripherals like external disk drives, portable media players, digital cameras and so on, there were an increasing demand of faster and reliable communication mediums.

Twenty-six years later, the *Universal Serial Bus* specification is still here, and it has become the most relevant and widespread industry standard in consumer electronics. During the years, several specifications have been released by the *USB Implementers Forum* (USB-IF), the committee that defines all USB-related standards. Every new release gives us a faster and more versatile communication standard as well as more portable and flexible physical mediums and connectors. The first USB 1.0 specification allowed to design devices able to exchange data with a bandwidth up to 1.5Mbit/s. The most recent specification, the USB4 released in 2019, supports a maximum bandwidth of 40Gbit/s.

The market importance of the USB standard forced all the MCUs vendors to adopt it in their most advanced devices, and obviously this is true for STM too. All STM32 families include several P/Ns that provide complete hardware support to at least the USB-device specification. Moreover, ST freely distributes a Cube middleware that simplifies the effort needed to develop USB-based applications.

From a conceptual point of view, the USB standard is not that obscure and complex protocol. Thanks to internal and dedicated phythers, STM32 developers can totally ignore all hardware-related aspects of the protocol (voltage signaling, encoding of messages, etc.), while focusing exclusively on the software part.
However, things can become hard to rule when implementing "logical protocols" on the top of the hardware medium. That's the reason why USB is still considered a hard-to-learn protocol by the majority of embedded programmers and small companies, which prefer to use simpler and still widespread communication protocols like the RS-232 and similar.

This chapter aims to provide a brief introduction to the topic. A complete dissertation of the USB specification, especially of the most recent ones like the USB 3.0, is outside of the scope of this book, and it would require a dedicated text with almost the same number of pages. Instead, we will focus our attention on the main concepts at the basis of the USB 2.0 specification, and we will analyze the structure of the ST's middleware designed for its STM32 microcontrollers.

[1] The early group was founded by Compaq, DEC, IBM, Intel, Microsoft, NEC, and Nortel.

27.1 USB 2.0 Specification Overview

For those of you totally new to USB[2], it is important to master some relevant concepts before starting to write code. To find good introductory guides to the subject is hard. The most relevant work is the series of books by Ms. Janet Louise Axelson[3] (aka Jan Axelson), especially the "USB Complete"[4] one. I think that almost every USB developer has one copy of this book on his desk, and I strongly suggest giving it a look if you are totally new to USB. Another good starting point is represented by the AN57294[5] from Cypress (now part of Infineon). This is an application note written by Robert Murphy, and the most of it is independent from the Cypress's hardware. I was inspired by that document while writing down the introductory part of this chapter, and I suggest you give it a look. ST made a good series of videos about the USB 2.0 and the complete USB stack available in the STM32Cube library. The videos are freely available on Youtube[6].

In this section I will provide the most essential possible introduction. However, I think that it is quite easy to get lost among the most relevant concepts. For this reason, I am going to provide a quick overview to USB before showing several details, so that you can have an overview to keep on hand while reading the next sub-paragraphs.

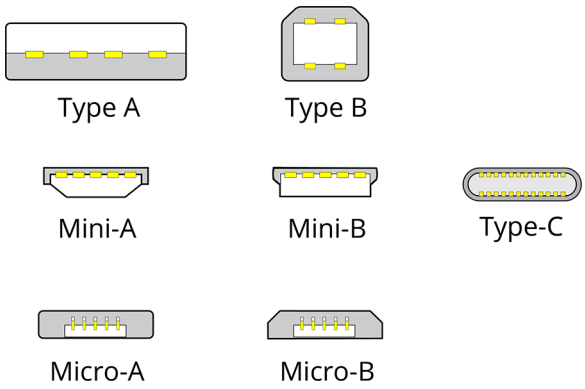


Figure 1: The most widespread USB plugs and receptacles

27.1.1 The "Before-To-Die" Guide to USB

Ok. There is no time to lose. This is the last opportunity to learn USB. Let us go.

The USB specification is declined in several releases. The most widespread one, the USB 2.0, supports three bus speeds: 480Mbit/s (also named *High Speed* - HS), 12Mbit/s (named *Full Speed* - FS), 1.5Mbit/s (named *Low Speed* - LS). The last two are inherited from the USB 1.1 specification. This indicates another important characteristic of the standard: new releases are designed to be backward compatible.

[2] Please take note that, from now on, with the term "USB" we are referring to the USB 2.0 protocol, unless different stated.
[3] http://janaxelson.com/ordering.htm#titles
[4] http://janaxelson.com/usbc.htm
[5] https://bit.ly/3zvecY6
[6] https://bit.ly/3Cyj6V6

During the years, several cables and connectors have been standardized. The standard connectors were deliberately intended to enforce the directed topology of a USB network: Type-A receptacles on host devices that supply power and Type-B receptacles on target devices that draw power. This prevents users from accidentally connecting two USB power supplies to each other, which could lead to short circuits and dangerously high currents, circuit failures, or even fire. The **Figure 1** shows seven of the most widespread USB plugs and receptacles.

USB is *host-centric*. Every communication is between a host (usually a computer) and a device. The host manages traffic on the wires, and the device responds to communications from the host. USB is a bus. This means that when the host is transmitting a packet of data, this is sent to every device connected to it. Only one device, the addressed one, accepts the data (the others all receive it, but being the address different, they silently ignore it).

A host exchanges packets with devices through "virtual pipes", which are called *endpoints* in the USB terminology. Each endpoint has one unique address, and there exist a limited number of endpoints a device can use. This means that, from the bus point-of-view, a packet is identified by a destination device address plus an endpoint number. From the developer point-of-view, a device endpoint is a memory buffer that stores received data or data to transmit. Together with an address number, each endpoint is characterized by a direction and the maximum dimension of the buffer, which clearly corresponds to the maximum number of data bytes the endpoint can send or receive in a transaction. A special endpoint, called endpoint zero (EP0) is always present in a USB device and it is used to configure it by the host.

Each USB transfer consists of one or more transactions that can carry data to or from an endpoint. A transaction begins with a "token packet" that specifies the endpoint number and the transfer direction. A **SETUP** token packet is used to configure the whole device or some features. An **IN** token packet requests a data packet from the endpoint. An **OUT** token packet precedes a data packet from the host. In addition to data, each data packet contains error-checking bits and a *Packet ID* (PID) with a data-sequencing value. Many transactions also have a handshake packet where the receiver of the data reports success or failure of the transaction. To clarify: we use an **IN** endpoint to transfer data *from the device to the host*, while we use an **OUT** endpoint to transfer data *from the host to the device*. Remember: USB is a *host*-centric protocol.

USB 2.0 supports four transfer types: *control*, *bulk*, *isochronous* and *interrupt*.
In a **control** transfer, the host sends a defined request to the device. On device attachment, the host uses control transfers to request a series of data structures called *descriptors* from the device. The descriptors provide information about the device's capabilities and help the host decide what driver to assign to the device. Control transfers have up to three stages: **SETUP**, **DATA** (optional), and **STATUS**. The **SETUP** stage contains the request. The **DATA** stage contains data from the host or device, depending on the request. The **STATUS** stage contains information about the success of the transfer.

The other transfer types don't have defined stages and it is up to the higher-level software to define how to use and interpret the data contained in a message.

Bulk transfers are the fastest on an otherwise idle bus but have no guaranteed timing. Printers and

USB *Virtual COM-Port* (VCP) data use bulk transfers.

Isochronous transfers have guaranteed timing but no error correcting. Streaming audio and video use these types of transfers.

Interrupt transfers have guaranteed maximum latency, or time between transaction attempts. Mice, and keyboards use interrupt transfers. There is an important thing to clarify: interrupt transfers have nothing related to microcontroller's interrupts. There are no dedicated interrupt lines that flow between the host and connected devices. The USB 2.0 physical medium is made of just two wires, named D+ and D-, apart from VCC and GND wires. An interrupt transfer uses a dedicated endpoint which is constantly polled by the host. These types of transfers allow to signal to the host that it must perform an action, which depends on the specific protocol.

USB specification says nothing about the meaning of transferred bytes at application level. This aspect is demanded to device *classes*, application protocols that standardize the messages' structure, their meaning, the number and type of used endpoints and so on. The USB-IF standardizes some relevant classes. For example, the *Human Interface Device* (HID) is the class dedicated to standardizing devices such as mice, keyboards, joysticks and so on.

All USB devices have a hierarchy of descriptors which provide to the host information such as what the device is, who makes it, what version of USB it supports, how many ways it can be configured, the number of endpoints and their types, etc. The more common USB descriptors are

- **Device Descriptors**: they provide an overall overview of the device functionalities.
- **Interface Descriptors**: they provide the description of the functionalities of a given device interface (for example, in a mouse the interface descriptors provide information regarding the HID class).
- **Endpoint Descriptors**: these descriptors are used to describe endpoints other than endpoint zero. The host will use the information returned from these descriptors to determine the bandwidth requirements of the bus.
- **String Descriptors**: String descriptors provide human readable information.

And that's all folks. We are now gurus of the USB protocol.
More or less.

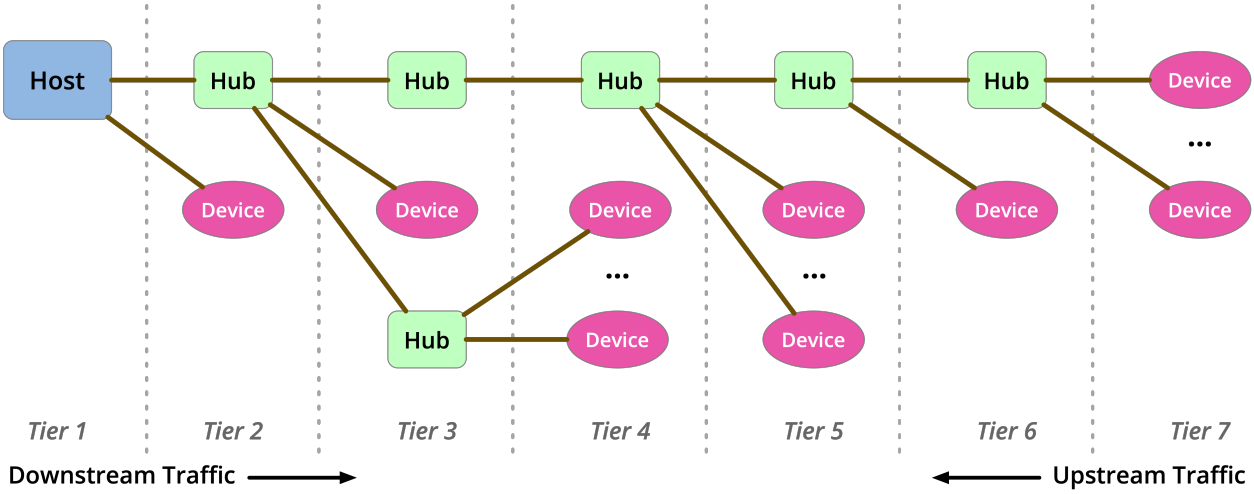


Figure 2: USB tiered topology

27.1.2 USB Physical Architecture Overview

As said before, USB is a host-centric serial bus protocol. On a USB bus can exist just one host that acts as a "master", which schedules the messages flowing on the bus. Up to 127 "slave" devices can coexist on the same bus. Host and devices are interconnected through dedicated and transparent devices, named hubs. The USB has a tiered star topology (**Figure 2**). At the root tier is the USB host. All devices connect to the host either directly or via a hub. According to the USB specification, a USB host can only support a maximum of seven tiers.

Basically, every host device is made of two fundamental components: a *Host Controller Interface* (HCI) and a root hub. A host controller is a dedicated IC that implements all the hardware requirements and part of the protocol logic to handle the flowing of messages between host and devices. There exist four main "standardized" PC host controllers on the market (OHCI, UHCI, EHCI and xHCI). From this dissertation's point-of-view, it is important to know that an HCI is a complex piece of hardware. As far as I know, there not exist on the market Cortex-M microcontrollers that provide a USB 2.0 HCI. Instead, several MCUs, like some STM32 microcontrollers, provide the ability to work as USB FS host implementing a limited set of capabilities. This allows to develop USB host devices able to interface some common and not so fast USB devices (mice, keyboards, etc.).

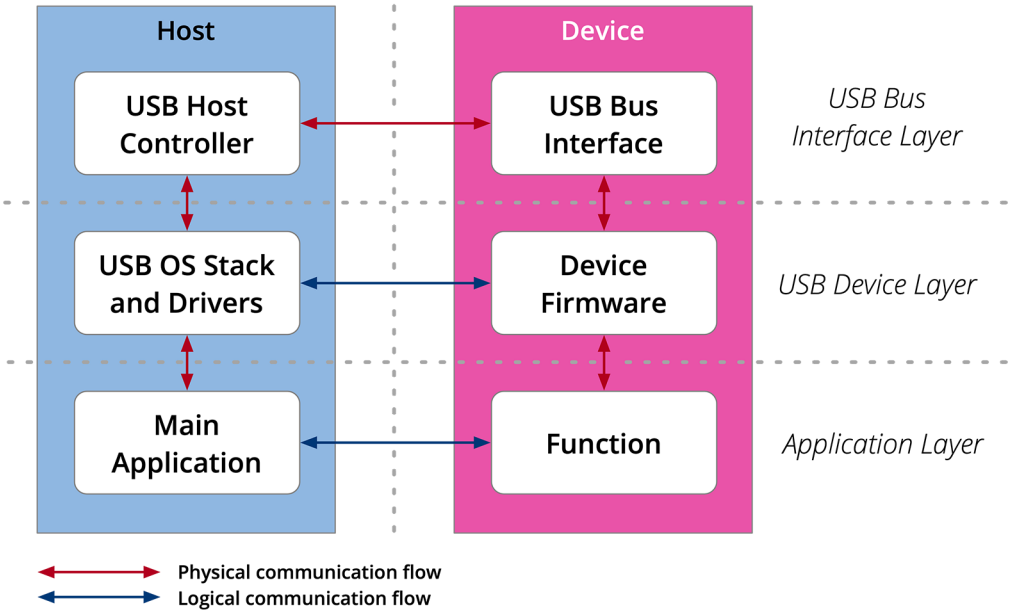


Figure 3: USB architecture overview

The USB specification can also be seen as a tiered architecture, which is schematized in **Figure 3**. USB can be divided in three main tiers: the *bus interface*, the *device* and the *application* layer.

The *bus interface layer* is represented by the hardware controller. A PC integrates an HCI, while an STM32 MCU provides a dedicated hardware phyther, compatible with the signaling of the USB 2.0 specifications, plus a *Serial Interface Engine* (SIE). The SIE is the frontend of the hardware layer and handles most of the protocol described in the USB specification. The SIE typically comprehends signaling up to the transaction level. The functions that it handles include:

- Packet recognition, transaction sequencing
- SOP, EOP, RESET, RESUME signal detection/generation
- Clock/Data separation
- NRZI Data encoding/decoding and bit-stuffing
- CRC generation and checking (Token and Data)
- Packet ID (PID) generation and checking/decoding
- Serial-Parallel/Parallel-Serial Conversion

The *device layer* corresponds to the software low-level part in a USB connection. It contains both some generic USB functionalities (such as device descriptors used to discover device's capabilities, as we will see later) and protocol specific functionalities. For example, a *Mass Storage Class* (MSC) device implements all those functionalities that allow to interface an external disk drive to a PC. At the same time, a PC Operating System (e.g., Windows) needs a dedicated "driver" to interface such a USB device.

The *application layer* implements the necessary logic to use the device. From the USB device's point-of-view, this layer implements the actual code that provides a given functionality. For example,

our MSC device may implement the necessary code so that an SD card is interfaced using the SPI protocol, and it appears to a PC like any other MSC device (that is, an external disk). The MSC protocol is essentially made of SCSI commands that need to be adapted to specific *Memory Technology Device* (MTD) by the application code.

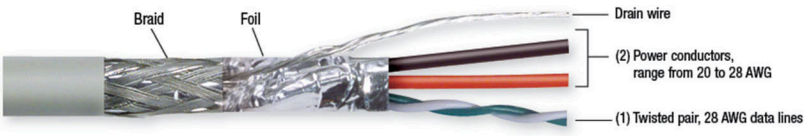


Figure 4: The wiring of a USB 2.0 cable

The USB-IF also standardizes the hardware mediums to interconnect USB devices: plugs, sockets and cables. **Figure 4** shows the typical structure of a USB 2.0 cable. A shield made of a braid and a foil (plus the drain wire that ensures correct grounding of the braid) surrounds power and signal wires. Power conductors are used to supply adequate energy to devices. The red wire, which is also named VBUS, gives a constant 4.40 - 5.25V supply to all attached devices. A USB 2.0 end device can draw up to 500mA from the USB line. More recent USB 3.x standard allows to supply up to 5A@20V (100W).

While USB supplies up to 5.25V to devices, the data lines (D+ and D-) function at 3.3V. Data lines are a differential twisted pair. The reason for using the differential D+ and D- signal is for rejecting common-mode noise. If noise becomes coupled into the cable, it will normally be present on all wires in the cable. With the use of a differential amplifier in the USB hardware internal to the host and device, the common-mode noise can be rejected. USB 2.0 uses the inverted non-return zero encoding (NRZI) schema to encode binary signals.

Over the years a series of connectors have been standardized by the USB-IF. Historically, connectors (which include plugs and sockets) was divided in two main categories: Type-A and Type-B. Hosts always use a Type-A ports and connectors, while devices use Type-B ports and connectors. Initially, the USB specification included only the larger Type-A and Type-B connectors for devices but later included the Mini and Micro connections (see **Figure 1**).
These Mini and Micro connectors were initially developed for USB On-the-Go (USB OTG), which is a USB specification that allows devices that would normally act as slaves to become hosts. However, due to the smaller size of the Mini-B and Micro-B connectors compared to Type-B, they were adopted in many electronics despite lacking USB OTG capabilities.

The evolution of standards, and the market pressure from other players (Apple with its Lightning standard for iPhone/IPad), forced the USB-IF to introduce a fully symmetrical connector. With the advent of the USB 3.1 standard in 2014, the USB-IF introduced Type-C connectors, which use the same layout both for the host and devices.

Currently, the USB specification defines seven speeds for a USB system: Low-Speed, Full-Speed, Hi-Speed, SuperSpeed 5/10/20/40GB. At the time of writing this chapter, the majority of STM32 MCUs with a USB peripheral support just Full-Speed mode, while the most performant ones also USB 2.0 High-Speed.

Low-, Full-, and High-Speed devices are often advertised as 1.5 Mb/s, 12 Mb/s, and 480 Mb/s,

respectively. However, these are bus rates and not data rates. The actual data rates are affected by bus loading, transfer type, overhead, OS, and so forth.

When a USB device is connected to a host, the speed of the device needs to be detected. This is done with pull-up resistors on the D+ or D- lines. A 1.5-kΩ pull-up on the D+ line indicates that the attached device is a Full-Speed device. A 1.5KΩ pull-up resistor on the D- line indicates the attached device is a Low-Speed device. High-Speed devices start as Full-Speed devices, so they have a 1.5KΩ pull-up on the D+ line. When the device is connected, it emits a sequence of signals over D+/D- lines during the reset phase of enumeration. If the hub supports High-Speed, then the pull-up resistor is removed. **Apart for MCU from the STM32F1-series**, the majority of STM32 MCUs provide internal 1.5KΩ pull-up resistors and there is no need to add them as external components.

A Low-Speed and Full-Speed USB device will use a 48-MHz clock for the SIE and the other USB clocking purposes. This 48-MHz clock and the bus speed is what will determine USB bit times:

- Full-Speed: 48 MHz / 12 Mb/s = 4 clocks per bit time.
- Low-Speed: 48 MHz / 1.5 Mb/s = 32 clocks per bit time.

Several STM32 MCUs, especially those belonging to low-cost value lines, provide an integrated 48MHz RC network, so that you can save space and BOM cost (they are called USB crystal-less MCUs).

27.1.3 USB Logical Architecture Overview

As STM32 developer, you do not need to dig into physical-related aspects of the USB 2.0 specification. ST engineers packed all the hardware circuitry that implements the phyther and the SIE inside the chip. Instead, as programmers, it is paramount to understand the USB 2.0 logical architecture and its state machine.

27.1.3.1 Device States

When a USB 2.0 device is inserted in the host/hub port, a series of non-trivial things take place. This process is named *Enumeration* and it is made up of an electrical phase and a software stage. We will start analyzing the electrical part.

When a USB 2.0 is inserted in a Type-A port, both the host/hub and the end device go through a well-defined set of states.

- **ATTACHED**: occurs when a device is attached to a host/hub but does not give any power to the VBUS line.
- **POWERED**: a device is attached to the USB and has been powered but has not yet received a reset request. A device in this state cannot draw more than 100mA.
- **DEFAULT**: a device is attached to the USB, is powered, and has been reset by the host. At this point, the device does not have a unique device address. The device responds to address 0.

- **ADDRESSED**: the device is attached to the USB, powered, has been reset, and has had a unique address assigned to it. The device however has not yet been configured.
- **CONFIGURED**: the device is attached to the USB, powered, has been reset, assigned a unique address, has been configured, and is not in suspend state. At this point, bus-powered devices can draw more than 100 mA.
- **SUSPENDED**: this state occurs when the device is attached and configured but has not seen activity on the bus for 3ms.

Let us analyze better each one of them.

ATTACHED
The user wants to use his USB device. He so gently inserts the device in the hub port. According to the limited experience of this author, what happens immediately after the device insertion is largely implementation dependent. Some hub controllers leave the VBUS line disconnected, and they are designed to sense a variation of the D+/D- lines capacitance. According to USB physical specification, data lines must have a well characterized bus capacitance, and some controllers are designed to measure it. On a capacitance variation (which drops to few pF), they start powering the VBUS line and begin the device speed identification phase.

Other hub controllers are designed to detect an over-current event on the VBUS line. When the device is inserted, the device starts drawing a lot of current due to the limited resistance in the very beginning insertion phase. The hub detects this event, and it detaches the VBUS line. It then re-powers the VBUS line and starts the device identification phase.

Finally, other controllers provide a standardized *Attach Detection Protocol* (ADP). The hub performs attach probing by discharging the VBUS line, then measuring the time required for a well-known current to charge the line to a defined voltage. If the line does not charge in the expected time, no device is present. The probing repeats about every 1.75s.

POWERED
Once the device insertion is complete (the device is hence in the **ATTACHED** state) the hub start powering it by supplying up to 100mA. In this phase starts the device speed identification. On attaching to a port, the device connects to the bus by bringing the appropriate pull-up line high (according to its speed). The value of the pull-up resistor allows to derive the device speed. The device is so in the **POWERED** state, and the hub knows which physical protocol (according to the speed) to use to exchange data with it.

DEFAULT
The host is notified by the hub of a new device. It so asks to the hub to reset the corresponding port. A well-defined reset sequence takes place, which consists in pulling down both the data lines for at least 10ms. This is the very first bus condition that the microcontroller notifies to the developer. This give us the opportunity to start the early configuration of the microcontrollers, so that the USB peripheral is able to answer to further requests coming from the host. When the reset condition ends, the device is said to be in the **DEFAULT** state. In this state, the host/hub also samples the data line to understand if the device supports the High-Speed mode, by issuing special signals over data lines.

ADDRESSED
A device in the **DEFAULT** state answers to host on the 0x0 address. As said before, USB 2.0 is a broadcast protocol. All devices receive the same messages, but they discard those not having its device address. When a device is inserted in the hub port, it automatically sets the default address to 0x0. Since the host *enumerates* one device at time (by physically powering a port at a time), only one device will answer to communications sent to 0x0 address. The host is so able to communicate with the device, and it start asking a series of bytes that describe the device. These bytes are called *device descriptors* and we will analyze them in depth later.

What happens next is dependent on the Operating System and application layer. Some older Windows releases reset again the device (and, sometimes, again again…). The device descriptor is polled again and if the operation succeeds, then the host assigns a unique address to the device. Some other OSes, instead, skip this additional reset and assign the address. The device is so **ADDRESSED**, and the host can ask for additional information.

CONFIGURED
The host learns about complete device's capability by reading device's descriptors using the assigned addressed. The list of required information may be long, and we will analyze the most of them later. Some of them include the device powering needs: if the device needs to draw more than 100mA, it must notify the host so that it "authorizes" it to increase the power consumption. Other information gathered by the host concerns the number and types of communication channels (endpoints), the maximum dimension of communication buffers, and so on. At the end of this inquiring stage, the Operating System also selects and activates the device driver needed to interact with the device. Once all these operations are complete, the device is in the **CONFIGURED** state, and it is ready to accept higher level messages from the application software.

SUSPENDED
A device enters in this state after detecting no bus activity for at least 3ms. In the **SUSPENDED** state the device should limit its use of the bus power. Both configured and unconfigured devices must support this state. We will see more about this later.

27.1.3.2 Communication Endpoints

In USB 2.0, host and devices communicate through virtual pipes. Every message is addressed to a specific device and a given communication pipe. There are two types of pipes in a USB 2.0 system: *control pipes* and *data pipes*. For every device exists just one control pipe and a variable number of data pipes, according to the device functionalities. For example, a mouse will typically have the control pipe plus one data pipe used to send pointer coordinates to the PC. The control pipe is the only bidirectional pipe in the USB system. All the data pipes are instead unidirectional.

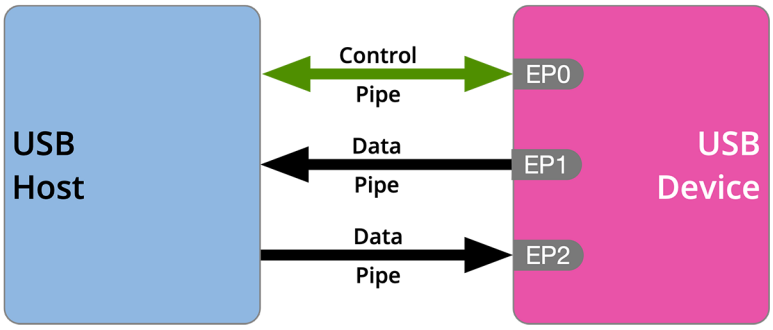


Figure 5: USB endpoint/pipe model

The USB specification defines four different data transfer types. The type of pipe used depends on the characteristic of the data to transfer. Let us analyze them.

CONTROL TRANSFERS

Control transfers send and receive device information across the bus. They are used for sending commands to the device, make inquiries, and configure the device. This transfer uses the *control pipe*. For example, the host sets the device's address by using a control transfer. The advantage of control transfers is guaranteed accuracy. Errors that occur are properly detected and the data is resent. Control transfers have a 10% reserved bandwidth on the bus in Low- and Full-Speed devices and 20% in High-Speed devices.

INTERRUPT TRANSFERS

These transfers are used on devices that must use a high reliability method to communicate a small amount of data. This is commonly used in HID devices, such as mice and keyboards. The name of this transfer can be misleading. It is not truly a dedicated interrupt line flowing from the device to the host but uses a polling rate. However, you get a guarantee that the host checks for data at a predictable interval. Interrupt transfers give guaranteed accuracy as errors are properly detected and faulty transactions are retried at the next transaction. Interrupt transfers have a guaranteed bandwidth of 90% in Low- and Full-Speed devices and 80% in High-Speed devices. This bandwidth is shared with isochronous transfers. Interrupt maximum packet size is a function of device speed. High-Speed capable devices support a maximum packet size of 1024 byes. Full-Speed capable devices support a maximum packet size of 64 bytes. Low-Speed devices support a maximum packet size of 8 bytes. This type of transfers uses a data pipe.

BULK TRANSFERS

These transfers are commonly used on devices that move relatively large amounts of data at highly variable times where the transfers can use any available bandwidth space. They are the most common transfer type for USB devices. Delivery time with a bulk transfer is variable because there is no set aside bandwidth for the transfer. The delivery time varies depending on how much bandwidth on the bus is available, which makes the actual delivery time unpredictable. Bulk transfers give guaranteed accuracy because errors are properly detected, and transactions are resent. Bulk transfers are useful in moving large amounts of data that are not time sensitive. A bulk transfer maximum packet size is a function of device speed. High-Speed capable devices support a maximum packet size of 512 bytes. Full-Speed capable devices support a maximum packet size of 64-bytes. Low-Speed

devices do not support bulk transfer types. These transfers use a data pipe.

ISOCHRONOUS TRANSFERS
These transfers are continuous and real-time transfers with a pre-negotiated bandwidth. Isochronous transfers must support streams of error tolerant data because they do not have an error recovery mechanism or handshaking. Errors are detected through the CRC field, but not corrected. With isochronous transfers, you get the tradeoff of guaranteed delivery versus guaranteed accuracy. Streaming music or video are examples of an application that uses isochronous endpoints because the occasional missed data is ignored by the human ears and eyes. Isochronous transfers have a guaranteed bandwidth of 90% in Low- and Full-Speed devices (80% in High-Speed devices) that is shared with interrupt endpoints. High-Speed capable devices support a maximum packet size of 1024 bytes. Full-Speed devices support a maximum packet size of 1023 bytes. Low-Speed devices do not support isochronous transfer types.
There are special considerations with isochronous transfers. You generally want 3x buffering to ensure data is ready to go by having one actively transmitting buffer and another buffer being actively loaded. These transfers use a data pipe.

Control and data pipes are an abstraction built on the top of *device endpoints*. In the USB specification, a device endpoint is a uniquely addressable portion of a USB device that is the source or sink of information in a communication flow between the host and device. There are four types of *endpoints* that correspond to the transfer types seen before: *control*, *interrupt*, *bulk* and *isochronous* endpoints.

An endpoint is defined by two parameters: a unique address for a given device and a direction. An **IN** endpoint is used to transfer data *from the device to the host*. An **OUT** endpoint to transfer data *from the host to the device*. The control endpoint, named **SETUP**, is a special endpoint that has the `0x0` address and it is the sole bidirectional endpoint in a USB system: programmers will refer to it to send and receive data from the host. The control endpoint is the only required endpoint in a USB system, and there exist classes that just communicate through the **SETUP** endpoint (for example, the DFU class). During the enumeration process, the host sends and receive messages through the control endpoint to acquire device capabilities and to configure it. The control endpoint ensures that the host has a default way to access to device without knowing the actual implemented functionalities. Endpoints are bidirectional by nature. It is not until they are configured that they take on a single direction (becoming unidirectional). Endpoint 1 (shortened *EP1*), for example, can be either an IN or OUT endpoint. It is in the device configuration that it will officially make EP1 an IN endpoint.

At the basis of USB 2.0 specification there is an important concept to keep in mind. As said several other times before, USB is a host centric protocol. All transactions originate by the host, regardless of the endpoint direction. If the host needs to send data to device, it writes a given number of bytes in an OUT endpoint. If, instead, the device needs to send data to the host, it writes the data in an IN endpoint and waits until the host reads it. From the firmware point of view, the device is notified of transfer completion (in both directions) thanks to a dedicated interrupt. As we will see later, the USB protocol is a complete interrupt-driven protocol, and this heavily impacts on the structure of our embedded application (and on the possibility to debug it easily).

In addition to the EP0, the number of endpoints supported in any device is based on its design

requirements. A simple design such as a mouse may need only one IN endpoint. More complex designs may need several data endpoints. The USB specification sets a limit on the number of endpoints to 16 for each direction (16 IN/16 OUT – 32 Total) for High-Speed and Full-Speed devices, which does not include the control endpoints 0 IN and 0 OUT. However, STM32 microcontrollers provide less endpoints (usually a total of eight, including EP0, but for some STM32MCU even less). Low-Speed devices are limited to two endpoints.

Endpoints use cyclic redundancy checks (CRCs) to detect errors in transactions. The CRC is a calculated value used for error checking. The actual calculation equation is explained in the USB specification and the handling of these calculations is taken care of by the USB hardware so that the proper response can be issued. The recipient of a transaction checks the CRC against the data. If the two match, then the receiver issues an ACK. If the data and the CRC do not match, then no handshake is sent. This lack of a handshake tells the sender to try again.

Table 1 summarizes the four endpoint types and their relevant characteristics.

Transfer Type	Control	Interrupt	Bulk	Isochronous
Typical Use	Device Initialization and Management	Mouse and Keyboard	Printer and Mass Storage	Streaming Audio and Video
Low-Speed Support	Yes	Yes	No	No
Error Correction	Yes	Yes	Yes	No
Guaranteed Delivery Rate	No	No	No	Yes
Guaranteed Bandwidth	Yes (10% FS - 20% HS)	Yes (90% FS - 80% HS)	No	Yes (90% FS - 80% HS)
Guaranteed Latency	No	Yes	No	Yes
Maximum Transfer Size	1024 bytes (HS) 64 bytes (FS) 8 bytes (LS)	1024 bytes (HS) 64 bytes (FS) 8 bytes (LS)	512 bytes (HS) 64 bytes (FS)	1024 (HS) 1023 (FS)
Maximum Transfer Speed	832 KB/s	1.216 MB/s	1.216 MB/s	1.023 MB/s

Table 1: Endpoint types and their relative features

27.1.4 USB 2.0 Communication Protocol Overview



If you are new to the USB 2.0 protocol, it is not strictly required that you study this paragraph and its subparagraphs in this phase. The information provided here is mostly handled in hardware and by the CubeHAL. The transaction protocol is essentially transparent for the programmer, unless he needs to debug the device using a protocol analyzer (more about this later). I suggest skipping this part and to go to the next paragraph about device descriptors.

From the developer's point-of-view, a USB 2.0 message consists of a stream of bytes whose meaning depends on the message type. For example, a mouse sends to the host several data messages using

an interrupt endpoint. The exchanged data contains relative coordinates that are used to move the pointer on the screen.

From the USB point-of-view, a message is a series of *frames*, as shown in **Figure 6**. Each frame consists of a *Start of Frame* (SOF) followed by one or more *transactions*. Each transaction is made up of a series of *packets*. A packet is preceded with a packet id and a sync pattern and ends with an *End of Packet* (EOP) pattern. Depending on the transaction type there may be one or more data packets and some transactions may or may not have a handshake packet. At a minimum, a transaction has a token packet.

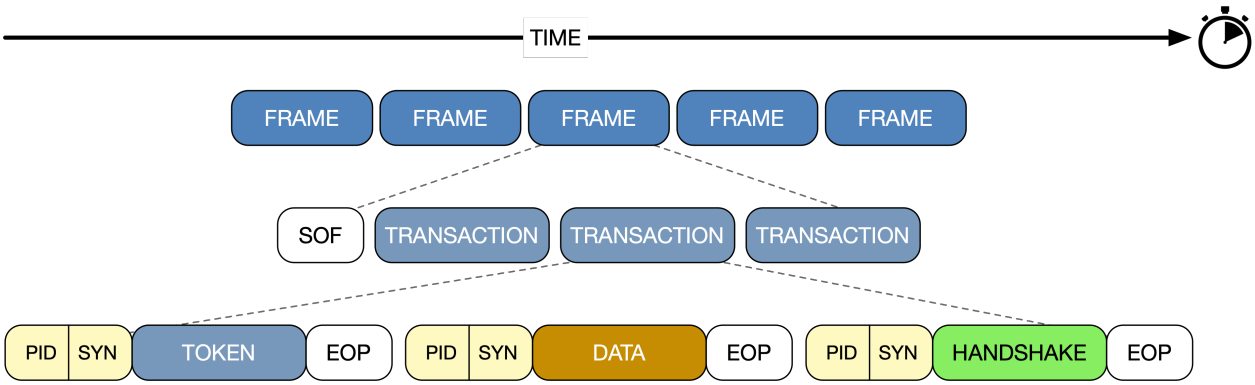


Figure 6: USB communication in a given time-frame

Each *packet* can contain different pieces of information. What information is included depends on the packet type. The following is a list of the potential information that can be included with a packet:

- **Packet ID (PID)**: declares a transaction type as an **IN/OUT/SETUP/SOF**.
- **Optional Device Address**: specifies the device which the message is addressed to.
- **Optional Endpoint Address**: specifies the endpoint address between 1 and 16 (between 1 and 7 for all STM32 MCUs).
- **Optional Payload Data**: depending on the packet type and USB version, this can contain up to 1024 bytes of data, whose mining is related to the packet type.
- **Optional CRC**.

27.1.4.1 Packet Types

According to USB 2.0 specification, there exist four types of packets with a given fixed structure:

- **Token packets**
  - Initiate transaction
  - Identify device involved in transaction
  - Always sourced by the host
- **Data packets**

  - Deliver payload data
  - Sourced by host or device
- **Handshake packets**:
  - Acknowledge error-free data receipt
  - Sourced by host or device
- **Special packets**
  - Facilitates speed differentials
  - Sourced by host-to-hub devices

Let us analyze these packet types in depth.

TOKEN PACKET
Token packets always originate from the host and are used to control the traffic on the bus. The function of the token packet depends on the activity performed.
IN tokens are used to request that devices send data to the host.
OUT tokens are used to precede data from the host.
SETUP tokens are used to precede control commands from the host.
SOF tokens are used to mark time frames. In every **IN**, **OUT**, and **SETUP** token packet there is a 7-bit device address, 4-bit endpoint ID, and 5-bit CRC. **Figure 7** shows the structure of the five types of token packets.

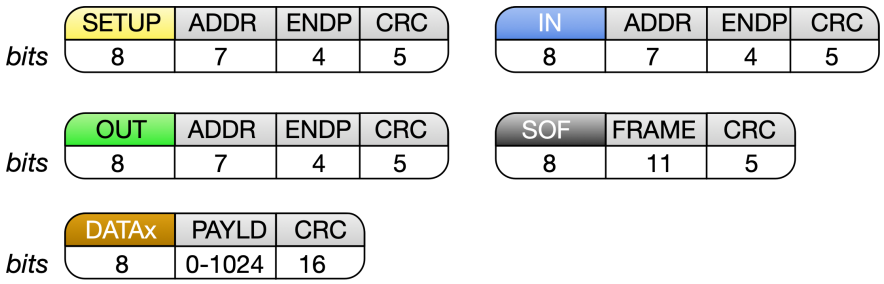


Figure 7: Types of Token packets in USB 2.0

SOF packets give a way for devices to identify the beginning of a frame and synchronize with the host. They are also used to prevent a device from entering suspend mode (which it must do if 3ms pass without an SOF). SOF packets are only seen on full- and high-speed devices and are sent every millisecond. A handshake packet does not occur for an SOF packet. High-speed communication goes a step further with *microframes*. With a High-Speed device, an SOF is sent out every 125µs and frame count is only incremented every 1ms.

DATA PACKET
Data packets follow **IN**, **OUT**, and **SETUP** token packets. The size of the payload data ranges from 0 to 1024 bytes depending on the transfer type and the USB version. The Packet ID toggles between **DATA0** and **DATA1** for LS/FS devices, and **DATA0/1/2/MDATA** for HS devices. The packet closes with a 16-bit CRC. The composition of a data packet can be seen in **Figure 7**. The data toggle is updated at the host and the device for each successful data packet transfer. One advantage to the

data toggle is that it acts as an additional error detection method. If a different packet ID is received than what is expected, the device will be able to know there was an error in the transfer and it can be handled appropriately. An example where the data toggle is used is if an ACK is sent but not received. In this instance, the sender updates the data toggle from `1` to `0` but the receiver does not. The receiver remains at `1`. This causes the host and device to be out of sync on the next data stage, which indicates an error.

HANDSHAKE PACKETS
Handshake packets conclude each transaction. Each handshake includes an 8-bit packet ID and is sent by the receiver of the transaction. Each USB speed has several options for a handshake response. Which ones are supported depend on the USB speed:

- **ACK**: acknowledges successful completion. (LS/FS/HS)
- **NAK**: negative acknowledgement. (LS/FS/HS)
- **STALL**: Error indication sent by a device. (LS/FS/HS)
- **NYET**: indicates the device is not ready to receive another data packet. (HS Only)

SPECIAL PACKETS
The USB specification defines four special packets.

- **PRE**: it is issued to hubs by the host to indicate that the next packet is low speed.
- **SPLIT**: Precedes a token packet to indicate a split transaction. (HS Only)
- **ERR**: Returned by a hub to report an error in a split transaction. (HS Only)
- **PING**: Checks the status for a Bulk OUT or Control Write after receiving a NYET handshake. (HS Only)

27.1.4.2 Transaction Types

SETUP, **IN** and **OUT** tokens determine three different transaction types. Let us briefly look at them.

27.1.4.2.1 Control Transactions

Control transactions identify, configure, and control devices. They enable the host to read information about a device, set the device address, establish configuration, read an endpoint configuration, issue certain class-related commands and so on. In the enumeration phase, a control transfer is always directed to the control endpoint of a device (which is always the EP0). After the device is completely enumerated, control transfers can be addressed to other endpoints to interact with other device aspects, such as its class (but this is quite uncommon).

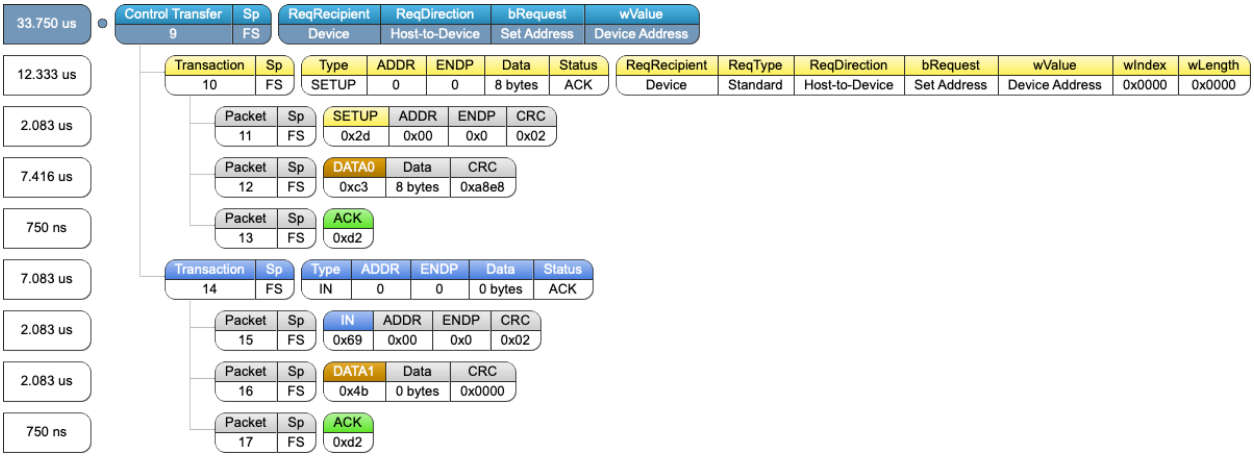


Figure 8: An example of a Control transaction

Control transactions have three stages: the *setup stage* (mandatory), the *data stage* (optional), and the *status stage* (mandatory). The *status stage* includes a single **IN** or **OUT** transaction that reports on the success or failure of the previous stages. The data packet for this stage is always **DATA1** (unlike normal **IN** and **OUT** transactions that toggle between **DATA0** and **DATA1**) and contains a zero-length data packet. The status stage ends with a handshake transaction that is sent by the receiver of the preceding packet.

Figure 8 shows a complete control transaction issued by the host to setup the device address. The packets 11, 12 and 13 identify the *setup stage*.

Packet 11 is the *token packet* with a PID equal to **SETUP**, a destination address equal to 0x0 (corresponding to the new device which is currently in **DEFAULT** state) and an endpoint ID corresponding to the control endpoint EP0.

Packet 12 is the *data packet* that contains the effective request to the device. It contains the `SET_ADDRESS` command and the corresponding address ID.

Packet 13 is the **ACK** packet issued by the device. This control transaction has no *data stage*.

Finally, transaction 14 (made of packets 15, 16 and 17) corresponds to the *status stage* (see the empty **DATA1** packet).

Offset	Field	Size (byte)	Value	Description
0	**bmRequestType**	1	Bit-Map	**D7 Data Phase Transfer Direction** 0 = Host to Device 1 = Device to Host **D6..5 Type** 0 = Standard 1 = Class 2 = Vendor 3 = Reserved **D4..0 Recipient** 0 = Device 1 = Interface 2 = Endpoint 3 = Other **4..31 = Reserved**
1	**bRequest**	1	Value	Request
2	**wValue**	2	Value	Value
4	**wIndex**	2	Index or Offset	Index
6	**wLength**	2	Count	Number of bytes to transfer if there is a data phase

Table 2: Structure of a SETUP token

The content of the data packet right after the **SETUP** token packet has a well-defined structure which is reported in **Table 2**. As you can see, a **SETUP** packet can be sent to a device, an interface (more about this later) or an endpoint. When a **SETUP** packet is addressed to a device (`bmRequestType.Type=0/bmRequestType.Recipient=0`), we talk about a *Standard Device Request*, and the content of the fields `bRequest`, `wValue`, `wIndex` and `wLength` correspond to one of the rows in the **Table 3**. As you can see, the `SET_ADDRESS` request has this structure:

- `wValue`: it is the address assigned to the new device;
- `wIndex`: not used field and set to zero;
- `wLength`: not used field and set to zero;
- `Data`: the request has no *data stage*.

The rest of *Standard Device Requests* have the following usages:

- `GET_STATUS`: the request is directed at the device, which will return two bytes during the *data stage* indicating if the device is self-powered and if it has a remote wakeup feature and it can wake the host up during suspend (for example, a click on a mouse button can wake-up the host).
- `CLEAR_FEATURE` and `SET_FEATURE`: these requests can be used to set Boolean features. When the designated recipient is the device, the only two feature selectors available are `DEVICE_REMOTE_WAKEUP` and `TEST_MODE`. Test mode allows the device to exhibit various conditions. These are further documented in the USB Specification Revision 2.0.

- `GET_DESCRIPTOR` and `SET_DESCRIPTOR`: these requests are used to set/return the specified descriptor in `wValue`. A request for the configuration descriptor will return the device descriptor and all interface and endpoint descriptors in the one request. More about descriptors in a while.
- `GET_CONFIGURATION` and `SET_CONFIGURATION`: these requests are used to return or set the current device configuration. In the case of a `GET_CONFIGURATION` request, a byte will be returned during the *data stage* indicating the devices status. A zero value means the device is not configured and a non-zero value indicates the device is configured. `SET_CONFIGURATION` is used to enable a device. It should contain the value of `bConfigurationValue` of the desired configuration descriptor in the lower byte of `wValue` to select which configuration to enable.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000 0000b	**GET_STATUS (0x00)**	Zero	Zero	Two	Device Status
0000 0000b	**CLEAR_FEATURE (0x01)**	Feature Selector	Zero	Zero	None
0000 0000b	**SET_FEATURE (0x03)**	Feature Selector	Zero	Zero	None
0000 0000b	**SET_ADDRESS (0x05)**	Device Address	Zero	Zero	None
1000 0000b	**GET_DESCRIPTOR (0x06)**	Descriptor Type & Index	Zero or Language ID	Descriptor Length	Descriptor
0000 0000b	**SET_DESCRIPTOR (0x07)**	Descriptor Type & Index	Zero or Language ID	Descriptor Length	Descriptor
1000 0000b	**GET_CONFIGURATION (0x08)**	Zero	Zero	1	Configuration Value
0000 0000b	**SET_CONFIGURATION (0x09)**	Configuration Value	Zero	Zero	None

Table 3: List of requests in a *Standard Device Request*

When a control transaction is addressed to an interface (`bmRequestType.Type=0 / bmRequestType.Recipient=1`), we talk about a *Standard Interface Request*. In this case, the fields of the request can assume the values reported in **Table 4**. Most of the requests reported in **Table 4** are reserved for future usage, apart for the `GET_INTERFACE` and `SET_INTERFACE` requests that allows to switch between alternate interfaces configuration, a feature we will describe later.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000 0001b	**GET_STATUS (0x00)**	Zero	Interface	Two	Interface Status
0000 0001b	**CLEAR_FEATURE (0x01)**	Feature Selector	Interface	Zero	None
0000 0001b	**SET_FEATURE (0x03)**	Feature Selector	Interface	Zero	None
1000 0001b	**GET_INTERFACE (0x0A)**	Zero	Interface	One	Alternate Interface
0000 0001b	**SET_INTERFACE (0x11)**	Alternative Setting	Interface	Zero	None

Table 4: List of requests in a *Standard Interface Request*

When a control transaction is addressed to an endpoint (`bmRequestType.Type=0 / bmRequestType.Recipient=2`), we talk about a *Standard Endpoint Request*. In this case, the fields of the request can assume the values reported in **Table 5**. The `GET_STATUS` request returns two bytes indicating the status (Halted/Stalled) of an endpoint. `CLEAR_FEATURE` and `SET_FEATURE` are used to set Endpoint Features. The standard currently defines one endpoint feature selector, `ENDPOINT_HALT`

(`0x00`) which allows the host to stall and clear an endpoint. Only endpoints other than the default endpoint are recommended to have this functionality. Finally, a `SYNCH_FRAME` request is used to report an endpoint synchronization frame.

bmRequestType	bRequest	wValue	Windex	wLength	Data
1000 0010b	**`GET_STATUS (0x00)`**	Zero	Endpoint	Two	Endpoint Status
0000 0010b	**`CLEAR_FEATURE (0x01)`**	Feature Selector	Endpoint	Zero	None
0000 0010b	**`SET_FEATURE (0x03)`**	Feature Selector	Endpoint	Zero	None
1000 0010b	**`SYNCH_FRAME (0x12)`**	Zero	Endpoint	Two	FrameNumber

Table 5: List of requests in a *Standard Endpoint Request*

Finally, when a control transaction is addressed to an interface and its type is *Class* (`bmRequestType.Type=1 / bmRequestType.Recipient=1`), we talk about a *Class Request*. *Class requests* are a set of class-specific requests defined by the class itself. These requests allow the host to inquire about the capabilities and state of a specific device and to set the state of output and feature items. We will analyze some class-specific requests later in this chapter.

27.1.4.2.2 IN/OUT Transactions

IN transactions refer to a data transfer from the device to the host. They are addressed to an endpoint different from EP0. These transactions are initiated by the host by sending an **IN** token packet. The target device responds by sending one or more data packets, and the host responds with a handshake packet. The device could not be able to answer to the host request: if this the case, the device needs to answer with a NAK. The host will keep retrying the request by sending again the same transaction.

Conversely, **OUT** transactions refer to a transfer that occurs from the host to the device. In this type of transaction, the host sends the appropriate token packet (either `OUT` or `SETUP`) and follows with one or more data packets. The receiving device ends the transaction by sending the appropriate handshake packet. The device could not be able to accept the data from the host: if this the case, the device needs to answer with a **NYET** or **NAK**[7]. The host will keep retrying the request by sending again the same transaction.

For example, look at the *bulk* transaction in **Figure 9**. The transaction 5244 is an **OUT** transaction ending with a **NYET** from the device. The host send a **PING** packet, and even if the device acknowledges the **PING**, it refuses the subsequent data in the **OUT** transaction (packet 5249) by issuing a **NYET**. Successive transaction (5252) will succeed instead (look to the **ACK** packet #5258).

[7]The reason for the introduction of the **NYET** handshake packet in USB 2.0 were bandwidth utilization efficiency considerations. If a device responds with a **NYET**, the host knows that the device will very likely **NAK** the next **OUT** transaction which means that the whole frame time the data is being transmitted is wasted: The exact same data will have to be sent again. That's why NAKing an **OUT** transaction wastes a lot of frame time since the **OUT** transaction occupies the bus without purpose and it competes with other transactions/devices as well, taking frame time from them. Imagine the protocol without the **NYET** handshake: The host would have to send the same whole block of data (i.e., up to 512 bytes for bulk endpoints) every time the device **NAK**s just to inquiry if the device is ready. If the host gets a **NYET** instead, it will start **PING**ing the device, asking if the device is ready to receive more data. A **PING** transaction is very short compared to a large data **OUT** transaction. Hence, if the device **NAK**s the **PING**, the host can use the rest of the frame for other transactions instead which leads to better utilization of the bus.

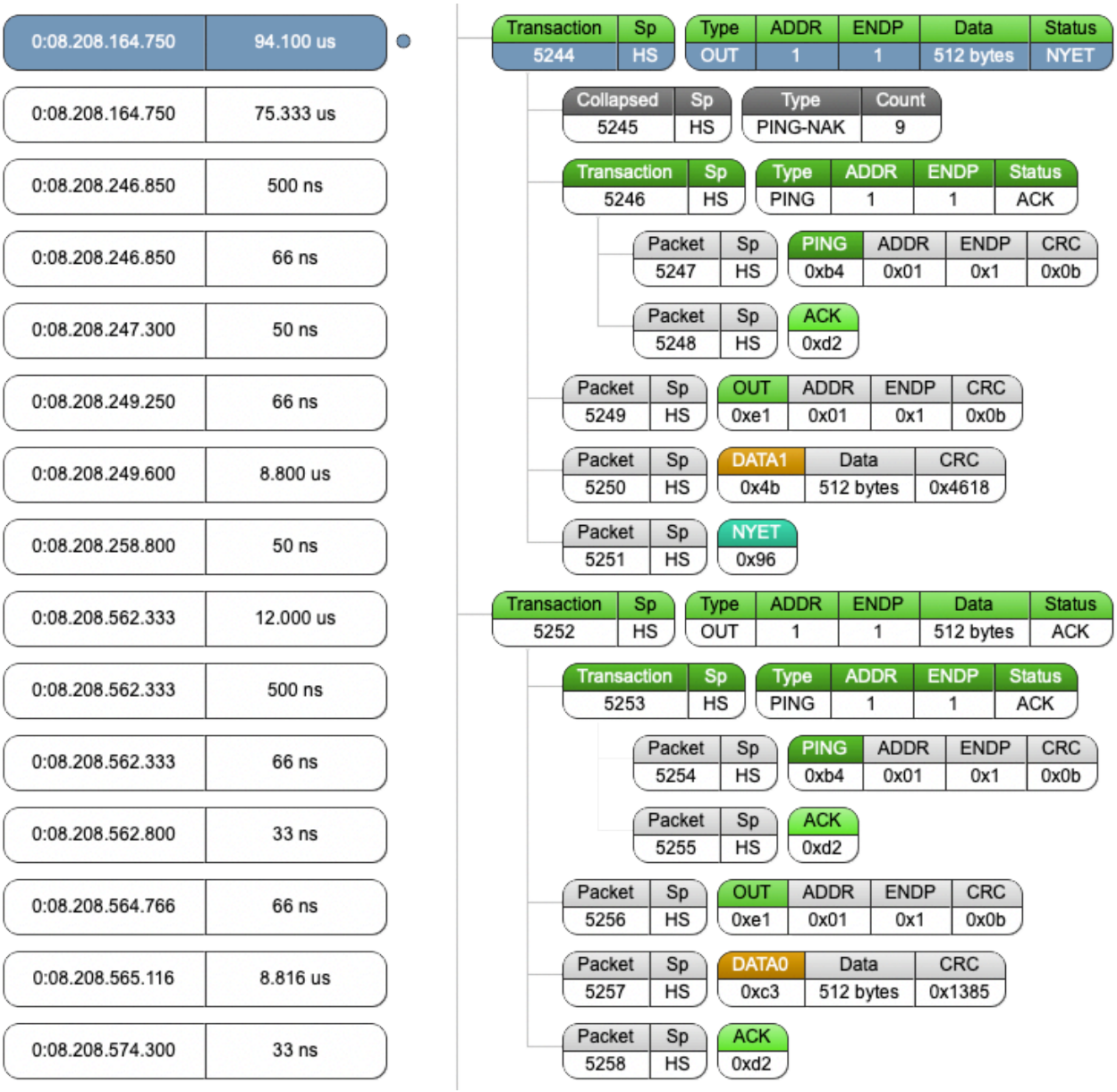


Figure 9: An example of a OUT transaction

27.1.4.3 Device and Interface Descriptors

A communication between a host and a USB device is just a matter of bytes flowing through one or more endpoints. Apart for the control endpoint (EP0), every endpoint can assume both the IN and the OUT function, and the bytes transferred assume a given meaning according to the device characteristics, especially the implemented USB class (HID, CDC, MSC, etc.). So, there should be a way so that a device communicates to the host what features it implements, how many endpoints uses, how much current it draws, and so on. Finally, the same device must provide to the Operating System other non-neglectable information such as device and vendor names, so that the end user

can be warned regarding the device activity and usage. In the USB protocol this task is demanded to descriptors, tables hardcoded inside the FW that strictly adhere to a well-defined structure and that are exchanged with the host during the *enumeration* phase. USB 2.0 defines four different types of descriptors, as shown in **Figure 10**. Let us briefly analyze them.

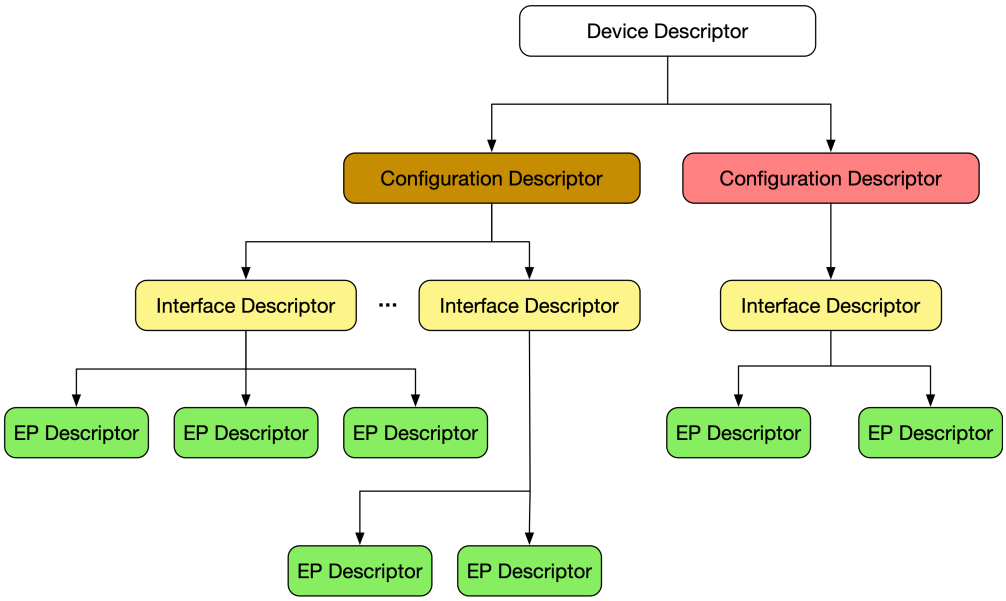


Figure 10: Complete hierarchy of a device descriptors

27.1.4.3.1 Device Descriptors

Device descriptors give the host information such as the USB specification to which the device conforms, the number of device configurations, protocols (classes) supported by the device, Vendor Identification (also known as VID, which is an unique identifiers that each company gets from the USB Implementers Forum), Product Identification (also known as PID, different from a packet ID, which every company choose uniquely), and a serial number if the device has one. The device descriptor is where some of the most crucial information about the USB device is contained. **Table 6** shows the structure for a device descriptor. Let us analyze individual fields.

Offset	Field	Size (Bytes)	Description
0	**bLength**	1	Length of this descriptor = 18 bytes
1	**bDescriptorType**	1	Descriptor type = DEVICE (01h)
2	**bcdUSB**	2	USB specification version (BCD)
4	**bDeviceClass**	1	Device class
5	**bDeviceSubClass**	1	Device subclass
6	**bDeviceProtocol**	1	Device Protocol
7	**bMaxPacketSize0**	1	Max Packet size for endpoint 0
8	**idVendor**	2	Vendor ID (or VID, assigned by USB-IF)
10	**idProduct**	2	Product ID (or PID, assigned by the manufacturer)
12	**bcdDevice**	2	Device release number (BCD)
14	**iManufacturer**	1	Index of manufacturer string
15	**iProduct**	1	Index of product string
16	**iSerialNumber**	1	Index of serial number string
17	**bNumConfigurations**	1	Number of configurations supported

Table 6: Structure of a device descriptor

- `bLength`: this field corresponds to the total length in bytes of the device descriptor. This value it is fixed by the USB standard.
- `bDescriptorType`: identifies the device descriptor type and it corresponds to 01h for device descriptors table.
- `bcdUSB`: this field reports the USB revision that the device supports, which should be latest supported revision. This is a binary-coded decimal value that uses a format of 0xAABC, where A is the major version number, B is the minor version number, and C is the sub-minor version number. For example, a USB 2.0 device would have a value of 0x0200 and USB 1.1 would have a value of 0x0110. This is normally used by the host in determining which driver to load.
- `bDeviceClass`, `bDeviceSubClass`, `bDeviceProtocol`: these fields are used by the operating system to identify a driver for a USB device during the enumeration process. Filling in this field in the device descriptor prevents different interfaces from functioning independently, such as a composite device. Most USB devices define their classes in the interface descriptor and leave these fields as 00h.
- `bMaxPacketSize`: this field reports the maximum number of packets supported by the EP0. Depending on the device, the possible sizes are 8 bytes, 16 bytes, 32 bytes, and 64 bytes.
- `iManufacturer`, `iProduct`, `iSerialNumber`: these fields are indexes in the table of string descriptors. String descriptors give details about the manufacturer, product, and serial number. If string descriptors exist, these variables should point to their index location in the strings tables. If no string exists, then the respective field should be assigned a value of zero.
- `bNumConfigurations`: this field defines the total number of configurations the device can support. Multiple configurations allow the device to be configured differently depending on

certain conditions such as being bus powered or self-powered. More details regarding this stuff are discussed in the next paragraph.

27.1.4.3.2 Configuration Descriptors

Configuration descriptors provide information regarding a specific configuration of the device, such as the number of interfaces, if the device is self-powered or bus-powered, if it can wake up the host (remote wake-up) upon an event (for example, a mouse can wake up the Host PC if one of it is button is pressed). A device must provide at least one configuration, but it may provide multiple configurations. For example, a device may expose some functionalities just if it is self-powered by an external power adapter. However, while the USB standard provides such flexibility for both FW and HW developers, it is quite uncommon to find USB devices with multiple configurations in practice. **Table 7** shows the structure for a configuration descriptor. Let us analyze individual fields.

- `wTotalLength`: this field corresponds to the total length in bytes of the configuration descriptor, including the interface and endpoint descriptors. This value it is fixed by the USB standard.
- `bNumInterfaces`: this field defines the total number of possible interfaces in this configuration. This field has a minimum value of 1.
- `bConfigurationValue`: it defines a value to use as an argument to the `SET_CONFIGURATION` request to select this configuration.
- `bmAttributes`: it defines parameters for the USB device. If the device is bus powered, bit 6 is set to 0, if the device is self-powered, then bit 6 is set to 1. If the USB device supports remote wake-up, bit 5 is set to 1. If remote wake-up is not supported, bit 5 is set to 0.
- `bMaxPower`: this field defines the maximum power consumption drawn from the bus when the device is fully operational, expressed in 2 mA units. If a self-powered device becomes detached from its external power source, it may not draw more than the value indicated in this field.

Offset	Field	Size (Bytes)	Description
0	**bLength**	1	Length of this descriptor = 9 bytes
1	**bDescriptorType**	1	Descriptor type = CONFIGURATION (02h)
2	**wTotalLength**	2	Total length including interface and endpoint descriptors
4	**bNumInterfaces**	1	Number of interfaces in this configuration
5	**bConfigurationValue**	1	Configuration value used by `SET_CONFIGURATION` to select this configuration
6	**iConfiguration**	1	Index of string that describes this configuration
7	**bmAttributes**	1	Bit 7: Reserved (set to 1) Bit 6: Self-powered Bit 5: Remote wakeup
8	**bMaxPower**	1	Maximum power required for this configuration (in 2 mA units)

Table 7: Structure of a configuration descriptor

27.1.4.3.3 Interface Descriptors

An interface descriptor describes a specific interface within a configuration. The number of endpoints for an interface is identified in this descriptor. The interface descriptor is also where the USB Class of the device is declared. There are many predefined classes that a USB device can be, many of which are listed in **Table 11**. A USB device class identifies the device functionality and aids in the loading of a proper driver for that specific functionality. **Table 8** shows the structure for an interface descriptor. Let us analyze individual fields.

- `bInterfaceNumber`: this field corresponds to the interface number.
- `bAlternateSetting`: an interface may have alternative configurations selected by using the `SET_INTERFACE` request. For example, a USB interface endpoints may act as INTERRUPT pipes in normal settings, but might act as BULK pipe in alternate settings providing you the facility of two different mode on the same interface. Like for the multi-configuration support, this possibility is offered by the USB protocol, but it has few practical applications.
- `bNumEndpoints`: specifies the number of endpoints used by the interface excluding the control endpoint EP0.
- `bInterfaceClass`, `bInterfaceSubclass`, `bInterfaceProtocol`: these fields refer to the USB class and protocol exposed by the current interface. Some classes are well defined and standardized by the USB-IF. We will talk about this later.
- `iInterface`: this field refers to the index in the string descriptor containing the human-readable description.

Offset	Field	Size (Bytes)	Description
0	**bLength**	1	Length of this descriptor = 9 bytes
1	**bDescriptorType**	1	Descriptor type = INTERFACE (04h)
2	**bInterfaceNumber**	1	Zero based index of this interface
3	**bAlternateSetting**	1	Alternate setting value
4	**bNumEndpoints**	1	Number of endpoints used by this interface (not including EP0)
5	**bInterfaceClass**	1	Interface class
6	**bInterfaceSubclass**	1	Interface subclass
7	**bInterfaceProtocol**	1	Interface protocol
8	**iInterface**	1	Index to string describing this interface

Table 8: Structure of an interface descriptor

27.1.4.3.4 Endpoint Descriptors

Each endpoint used for an interface has its own descriptor. This descriptor contains the information required by the host to determine the characteristics of each endpoint. An endpoint descriptor is always returned as part of the configuration information. There is no descriptor for endpoint zero. **Table 9** shows the structure for an endpoint descriptor. Let us analyze individual fields.

- `bEndpointAddress`: this field defines the endpoint address and its direction. In all STM32 it is possible to configure just eight endpoints including the EP0. The EP0 is the sole bidirectional endpoint. Instead, for the other seven endpoints a direction must be specified by setting the eighth bit in this field. For example, a value 0x01 corresponds to an **OUT** endpoint with address 0x01, while a value 0x82 corresponds to an **IN** endpoint with address 0x02.
- `bmAttributes`: this field is a bitmask used to set type of endpoint. The first two bits set the endpoint type. If the endpoint type is set to isochronous, then the next two bits in the bitmask set the type of synchronization[8] and the next twos are used to set feedback endpoints for audio feedback loop.

Offset	Field	Size (Bytes)	Description
0	**bLength**	1	Length of this descriptor = 7 bytes
1	**bDescriptorType**	1	Descriptor type = ENDPOINT (05h)
2	**bEndpointAddress**	1	**Bit 3...0**: The endpoint number **Bit 6...4**: Reserved, reset to zero **Bit 7**: Direction. Ignored for Control Endpoint **0** = OUT endpoint **1** = IN endpoint
3	**bmAttributes**	1	**Bits 1..0**: Transfer Type **00** = Control **01** = Isochronous **10** = Bulk **11** = Interrupt If not an isochronous endpoint, bits 5...2 are reserved and must be set to zero. If isochronous, they are defined as follows: **Bits 3..2**: Synchronization Type 00 = No Synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous **Bits 5..4**: Usage Type 00 = Data endpoint 01 = Feedback endpoint 10 = Implicit feedback Data endpoint 11 = Reserved
4	**wMaxPacketSize**	2	Maximum packet size for this endpoint
6	**bInterval**	1	Polling interval in milliseconds for interrupt endpoints (1 for isochronous endpoints, ignored for control or bulk)

Table 9: Structure of an endpoint descriptor

- `wMaxPacketSize`: this field sets the dimension of the packet size on the endpoint. Every endpoint type can accept a maximum amount of data depending on the USB protocol specifications (LS/FS/HS). Please, refer to the **Table 1** for this value.

[8] This topic is related to audio/video transmission over USB. For a detailed description of the three possible synchronization types, please refer to this thread.

- `bInterval`: this field sets the polling interval in milliseconds for interrupt endpoints. Remember: interrupt endpoints are not truly "interrupts" since every transaction on the USB bus is always initiated by the host. They are just endpoints polled at regular interval by the host.

Offset	Field	Size (Bytes)	Description
0	**bLength**	1	Length of this descriptor = 7 bytes
1	**bDescriptorType**	1	Descriptor type = STRING (03h)
2..n	**bString** **-or-** **wLangID**	Varies	Unicode encoded text string -or- LANGID code

Table 10: Structure of a string descriptor

27.1.4.3.5 String Descriptors

The string descriptor gives user-readable information about the device. Usually, this descriptor contains the name of the device, the manufacturer, the serial number, or names for the various interfaces or configurations. If strings are not used in a device, any string index field of the descriptors mentioned earlier must be set to 00h. Strings are defined using Unicode UTF-16 encodings in little endian and can support multiple languages with a language ID code. In a Windows system, these stings can be seen in the device manager. **Table 10** shows the structure for a string descriptor.

27.1.4.4 USB Classes

So far, we talked about the way the communication is arranged between the host and the USB device. We provided a lot of information about *endpoints*, virtual channels where bytes are transferred, and the role that some of them play during the configuration stage of a USB device. But what about the bytes transferred during a regular transaction? For example, what about the way a mouse communicates with the PC to signal its movement or the click of one of its buttons?

USB specification demands this task to the USB *classes*, code information that is used to identify a device's functionality and to nominally load a device driver based on that functionality. The information is contained in three bytes of the device descriptor (`bDeviceClass`, `bDeviceSubClass`, `bDeviceProtocol` - see **Table 6**).

There are two places on a USB device where class code information can be placed. One is in the Device Descriptor, and the other is in the Interface Descriptors. Some defined class codes are allowed to be used only in a Device Descriptor, others can be used in both Device and Interface Descriptors, and some can only be used in Interface Descriptors. The **Table 11** shows the currently defined set of standard class values, what the generic usage is, and where that class can be used (either Device or Interface Descriptors, or both).

Offset	Usage	Description	Examples
00h	`Device`	Use class information in the Interface Descriptors	Device class is unspecified, interface descriptors are used to determine needed drivers
01h	`Interface`	Audio	Speaker, microphone, sound card, MIDI
02h	`Both`	Communications and Communications Device Class (CDC) Control	Modem, ethernet adapter, Wi-Fi adapter, RS232/RS485 serial adapter
03h	`Interface`	Human Interface Device (HID)	Keyboard, mouse, joystick
05h	`Interface`	Physical Interface Device (PID)	Force feedback joystick
06h	`Interface`	Image	Camera, scanner
07h	`Interface`	Printer	Printers, CNC machine
08h	`Interface`	Mass Storage Class (MSC)	External hard drives, flash drives, memory cards
09h	`Device`	USB Hub	USB hubs
0Ah	`Interface`	CDC-Data	Used in conjunction with class 02h.
0Bh	`Interface`	CCID	USB smart card reader
0Dh	`Interface`	Content Security	Fingerprint reader
0Eh	`Interface`	Video	Webcam
0Fh	`Interface`	Personal Healthcare	Heart rate monitor, glucose meter
DCh	`Both`	Diagnostic Device	USB compliance testing device
E0h	`Interface`	Wireless Controller	Bluetooth adapter
EFh	`Both`	Miscellaneous	ActiveSync device
FEh	`Interface`	Application Specific	IrDA Bridge, Test & Measurement Class (USBTMC), USB DFU (direct firmware update)
FFh	`Both`	Vendor Specific	Indicates a device needs vendor specific drivers

Table 11: USB classes standardized by the USB-IF

What advantages gives the usage of a USB standard class? The first advantage is that a "standard" device can be used easily by the end-user. This fosters the adoption of the device. For example, consider how it is easy nowadays to connect a mouse or a keyboard to a PC. But the biggest advantage to using a predefined USB device class is the cross-platform support across various operating systems. All major operating systems include a driver in the OS for many of the predefined USB classes that eliminates the need to create a custom driver. And this removes one of the biggest obstacles in the USB adoption especially for small companies and individual developers. To develop a custom driver, especially for the Windows OS, is not a trivial task, that requires adequate investments and time.

For this reason, it is quite common to find USB devices implementing the HID class that are either no mice nor keyboards, and that exchange custom messages between a specific application and the device through custom HID reports.

STM32 Family	USB Device FS Only	USB OTG FS	USB OTG HS
STM32F2		✓	✓
STM32F4		✓	✓
STM32F7		✓	✓
STM32H7		✓	✓
STM32F0	✓		
STM32F1	✓	✓	
STM32F3	✓		
STM32G0	✓		
STM32G4	✓		
STM32L0	✓		
STM32L1	✓		
STM32L4	✓	✓	
STM32L4+		✓	
STM32L5	✓		

Table 12: Implementation of the USB peripheral in all STM32-series

27.2 STM32 USB Device Library

Chances are that you are a little bit mazed by all that information, especially if this is the first time you deal with the USB specification. This is normal, and you do not have to be worried too much. This for two reasons. First, the more you will work with the USB the more you will master relevant topics. Second, you do not have to implement the whole USB stack protocol from scratch.
To develop a complete USB stack it is not for the faint of heart. A lot of details were omitted in the previous paragraphs. Moreover, without adequate instrumentations (a USB protocol sniffer) it is unlikely that you can implement a complete stack by yourself. As we will see later, to debug a USB communication is impossible with regular debug tools because, in a USB transaction, things happen too fast. Even a print on the UART can affect a communication session.

Luckily for us, ST already designed a complete USB device and host stack that can be easily integrated in our firmware. However, not all STM32 families provide a USB controller able to work both in device and host mode. **Table 12** lists the implementation of the USB peripheral in all STM32-series. As you can see, just some of them provides a USB *On-The-Go* (OTG) that allows to design devices able to work both in host and device mode, and just few of them are able to work in *High Speed* mode by using an external phyther. For the complete list of the USB features in all STM32

families (including the suggested USB hardware and PCB guidelines), refer to the AN4879[9].

ST also developed the support to the most widespread USB classes, and it is relatively easy to adapt the code at your needs. In the next paragraph we will study how to configure CubeMX so that it automatically generates for us all the necessary code to add USB support in our firmware. Next, we will study the architecture of the ST device stack. At the first instance, the generated code may appear hard to understand, especially for the tight interconnection between the USB device library and the CubeHAL. Finally, we will develop examples to play with some of the standard USB classes. Let us go.

To add the ST USB Device library in an existing project is a straightforward operation. You just have to follow this simple two-steps process. First, you have to enable the USB peripheral in CubeMX (you can find it in the *Connectivity* section of the **Category** pane). Next, you have to enable the USB Device Middleware by selecting the wanted USB class as shown in **Figure 11**.

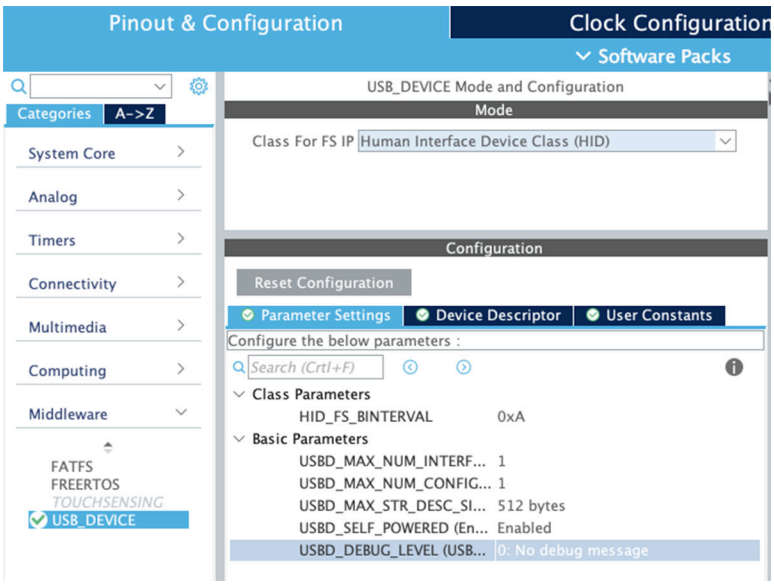


Figure 11: CubeMX options to configure the STM32 USB Device Library

27.2.1 Understanding Generated Code

The **Figure 12** shows the new project structure once that CubeMX generates the code with the USB Device library and the wanted USB class[10]. At the same time, the **Figure 13** shows the whole application architecture: keep it on hands while reading the rest of this paragraphs.

As you can see, three new files are added in the CubeHAL. Despite the different name, all the three files are related to the configuration and the handling of the USB data processing.

[9] https://bit.ly/3Fagq0f

[10] Several files in the **Drivers/STM32XXxx_HAL_Driver/Src** folder was omitted to simplify the image.

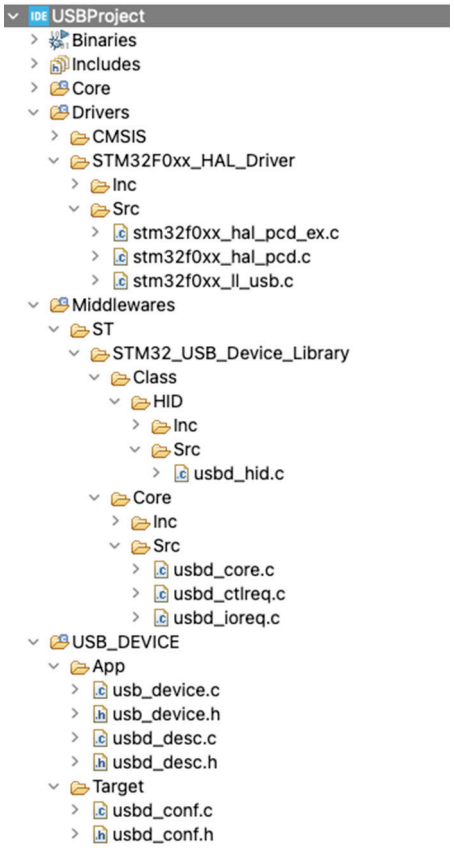


Figure 12: Structure of the generated project

- **stm32XXxx_hal_pcd.c**: this module corresponds to the Low-Layer USB *Peripheral Control Driver* (PCD). This module does not correspond to a real peripheral in the MCU. All operations carried out by the functions contained in the PCD module are all related to activities of the USB peripheral (mostly, IRQ management). As we will see next, all the operations performed on the USB peripheral (initialization, endpoint allocation, data transfer, class management, etc.) are carried out in the handler of the USB peripheral interrupt. This requires that it is important to keep track of the different states and transitions of the USB peripheral, so that the correct USB state machine is implemented. The PCD module is designed so that all events generated by the USB peripheral are routed to dedicated callbacks. These callbacks are "glued" with the main USB core library by the functions inside the **USB_DEVICE/target/usbd_conf.c** file.

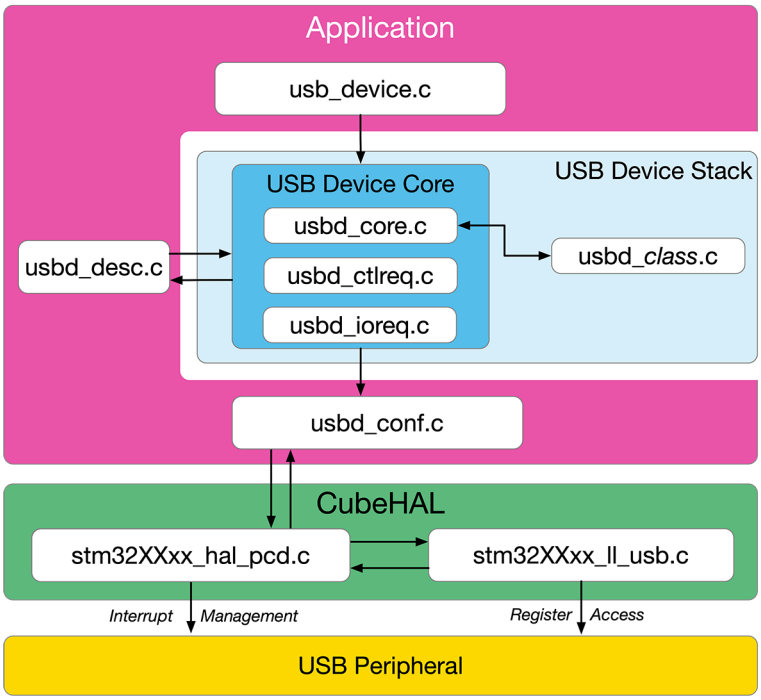


Figure 13: Architecture of the generated application with STM32 USB Device Library

- **stm32XXxx_hal_pcd_ex.c**: some more recent STM32 MCUs implement advanced USB features such as the handling of the *Low-Power Mode* (LPM) or a dedicated buffer memory called *Packet Memory Address* (PMA). This buffer memory is present in those STM32 MCUs with less SRAM memory so that the usage of PMA avoids allocating buffers in SRAM.
- **stm32XXxx_ll_usb.c**: this file contains the *Cube Low-Level* Library (Cube-LL) for the USB peripheral. As said several times before, the timing in USB communications is so stringent that it is important to have the most optimized code to avoid unwanted delays. For this reason, all routines related to the USB registers are designed to be optimized and dedicated to the actual peripheral in the given STM32 MCU.

CubeMX also adds two main folders to the project tree. The first one is **Middleware**, which contains the STM32 USB Device library. This in turn is organized in subfolders:

- **Class/<class>**: this folder contains all C source and header files related to the selected USB Class. For example, in **Figure 12** you can see the file **usbd_hid.c** containing all the necessary code to handle the HID class.
- **Core**: this folder contains the main core USB stack, which is implemented in the following files:
  - **usbd_core{.c,.h}**: these files include the implementation of the USB core state machine. The **Table 13** shows the main functions description.

– **usbd_ctlreq{.c,.h}**: these files include variables and functions ensuring the processing of all requests of the Chapter 9 in the USB 2.0 specification[11]. The **Table 14** shows the main functions description.
– **usbd_ioreq{.c,.h}**: these files include the implementation of data sending and reception on the endpoint 0. The **Table 15** shows the main functions description.

Table 13: API of the USB core stack

Function	Description
USBD_Init	Initializes the Device Handler and state machine
USBD_DeInit	De-initializes the Device Handler
USBD_RegisterClass	Links the device structure handler to the class structure handler
USBD_Start	Starts the device core
USBD_Stop	Stops the device core and class
USBD_SetClassConfig	Initializes the device's class with the input configuration
USBD_ClrClassConfig	Clears a device's class configuration
USBD_LL_SetupStage	Handles the setup stage requests
USBD_LL_DataOutStage	Handles the data reception from the host on the convenient endpoint
USBD_LL_DataInStage	Handles the data sending to the host on the convenient endpoint
USBD_LL_Reset	Reinitializes the Device's handler
USBD_LL_SetSpeed	Sets the device's speed
USBD_LL_Suspend	Modifies the device status into suspended
USBD_LL_Resume	Resumes the device old status before being suspended
USBD_LL_SOF	Handles the Start Of Frame (SOF) event
USBD_LL_IsoINIncomplete	Handles the Isochronous IN Incomplete event
USBD_LL_IsoOUTIncomplete	Handles the Isochronous OUT Incomplete event
USBD_LL_DevConnected	Handles the Device connected event
USBD_LL_DevDisconnected	Handles the Device disconnected event

Table 14: API to process the USB 2.0 Chapter 9 specification

Function	Description
USBD_StdDevReq	Handles all the requests that are addressed to a USB device
USBD_StdItfReq	Handles all the requests that are addressed to an interface
USBD_StdEPReq	Handles all the requests that are addressed to an endpoint
USBD_GetDescriptor	Handles the different types of GET_DESCRIPTOR request
USBD_SetAddress	Handles the SET_ADDRESS request
USBD_GetConfig	Handles the GET_CONFIGURATION request
USBD_SetConfig	Handles the GET_CONFIGURATION request
USBD_GetStatus	Handles the GET_STATUS request
USBD_SetFeature	Handles the SET_FEATURE request
USBD_ClrFeature	Handles the CLEAR_FEATURE request
USBD_ParseSetupRequest	Organizes the received data buffer into setup request structure
USBD_CtlError	Handles USB low level Error

[11] https://bit.ly/3ES3H2K

Table 14: API to process the USB 2.0 Chapter 9 specification

Function	Description
`USBD_GetString`	Converts Ascii string into Unicode one
`USBD_GetLen`	Returns the string length

Table 15: API for data exchange over EP0

Function	Description
`USBD_CtlSendData`	Starts data sending on the EP0
`USBD_CtlContinueSendData`	Sends the remaining data on the EP0
`USBD_CtlPrepareRx`	Prepares the EP0 for receiving data
`USBD_CtlContinueRx`	Continues the reception of data on EP0
`USBD_CtlSendStatus`	Sends zero length packet on the EP0
`USBD_CtlReceiveStatus`	Receives zero length packet on the EP0
`USBD_GetRxCount`	Returns the received data length

The second folder added to the project by CubeMX is **USB_DEVICE**. This in turn is organized in two subfolders.

- **App**: this folder contains the C file related to the initialization phase of the USB stack and the main device descriptor.
  - **usb_device{.c,.h}**: these files contain the initialization sequence of the USB Stack. It contains just the `MX_USB_DEVICE_Init()` routine, which initializes the USB stack, registers the USB class and starts the USB peripheral.
  - **usbd_desc{.c,.h}**: these files contain the USB device descriptors.
- **Target**: this folder contains just two C files.
  - **usbd_conf{.c,.h}**: these files contain the "glue" code to link USB Device Core part to the actual STM32 device. The content of this folder plays the same role of the *Board Support Packages* (BSP) seen in other ST examples or projects. In this file you can find the effective implementation of all *callbacks* defined by the PCD module and the implementation of those `USBD_LL` routines related to the actual USB peripheral.

27.2.2 USB Initialization Sequence

To better understand the relation between the different source files seen before, it is useful to get a look to the initialization sequence of the USB stack. While reading the following text, keep on hands the sequence diagram on the next page[12].

1 - The first step of the sequence takes place inside the `MX_USB_DEVICE_Init()` (**usb_device.c** file). Here the instance of the `struct USBD_HandleTypeDef` is defined. The routine invokes in turn the following functions:

[12]The diagram has been taken from this ST webpage.

- `USBD_Init()` (**usbd_core.c**): this function checks whether all USB descriptors are ok, assigns the Device descriptors, sets the default state and passes the control to the `USBD_LL_Init()` function.
  - `USBD_LL_Init()` (**usbd_conf.c**): this function takes care of the initialization of USB peripheral descriptors and the PMA memory if supported by the MCU. It then calls the `HAL_PCD_Init()` function.
    - `HAL_PCD_Init()` (**stm32XXxx_hal_pcd.c**): this function sets all the PCD related callbacks and configure the control endpoint EP0. It then passes the control to the `USB_DevInit()`.
      - `USB_DevInit()` (**stm32XXxx_ll_usb.c**): this function performs the effective configuration of the USB peripheral by setting its registers.

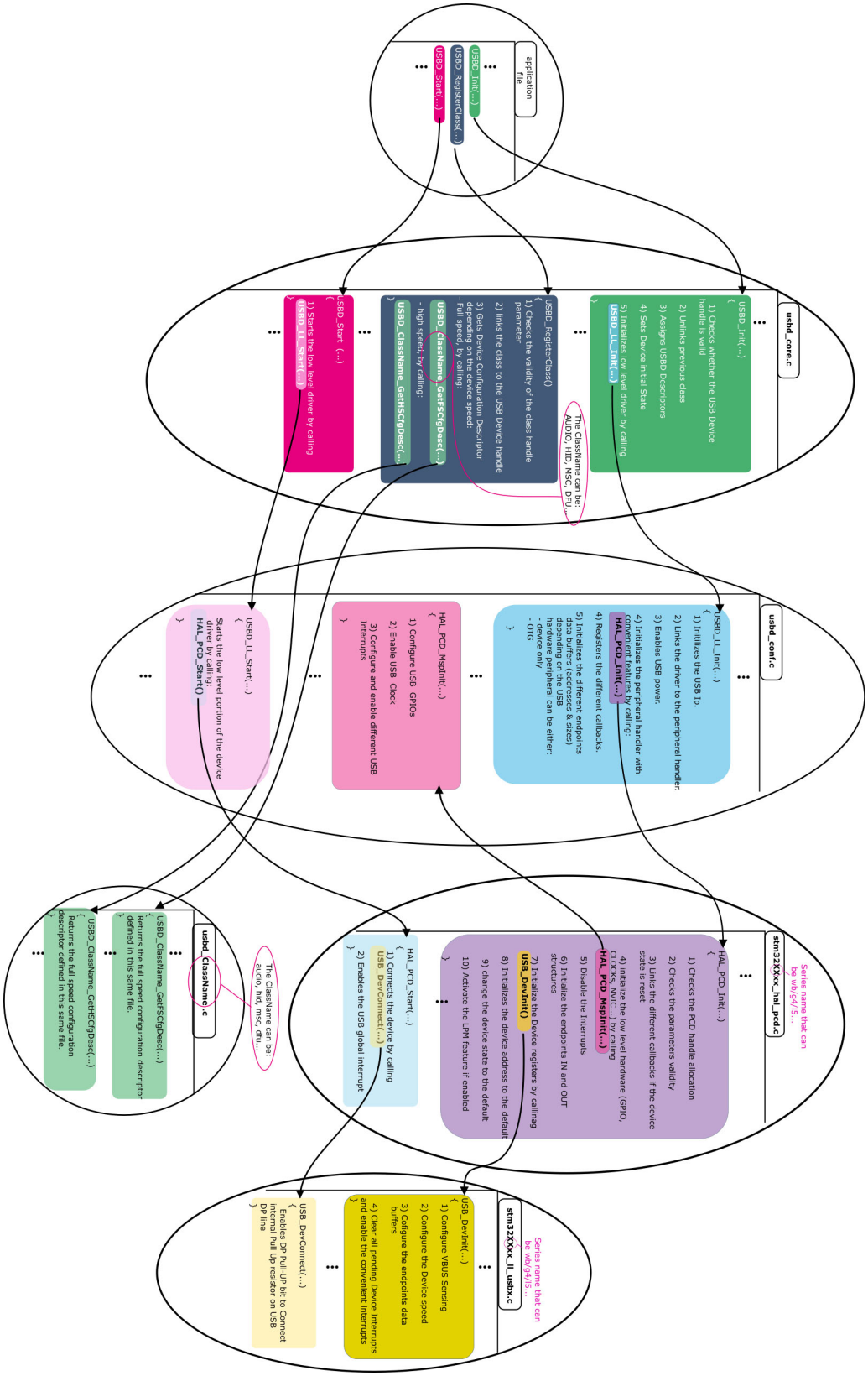
2 - The control is back in the `MX_USB_DEVICE_Init()` (**usb_device.c** file). Now the function invokes the `USBD_RegisterClass()` routine, which links the instance of the `struct USBD_ClassTypeDef` - corresponding the USB class selected - to the main `USBD_HandleTypeDef` instance. This will be used later by the USB core stack to initialize the USB class accordingly.
3 - Finally, the `MX_USB_DEVICE_Init()` (**usb_device.c** file) routine invokes the `USBD_Start()` routine:

- `USBD_Start()` (**usbd_core.c**): this function takes care of calling the Low-Level part of the stack to start the USB operations, by calling the `USBD_LL_Start()` routine.
  - `HAL_PCD_Start()`(**stm32XXxx_hal_pcd.c**): this function simply passes to control the function `USB_DevConnect()`.
    - `USB_DevConnect()`(**stm32XXxx_ll_usb.c**): this function enable the USB peripheral by enabling D+ Pull-UP bit to Connect internal resistor on USB D+ line

After these three main calls, the whole USB stack is ready to accept connections by the host. It is now the good time to connect our device to the host PC. In all nine Nucleo boards used in this text the USB signal are routed to the Morpho connector, as shown in **Figure 14**. You can use the USB Type-B connector you want. The wiring scheme shown assumes that the USB device is "self-powered". For this reason, we are not going to connect the VDD (+5V).

F1 F3

Owners of STM32F1/F3 MCU based boards need to take in account that the USB phyther does not provide internal pull-up resistor for the D+ line, which is mandatory in the USB 2.0 specification. So, it is required to add a 1.5k resistor between PA12 and +3V3.


application file
USBD_Init(...)
USBD_RegisterClass(...)
USBD_Start(...)
usbd_core.c
USBD_Init(...)
1) Checks whether the USB Device handle is valid
2) Uninks previous class
3) Assigns USBD Descriptors
4) Sets Device initial State
5) Initializes low level driver by calling USBD_LL_Init(...)
USBD_RegisterClass()
1) Checks the validity of the class handle parameter
2) links the class to the USB Device handle
3) Gets Device Configuration Descriptor depending on the device speed:
- Full speed, by calling:
USBD_ClassName_GetFSCfgDesc(...
- high speed, by calling:
USBD_ClassName_GetHSCfgDesc(...
The ClassName can be: AUDIO, HID, MSC, DFU,...
USBD_Start (...)
1) Starts the low level driver by calling USBD_LL_Start(...)
usbd_conf.c
USBD_LL_Init(...)
1) Initilizes the USB Ip.
2) Links the driver to the peripheral handler.
3) Enables USB power.
4) Initializes the peripheral handler with convenient features by calling:
HAL_PCD_Init(...)
4) Registers the different callbacks.
5) Initializes the different endpoints data buffers (addresses & sizes) depending on the USB hardware peripheral can be either:
- device only
- OTG
HAL_PCD_MspInit(...)
1) Configure USB GPIOs
2) Enable USB Clock
3) Configure and enable different USB Interrupts
USBD_LL_Start(...)
Starts the low level portion of the device driver by calling:
HAL_PCD_Start()
stm32XXxx_hal_pcd.c
Series name that can be wb/g4/l5...
HAL_PCD_Init(...)
1) Checks the PCD handle allocation
2) Checks the parameters validity
3) Links the different callbacks if the device state is reset
4) initialize the low level hardware (GPIO, CLOCKs, AVIC...) by calling
HAL_PCD_MspInit(...)
5) Disable the Interrupts
6) Initialize the endpoints IN and OUT structures
7) Initialize the Device registers by calling
USB_DevInit()
8) Initializes the device address to the default
9) change the device state to the default
10) Activate the LPM feature if enabled
HAL_PCD_Start(...)
1) Connects the device by calling
USB_DevConnect(...)
2) Enables the USB global interrupt
usbd_ClassName.c
USBD_ClassName_GetFSCfgDesc(...)
Returns the full speed configuration descriptor defined in this same file.
USBD_ClassName_GetHSCfgDesc(...)
Returns the full speed configuration descriptor defined in this same file.
The ClassName can be: audio, hid, msc, dfu,...
stm32XXxx_ll_usbx.c
Series name that can be wb/g4/l5...
USB_DevInit(...)
1) Configure VBUS Sensing
2) Configure the Device speed
3) Configure the endpoints data buffers
4) Clear all pending Device Interrupts and enable the convenient interrupts
USB_DevConnect(...)
Enables DP Pull-UP bit to Connect internal Pull Up resistor on USB DP line

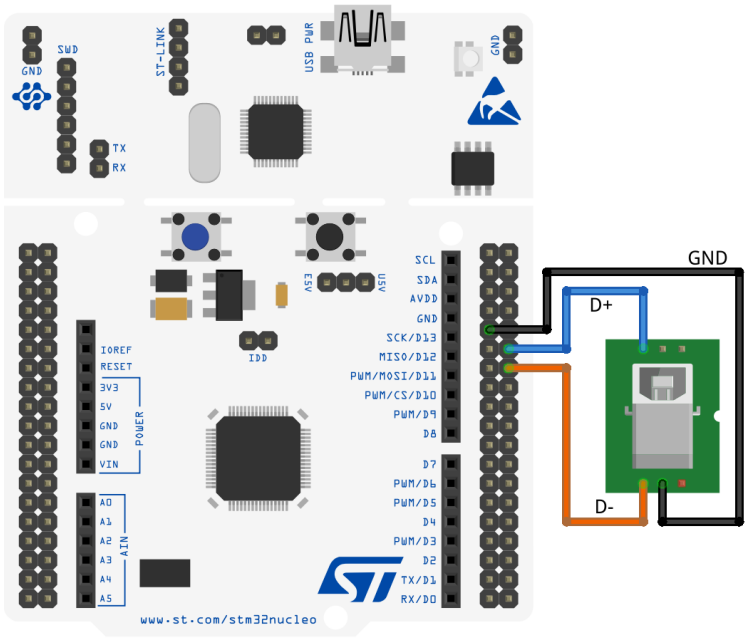


Figure 14: How to connect a USB Type-B connector to a Nucleo board

27.2.3 USB Enumeration Sequence

I will never get tired of repeating this: USB protocol tends to appear hard to master because all the operations involving its usage take place in an interrupt handler. And this because of one simple reason: all operations performed on the USB bus are always initiated by the master, which rules the bus, avoiding bus conflicts by individual devices. This implies that the device needs to be ready to process asynchronous events, to avoid that the application stalls waiting for USB events.

So, what precisely happens when we connect the USB cable into the device port? Well, the answer is not trivial, and it is mostly OS dependent. Unfortunately, details change a lot between Windows, Mac and Linux, and to be precise things can change a lot between the several releases of a given OS (especially Windows). Finally, the USB Host controller of the PC may impact on the enumeration. However, the enumeration phase can be summarized in the next steps. Refer to the USB sniffing sequence shown in **Figure 15**[13] while reading the text.

[13]Please, take note that the enumeration sequence shown in **Figure 15** is captured while connecting a USB Virtual COM Port device to a MacOS 11.x. The enumeration could slightly change in Windows.

Sp	Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Summary
	6	0:09.957.382	716 ns				<Reset> / <Chirp K> / <Tiny K>	
	7	0:09.957.382	4.58 ms				<Reset> / <Target disconnect…	
FS	8	0:09.961.970					<Full-speed>	
FS	9	0:09.964.970	147 ms				<Suspend>	
FS	10	0:10.112.313	11.0 ms				<Reset> / <Chirp J> / <Tiny J>	
FS	11	0:10.123.314					<Full-speed>	
FS	13	0:10.154.335	0 B		00	00	Set Address	Address=09
FS	24	0:10.161.015	8 B		09	00	Get Device Descriptor	Index=0 Length=8
FS	44	0:10.164.300	18 B		09	00	Get Device Descriptor	Index=0 Length=18
FS	64	0:10.164.577	2 B		09	00	Get String Descriptor	Index=2 Length=2
FS	83	0:10.164.811	44 B		09	00	Get String Descriptor	Index=2 Length=44
FS	102	0:10.165.090	2 B		09	00	Get String Descriptor	Index=1 Length=2
FS	122	0:10.165.313	38 B		09	00	Get String Descriptor	Index=1 Length=38
FS	141	0:10.165.661	2 B		09	00	Get String Descriptor	Index=3 Length=2
FS	160	0:10.165.882	26 B		09	00	Get String Descriptor	Index=3 Length=26
FS	175	0:10.170.212	9 B		09	00	Get Configuration Descriptor	Index=0 Length=9
FS	195	0:10.170.499	67 B		09	00	Get Configuration Descriptor	Index=0 Length=67
FS	212	0:10.170.926	0 B		09	00	Set Configuration	Configuration=1
FS	223	0:10.399.753	4 B		09	00	Get String Descriptor	Index=0 Length=255

Figure 15: Enumeration sequence of a VCP device connected to a MacOS 11.x

1 - *High-Speed* (HS) devices begin the enumeration process as *Full-Speed* (FS) devices. During the reset phase, HS devices and HS capable hubs begin a negotiation process to determine if they can mutually move into the HS mode. This process starts by asserting a voltage sequence on D+ and D- called "*chirping*". After this phase the device will end up in operating either in FS or HS mode.
2 - The phyther inside the USB peripheral is now ready to exchange data with the Host controller. The device needs to set its USB address and the Host starts the *setup sequence* by issuing a `SET_ADDRESS`. The USB interrupt fires and the `HAL_PCD_IRQHandler()` (**stm32XXxx_it.c**) is called. The handler detects that the message is a `SET_ADDRESS` and calls the `HAL_PCD_SetAddress()`(**stm32XXxx_hal_pcd.c**) by passing the address ID received by the host. This routine in turn calls the `USB_SetDevAddress()`(**stm32XXxx_ll_usb.c**) which finally configures the USB interface.
3 - The address is now configured, and the Host asks for the device descriptors, so that it can start understanding its capabilities. The Host starts a new *setup sequence* by issuing a `GET_DESCRIPTOR`. The USB interrupt fires and the `HAL_PCD_IRQHandler()` (**stm32XXxx_it.c**) is called. The handler detects that the message is addressed to the control endpoint (EP0) and it passes the control to the `PCD_EP_ISR_Handler()` routine.

- `PCD_EP_ISR_Handler()`(**stm32XXxx_hal_pcd.c**): the routines immediately extracts the endpoint number (0) and decodes the service endpoint interrupt. This contains the ID for the **SETUP** transaction. The routine passes so the control to the `HAL_PCD_SetupStageCallback()`(**usbd_conf.c**) routine, which in turn calls the `USBD_LL_SetupStage()` routine.
  - `USBD_LL_SetupStage()`(**usbd_core.c**): the routine analyzes the requests to detect the recipient of the request. The request is addressed to the device, and so it passes the control to the `USBD_StdDevReq()` routine.
    * `USBD_StdDevReq()`(**usbd_ctlreq.c**): the routine analyzes the content of the request and it detects that the host is asking for the device descriptors. It so passes the control to the `USBD_GetDescriptor()`(**usbd_ctlreq.c**) routine, which in turns calls the `USBD_FS_DeviceDescriptor()`(**usbd_desc.c**) that passes the pointer to the device descriptor inside the **usbd_desc.c** file. The `USBD_GetDescriptor()` is so ready to transfer the device descriptor to the host. It calls the `USBD_CtlSendData()` routine.

- `USBD_CtlSendData`(**usbd_ioreq.c**): this routine configures the EP0 so that it can transfer the content of the array containing the descriptors. The control is so transferred to the `USBD_LL_Transmit()`(**usbd_conf.c**), which in turns call the `HAL_PCD_EP_Transmit()`.
- `HAL_PCD_EP_Transmit()`(**stm32XXxx_hal_pcd.c**): this routine, after few setup instructions, passes the control to the `USB_EPStartXfer()`(**stm32XXxx_ll_usb.c**), which configure the EP0 (by writing in the PMA memory, if supported) and then sets the USB peripheral so that it starts the transfer (which starts effectively once the host will ask for the data). The **Figure 16** shows the whole call graph.



Figure 16: Complete call-graph generated upon a `GET_DESCRIPTOR` request

4 - The Host uses the information in the device descriptors to learn about the device and its abilities. This information includes the USB class and subclass, VID/PID and index of the strings containing Manufacturer, Product and Serial Number. The host so start asking for strings, by issuing several `GET_DESCRIPTOR` commands asking for given string IDs.
5 - The host issues another `GET_DESCRIPTOR` command asking for the configuration descriptor. This request not only returns the configuration descriptor, but all other descriptors associated with it, such as the interface descriptor and the endpoint descriptor. The configuration descriptors contain relevant information such as the number of interfaces, device powering mode, number and types of endpoint and so on.
6 - The Host to successfully use the device must load a device driver. The Host searches for a driver to manage communication between itself and the device. Windows uses its `.inf` files to locate a match for the devices Product ID and Vendor ID. Device release version numbers can optionally be used. If Windows cannot find a match, then it looks at the driver from a different perspective by looking for a match with any class, subclass, and protocol retrieved from the device. If a device was previously enumerated, Windows uses its registry to search for the proper driver. When a driver is identified, the Host may request additional descriptors that are specific to the device class or request that descriptors are resent.
7 - After all descriptors are received, the Host sets a specific device configuration using the `SET_CONFIGURATION` request. Most devices have only one configuration. Devices that support multiple configurations can allow the user or the driver to select the proper configuration. In this stage the firmware needs to perform all the operations related to the communication between the host and the device on the given interface's endpoints, as we will see later in this chapter.

8 - The device is now in the configured state. It took on its proprieties that were defined with the descriptors. The defined maximum power can be drawn from VBUS and the device is now ready for use in an application.



The enumeration sequence described before is just a simplification of the actual process. Too many details are hard to be reported here and, as said before, things can slightly change depending on the very specific Operating System on the Host PC. I strongly suggest to start experiencing with the ST stack by using a dedicated USB sniffer, so that it is easier to understand which types of calls are coming from the host and which answers the stack is providing. For very simple USB projects, it is ok to start using the ST stack without worrying too much of the underlying details. But if you are going to do more complex works (for example, developing a custom class not covered by the stack), then you will mostly likely be ending in buying a dedicated tool.

27.2.4 The USB CDC Class

USB-IF standardizes several USB classes that represent really common devices and applications. One of these classes is the USB *Communication Device Class* (CDC)[14].

USB *Communication Device Class* is a composite[15] USB device class that enables telecommunication devices like phones, ISDN terminal adapters, networking devices like PSTN/ADSL modems, Ethernet adapters/hubs, etc., to connect to a USB host machine. USB CDC specifies multiple models to support different types of communication devices. The most relevant ones are:

- Direct Line Control Model
- Telephone Control Model
- CAPI Control Model
- Ethernet Networking Control Model
- ATM Networking Control Model
- Abstract Control Model

The *Abstract Control Model* (ACM) is defined to support legacy modem devices and an advantage of ACM is the *serial emulation* feature (the so-called *Virtual COM Port* - VCP). Serial emulation of a USB device eases the development of host PC applications, provides software compatibility with RS-232 based legacy devices, enables USB to RS-232 conversions and gives good abstraction over the USB for application developers. Serial emulator USB ICs (like the FTDI FT232 or the Prolific PL2303) are popular in the embedded industry. By implementing the USB CDC class, we can provide our STM32 devices with a USB serial converter, without adding dedicated peripherals. Finally, CDC

[14]The standard USB CDC class specification can be found in the USB-IF website

[15]According to USB specification, a device can have multiple interfaces, and so multiple interface descriptors. A USB device with multiple interfaces that perform different functions is called a *composite* device. An example is a keyboard with an integrated smart card reader. In this case you have a single USB device with two interfaces. One interface is for the smart card reader (implementing the CCID class) and another interface is for the keyboard (implementing the HID class). Multiple interfaces can be active at the same time.

class is natively supported by Windows starting from Windows 10, without the need of a dedicated driver.

The whole CDC class implementation is contained in just one file: **Middlewares/ST/STM32_-USB_Device_Library/Class/CDC/Src/usbd_cdc.c**. Instead, the functions which handle the VCP operations are implemented in the **USB_DEVICE/App/usbd_cdc_if.c** file. In the next paragraphs we are going to explain the CDC class implementation deeply. This will give us also the possibility to explain how USB classes are handled inside the STM32 USB Device Library.

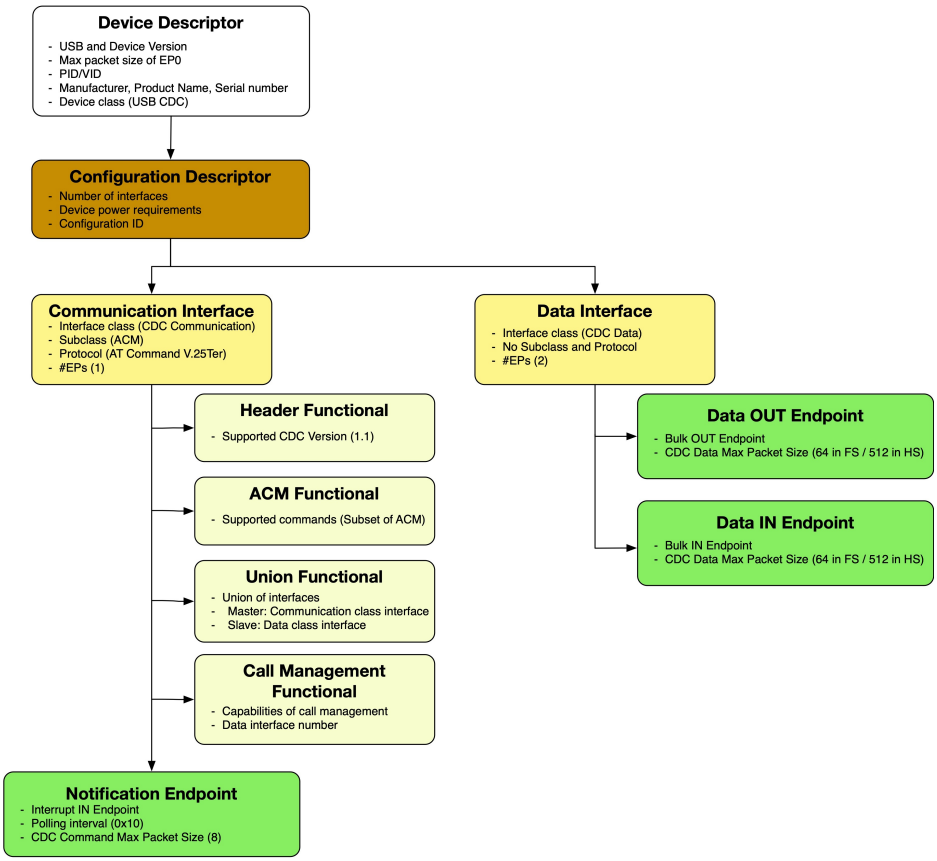


Figure 17: Descriptors hierarchy in a VCP device

27.2.4.1 USB CDC Descriptors

Before going into details of CDC descriptors, it is best to do a quick recap.
In the previous paragraph we have seen that, during the enumeration phase, the Host starts asking to the device for its configuration descriptors right after the `SET_ADDRESS` operation. Each configuration defines a set of *interfaces*, each one corresponding to a given class feature. To foster the compatibility among different devices implementing the same USB class, every interface in each USB class standardizes the number of communication endpoints and their communication behavior (*interrupt*, *bulk*, *control*, *isochronous*).

Every USB CDC device implementing the *Abstract Control Model* shall provide the descriptors

shown in **Figure 17**. Let us describe them.

- **Device Descriptor**: this descriptor, which is implemented inside the **USB_DEVICE/App/usbd_desc.c** file, provides general information regarding device capabilities. Information such as implemented class and subclass, maximum packet size on the EP0, VID/PID, device release[16] and other informative strings such as Manufacturer, Product and Serial strings. This last string is very important in VCP devices: the device serial will be used by Windows to prevent unwanted "COM-port proliferation." A device with a serial number retains its COM-port number if moved to a different USB port on a Windows PC. A device that doesn't contain a serial number gets a new port number on each attachment to a different port on a PC. In Linux and MacOS the serial number will be used to generate the corresponding **/dev/** endpoint. For example, a USB device with a serial equal to `316E355B4236` will generate the endpoint **/dev/tty.usbmodem316E355B4236**.
  The following code shows the standard CDC device descriptor generated by CubeMX.

Filename: `USB_DEVICE/App/usbd_desc.c`

```
148 __ALIGN_BEGIN uint8_t USBD_FS_DeviceDesc[USB_LEN_DEV_DESC] __ALIGN_END = {
149   0x12,                       /*bLength */
150   USB_DESC_TYPE_DEVICE,       /*bDescriptorType*/
151   0x00,                       /*bcdUSB: BCD (2.00) */
152   0x02,
153   0x02,                       /*bDeviceClass: CDC*/
154   0x02,                       /*bDeviceSubClass: ACM*/
155   0x00,                       /*bDeviceProtocol: No class specific protocol */
156   USB_MAX_EP0_SIZE,           /*bMaxPacketSize*: EP0 size = 64 bytes/
157   LOBYTE(USBD_VID),           /*idVendor*/
158   HIBYTE(USBD_VID),           /*idVendor*/
159   LOBYTE(USBD_PID_FS),        /*idProduct*/
160   HIBYTE(USBD_PID_FS),        /*idProduct*/
161   0x00,                       /*bcdDevice rel. 2.00*/
162   0x02,
163   USBD_IDX_MFC_STR,           /*Index of manufacturer string in string descriptor*/
164   USBD_IDX_PRODUCT_STR,       /*Index of product string in string descriptor */
165   USBD_IDX_SERIAL_STR,        /*Index of serial number string in string descriptor*/
166   USBD_MAX_NUM_CONFIGURATION  /*bNumConfigurations*/
167 };
```

- **Configuration Descriptor**: a typical CDC VCP device has just one configuration. The configuration descriptor specifies power requirements and the number of interfaces in the configuration. The configuration descriptor is defined inside the **Middlewares/ST/STM32_USB_Device_Library/Class/CDC/Src/usbd_cdc.c** file.

[16] This field may contain additional information particularly useful to debug devices in field. For example, it is useful to store here also the FW version, so that it is easy to keep track of given behaviors of the FW during support sessions. The same value may be useful to drive specific operations at application level.

Filename: `Middlewares/ST/STM32_USB_Device_Library/Class/CDC/Src/usbd_cdc.c`

```
__ALIGN_BEGIN uint8_t USBD_CDC_CfgFSDesc[USB_CDC_CONFIG_DESC_SIZ] __ALIGN_END = {
  /* Configuration Descriptor */
  0x09,   /* bLength: Configuration Descriptor size */
  USB_DESC_TYPE_CONFIGURATION,  /* bDescriptorType: Configuration */
  USB_CDC_CONFIG_DESC_SIZ,      /* wTotalLength:no of returned bytes: 67 bytes */
  0x00,
  0x02,   /* bNumInterfaces: 2 interface */
  0x01,   /* bConfigurationValue: Configuration value */
  0x00,   /* iConfiguration: Index of string descriptor describing the configuration */
  0xC0,   /* bmAttributes: self powered */
  0x32,   /* MaxPower 50 mA */
```

- **Interface Descriptor**: the USB CDC class defines two distinct interfaces for each configuration.
  - The first interface (shown next) is called *Communication Interface* and it is used to describe the very specific capabilities of the communication device. Generic COM-port devices should specify the compatibility with the V.25ter protocol even if they don't use AT commands. The communication interface has four class-specific descriptors plus an endpoint descriptor. The *Header Functional* descriptor specifies the version of the CDC specification (v1.1). The *Abstract Control Model* descriptor specifies what class-specific requests and notifications the device supports. The *Union Functional* descriptor identifies the interfaces that belong to the CDC function, which are typically the communication interface plus the data interface. The *Call Management Functional* descriptor tells how the device manages calls. Because a generic COM-port device has no calls to handle, the descriptor says the device doesn't handle call management. An interrupt endpoint (**OUT** endpoint with address 0x82) sends status notifications to the host. The endpoint descriptor provides the endpoint's number, direction, and maximum packet size (8 bytes). In the current implementation of the CDC VCP interface provided by ST this endpoint is not used at all, and usually no data transit over it with all major OSes.

Filename: `Middlewares/ST/STM32_USB_Device_Library/Class/CDC/Src/usbd_cdc.c`

```
  /*Interface Descriptor */
  0x09,   /* bLength: Interface Descriptor size */
  USB_DESC_TYPE_INTERFACE,  /* bDescriptorType: Interface */
  /* Interface descriptor type */
  0x00,   /* bInterfaceNumber: Id of the Interface */
  0x00,   /* bAlternateSetting: Alternate setting */
  0x01,   /* bNumEndpoints: One endpoints used */
  0x02,   /* bInterfaceClass: Communication Interface Class */
  0x02,   /* bInterfaceSubClass: Abstract Control Model */
  0x01,   /* bInterfaceProtocol: Common AT commands */
  0x00,   /* iInterface: */
```

```
294
295   /*Header Functional Descriptor*/
296   0x05,   /* bLength: Endpoint Descriptor size */
297   0x24,   /* bDescriptorType: CS_INTERFACE */
298   0x00,   /* bDescriptorSubtype: Header Func Desc */
299   0x10,   /* bcdCDC: spec release number */
300   0x01,
301
302   /*Call Management Functional Descriptor*/
303   0x05,   /* bFunctionLength */
304   0x24,   /* bDescriptorType: CS_INTERFACE */
305   0x01,   /* bDescriptorSubtype: Call Management Func Desc */
306   0x00,   /* bmCapabilities: D0+D1 */
307   0x01,   /* bDataInterface: 1 */
308
309   /*ACM Functional Descriptor*/
310   0x04,   /* bFunctionLength */
311   0x24,   /* bDescriptorType: CS_INTERFACE */
312   0x02,   /* bDescriptorSubtype: Abstract Control Management desc */
313   0x02,   /* bmCapabilities */
314
315   /*Union Functional Descriptor*/
316   0x05,   /* bFunctionLength */
317   0x24,   /* bDescriptorType: CS_INTERFACE */
318   0x06,   /* bDescriptorSubtype: Union func desc */
319   0x00,   /* bMasterInterface: Communication class interface */
320   0x01,   /* bSlaveInterface0: Data Class Interface */
321
322   /*Endpoint 2 Descriptor*/
323   0x07,                         /* bLength: Endpoint Descriptor size */
324   USB_DESC_TYPE_ENDPOINT,       /* bDescriptorType: Endpoint */
325   CDC_CMD_EP,                   /* bEndpointAddress */
326   0x03,                         /* bmAttributes: Interrupt */
327   LOBYTE(CDC_CMD_PACKET_SIZE), /* wMaxPacketSize: 8 bytes */
328   HIBYTE(CDC_CMD_PACKET_SIZE),
329   CDC_FS_BINTERVAL,             /* bInterval: 10ms*/
```

- – The second interface is called *Data Interface* and it is responsible for sending and receiving the COM-port data. The interface descriptor (shown next) tells the host the interface has two bulk endpoints, one for each direction (0x1 is for Bulk **OUT** and 0x81 is Bulk **IN**). Each endpoint has an endpoint descriptor.

Filename: `Middlewares/ST/STM32_USB_Device_Library/Class/CDC/Src/usbd_cdc.c`

```
332   /*Data class interface descriptor*/
333   0x09,   /* bLength: Endpoint Descriptor size */
334   USB_DESC_TYPE_INTERFACE,  /* bDescriptorType: */
335   0x01,   /* bInterfaceNumber: Id of the Interface */
336   0x00,   /* bAlternateSetting: Alternate setting */
337   0x02,   /* bNumEndpoints: Two endpoints used */
338   0x0A,   /* bInterfaceClass: CDC Data */
339   0x00,   /* bInterfaceSubClass: */
340   0x00,   /* bInterfaceProtocol: */
341   0x00,   /* iInterface: */
342
343   /*Endpoint OUT Descriptor*/
344   0x07,   /* bLength: Endpoint Descriptor size */
345   USB_DESC_TYPE_ENDPOINT,      /* bDescriptorType: Endpoint */
346   CDC_OUT_EP,                        /* bEndpointAddress */
347   0x02,                              /* bmAttributes: Bulk */
348   LOBYTE(CDC_DATA_FS_MAX_PACKET_SIZE),  /* wMaxPacketSize: 64 for FS, 512 for HS*/
349   HIBYTE(CDC_DATA_FS_MAX_PACKET_SIZE),
350   0x00,                              /* bInterval: ignore for Bulk transfer */
351
352   /*Endpoint IN Descriptor*/
353   0x07,   /* bLength: Endpoint Descriptor size */
354   USB_DESC_TYPE_ENDPOINT,      /* bDescriptorType: Endpoint */
355   CDC_IN_EP,                         /* bEndpointAddress */
356   0x02,                              /* bmAttributes: Bulk */
357   LOBYTE(CDC_DATA_FS_MAX_PACKET_SIZE),  /* wMaxPacketSize: 64 for FS, 512 for HS */
358   HIBYTE(CDC_DATA_FS_MAX_PACKET_SIZE),
359   0x00                               /* bInterval: ignore for Bulk transfer */
360 } ;
```

27.2.4.2 USB CDC Class Initialization

Again, before going into additional details, let us do a quick recap.

1. The device is connected to the host. After detecting the USB device speed mode (HS/FS), the host performs a `SET_ADDRESS` command by assigning a USB unique address on the bus.
2. The host then asks to the device for its **Device Descriptors** by issuing one or more `GET_DESCRIPTOR` commands.
3. The host asks in this stage - or in a later one - for string descriptors by issuing several `GET_DESCRIPTOR` commands.
4. The host next asks for device's **Configuration Descriptors** by issuing one or more `GET_DESCRIPTOR` commands. The device will provide the configuration descriptors and all **Interface Descriptors** implemented.

5. The host so choose the wanted configuration and performs a `SET_CONFIGURATION` command.

Up to the **step 5**, all the messages between the host and the device are exchanged on the EP0. You can clearly see this in **Figure 15**. So, the device needs to configure the other endpoints (EP2 for control interrupt and EP1 for IN/OUT) used by CDC drivers. This operation is performed by the `USBD_CDC_Init()`(**usbd_cdc.c**) function, which is invoked during the `SET_CONFIGURATION` command.

1 - The host issues a `SET_CONFIGURATION` command by passing `1` as the index of the wanted configuration (remember that our device provides just one configuration). The command is sent to the EP0 and so a new USB interrupt fires.
2 - The stack detects that the message is addressed on the EP0, the control interrupt, and the `HAL_PCD_SetupStageCallback()` (**usbd_conf.c**) is called. The control is then transferred to the `USBD_LL_SetupStage()` (**usbd_core.c**), which detects that the call is a standard setup request and calls the `USBD_StdDevReq()` routine.
3 - The `USBD_StdDevReq()`(**usbd_ctlreq.c**) detects that the command type is a `SET_CONFIGURATION` and passes the control to the `USBD_SetConfig`(**usbd_ctlreq.c**) routine. This in turn calls the `USBD_SetClassConfig()`(**usbd_core.c**). This other routine simply calls the initialization routine of the CDC class (`USBD_CDC_Init()`). If you give a look to the source code, you will see that the code is designed to be abstracted from the actual USB class used. This is possible thanks to the fact that the instance of the `struct USBD_ClassTypeDef` (which corresponds to `USBD_CDC` (**usbd_cdc.c**)) is linked to the USB handler `USBD_HandleTypeDef` by the call to the `USBD_RegisterClass()` inside the `MX_USB_DEVICE_Init()` (**usb_device.c**) routine.
4 - The function `USBD_CDC_Init()`(**usbd_cdc.c**) is so invoked, and the EP1 and EP2 are properly initialized. The routine also sets the USB peripheral so that the EP1 is ready to accept data by the host by calling the `USBD_LL_PrepareReceive()`.
5 - Finally, the CDC Interface is initialized inside the `USBD_CDC_Init()` routine by invoking the `CDC_Init_FS()`(**usbd_cdc_if.c**) routine, which simply configures the global TX and RX buffers used to exchange data over the serial interface.

27.2.4.3 USB CDC Class Operations

We are close to see a complete and practical example. But we need to understand just few more things. Again, a recap may be gold in this phase.

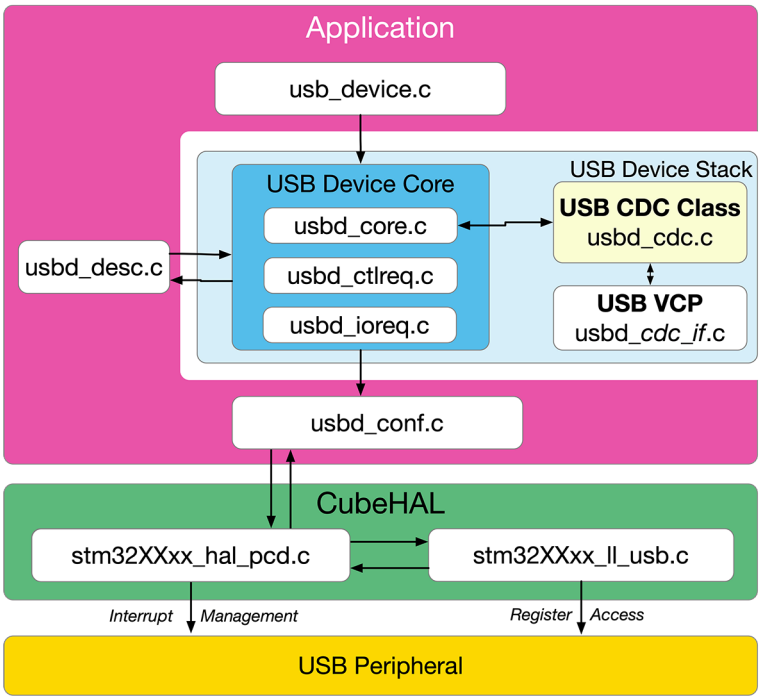


Figure 18: Typical architecture of a USB-CDC application

The CDC is the standard USB class for the communication devices exchanging data over a USB connection. The CDC is a generic specification, which defines several communication models specific to the real device. The *Abstract Control Model* is the model for serial interfaces, the so-called *Virtual COM Port*. The STM32 USB Device library is designed so that the CDC implementation (which is entirely contained in the **usbd_cdc.c** file) is abstract from the actual model (called *interface* in the library). The file **USB_DEVICE/App/usbd_cdc_if.c** file contains the implementation for the VCP model (see **Figure 18**).

This interface file contains just five routines:

- `CDC_Init_FS`: this function is called by the USB CDC class initializer during a `SET_CONFIGURATION` command (issued by the host).
- `CDC_DeInit_FS`: this function is called the USB CDC class de-initializer when the USB interface issues a reset operation (mostly because the USB device is disconnected from the host).
- `CDC_Control_FS`: this function handles all the specific CDC commands issued by the host driver or the application (in our case, a terminal emulator program connecting to the serial port). More about this function in a while.
- `CDC_Receive_FS`: this function is called by the USB stack once new data arrives on the endpoint EP1, that is once the data is exchanged over the serial port. Please, take note that you cannot use this routine in polling mode. This routine is a callback that will be called once the data is inside the RX buffer. Also take note that, once the data arrives on the EP1, `CDC_Receive_FS()` must call the `USBD_CDC_ReceivePacket()` routine that configures EP1 in receive mode again, so that it is ready to accept new data by the host.

- `CDC_Transmit_FS`: this function sets the transmit buffer and invokes the `USBD_CDC_TransmitPacket` that will configure the EP1 in transmit mode (remember that data will be "asked" by the host in an **IN** transaction).

Offset	Field	Size (Bytes)	Description
0	**dwDTERate**	4	Data terminal rate, in bits per second
4	**bCharFormat**	1	Stop bits 0 - 1 Stop bit 1 - 1.5 Stop bits 2 - 2 Stop bits
5	**bParityType**	1	Parity 0 - None 1 - Odd 2 - Even 3 - Mark 4 - Space
6	**bDataBits**	1	Number Data bits (5, 6, 7, 8 or 16).

Table 16: Structure of the `CDC_SET_LINE_CODING` request

By looking to the content of the `CDC_Control_FS`, you can see that it contains a `switch` block to process the CDC-related *Class Requests*[17] coming from the host (and the application level). None of the `case` statements contains an implementation. To proper establish a serial communication, it is mandatory to implement at least the `CDC_SET_LINE_CODING` and `CDC_GET_LINE_CODING` requests, which are used to configure the serial line communication parameters (baud rate, stop bits, parity, data size). The serial protocol requires that, once a `CDC_SET_LINE_CODING` is issued, the `CDC_GET_LINE_CODING` must return the same configuration values. The configuration values received by the `CDC_SET_LINE_CODING` request adhere to a well-defined structure, which is shown in **Table 16**. The STM32 USB Device Library already defines the struct `USBD_CDC_LineCodingTypeDef` used to store the configuration values.

The following code shows how to properly implement the line coding configuration. Moreover, it shows a modification to the `CDC_Receive_FS()` function (at line 239) so that every byte received on the serial port is echoed back to the host: this will allows us to see what is written on the terminal emulator. The complete source code is available in the book samples. Please also consider that all those `printf()` will allow you to see debug messages on the Nucleo VCP. This will clarify the way the VCP is configured.

[17] These requests represent just a fraction of the complete list of *Class Requests* defined by the CDC class. For more information, please refer to official CDC specification.

Filename: `USB_DEVICE/App/usbd_cdc_if.c`

```
179 static int8_t CDC_Control_FS(uint8_t cmd, uint8_t* pbuf, uint16_t length) {
180   switch(cmd) {
181     case CDC_SEND_ENCAPSULATED_COMMAND:
182       printf("Received CDC_SEND_ENCAPSULATED_COMMAND: %d\r\n", length);
183       break;
184
185     case CDC_GET_ENCAPSULATED_RESPONSE:
186       printf("Received CDC_GET_ENCAPSULATED_RESPONSE\r\n");
187       break;
188
189     case CDC_SET_COMM_FEATURE:
190       printf("Received CDC_SET_COMM_FEATURE\r\n");
191       break;
192
193     case CDC_GET_COMM_FEATURE:
194       printf("Received CDC_GET_COMM_FEATURE\r\n");
195       break;
196
197     case CDC_CLEAR_COMM_FEATURE:
198       printf("Received CDC_CLEAR_COMM_FEATURE\r\n");
199       break;
200
201     case CDC_SET_LINE_CODING:
202       lineCoding.bitrate = (uint32_t) (pbuf[0] | (pbuf[1] << 8)
203                           | (pbuf[2] << 16) | (pbuf[3] << 24));
204       lineCoding.format = pbuf[4];
205       lineCoding.paritytype = pbuf[5];
206       lineCoding.datatype = pbuf[6];
207       printf("Received CDC_SET_LINE_CODING: \r\n\tBaudrate: %lu\r\n\tCharFormat: %d\r\n\t
208               Parity: %d\r\n\tDataBits: %d\r\n", lineCoding.bitrate, lineCoding.format,
209               lineCoding.paritytype, lineCoding.datatype);
210       break;
211
212     case CDC_GET_LINE_CODING:
213       printf("Received CDC_GET_LINE_CODING\r\n");
214       pbuf[0] = (uint8_t) (lineCoding.bitrate);
215       pbuf[1] = (uint8_t) (lineCoding.bitrate >> 8);
216       pbuf[2] = (uint8_t) (lineCoding.bitrate >> 16);
217       pbuf[3] = (uint8_t) (lineCoding.bitrate >> 24);
218       pbuf[4] = lineCoding.format;
219       pbuf[5] = lineCoding.paritytype;
220       pbuf[6] = lineCoding.datatype;
221       break;
222
223     case CDC_SET_CONTROL_LINE_STATE:
224       printf("Received CDC_SET_CONTROL_LINE_STATE: %d\r\n", length);
```

```
225       break;
226
227     case CDC_SEND_BREAK:
228       printf("Received CDC_SEND_BREAK\r\n");
229       break;
230
231     default:
232       break;
233   }
234   return (USBD_OK);
235 }
236
237 static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len) {
238   CDC_Transmit_FS(Buf, *Len);
239   USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
240   USBD_CDC_ReceivePacket(&hUsbDeviceFS);
241   return (USBD_OK);
242 }
```

When the `CDC_SET_LINE_CODING` request is issued? And, more important, what is the calling sequence that generates the request? Again, the USB protocol analyzer can help us in understanding what under the hood. The **Figure 19** shows the request originated by the terminal emulator program during the connection to the COM port. As you can see, a **SETUP** transaction is issued over the EP0, with a direction **Host-to-Device** (this is different from a typical **SETUP** transaction during the enumeration phase, when the direction is **Device-to-Host** because the device needs to send to the host its descriptors). The recipient of the transaction is the *interface* (so, this is a *Standard Interface Request*) and the request is addressed to the CDC class. The data contained in the transaction corresponds to the `CDC_SET_LINE_CODING` request.

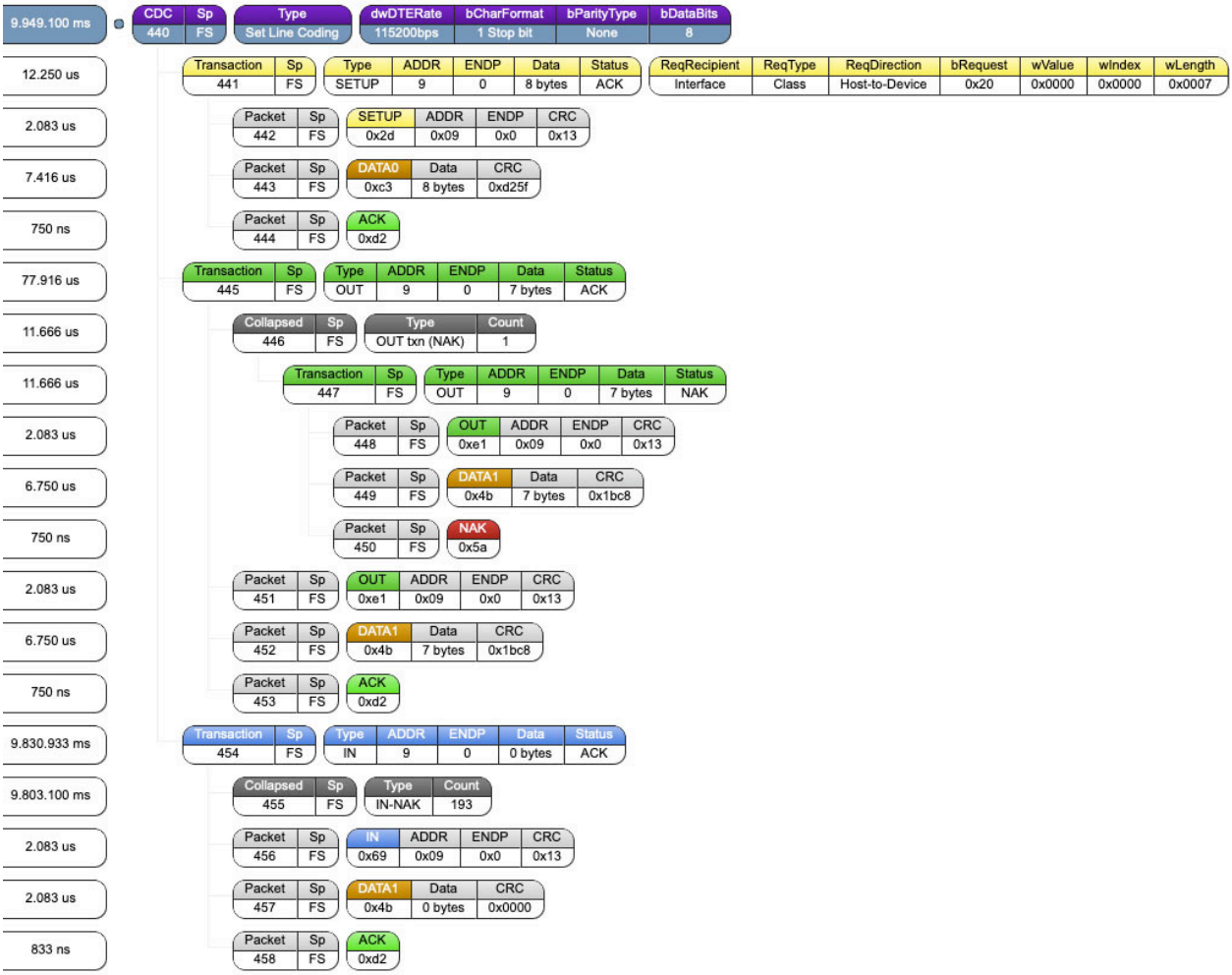


Figure 19: Transaction sequence originated by a `CDC_SET_LINE_CODING` request

27.3 Building Custom USB Devices

So far, we have seen that USB-IF standardizes several USB classes for given families of devices. Mice, keyboards, external USB disks are just few examples of standard and popular USB devices. However, chances are that, as firmware/hardware developer, you are going to design a USB device that falls outside the above categories. If this the case, you have to know that there are several options available with pro and cons.

The main issue to take in account while designing a vendor-specific device is that you need custom drivers for your device. And this is a relevant question mostly for Windows OSes. To design a device driver using the *Windows Driver Kit* (WDK) is not a trivial task. Moreover, you will need to sign the drivers with a *Code Signing Certificate* (and this can cost up to USD 500) and go through the WHQL certification (consider something between USD 2500 and USD 5000). Not that easy and not that affordable even for some companies in terms of complexity.

Depending on the degree of complexity and flexibility, these are the options to consider when designing a new USB device.

- **To use a standardized USB class**: well, this is still the best option to prefer when designing a new device. All recent versions of the three major OSes (Windows, MacOS and Linux) provide standard drivers for the HID and the CDC classes. If your device is suitable to exchange messages by using a serial communication, then a *Virtual COM Port* (VCP) is a perfect solution. This is the main reason why USB to serial converters are still so popular in the electronics industry. The only relevant drawback in using the CDC class is the maximum transmission speed, which is limited to ~80KB/s in USB FS devices. The HID class is another good alternative, since even the oldest Windows releases provide support to this USB class.
- **To design a WCID device**: a *Windows Compatible ID* (WCID) device is a USB device that provides additional information to a Windows system, to facilitate automated driver installation. This extra information is simply exchanged with the host through custom string descriptors. WCID allows a device to be used by a Windows application almost as soon as it is plugged in, as opposed to the usual scenario where a USB device that is neither HID nor CDC requires end-users to perform a manual driver installation. WCID is an extension of the *WinUSB Device* functionality, introduced by Microsoft with Windows 8. The purpose of a WinUSB device is to enable Windows to load `Winusb.sys` as the device's function driver without a custom INF file. For a WinUSB device, you are not required to distribute INF files for your device, making the driver installation process simple for end users. Conversely, if you need to provide a custom INF, you should not define your device as a WinUSB device and you should not specify the hardware ID of the device in the INF. Another advantage of this approach is that you do not need to undergo *Windows Hardware Quality Labs* (WHQL) testing for driver signing.
- **To use `libusb`**: `libusb` is an open-source project that provides generic access to USB devices. It is intended to be used by developers to facilitate the production of applications that communicate with USB hardware. `libusb` is a cross-platform library: it works under all Windows releases, Linux and MacOS. `libusb` also includes a Windows specific `libusb-win32` library, which allows user-space applications to access USB devices on Windows OS. This library can be used as a default driver for your device, and you can install it in two ways: by using the Zadig tool[18] by Pete Batard or the companion library libwdi (Windows Driver Installation library) by the same author. Pete also provides an excellent tutorial to build WCID devices[19] and some examples for the STM32 USB Device library. Finally, consider that in MacOS and Linux the usage of `libusb` is all you need to interface a vendor-specific device.

Table 17 summarizes the main features of the four available options when developing a new USB device. According to this author, unless speed is a strict requirement for your application, to design a custom HID device is still the best option. Moreover, for the STM32 families, ST provides a custom HID class that can be easily customized at your needs. Let us analyze it.

[18] https://zadig.akeo.ie/
[19] https://github.com/pbatard/libwdi/wiki/WCID-Devices

Feature	HID	CDC	WCID/WinUSB	libusb
Driver Support in Windows	Yes	Yes (from 10)	Yes (from 8)	Yes (with Zadig/libwdi)
Driver Support in MacOS / Linux	Yes	Yes	No	Yes
Custom installer for driver needed	No	No	No	Yes
Support for 64-Bit	Yes	Yes	Yes	Yes
Support for Control Transfers	Yes	No	Yes	Yes
Support for Interrupt Transfer	Yes	No	Yes	Yes
Support for Bulk Transfers	No	Yes	Yes	Yes
Support for Isochronous Transfers	No	Yes	No	Yes
Maximum Speed (Full-Speed)	~64 KB/s	~80 KB/s	~1 MB/s	~1 MB/s

Table 17: Comparison of the USB Device Class Driver features

27.3.1 The USB HID Class

The *Human Interface Device* (HID) is the USB class dedicated to peripherals usually used by humans, such as keyboards, mice, gamepads and so on. However, as we are going to see in a while, the HID specification is sufficiently generic so that virtually every device can be implemented with this specification, until transmission speed is not a strict requirement. The complete HID class specification is available on the official USB-IF website[20]. A HID device is essentially made of one interface and two data endpoints (in addition to the standard control endpoint EP0), even if several HID devices just implement the IN data endpoint (for example, simple USB mice just send position information through the data IN endpoint).

The HID protocol makes implementation of devices quite easy. Devices define the structure of their data packets (called *reports* in the HID protocol terminology) and then present a *Report Descriptor* to the host. The *Report descriptor* is a hard coded array of bytes that describes the device's data packets, called *Usage*. This includes how many packets the device supports, the size of the packets, and the purpose of each byte and bit in the packet, the minimum and maximum value an individual byte or bit can assume. For example, a mouse can tell the host that the left button's pressed/released state is stored as the 3rd bit in the first byte in data packet, and it can assume the logical values 0/1 to denote the button is released/pressed (note: these locations are only illustrative and are device-specific). The device typically stores the *Report Descriptor* in ROM and does not need to intrinsically understand or parse it.

Since the structure of the data packets can be shared between totally different devices (and so, the meaning of the data generates different behaviors in the host), the usage of individual groups of data is part of more generic *Usage Pages*. Every *Usage Page* has a standardized *Usage Page ID* and individual *Usages* have a standardized *Usage ID* in that *Usage Page*.
For example, the *Usage Page* **Generic Desktop Page** has a *Usage Page ID* equal to **0x01**, and the *Usage ID* **0x02** corresponds to the **Mouse** *Usage*, while the *Usage ID* **0x06** corresponds to the

[20] https://bit.ly/3qs2NV0

Keyboard *Usage*. When thinking to *Usage Pages* and *Usages* think them in term of *namespaces*: changing the *Usage Page* affects what *Usages* are available. The USB-IF standardizes many *Usage Pages* and related *Usages*. See the official documentation[21] for more information.

This mechanism allows devices to be self-descriptive and in the same way allows OSes to have just one driver for all HID devices: a powerful parser in the OS driver allows to "understand" the structure of the messages sent by actual devices, allowing manufacturers to build their new stylish and full-of-buttons mice without the need of dedicated drivers.

However, I am sure that you are totally puzzled by all that stuff. This is normal when dealing with USB and its standard classes. Don't worry: things will become clear soon.

27.3.1.1 USB HID Descriptors

Every USB HID device shall provide the descriptors shown in **Figure 19**. Let us describe them.

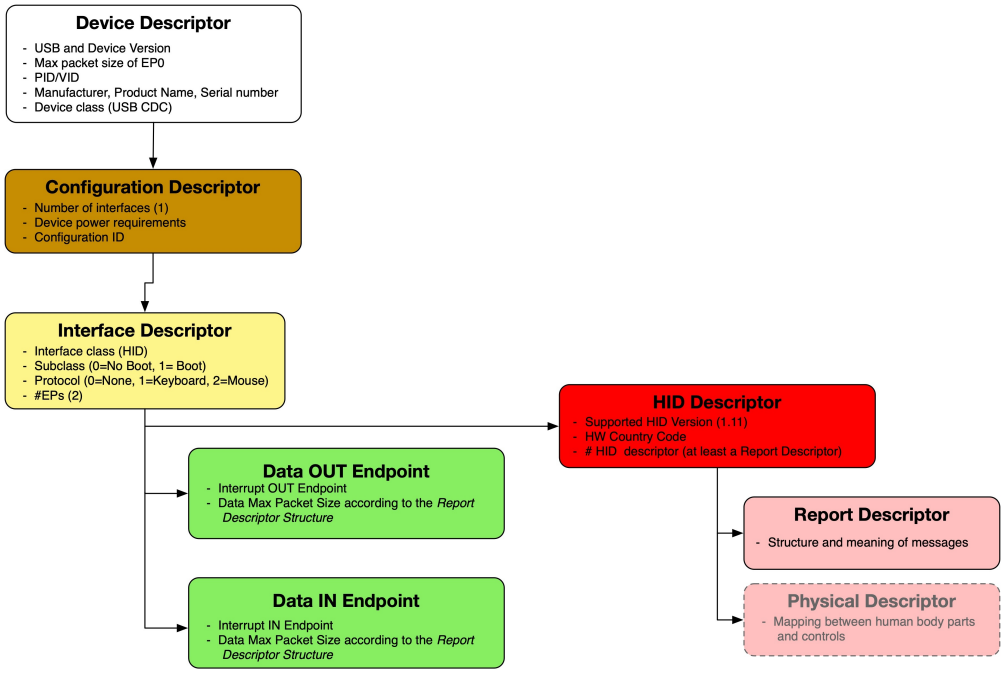


Figure 20: Descriptors hierarchy in a USB HID device

- **Interface Descriptor**: the HID class defines just one interface descriptor to announce to the host the device implements the HID protocol.
  The fields `bInterfaceSubClass` can assume the values `0=noboot` and `1=boot`: the value `1` indicates that the device adheres to the BOOT protocol. Because the parser for the *Report Descriptor* (shown next) represents a significant amount of code, a simpler method is needed to identify the device protocol for devices requiring BIOS support (BOOT devices). HID class devices use the `bInterfaceSubClass` field to indicate devices that support a predefined protocol

[21] https://bit.ly/3qw9dCj

for either mouse devices or keyboards (that is, the device can be used as a BOOT device). The boot protocol can be extended to include additional data not recognized by the BIOS, or the device may support a second preferred protocol for use by the HID class driver.
The field `nInterfaceProtocol` can assume the values `0=none`, `1=keyboard`, `2=mouse`. When this field assumes the values 1 or 2, the *Report Descriptor* shall adhere to a well-defined structure.

Offset	Field	Size (Bytes)	Description
0	**bLength**	1	Length of this descriptor = 9 bytes
1	**bDescriptorType**	1	Descriptor type = HID DESCRIPTOR (0x21)
2	**bcdHID**	2	Numeric expression identifying the HID Class Specification release (1.11)
4	**bCountryCode**	1	Numeric expression identifying country code of the localized hardware
5	**bNumDescriptors**	1	Numeric expression specifying the number of class descriptors (always at least one i.e. Report descriptor.)
6	**bDescriptorType**	1	Constant name identifying type of Report Descriptor (0x22)
7	**[bDescriptorType]...**	1	Constant name identifying type of optional Physical Descriptor (0x23)
8	**[wDescriptorLength]...**	1	Numeric expression that is the total size of the optional Physical Descriptor.

Table 18: Structure of a USB HID descriptor

- **HID Descriptor**: the HID descriptor provides information about the HID class specification release and identifies the length and type of subordinate descriptors for a device. The structure of the HID descriptor is shown in **Table 18**.
  The most relevant subordinate descriptor is the *Report Descriptor*. The *Report descriptor* is unlike other descriptors in that it is not simply a table of values. The length and content of a Report descriptor vary depending on the number of data fields required for the device's report or reports. The *Report Descriptor* is made up of items that provide information about the device. The content of the *Report Descriptor* will be used by the OS HID device driver to establish data exchange with the HID device. A bad formatted *Report Descriptor* will cause that no information flow between the device and the host or it is misinterpreted. We are going to provide additional information about *Report Descriptor* in the next paragraph.
  The *Physical Descriptor* is a data structure that provides information about the specific part or parts of the human body that are activating a control or controls. For example, a physical descriptor might indicate that the right-hand thumb is used to activate button 5. An application can use this information to assign functionality to the controls of a device. *Physical Descriptors* are entirely optional. They add complexity and offer very little in return for most devices.
- **Interrupt Endpoints**: a HID device communicates with the HID class driver using either the *Control* (default) EP0 endpoint or two *Interrupt* endpoints (called Data IN/OUT in **Figure 20**). The *Control* endpoint is used for receiving and responding to requests for USB control and class data, for transmitting data when polled by the HID class driver (using the `GET_REPORT` request) and for receiving data from the host. The two *Interrupt* endpoints are used for

receiving asynchronous (unrequested) data from the device and transmitting low latency data to the device. The *Interrupt* OUT endpoint is optional: if a device defines an *Interrupt* OUT endpoint, then output reports are transmitted by the host to the device through the Interrupt OUT endpoint. If no *Interrupt* OUT endpoint is declared then output reports are transmitted to a device through the *Control* EP0 endpoint, using `SET_REPORT` requests.

27.3.1.2 Overview of the *Report Descriptor*

When dealing with the HID protocol, the terminology itself can be a source of confusion. In the HID protocol, data packets sent over *Interrupt* IN and OUT endpoints are called *reports*. A *report* is a structured piece of information (for example, the coordinate of the mouse pointer or a keyboard's key) well described by the *Report Descriptor*. The mechanism around the report descriptor allows to have one generic driver in the host OS able to deal with a multitude of devices, each one with its features and its way to send structured data. Clearly, standard devices like keyboards and mice shall adhere to a fixed *Usage* and *Usage Pages*, otherwise the OS will not be able to use them without a dedicated driver[22].

The *Report Descriptor* does not have a fixed structure. It is just a sequence of bytes describing the structure of a report (a message). It is up to the specific *Usage* and *Usage Page* to give a meaning. Let us study a practical example. Here, we are building a mouse and we need to describe the structure of the message. The following C `struct` does the job:

```
struct mouse_report_t {
    uint8_t buttons;
    int8_t x;
    int8_t y;
}
```

The `x` and `y` fields correspond to the movement of the mouse on the X and Y axes. They are signed integers that take one byte, so their value goes between `-127` and `+127` (well, to be precise the range goes between `-128` and `+127`, but it is ok to keep things simple). The `buttons` field contains information regarding the three buttons of the mouse: just the first three nibbles in the byte are used. If the bit is set to `1`, then the button is pressed, otherwise it is released. Refer to **Table 19** for the complete interpretation of the message.

[22]Manufactures of modern and advanced HID devices use a dual mechanism to implement advanced features like gestures or to configure some device buttons. In this case, the device operates in a dual mode. By default, the device sends standard HID reports so that it works like any regular mouse until a dedicated driver is installed. Thanks to this driver, the device will operate in advanced mode. This is the case, for example, of the Logitech HID++ protocol.

Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
`buttons`	Unused	Unused	Unused	Unused	Unused	Left button {0,1}	Middle button {0,1}	Right button {0,1}
`x`	X Axis Relative Movement as Signed Integer {-127, +127}							
`y`	Y Axis Relative Movement as Signed Integer {-127, +127}							

Table 19: The structure of the mouse's message

How does that `struct` translate to a report descriptor? Well, the first step is to describe the buttons, three of them:

```
USAGE_PAGE (Button)
USAGE_MINIMUM (Button 1)
USAGE_MAXIMUM (Button 3)
```

Each button status is represented by a bit, 0 or 1:

```
LOGICAL_MINIMUM (0)
LOGICAL_MAXIMUM (1)
```

There are three of these bits:

```
REPORT_COUNT (3)
REPORT_SIZE (1)
```

This variable data is an input to the computer:

```
INPUT (Data,Var,Abs)
```

What about the five useless padding bits? We need to declare them as constant data:

```
REPORT_COUNT (1)
REPORT_SIZE (5)
INPUT (Cnst,Var,Abs)
```

And this is the result:

```
USAGE_PAGE (Button)
USAGE_MINIMUM (Button 1)
USAGE_MAXIMUM (Button 3)
LOGICAL_MINIMUM (0)
LOGICAL_MAXIMUM (1)
REPORT_COUNT (3)
REPORT_SIZE (1)
INPUT (Data,Var,Abs)
REPORT_COUNT (1)
REPORT_SIZE (5)
INPUT (Cnst,Var,Abs)
```

Now we have to define the X and Y axes movement:

```
USAGE_PAGE (Generic Desktop)
USAGE (X)
USAGE (Y)
```

We want it to be a signed integer that takes one byte, so it has a value between -127 and +127:

```
LOGICAL_MINIMUM (-127)
LOGICAL_MAXIMUM (127)
REPORT_SIZE (8)
REPORT_COUNT (1)
```

And send it to the computer as a variable relative coordinate:

```
INPUT (Data,Var,Rel)
```

And this is the result:

```
USAGE_PAGE (Button)
USAGE_MINIMUM (Button 1)
USAGE_MAXIMUM (Button 3)
LOGICAL_MINIMUM (0)
LOGICAL_MAXIMUM (1)
REPORT_COUNT (3)
REPORT_SIZE (1)
INPUT (Data,Var,Abs)
REPORT_COUNT (1)
REPORT_SIZE (5)
INPUT (Cnst,Var,Abs)
USAGE_PAGE (Generic Desktop)
USAGE (X)
USAGE (Y)
```

```
LOGICAL_MINIMUM (-127)
LOGICAL_MAXIMUM (127)
REPORT_SIZE (8)
REPORT_COUNT (2)
INPUT (Data,Var,Rel)
```

The above report descriptor is still not sufficient to make the OS to recognize the mouse movement. The HID specification requires a well-defined preamble:

```
USAGE_PAGE (Generic Desktop)
USAGE (Mouse)
COLLECTION (Application)
    USAGE (Pointer)
    COLLECTION (Physical)
        USAGE_PAGE (Button)
        USAGE_MINIMUM (Button 1)
        USAGE_MAXIMUM (Button 3)
        LOGICAL_MINIMUM (0)
        LOGICAL_MAXIMUM (1)
        REPORT_COUNT (3)
        REPORT_SIZE (1)
        INPUT (Data,Var,Abs)
        REPORT_COUNT (1)
        REPORT_SIZE (5)
        INPUT (Cnst,Var,Abs)
        USAGE_PAGE (Generic Desktop)
        USAGE (X)
        USAGE (Y)
        LOGICAL_MINIMUM (-127)
        LOGICAL_MAXIMUM (127)
        REPORT_SIZE (8)
        REPORT_COUNT (2)
        INPUT (Data,Var,Rel)
    END COLLECTION
END COLLECTION
```

Finally, the above report descriptor needs to be "compiled" so that it becomes a const array to be stored in FLASH memory. This can be done by using a dedicated tool provided by the USB-IF[23]. And so, the descriptor becomes a C array:

[23] https://bit.ly/3pzZs7e

```
const char ReportDescriptor[50] = {
    0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)
    0x09, 0x02,                    // USAGE (Mouse)
    0xa1, 0x01,                    // COLLECTION (Application)
    0x09, 0x01,                    //   USAGE (Pointer)
    0xa1, 0x00,                    //   COLLECTION (Physical)
    0x05, 0x09,                    //     USAGE_PAGE (Button)
    0x19, 0x01,                    //     USAGE_MINIMUM (Button 1)
    0x29, 0x03,                    //     USAGE_MAXIMUM (Button 3)
    0x15, 0x00,                    //     LOGICAL_MINIMUM (0)
    0x25, 0x01,                    //     LOGICAL_MAXIMUM (1)
    0x95, 0x03,                    //     REPORT_COUNT (3)
    0x75, 0x01,                    //     REPORT_SIZE (1)
    0x81, 0x02,                    //     INPUT (Data,Var,Abs)
    0x95, 0x01,                    //     REPORT_COUNT (1)
    0x75, 0x05,                    //     REPORT_SIZE (5)
    0x81, 0x03,                    //     INPUT (Cnst,Var,Abs)
    0x05, 0x01,                    //     USAGE_PAGE (Generic Desktop)
    0x09, 0x30,                    //     USAGE (X)
    0x09, 0x31,                    //     USAGE (Y)
    0x15, 0x81,                    //     LOGICAL_MINIMUM (-127)
    0x25, 0x7f,                    //     LOGICAL_MAXIMUM (127)
    0x75, 0x08,                    //     REPORT_SIZE (8)
    0x95, 0x02,                    //     REPORT_COUNT (2)
    0x81, 0x06,                    //     INPUT (Data,Var,Rel)
    0xc0,                          //   END_COLLECTION
    0xc0                           // END_COLLECTION
};
```

Going into details of the above report description declarations is outside the scope of this book. Moreover, I have to admit that this matter is mostly for people operating in the HID industry, since it is very poorly documented, and it is very tough to find good tutorials online. I suggest giving a look to this tutorial[24] and this other one[25] for a good introduction to the topic.

27.3.1.3 USB HID Class-Specific Requests

The USB-IF standardizes very few class-specific requests for the HID class. These requests are sent by the Host to the device through the control EP0, and they are summarized in **Table 20**.

[24] https://bit.ly/3qy8yk3
[25] https://bit.ly/3mFNx5T

Table 20: HID class-specific requests

Value	Request	Description
0x01	`GET_REPORT` [26]	The `GET_REPORT` request allows the host to receive a report via the Control pipe.
0x02	`GET_IDLE`	The `GET_IDLE` request reads the current idle rate for a particular Input report.
0x03	`GET_PROTOCOL` [27]	The `GET_PROTOCOL` request reads which protocol is currently active (either the boot protocol or the report protocol.)
0x04-0x08	Reserved	
0x09	`SET_REPORT`	The `SET_REPORT` request allows the host to send a report to the device, possibly setting the state of input, output, or feature controls.
0x0A	`SET_IDLE`	The `SET_IDLE` request silences a particular report on the Interrupt In pipe until a new event occurs or the specified amount of time passes
0x0B	`SET_PROTOCOL` [28]	The `SET_PROTOCOL` switches between the boot protocol and the report protocol (or vice versa).

27.3.2 Building a Vendor-Specific USB HID Device

Let us do a quick recap, this time focusing on shallow details without going too much in depth.
A HID compliant device is a USB peripheral exchanging messages, called *reports*, with the host PC. This is performed by using two *Interrupt* endpoints: one IN endpoint for messages flowing from the device to the host and one OUT endpoint (optional) for messages from the host to the device. In order to exchange data between the two endpoints, this data needs to be exactly described to the host device driver: how many bytes are exchanged, if this data varying or it is constant, the value these bytes can assume and how they relate each other. This operation is carried out by the *Report descriptor*, which shall adhere to a well-defined structure.
If we just focus on the above sentences, then we can forget about mice, keyboards, gamepads, human beings and so on. From the HID protocol point-of-view, a HID device is just a peripheral sending bunch of bytes over the USB communication. And we can build the custom protocol we want over this sequence of bytes.

Having this simple concept in mind, we can build how many vendor-specific devices we want,

[26] This request is mandatory and must be supported by all devices.
[27] This request is required only for boot devices.
[28] This request is required only for boot devices.

without dealing with the development and certification of custom device drivers, custom installers for drivers and so on. This is the reason why making vendor-specific HID devices is still the best option when designing a new USB device, and this is the reason why ST provides a Custom HID class in its STM32 USB Device Library.

As you can see, the most relevant issue to address when designing a vendor-specific HID device is the definition of the *Report Descriptor*. Without a good *Report Descriptor* we will not be able to exchange messages (reports) between the host and the device. The good news is that the HID specification provides a way to define "unstructured" reports, that do not need to adhere to a fixed structure like mice and keyboards. For example, we can declare a report that can assume the full range of 8-bit bytes:

```
4  LOGICAL_MINIMUM (0)
5  LOGICAL_MAXIMUM (255)
```

And then we can define the *Input* report (that is the message flowing from the device to the host) as a sequence of *n* variable bytes:

```
6  REPORT_ID (1)
7  REPORT_SIZE (8)
8  REPORT_COUNT (The amount of bytes in the report)
9  USAGE (Undefined)
10 INPUT (Data,Var,Abs,Vol)
```

where the `REPORT_ID` corresponds to a *tag* we add to the message so that we can specialize the reports according to given tags.
In the same way, we can define the *Output* report (that is the message flowing from the host to the device):

```
11 REPORT_ID (2)
12 REPORT_SIZE (8)
13 REPORT_COUNT (The amount of bytes in the report)
14 USAGE (Undefined)
15 OUTPUT (Data,Var,Abs,Vol)
```

Putting all together, we have the following *Report Descriptor*:

```
USAGE_PAGE (Generic Desktop)
USAGE (Undefined)
COLLECTION (Application)
    LOGICAL_MINIMUM (0)
    LOGICAL_MAXIMUM (255)
    REPORT_ID (1)
    REPORT_SIZE (8)
    REPORT_COUNT (The amount of bytes in the report)
    USAGE (Undefined)
    INPUT (Data,Var,Abs,Vol)
    REPORT_ID (2)
    REPORT_SIZE (8)
    REPORT_COUNT (The amount of bytes in the report)
    USAGE (Undefined)
    OUTPUT (Data,Var,Abs,Vol)
END_COLLECTION
```

Which translates to the following C array:

```
const uint8_t ReportDescriptor[32] = {
  0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)
  0x09, 0x00,                    // USAGE (Undefined)
  0xa1, 0x01,                    // COLLECTION (Application)
  0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
  0x26, 0xff, 0x00,              //   LOGICAL_MAXIMUM (255)
  // IN report
  0x85, 0x01,                    //   REPORT_ID (1) - The report tag
  0x75, 0x08,                    //   REPORT_SIZE (8)
  0x95, IN_REPORT_SIZE-1,        //   REPORT_COUNT (The amount of bytes in the report)
  0x09, 0x00,                    //   USAGE (Undefined)
  0x81, 0x82,                    //   INPUT (Data,Var,Abs,Vol)
  // OUT report
  0x85, 0x02,                    //   REPORT_ID (2) - The report tag
  0x75, 0x08,                    //   REPORT_SIZE (8)
  0x95, OUT_REPORT_SIZE-1,       //   REPORT_COUNT (The amount of bytes in the report)
  0x09, 0x00,                    //   USAGE (Undefined)
  0x91, 0x82,                    //   OUTPUT (Data,Var,Abs,Vol)
  0xc0                           // END_COLLECTION
};
```

Once we have defined the report descriptor, we can generate a new project by using CubeMX and by selecting the *Custom Human Interface Device Class (HID)* as wanted class for our device. CubeMX will generate a template project that can be adapted to our needs with a few modifications. The **Figure 21** shows the structure of the generated project. Let us briefly describe the most relevant source files and their main routines.

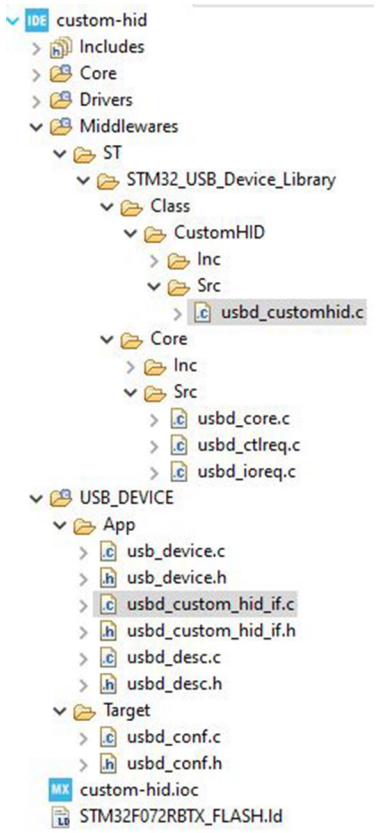


Figure 21: Structure of the Custom HID generated project

- **Class/CustomHID/Src/usbd_customhid.c**: this file contains the complete implementation of the HID class protocol.
  - The array `USBD_CUSTOM_HID_CfgFSDesc` contains the full implementation of the HID descriptors (*Configuration*, *Interface*, *HID* and *Endpoint* descriptors).
  - The function `USBD_CUSTOM_HID_Init()` is responsible of the IN and OUT endpoints initialization.
  - The function `USBD_CUSTOM_HID_Setup()` contains the full implementation of the *Class-specific Requests* list in **Table 20**.
  - The function `USBD_CUSTOM_HID_SendReport` allows to send a report (a message) with a given LEN through the IN endpoint.
- **USB_DEVICE/App/usbd_custom_hid_if.c**: this file contains the actual implementation of the custom HID protocol.
  - The array `CUSTOM_HID_ReportDesc_FS` contains the report descriptor, which will be passed to the host when the `GET_DESCRIPTOR` request is issued over the EP0.
  - The function `CUSTOM_HID_OutEvent_FS()` is a callback invoked once a report is sent by the host through the OUT endpoint.

We are now going to see a complete and very bare bone example that allows us to understand which modifications are needed to the generated project so that we can exchange reports in both the ways. The idea is to have a report message made of four bytes:

- The first byte is the report ID. The report ID is used to "tag" the reports, and we have just two IDs: the ID 1 is used for messages from the device to the host (INPUT report); the ID 2 is for messages flowing in the opposite way.
- The second byte corresponds to the status of the Nucleo's USER button (the blue one). The byte is used just for the IN report (with ID 1), since we cannot modify the status of a button.
- The third and fourth bytes correspond to the upper and lower part of an unsigned half-word (16 bits) used to setup the DAC output I/O connected to the Nucleo's LD2 LED. This will allow us to modify the output voltage of the PA5 GPIO so that we can dimmer the LD2 LED.

Nucleo-F401 **Nucleo-F103**

Owners of the Nucleo-F401 and Nucleo-F103 will find a slightly different example. Since those STM32 MCUs do not provide a DAC, the example is made so that the LD2 LED blinks at 1Hz rate. For this reason, the third byte of the report message will never change.

Filename: `USB_DEVICE/App/usbd_custom_hid_if.c`

```
24  /* Private variables ---------------------------------------------------------*/
25  extern DAC_HandleTypeDef hdac;
26  extern USBD_HandleTypeDef hUsbDeviceFS;
27
28  /** Usb HID report descriptor. */
29  static const uint8_t CUSTOM_HID_ReportDesc_FS[USBD_CUSTOM_HID_REPORT_DESC_SIZE] = {
30    0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)
31    0x09, 0x00,                    // USAGE (Undefined)
32    0xa1, 0x01,                    // COLLECTION (Application)
33    0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
34    0x26, 0xff, 0x00,              //   LOGICAL_MAXIMUM (255)
35    // IN report
36    0x85, 0x01,                    //   REPORT_ID (1)
37    0x75, 0x08,                    //   REPORT_SIZE (8)
38    0x95, USBD_CUSTOMHID_OUTREPORT_BUF_SIZE-1,      //  REPORT_COUNT (3)
39    0x09, 0x00,                    //   USAGE (Undefined)
40    0x81, 0x82,                    //   INPUT (Data,Var,Abs,Vol)
41    // OUT report
42    0x85, 0x02,                    //   REPORT_ID (2)
43    0x75, 0x08,                    //   REPORT_SIZE (8)
44    0x95, USBD_CUSTOMHID_OUTREPORT_BUF_SIZE-1,       //  REPORT_COUNT (3)
45    0x09, 0x00,                    //   USAGE (Undefined)
46    0x91, 0x82,                    //   OUTPUT (Data,Var,Abs,Vol)
47    0xC0                           //   END_COLLECTION
48  };
49
50  static int8_t CUSTOM_HID_Init_FS(void);
51  static int8_t CUSTOM_HID_DeInit_FS(void);
```

```
52  static int8_t CUSTOM_HID_OutEvent_FS(uint8_t *report, uint8_t report_len);
53  static int8_t CUSTOM_HID_GetData(uint8_t *report, uint8_t *report_len);
54
55  USBD_CUSTOM_HID_ItfTypeDef USBD_CustomHID_fops_FS = {
56    CUSTOM_HID_ReportDesc_FS,
57    CUSTOM_HID_Init_FS,
58    CUSTOM_HID_DeInit_FS,
59    CUSTOM_HID_OutEvent_FS,
60    CUSTOM_HID_GetData
61  };
62
63  static int8_t CUSTOM_HID_Init_FS(void) {
64    return (USBD_OK);
65  }
66
67  static int8_t CUSTOM_HID_DeInit_FS(void) {
68    return (USBD_OK);
69  }
70
71  static int8_t CUSTOM_HID_OutEvent_FS(uint8_t *report, uint8_t report_len) {
72    HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R,
73                     (uint32_t)((report[2] << 8) | report[3]));
74
75    return (USBD_OK);
76  }
77
78  static int8_t CUSTOM_HID_GetData(uint8_t *report, uint8_t *report_len) {
79    uint32_t dacValue = 0;
80
81    report[0] = 0x1;
82    report[1] = !HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin);
83
84    dacValue = HAL_DAC_GetValue(&hdac, DAC_CHANNEL_2);
85    report[2] = dacValue >> 8;
86    report[3] = dacValue & 0xFF;
87
88    *report_len = 4;
89
90    return (USBD_OK);
91  }
```

The above code shows the modifications to the **usbd_customhid-if.c** file. Lines [29:48] correspond to the report descriptor described so far. At lines [55:61] there is an instance of the `struct USBD_CUSTOM_HID_ItfTypeDef` (defined in the file **usbd_customhid.h**), which has been modified to add the pointer to an additional function to retrieve the report data: the function is implemented at lines [78:91] and it is self-explanatory. Finally, the callback `CUSTOM_HID_OutEvent_FS()` is modified so that

it sets the DAC channel 2 to the value received by the host.

The auto-generated code assumes a dimension of the report message equal to two and a smaller dimension of the report descriptor. We so need to modify the macros `USBD_CUSTOMHID_OUTREPORT_BUF_SIZE` and `USBD_CUSTOM_HID_REPORT_DESC_SIZE` in the **usbd_conf.h** file, as shown below at lines [75:77]:

Filename: `USB_DEVICE/App/usbd_conf.h`

```
64 /*---------- -----------*/
65 #define USBD_MAX_NUM_INTERFACES     1
66 /*---------- -----------*/
67 #define USBD_MAX_NUM_CONFIGURATION     1
68 /*---------- -----------*/
69 #define USBD_MAX_STR_DESC_SIZ     512
70 /*---------- -----------*/
71 #define USBD_DEBUG_LEVEL     0
72 /*---------- -----------*/
73 #define USBD_SELF_POWERED     1
74 /*---------- -----------*/
75 #define USBD_CUSTOMHID_OUTREPORT_BUF_SIZE     4
76 /*---------- -----------*/
77 #define USBD_CUSTOM_HID_REPORT_DESC_SIZE     32
78 /*---------- -----------*/
79 #define CUSTOM_HID_FS_BINTERVAL     0x5
```

In the same way, we need to increase the dimension of the two endpoints' buffer (IN/OUT) in the file **usbd_custom_hid.h**:

Filename: `Middlewares/ST/STM32_USB_Device_Library/Class/CustomHID/Inc/usbd_customhid.h`

```
44 #define CUSTOM_HID_EPIN_ADDR                0x81U
45 #define CUSTOM_HID_EPIN_SIZE                USBD_CUSTOMHID_OUTREPORT_BUF_SIZE
46
47 #define CUSTOM_HID_EPOUT_ADDR               0x02U
48 #define CUSTOM_HID_EPOUT_SIZE               USBD_CUSTOMHID_OUTREPORT_BUF_SIZE
```



Read Carefully

In the same header file we need to modify the OUT endpoint address from 0x1 to 0x2, as shown at line 47. This because the current HID class implementation does not take care of the concurrent access (in reading/writing) on the IN/OUT endpoints, and this could break the endpoint configuration causing the stall of the USB stack. So, the simplest workaround is to use two different endpoint numbers at all. This will cause a lot of headaches to novice users and this is very subtle to debug if you do not have an hardware USB protocol analyzer.

Finally, we need to perform a few modifications to the **usbd_custom_hid.c**. First, we need to modify the `USBD_CUSTOM_HID_Setup()` function (which is responsible of the processing of the *Class Requests*)

so that it correctly handles the `GET_REPORT` request. This request is not implemented by ST, while it is the sole mandatory *Class Request* as shown in **Table 20**.

Filename: `Middlewares/ST/STM32_USB_Device_Library/Class/CustomHID/Src/usbd_customhid.c`

```
static uint8_t  USBD_CUSTOM_HID_Setup(USBD_HandleTypeDef *pdev,
                                      USBD_SetupReqTypedef *req)
{
  USBD_CUSTOM_HID_HandleTypeDef *hhid = (USBD_CUSTOM_HID_HandleTypeDef *)pdev->pClassData;
  uint16_t len = 0U;
  uint8_t  *pbuf = NULL;
  uint16_t status_info = 0U;
  uint8_t ret = USBD_OK;
  uint8_t report[USBD_CUSTOMHID_OUTREPORT_BUF_SIZE];
  uint8_t reportLen = 0;

  switch (req->bmRequest & USB_REQ_TYPE_MASK)
  {
    case USB_REQ_TYPE_CLASS :
      switch (req->bRequest)
      {
        case CUSTOM_HID_REQ_SET_PROTOCOL:
          hhid->Protocol = (uint8_t)(req->wValue);
          break;

        case CUSTOM_HID_REQ_GET_PROTOCOL:
          USBD_CtlSendData(pdev, (uint8_t *)(void *)&hhid->Protocol, 1U);
          break;

        case CUSTOM_HID_REQ_SET_IDLE:
          hhid->IdleState = (uint8_t)(req->wValue >> 8);
          break;

        case CUSTOM_HID_REQ_GET_IDLE:
          USBD_CtlSendData(pdev, (uint8_t *)(void *)&hhid->IdleState, 1U);
          break;

        case CUSTOM_HID_REQ_GET_REPORT:
          ((USBD_CUSTOM_HID_ItfTypeDef *)pdev->pUserData)->GetData(report, &reportLen);
          if(reportLen == USBD_CUSTOMHID_OUTREPORT_BUF_SIZE) {
                  USBD_CtlSendData(pdev, report, reportLen);
                  ret = USBD_OK;
          } else {
                  USBD_CtlError(pdev, req);
                  ret = USBD_FAIL;
          }
          break;
```

Next, we need to modify the functions `USBD_CUSTOM_HID_DataOut()` and `USBD_CUSTOM_HID_EP0_RxReady()` so that we can pass to the `CUSTOM_HID_OutEvent_FS()` a report message with a length higher than two.

Filename: `Middlewares/ST/STM32_USB_Device_Library/Class/CustomHID/Src/usbd_customhid.c`

```
649 static uint8_t  USBD_CUSTOM_HID_DataOut(USBD_HandleTypeDef *pdev, uint8_t epnum) {
650   USBD_CUSTOM_HID_HandleTypeDef *hhid = (USBD_CUSTOM_HID_HandleTypeDef *)pdev->pClassData;
651
652   ((USBD_CUSTOM_HID_ItfTypeDef *)pdev->pUserData)->OutEvent(
653                                 hhid->Report_buf, USBD_CUSTOMHID_OUTREPORT_BUF_SIZE);
654
655   USBD_LL_PrepareReceive(pdev, CUSTOM_HID_EPOUT_ADDR, hhid->Report_buf,
656                          USBD_CUSTOMHID_OUTREPORT_BUF_SIZE);
657
658   return USBD_OK;
659 }
660
661 static uint8_t USBD_CUSTOM_HID_EP0_RxReady(USBD_HandleTypeDef *pdev) {
662   USBD_CUSTOM_HID_HandleTypeDef     *hhid = (USBD_CUSTOM_HID_HandleTypeDef *)pdev->pClassData;
663
664   if (hhid->IsReportAvailable == 1U) {
665           ((USBD_CUSTOM_HID_ItfTypeDef *)pdev->pUserData)->OutEvent(
666                                 hhid->Report_buf, USBD_CUSTOMHID_OUTREPORT_BUF_SIZE);
667
668     hhid->IsReportAvailable = 0U;
669   }
670
671   return USBD_OK;
672 }
```

The example is fully working but with just one notably limitation: the only way to transfer data from the device to the host is by issuing a `GET_REPORT` request. This is a strong limitation that does not allow us to capture once the USER BUTTON is pressed. We can so modify the example so that a report message is sent to the host once the interrupt connected to GPIO13 fires.

Filename: `USB_DEVICE/App/usb_device.c`

```
22 #include "usb_device.h"
23 #include "usbd_core.h"
24 #include "usbd_desc.h"
25 #include "usbd_customhid.h"
26 #include "usbd_custom_hid_if.h"
27
28 /* Private variables ---------------------------------------------------------*/
29 volatile int8_t userBtnStatus = -1;
30
31 /* USB Device Core handle declaration. */
```

```
32  USBD_HandleTypeDef hUsbDeviceFS;
33
34  void MX_USB_DEVICE_Init(void) {
35    uint8_t report[4], reportLen;
36
37    /* Init Device Library, add supported class and start the library. */
38    if (USBD_Init(&hUsbDeviceFS, &FS_Desc, DEVICE_FS) != USBD_OK) {
39      Error_Handler();
40    }
41    if (USBD_RegisterClass(&hUsbDeviceFS, &USBD_CUSTOM_HID) != USBD_OK) {
42      Error_Handler();
43    }
44    if (USBD_CUSTOM_HID_RegisterInterface(&hUsbDeviceFS, &USBD_CustomHID_fops_FS) != USBD_OK) {
45      Error_Handler();
46    }
47    if (USBD_Start(&hUsbDeviceFS) != USBD_OK) {
48      Error_Handler();
49    }
50
51    while(1) {
52      if(userBtnStatus >= 0) {
53        HAL_Delay(10); //Adding a little bit of debouncing
54        if(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == userBtnStatus) {
55          USBD_CustomHID_fops_FS.GetData(report, &reportLen);
56          USBD_CUSTOM_HID_SendReport(&hUsbDeviceFS, report, reportLen);
57        }
58        userBtnStatus = -1;
59      }
60    }
61  }
62
63  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
64    if(GPIO_Pin == B1_Pin) {
65      userBtnStatus = HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin);
66    }
67  }
```

How to test the example? Well, we have several options depending on the HOST Operating System.

- **Windows**: if you just want to perform a test of the sample firmware, then you can consider the SimpleHIDWrite[29] tool by Jan Axelson. This tool works very easily. First, you need to select the device you want to exchange data with. The tool will automatically show as much text boxes as many bytes are in the report message. In our case, it will show one byte for the *Report ID* and three bytes for the rest of the message (see **Figure 22**).

[29] http://janaxelson.com/files/SimpleHIDWrite3.zip

By setting `01` as the *Report ID*, the tool will allow to retrieve the report message by clicking on **Get Report** button (the tool, will send a `GET_REPORT` request through the control EP0). The received message will have the form `**rd** 01 00 00 00`. Instead, if we press the USER BUTTON, then the tool will automatically display the report in the form `**RD** 01 01 00 00`. To set the DAC output we can setup the *Report ID* to `02` and the last two bytes to `0A 00`.

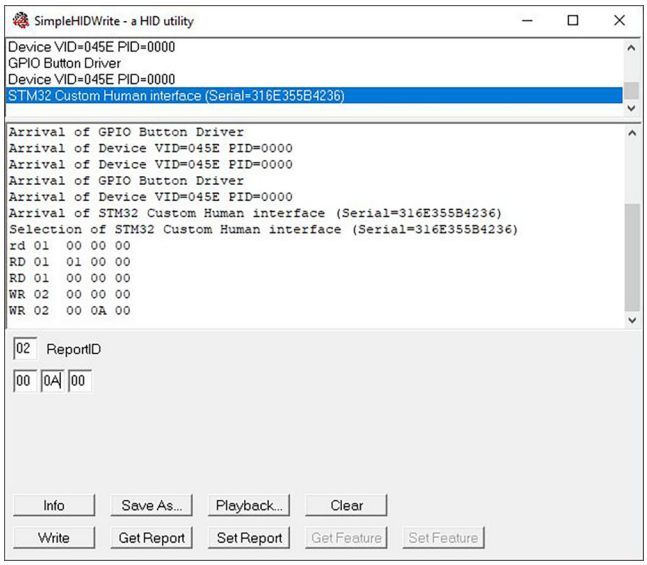


Figure 22: The SimpleHIDWrite utility

- **Windows**, **MacOS** and **Linux**: if you need to programmatically access to a HID device, then the `hidapi`[30] is one of the best option to consider, especially if you are making a cross-platform application. `hidapi` is a library part of the `libusb` project and it is dedicated to the handling of USB-HID devices. It works on the three major OSes offering the same API. Moreover, there exist several portings of the `hidapi` library to other languages.
  For example, the following snippet is written in Python 3 and it uses the `cython-hidapi`[31] wrapper to interface our HID example.

[30] https://github.com/libusb/hidapi
[31] https://github.com/trezor/cython-hidapi

Filename: `hid_test.py`

```
import hid, time, threading, struct

class myT(threading.Thread):
    def run(self):
        global hidhandle
        while True:
            report = hidhandle.read(0x4)
            if report[1] == 1:
                print("USER BUTTON PRESSED")
            elif report[1] == 0:
                print("USER BUTTON RELEASED")

hidhandle = hid.device()
hidhandle.open(0x483, 0x5750)

print("Manufacturer: %s" % hidhandle.get_manufacturer_string())
print("Product: %s" % hidhandle.get_product_string())
print("Serial No: %s" % hidhandle.get_serial_number_string())

t = myT()
t.start()

STEP = 50
ledStatus = 2000
incr = STEP
while True:
    p = tuple(struct.pack(">H", ledStatus))
    hidhandle.write((2,0)+p)
    if ledStatus >= 3500:
        incr = -STEP;
    elif ledStatus <= 2000:
        incr = STEP
    ledStatus += incr
    time.sleep(0.05)
```

The code should be easy to understand. The class `myT` is a thread running in background that performs a blocking `read()` to the device: once the Nucleo's USER BUTTON is pressed, a new report is issued through the IN endpoint. If the second byte in the report is equal to `1`, then the USER BUTTON has been pressed, otherwise it has been released.
The other part of the example is just a continuous writing to the OUT endpoint of an unsigned integer ranging from 2000 up to 3500, and vice versa: this will cause the LD2 LED to fade IN and OUT.

Finally, another solution to test a HID device is to use the `hidapitester` tool[32].

[32] https://github.com/todbot/hidapitester/releases

27.4 Debugging USB Devices

To develop USB devices is not a trivial task and, especially if you are new to the USB protocol, you will need a way to debug the data exchanged between the device and the host. There are two main options to perform this dirty job: hardware protocol sniffer and analyzers and software tools.

27.4.1 Software Sniffers and Analyzers

There are several free and commercial USB software sniffers around. Software analyzers can be used to capture device data transfers and protocols. They can also be used to send commands to the device. Software analyzers work by replacing the USB software stack on the host machine. This is a required step to monitor the USB traffic. As a result, the software analyzer will be dependent on the host hardware with regard to what information is available to be presented. Software analyzers have one important advantage: they display communications at the driver level, allowing to understand what's going wrong between your device and the host OS's device driver.
However, software analyzers have also some disadvantages. First, a host cannot monitor the USB traffic without being part of the bus itself. This means observing conditions such as suspend or reset is not possible. Additionally, depending on the host hardware and software stack, there are other bus conditions that the host-controller handles without software involvement. Another con is the cross-platform availability of the software. If the software analyzer runs just on Windows, then you will be able to debug just the communication with Windows OS. And this is a serious limit nowadays. One final disadvantage is related to timing. The accuracy of timing with a software analyzer is dependent on the host operating system, which could induce varying levels of inaccuracies.

You will find several USB software sniffers. Wireshark[33] is the only one available for Windows, MacOS and Linux. It is completely free, and it works well in most situations.
However, if you are new to the USB protocol, you can find the Wireshark output a little hard to decode. Several commercial and dedicated software are available. USB Monitor[34] (successfully used by this author in the past) and USBTrace[35] are two good alternatives to consider.

27.4.2 USB Hardware Analyzers

If you need to do serious things you need serious tools. Trust this author: buy a hardware protocol analyzer and you will save a lot of time.
Hardware protocol analyzers are custom-manufactured pieces of hardware (mainly an FPGA with a USB capable MCU) that are inserted between the host and the device to monitor the bus traffic. Some of these devices even have the capability to generate bus traffic if there is a specific condition that is trying to be produced. Prices for these types of devices can vary greatly depending on manufacturer, desired USB specification (USB 2.0 versus USB 3.0), and traffic generation capabilities. In general, these analyzers cost between few hundred dollars and several thousand dollars.

[33] https://wiki.wireshark.org/CaptureSetup/USB
[34] https://www.hhdsoftware.com/usb-sniffer
[35] https://www.sysnucleus.com/

While there are several companies that produce hardware analyzers, the two more common companies are TotalPhase and LeCroy. This author uses the Beagle USB 480[36] from TotalPhase (see **Figure 23**), which comes with a free of use software, called TotalPhase Data Center, which allows to analyze the traffic between the host and the device. The software runs on Windows, MacOS and Linux.



Figure 23: Beagle USB 480 protocol analyzer by TotalPhase

27.5 Optimizing the STM32 USB Device Library

Let us face it clearly: the STM32 USB Device Library is far from being an optimized library. And this for one simple reason: ST designed the most possible generic USB library, abstracting from the very specific STM32 MCU, the given USB phyther (LS-FS/HS) and the USB class. Moreover, it is designed so that everyone can design his own class by simply adapting existing code. However, this makes the library full of replicated structures and full of *if-then-else*. The result is a library with a relevant FLASH and SRAM footprint that can give you a lot of issues when designing your application.

You can reach better results without upsetting the structure of the library by performing these simple operations:

- If you are going to design a device that operates just in Full-Speed or High-Speed mode, then you can remove from the library all those descriptors related to the USB speeds you are not considering. For example, if your device operates just in FS mode, then you can remove the following giant arrays at all:
  - `USBD_<CLASS>_CfgHSDesc`.
  - `USBD_<CLASS>_OtherSpeedCfgDesc`.
- The other descriptors can be defined as `const`, so that they go in FLASH memory instead of SRAM. This will save a lot of SRAM memory. While doing this operation, comment the following lines in the file **usbd_ctlreq.c**, around at lines 436 and 441 (honestly speaking, I do not know what those lines are used for...):

[36] https://www.totalphase.com/media/datasheet/TP_Beagle_USB_480_Protocol_Analyzer-Datasheet.pdf

Filename: `Middleware/ST/STM32_USB_Device_Library/Core/Src/usbd_ctlreq.c`

```
428     case USB_DESC_TYPE_DEVICE:
429       pbuf = pdev->pDesc->GetDeviceDescriptor(pdev->dev_speed, &len);
430       break;
431
432     case USB_DESC_TYPE_CONFIGURATION:
433       if (pdev->dev_speed == USBD_SPEED_HIGH)
434       {
435         pbuf = pdev->pClass->GetHSConfigDescriptor(&len);
436         pbuf[1] = USB_DESC_TYPE_CONFIGURATION;
437       }
438       else
439       {
440         pbuf = pdev->pClass->GetFSConfigDescriptor(&len);
441         pbuf[1] = USB_DESC_TYPE_CONFIGURATION;
442       }
443       break;
```

- In the same way, you can remove all those functions designed to work just with one speed mode, and you can strip all those *if-then-else* related to USB speed selection. For example:

```
if(speed == USBD_SPEED_HIGH) {
    USBD_GetString((uint8_t *)USBD_CONFIGURATION_STRING_FS, USBD_StrDesc, length);
} else {
    USBD_GetString((uint8_t *)USBD_CONFIGURATION_STRING_FS, USBD_StrDesc, length);
}
return USBD_StrDesc;
```

this `if` can be simplified to just one instruction if your device operates just in FS mode, for example.

Finally, consider that there exist other complete USB stacks that are more optimized (and even more supported) than the ST's library. tinyusb[37] is a framework to seriously take in account when designing USB devices and consider that all STM32 families are officially supported.

27.6 Going to the Market

Before start selling a USB device, there are several steps needed depending on the level of compliance you want/need to acquire.

First, there are some mandatory steps you need to do. The most relevant (and expensive) one is the acquiring of an official *Vendor ID* (VID). The VID is a unique identifier that each company gets from the USB Implementers Forum. There are three options to get a VID:

[37] https://github.com/hathach/tinyusb

- Become a member of USB-IF and pay the annual membership dues. At the time of writing this chapter, the dues were $5000/yr. Membership has several benefits. Some of the most important include
  - A free Vendor ID (if one has not been previously assigned)
  - Eligibility to participate in free USB-IF sponsored quarterly Compliance Workshops
  - Opportunities to participate in USB-IF marketing programs and events, such as retail newsletters, store endcaps, featured products, etc.
  - A company listing in the USB key contacts list
  - Eligibility for inclusion in the USB current products list on the usb.org web site and in periodic USB-IF retail newsletters
  - A waived logo administration fee when joining the USB-IF logo program
  - Discounts on Developer Conferences, products in the e-store, etc
  - Eligibility to participate in Device Working Groups
- Purchase a Vendor ID from USB-IF for a one-time fee of $6000. Note however that simply buying a VID does not authorize you to use the USB logo with a product. In order to do so, you need to become a USB-IF non-member logo licensee. This licensing fee comes at a cost of $3,500 for a two-years term. Also, be aware that this licensing fee does not entitle you to other USB-IF membership benefits, such as compliance workshops.
- Buy an individual ***PID*** from MCS Electronics[38]. Well, this looks more like an Italian[39] way of getting around the obstacle, but MCS Electronics is a Dutch company that acquired many years ago a VID and started selling individual PIDs from their large pool. They had a dispute with USB-IF, but it looks like that according to the Dutch law they can do that and USB-IF stopped to fight against them. So, if you need the USB Logo compliance (more about this next), then you need to buy your own VID and to invest ∼10K USD. If, instead, you need just a unique VID/PID pair, it is ok to buy an individual PID from MCS for less than USD 20 (Yes, you read that right).

The next mandatory step is to perform compliance tests at least for European CE and for the USA FCC. With these two certifications you can sell your device in the majority of countries on the earth. When choosing a test lab, you have two main options:

- Use a more expensive test-lab, but close to you.
- Use a far but a lot cheaper test-lab in China.

Consider that you can obtain a CE/FCC certification in China for less than USD 500. In China there are several companies notified for regulatory tests that work fast and well in many circumstances. I personally visited a couple of relevant test labs in ShenZhen and I can guarantee that you can find the same quality and professionalism offered by several “well-known” western companies.
When testing your device, I suggest you to perform at least the following tests:

[38] https://bit.ly/34sPTic
[39] Being the author an Italian citizen, he is fully authorized to say that :-D

- Tests for the Electromagnetic Compatibility (EMC) Directive 2014/30/EU[40].
- Tests for FCC Title 47 CFR Part 15[41].
- Tests for the Low-Voltage directive (LVD) (2014/35/EU)[42].
- Tests for the REACH-SVCH[43].

All those tests will cost you less than USD 1000 in a Chinese test-lab.
If you are going to design a 5V USB device, then the LVD test is not mandatory. However, to obtain this certification, together with the REACH-SVCH one, will save you a lot of headaches when shipping your devices around the world: people from the customs can transform every happy deal in a nightmare. Trust this author.

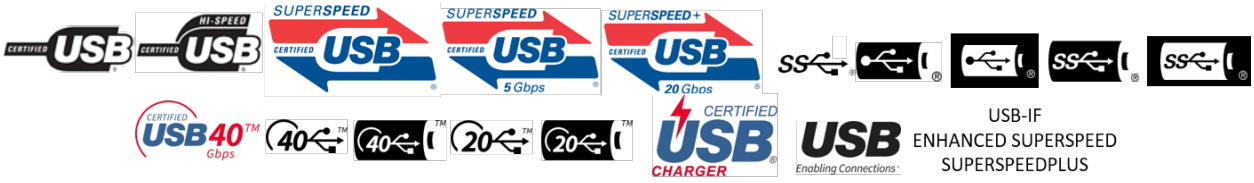


Figure 24: One of the USB-IF Logos - Can you spot which one you need?

Now there are a couple of additional and optional certifications you may need.
In case you want/need to acquire the USB Logo compliance[44] (see **Figure 24**), then you need to follow a dedicated path. To ensure the interoperability between USB-based devices (hosts, device, hubs, etc.) the USB-IF strictly rules the way your device adhere to the standards. So, you cannot use any USB logo on your device freely. Instead, you must undergo a series of compliance tests that are administered by USB-IF. These compliance tests will ensure that the device under test confirms to any relevant sections of the USB specification. Upon passing, the product will receive the ability to use the relevant USB—Certified® logo, and in addition, have the product put on the USB integrators' list, which is simply a list maintained by USB-IF that contains the list of USB products that have met the mandatory compliance criteria. There are two components that must be completed on the path to being certified as USB compliant: checklists and compliance testing. For more information, refer to the dedicated page on the USB-IF website[45]. When following this path, consider that this will cost to you no less than USD 10K and it is strongly suggested to receive support from specialized consultants.

Finally, you may need to acquire the *Windows Hardware Quality Labs* (WHQL) certification for your device and drivers. Well, this is not a trivial step. Microsoft offers a dedicated certification program[46], together with a suite of dedicated tools[47] to perform "pre-compliance" tests. Again, my suggestion is to ask for help to an experienced test lab. Be prepared to invest a lot of money.

[40] https://bit.ly/3t20c72
[41] https://bit.ly/3qXY3a1
[42] https://bit.ly/3qRZsi3
[43] https://bit.ly/3t0eDbF
[44] https://www.usb.org/compliance
[45] https://www.usb.org/compliance
[46] https://bit.ly/3HDiYWs
[47] https://bit.ly/3n2VHp4

28. Getting Started with a New Design

If you use STM32 microcontrollers for work, or you are going to create your latest funny project as a hobbyist, soon or later you will need to leave a development kit like the Nucleo, and you have to design a custom board around a given STM32 MCU. For every hardware engineer this is always an exciting process. You start from an idea, or a list of requirements, and you will obtain a piece of hardware able to do magic things.

The development process of a new board can be divided in two main steps: the hardware design part, related to components selection and placement, and the software development part, that consists in a starting configuration and all the code needed to make the board working. This chapter aims to provide a brief introduction to this topic. The chapter is logically divided in two parts: one related to the hardware design and one to software. Even if you are one of those lucky people working in companies where the hardware engineer is a separated figure from the firmware developer, it is strongly suggested to have a look at this chapter, which is essentially based on the hardware design. Otherwise, if you are the classical *one-man band*[1], reading this chapter at least once could help you if you are totally new to the STM32 world.

28.1 Hardware Design

If you come from simpler microcontroller architectures, like the ATMEL AVR ATMega328p used for the Arduino UNO, you may be familiar with some "artistic things" that often appear on the web (**Figure 28.1** is an example[2]). A lot of projects arise from a breadboard, few passives and several tons of wires. And they work great too.

However, if you are going to make a new board with an STM32 MCU, you have to completely forget this kind of design. This is because not only do not exist STM32 microcontrollers provided in a THT package. These MCUs require that special attention must be placed to the PCB layout process, even for the low-cost line STM32F030. The PCB design become critical if you are planning to use the fastest STM32 MCUs, like the F4 and F7 series, in conjunction with external devices like fast QSPI memories and external SDRAM.

[1] Like this author is :-)

[2] The picture was taken from this site(https://degreesplato.wordpress.com)

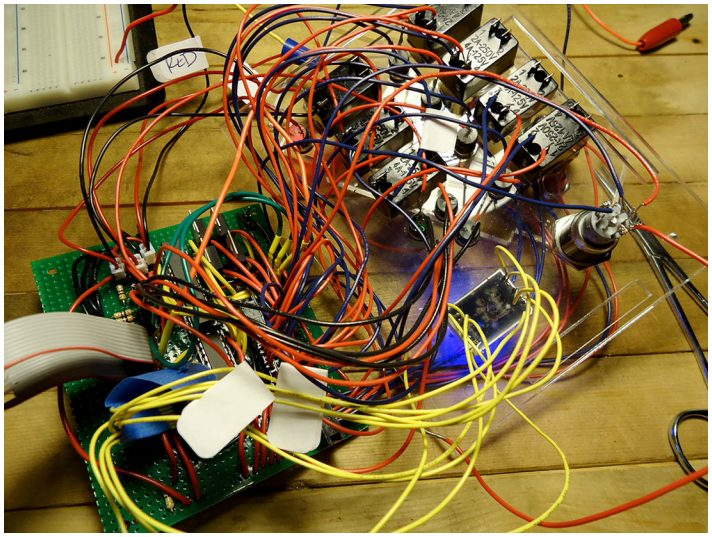


Figure 28.1: A creative "thing" made with an ATMega328 plus 1 mile of wires

For each STM32 family, ST provides a dedicated datasheet named "*Getting started with STM32xx hardware development*". For example, for the STM32F4 family, the AN4488[3] is the corresponding document. It is strongly suggested to carefully read these documents, since they contain the most important information to design a new PCB correctly. For all my designs based on these MCUs, I have always followed the information provided by ST engineers, and I have never had any issues. The next paragraphs summarize the most important aspects and decision steps, according to me, to follow during the design process of a new board based on an STM32 MCU.

28.1.1 PCB Layer Stack-Up

Every time you start a new design based on a microcontroller, you need to decide which PCB technology best fits your design and BOM cost, keeping in mind this important axiom: the faster your board goes, the more PCB layers are required. And this also true for STM32 MCUs. Even if it is not rare to see some low-cost 8-bit MCUs soldered on a single layer CEM PCB[4], for an STM32 MCU a 2-layers board is the minimum requirement. But, if you are planning to use the fastest versions of the STM32F4 series (like the STM32F446 MCU able to run up to 180MHz) or the latest STM32H7 series, then you have to consider a 4-layers PCB as minimum requirement[5].

Multi-layers PCBs have several advantages compared to 2-layers ones:

- More layers simplify the routing process, and this is really important if you have space constraints or if you need to route differential pair nets.
- They allow better routing for power as well as ground connections; if the power is also on a plane, it is available to all points in the circuit simply by adding vias.

[3] http://bit.ly/1NVb6ly
[4] This especially true for low-cost productions.
[5] Consider that the STM32F746-Discovery KIT is made with an eight layers PCB.

- They provide an intrinsic distributed capacitance between the power and ground planes, reducing high-frequency noise especially if your board relies on an external SRAM or a fast flash.
- For the same reason as before, they allow to significantly reduce EMI/RFI emissions, simplifying the development cost and the CE/FCC certification phase.

However, 4-layers PCBs have a higher cost compared to 2-layers ones, and this cost is often not affordable for some low-cost and higher volumes productions. Moreover, it is right to say that the Cortex-M portfolio (and hence the STM32 one) ranges from "low-cost" solutions able to run correctly on 2-layers boards to more powerful MCUs close to general purpose microprocessors (like the Cortex-M7 series), which demand a more advanced PCB stackup.

My personal experience is based on PCB designed with STM32F030 and STM32F401 MCUs, both implemented with 2-layers PCBs, and I had not significant issues during the boards testing. Using ground-planes on both layers allow to simplify the routing process and to reduce overall EMI emissions of the board.

28.1.2 MCU Package

The MCU package choice is often related to the whole PCB technology. STM32 MCUs are provided in several package variants. The most common and "simple to use" packages are the LQFP ones, like the LQFP-64 package used for all Nucleo boards. Packages with exposed pins have several advantages:

- They are easy to solder, even by hand for low-volume productions or for prototypes. With a little bit of practice, they can be soldered with the *drag soldering* technique[6], or simply placed on a PCB pre-covered with the solder paste using a stencil.
- They are easy to inspect using conventional *Automatic Optical Inspection* (AOI) machines, and they do not require x-ray inspection, which increases the production cost of your boards.
- They cost less for low and mid-volume productions, compared to other type of packages.
- They can be used on 2-layers low class PCBs (even a pattern class equal to 6 is sufficient[7]), different from other packages (like the BGA ones) that usually require more advanced PCB due the use of vias with a reduced annular ring.
- They provide a lot of signal I/O to interface external peripherals (this is obviously, but it is always good to remark it).

However, if space is a strict requirement for your design, then you have to consider BGA and similar packages, which offer more signal I/O in a smaller footprint.

[6]Youtube is full of videos that show how to solder SMD packages with this technique.
[7]Look at this document(http://bit.ly/1NVgYeI) from Eurocircuits to discover more about PCB production classes.

28.1.3 Decoupling of Power-Supply Pins

An important design step is the decoupling of every power supply pair (VDD, VSS). The key aspects can be summarized here:

- Each power couple (VDD, VSS) should be connected to a parallel ceramic capacitor of about 100nF (which is a widespread proven value) plus one 4.7µF ceramic capacitor for the overall MCU. It is best to choose 0805 or smaller capacitors (the smaller is the better is, since smaller capacitors have less ESR - for an STM32F7, 0402 capacitors is an option to consider). These capacitors need to be placed as close as possible to the appropriate pins, or the underside of the PCB if a BGA package is used for the fastest STM32 MCUs. If a ground plane is used, it is safe to connect VSS pins directly to the ground plane if this is extensive around that pin.
- This author also uses a large electrolytic capacitor (typically 10µF - a tantalum capacitor is also OK if your budget allows it) no more than 3cm away from the chip. The purpose of this capacitor is to be a reservoir of charge to supply the instantaneous charge requirements of the circuits locally, so the charge need not come through the inductance of the power trace.
- A small ferrite bead (with an impedance ranging from 600 to 1000Ω) placed in series between the analog power supply (AVDD) and digital power supply (VDD)[8]. It is used to:
  - Localizes the noise in the system.
  - Keeps external high frequency noise from the IC.
  - Keeps internally generated noise from propagating to the rest of the system.
- If your STM32 MCU provides a VBAT pin, it can be connected to the external battery (1.65 V < VBAT < 3.6 V). If no external battery is used, it is recommended to connect this pin to VDD with a 100nF external ceramic decoupling capacitor.

Figure 28.2 shows the reference schematics of an STM32F030CC MCU, while **Figure 28.3** shows the typical layout style used by this author to proper decouple power pins. As you can see, a solid ground plane ensures that decoupling capacitors are connected to the ground with the shortest possible path[9].

This document[10] from Texas Instruments is a good introduction to this topic.

[8] ST discourages the use of this ferrite if VDD is below 1.8V.

[9] However, keep in mind that the grounding scheme depends on the actual implementation. Some designs need a strong separation between analog and digital ground, plus some EMC-friendly devices (like ferrite beads) to connect them. Welcome to the "obscure" world of EMC :-)

[10] http://bit.ly/29pk0J9

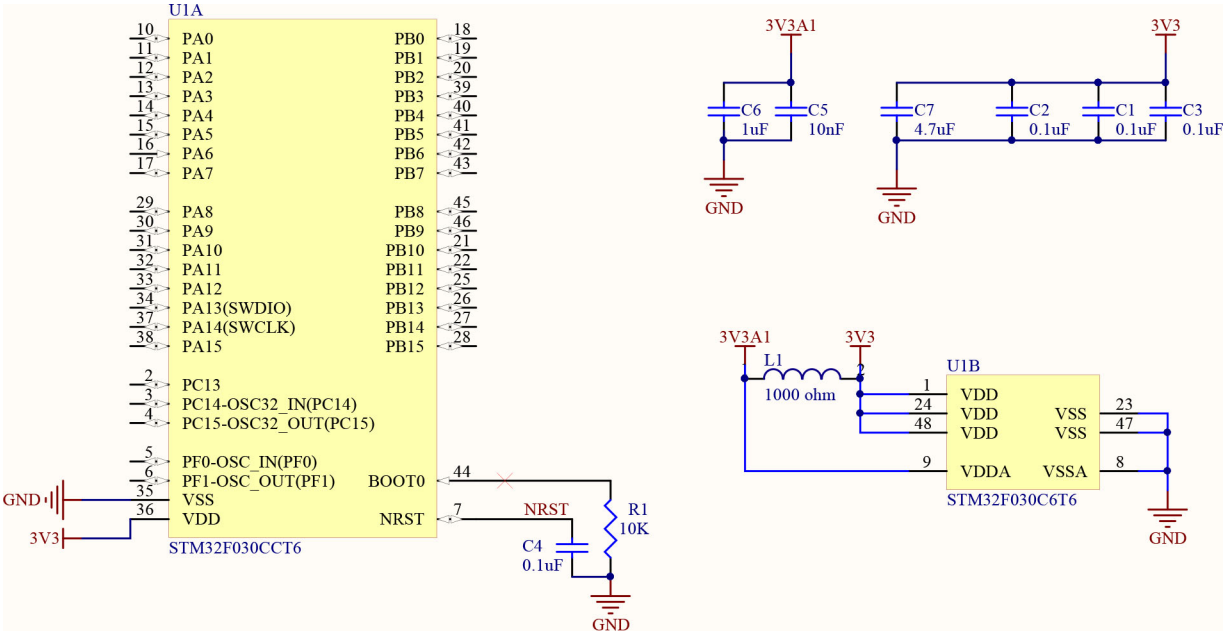


Figure 28.2: The minimal reference schematics for an STM32F030 MCU

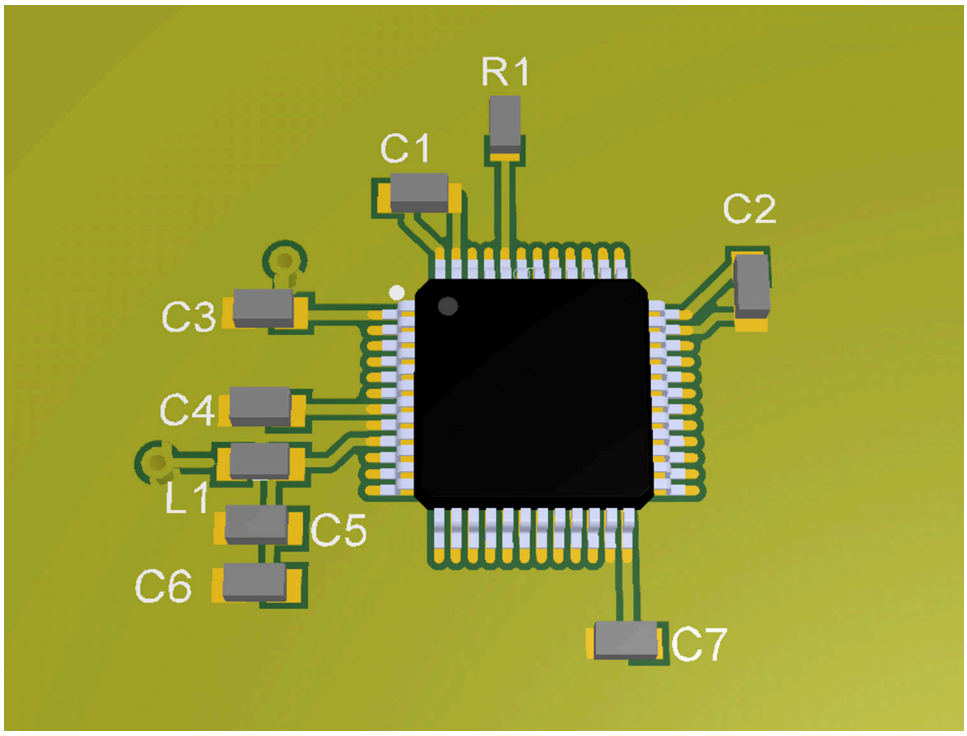


Figure 28.3: The preferred way by the author of this book to place decoupling capacitors

28.1.4 Clocks

If your design needs an external clock source, either the LSE or HSE one, special attention must be placed to the position of the external crystal and the selection of the capacitors used to match its load capacitance (this value is established by the crystal manufacturer, and it must be carefully checked during the selection process).

ST provides a really excellent guide (AN2867[11]) about oscillator design. Summarizing that guide is outside the scope of this paragraph, so I strongly suggest having a look at that application note. However, it is important to underline some things.

The most starting up errors (that is, the MCU does not want to properly boot in our final design when the external crystal is used) arises from bad choice of the external capacitors and bad placing of the crystal. For example, assuming a stray capacitance equal to 5pF and a crystal capacitance equal to 15pF, the following formula can be used to compute the value of external capacitors:

$C_{1,2} = 2(C_L - C_{stray}) = 2(15\text{pF} - 5\text{pF}) = 20\text{pF}$.

Moreover, it is best to place the crystal as close as possible to the MCU pins, surrounding it by a separated ground plane, in turn connected to the bottom ground plane, as shown in **Figure 28.4** (the bottom ground plane is not shown).

ST shows several "bad examples" in its Application Note. Moreover, all STM32 MCUs provide a useful feature to debug external oscillator issues: the *Clock Security System* (CSS). CSS is a self-diagnostic peripheral that detects the failure of the HSE. If this happens, HSE is automatically disabled (this means that the internal HSI is automatically enabled) and a NMI interrupt is raised to inform the software that something is wrong with the HSE. So, if your board refuses to work correctly, I strongly suggest you write down the exception handler for NMI, as described in Chapter 10. If the code hangs inside it, then there is a problem with your oscillator design.

Finally, consider that a lot of EMC issues come from bad placing of external clocks. Pay attention to the instructions contained in the ST application note.



The most of STM32 MCUs allow to connect an external or internal clock source (a PLL, the HSI or HSE and so on) to an output pin, called *Master Clock Output*(MCO). This is useful in some application, where this clock source may be used to drive an external IC or in audio applications. However, pay attention to avoid long traces between the MCU and the device connected to the MCO pin. In this case you have to consider the MCU like a normal clock source, and hence you have to pay attention both to the length of the trace and to cross-talks between MCO and other adjacent or underlying traces.

[11] http://bit.ly/1RFYZbZ

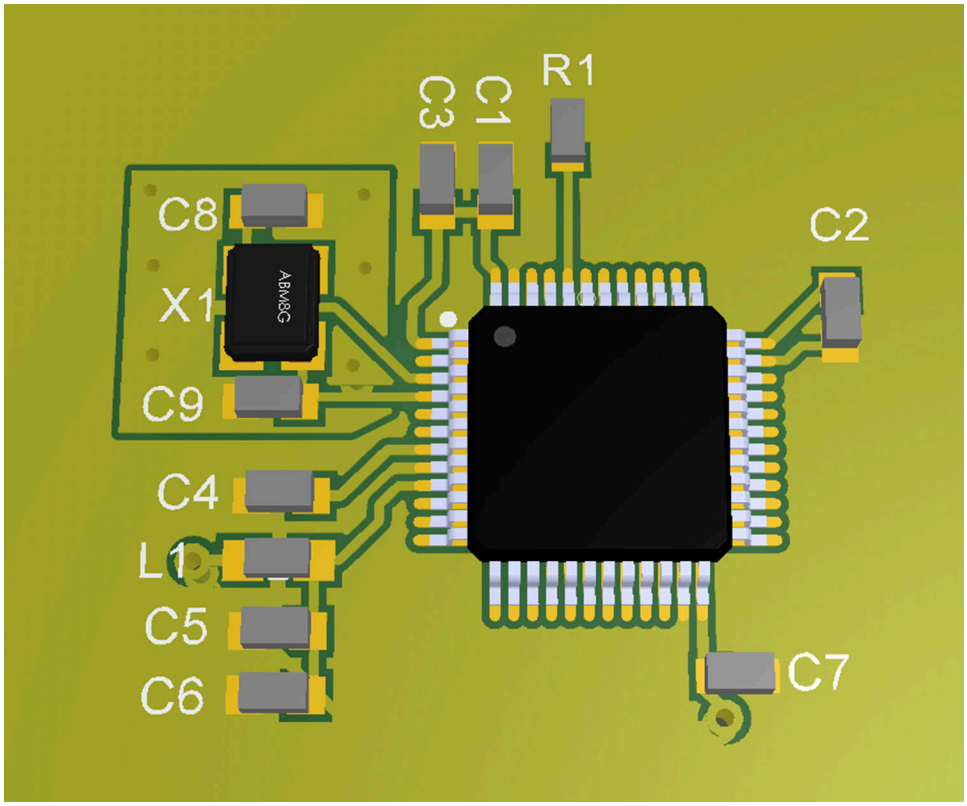


Figure 28.4: A good design way to place external crystals using a separated ground plane

28.1.5 Filtering of RESET Pin

To avoid unwanted reset of your board, it is strongly recommended to connect a decoupling capacitor (100nF is a proven value) between the RESET pin (named `NRST`) and the ground, even if your design does not require the use of the reset pin.

28.1.6 Debug Port

In order to develop and test the firmware for the new board, or to simply upload it to production devices, you need a way to interact with the target STM32 MCU using its debug port. STM32 MCUs offer several ways to debug them. One of this is through the use of the *Serial Wire Debug* (SWD) interface. SWD replaces the traditional JTAG port, using a clock line (named `SWDCLK`) and a single bi-directional data pin (named `SWDIO`[12]), providing all the normal JTAG debug and test functionality plus real-time access to system memory without halting the processor or requiring any target resident code (the condition for this to happen is that the SWD related I/O are not remapped to a different function - e.g., a general purpose output GPIO). Moreover, it is possible to use any ST-LINK debugger as debug device for your custom boards: all ST development boards (and, hence, the Nucleo too) are

[12]Sometimes, ST refers to these lines also as `SWCLK` and `SWIO`.

designed so that you can disconnect the target MCU from the ST-LINK interface and connect it to your board.

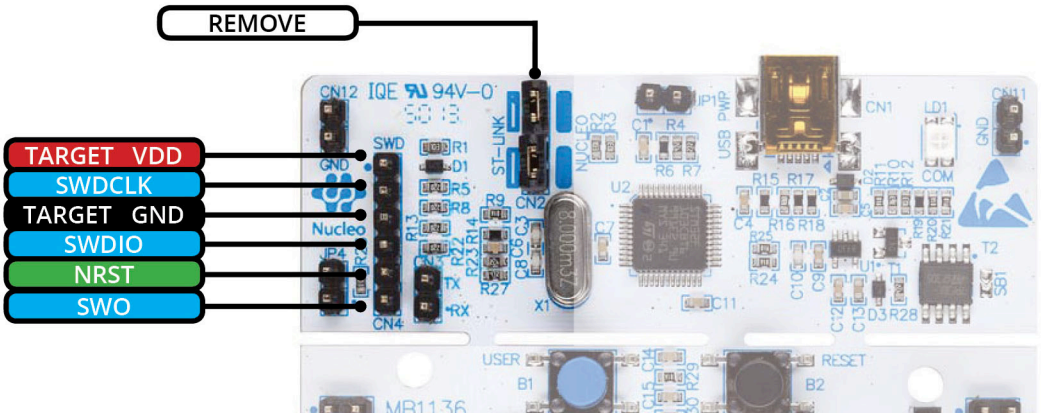


Figure 28.5: How to use the Nucleo as ST-LINK debugger

Figure 28.5 shows how to use a Nucleo as external debugger for a custom board. First, remove the two jumpers from the CN2 pin header. Next, connect the PIN1 of SWD pin header to a VDD (3.3V or lower) source of your custom board, PIN2 to the `SWDCLK` pin of the STM32 MCU in your board, PIN3 to the GND, PIN4 to `SWDIO` pin and finally the PIN5 to the `NRST` pin of the target STM32 MCU (this step is optional, at least in theory). The connection may be easily done simply routing those signals to a convenient pin header, which plays the role of debug port for your custom board. The `SWO` pin is also available on the SWD pin header, and it corresponds to the PIN6. However, the `SWO` is connected to the target MCU through a SMD jumper (SB15). So, if you want to use SWV functionality on you board, you will need to desolder that jumper.

Another useful feature to have on this debug port may be at least the USART TX pin of one of the available MCU USARTs. This could help you a lot during the development process, using it to print messages on a console to trace the firmware execution, even if it is not under debugging. Again, you could use the Nucleo board to interface the target MCU TTL USART to the Nucleo VCP, connecting USART pins to the CN3 connector on the Nucleo board, as shown in **Figure 28.6**. If so, you may need to desolder SB13 and SB13 jumper on your Nucleo or leave PA2 and PA3 of the target Nucleo MCU floating.

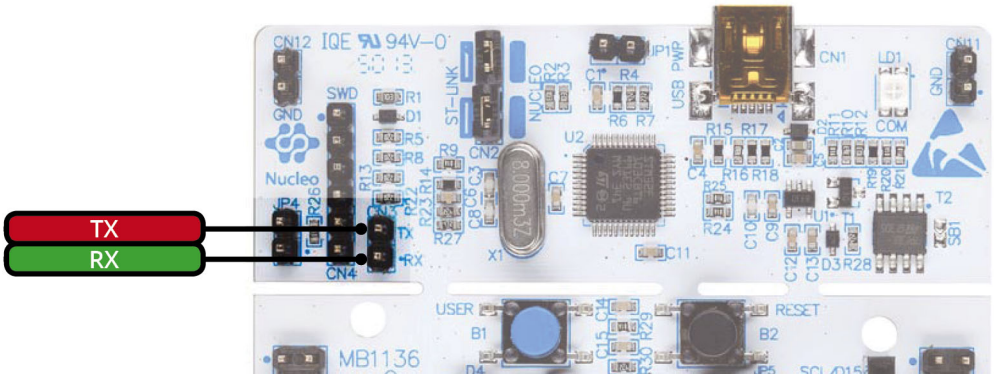


Figure 28.6: The CN3 connector allow to use the ST-LINK VCP with any other USART



Read Carefully

As said before, the SWD interface requires just two pins. These are named `SWDIO` and `SWDCLK`. You can easily identify them using CubeMX (more about this later) or downloading the right datasheet for your MCU. However, it is strongly suggested to use also the `NRST` pin for debugging. This is required because the STM32 microcontrollers allow to change the function of SWD pins, both for wanted design reason and for an invalid firmware state after a fault condition (e.g., an invalid memory access has corrupted the peripheral memory). Without routing the `NRST` signal to the debug port, it is impossible to connect to the target MCU "under reset", that is resetting the MCU just few CPU cycles before the MCU is placed under debug. This will help you in some critical situations. **So, to resume, always route to the custom "debug connector" on your board at least `SWDIO`, `SWDCLK` and `NRST` pins, plus VDD and GND.**

28.1.7 Boot Mode

Depending on the specific microcontroller model you are going to use in your design, STM32 MCUs can load firmware from different sources: internal or external flash, internal or external SDRAM, USART and USB are the most common sources for starting the firmware execution. This is an exciting feature of this platform described in Chapter 22.

This happens thanks to the fact that several boot loaders are pre-programmed in the *System memory* (the sub-region of code area starting from `0x1FFF F000`) during the MCU production. Each boot loader can be selected configuring one or two pins named `BOOT0` and `BOOT1`[13].

The default behavior, that is the regular boot from the internal flash, is obtained pulling to the ground at least BOOT0 pin and leaving BOOT1 pin (if present) floating. Once the firmware starts the execution, you can reuse BOOT pins as general I/O.

[13]The actual implementation of these pins depends on the specific STM32 series. For example, the STM32F030 provides only `BOOT0` pin, and substitutes the `BOOT1` pin with a specific bit inside the *Option Bytes* memory region.

28.1.8 Pay attention to "pin-to-pin" Compatibility...

A lot of STM32 microcontrollers are designed to be pin-to-pin compatible with other MCUs in the same series and between different series. This allows you to "simply" switch to a more/less performant model in case you need to adapt your design for budget reasons or if you are looking for a more powerful MCU.

However, the pin-to-pin compatibility is a feature that needs to be planned during the MCU selection process, even for MCUs belonging to the same STM32 series. Let us consider this example[14]. Suppose that you decide to use an STM32 MCU from the STM32F030 catalogue, and suppose that you choose the STM32F030R8 MCU, the one equipping the STM32F030 Nucleo. As soon as the board design is finished, and gerber files are sent to the PCB fab, you start developing the firmware (this is what often happens especially if you have to complete the project one day before you start developing it). After a while, you discover that the 8k of SRAM provided by this MCU are not sufficient for your project. So, you decide to switch to the STM32F030RC model, which provides 32K of SRAM and 256K of internal flash. However, after struggling several hours trying to understand why you cannot flash it, you discover that this model requires four additional power sources (PF4, PF5, PF6 and PF7), as you can see in **Figure 28.7**.

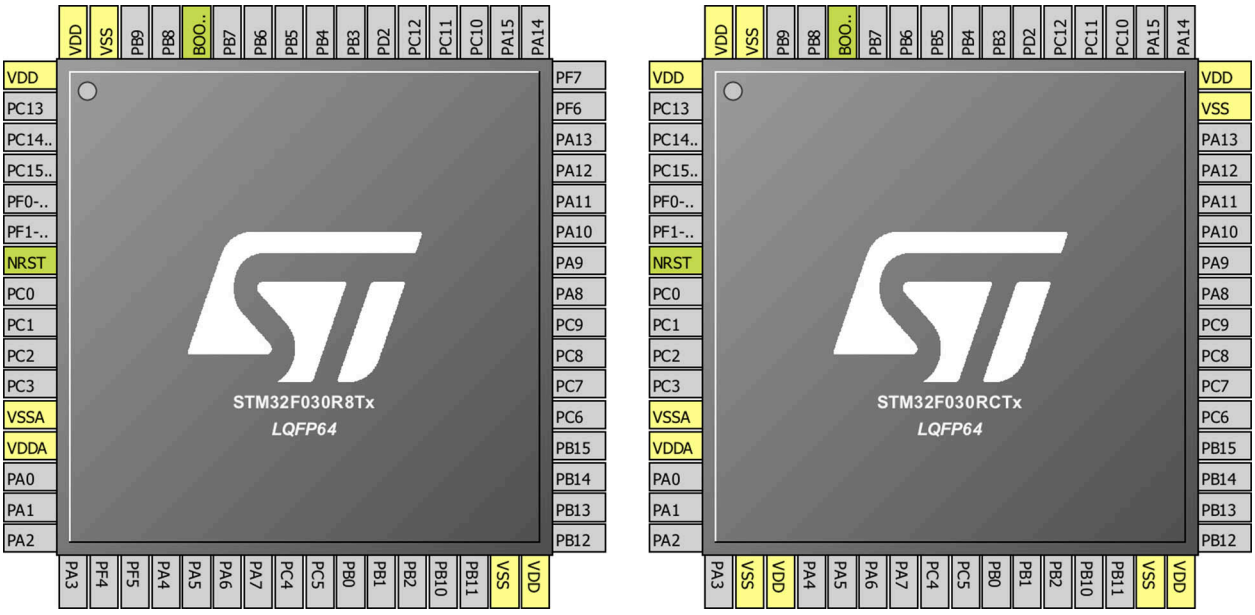


Figure 28.7: The STM32F030RC MCU requires four additional power sources compared to the STM32F030R8 one

So, how to avoid these kinds of mistakes? The best option is to *plan for the worst case*. In this specific case you may do a layout of your board that connects those pins (PF4, PF5, PF6 and PF7) to power sources even if you are going to use the STM32F030R8 model (being those pins regular I/O pins, it is ok to connect them to VDD and VSS, in parallel with decoupling capacitors).

Take note that CubeMX provides an excellent feature to check the pin-to-pin compatibility. You can

[14]This film is based on a true and sad story happened to this author :-)

find it in **Pinout->List Pinout Compatible MCUs** (see **Figure 28.8**). It automatically warns you if the MCU is 100% compatible or not with the current one.

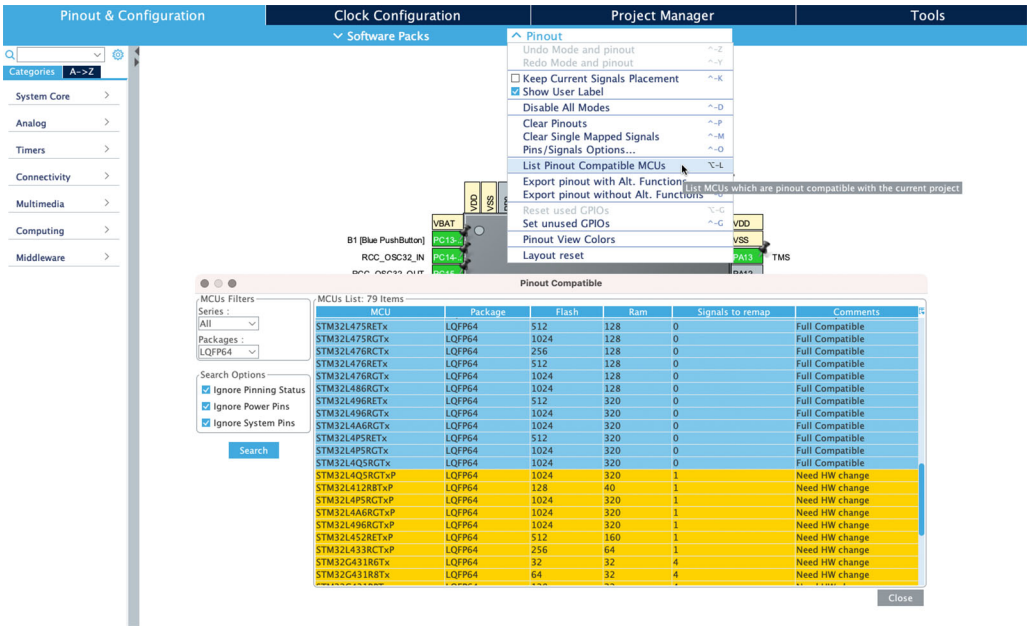


Figure 28.8: How CubeMX can help in identify pin-to-pin compatible MCUs

28.1.9 ...And to Selecting the Right Peripherals

The most of STM32 MCUs have multiple peripherals of a given type (SPI1, SPI2, etc.). This is a good thing for complex designs with multiple modules, but special care must be placed during the peripheral selection even for simple designs. And this is not only a problem related to the I/Os allocation. For example, let us suppose that you are basing your design on an STM32F030 MCU, and suppose that your design needs an UART and a SPI interface. You decide to use UART1 and SPI2 peripherals. During the firmware development, for performance reasons you decide to use both in DMA mode. However, looking to Table 9.2 you can see that it is not allowed to use `SPI2_TX` and `USART1_RX` in DMA mode simultaneously (they share the same channel). So, it is best to plan these software design choices while you are writing down the schematics.

If you are designing a device that will enter deeper sleep modes, like the *standby* one, and you want your device to be woken up by the user (maybe by pressing a dedicated button), then remember that usually just two I/Os can be used to this task (they are called *wake up pins*). So, avoid assigning those pins to other usages.

28.1.10 The Role of CubeMX During the Board Design Stage

Sometimes, I talk with people about CubeMX. A lot of them have a wrong consideration of what CubeMX is. Some of them consider it as a totally useless tool. Others limit its usage to the software development stage. **There is nothing more wrong**.

CubeMX is probably more useful during the hardware design process (both when drawing schematics and when doing board layout) than in the firmware development stage. Once you get familiar with the CubeHAL, you will stop to use CubeMX as a tool for the IDE project generation[15]. But CubeMX is essential during the design stage unless you are going to reuse previous designs or to base your projects always on few types of STM32 MCUs.

The most important part of CubeMX during the board design is the *Chip view*. Thanks to this representation you can "preview" in your mind the layout of the MCU part, and eventually adopt different layout strategies.

CubeMX is a tool that can be used iteratively. Let us me explain this concept better with an example. Suppose that you need to design a board based on an STM32F030C8Tx MCU. It is an LQFP-48 MCU from the F0 line. Suppose also that you need to use:

- Two SPI interfaces (SPI1 and SPI2).
- An I²C interface (I2C1).
- An external low speed clock source (LSE).
- Five GPIOs.
- An UART (UART2).

Once you have started a new project with this MCU, CubeMX shows you the MCU representation in the *Chip view*, as shown below.

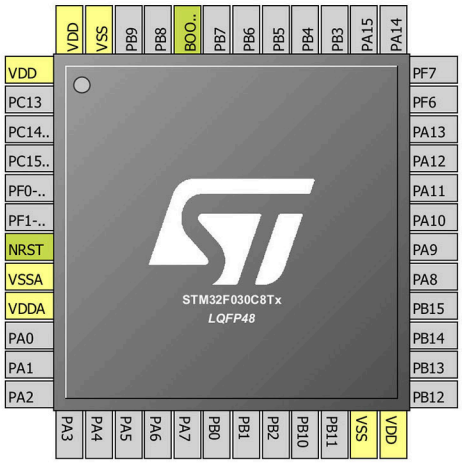


Figure 28.9: CubeMX shows a graphical representation of the MCU when a new project is started

This immediately gives you three facts:

- You can quickly derive that your board will need 6 decoupling capacitors, 5 for the power sources (4x100nF + 1x4.7uF) and 1x100nF for the NRST pin.
- PIN7 is the NRST pin and it must be decoupled.

[15]Honestly speaking, what CubeMX generates is not so good from a project organization point of view.

- PIN44 is the BOOT0 pin and it must be pulled-down.



Read Carefully

Never forget to tie to the ground the BOOT0 pin using a pull-down resistor (this reduce the power leakage). It is a common mistake for novices of this platform to leave that pin unconnected, or worst connecting it to a voltage source. STM32 hardware designers are divided in two groups: those that have forgotten to tie BOOT0 to the ground and those that will forget to do it.

The next step involves enabling all the required peripherals, the LSE and the SWD interface, leaving out the 5 GPIOs for the moment. We obtain the following representation in CubeMX:

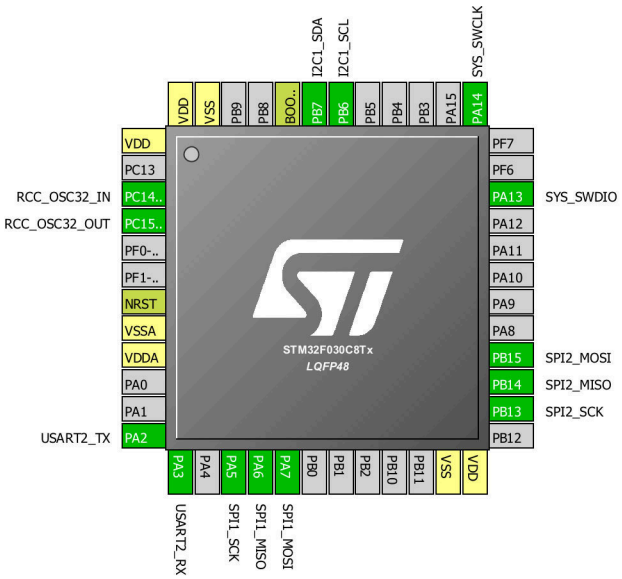


Figure 28.10: How CubeMX shows you the MCU when new peripherals are enabled

Ok. Now it is the good time to start writing down the board schematics, connecting the other devices to the MCU pins. Once you have completed this part of the schematics, you can start doing the layout process. In this phase, you discover that it is not simple to route the SPI1 signals to PA5, PA6 and PA7. So, doing a Ctrl+Click on the SPI1 signals you discover that you can remap them to PB3, PB4 and PB5, obtaining the following representation:

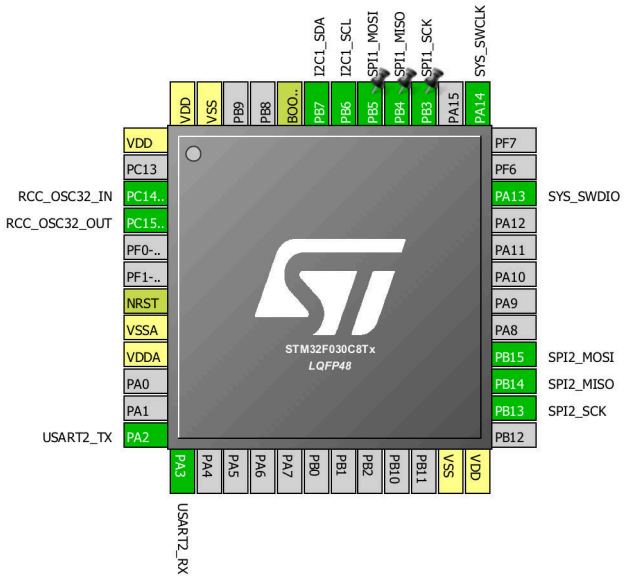


Figure 28.11: Pre-visualizing the MCU can help you during board layout

Now you can update your schematics and hence complete the layout of this part. Once the layout is almost complete, you can assign the 5 GPIO to the MCU pins, deciding which one best fits your layout. This is the reason why CubeMX can be used iteratively.

Another important thing regarding CubeMX is the ability to give custom names to signals. This is simply accomplished going into **Pinout->Pins/Signal Options** (see **Figure 28.12**). CubeMX will use the custom labels to generate corresponding C macros inside the `main.h` file. For example, an I/O labeled "TX_EN" will generate a macro named `TX_EN_Pin` to indicate the pin and a macro named `TX_EN_GPIO_Port` to indicate the corresponding GPIO port. This is important especially if you keep synchronized the CAD documentation and the project source files. It will help you to write better and more portable code.

Finally, I prefer to prefix the name of all high-speed signals with "HS_". This will guide you during the design process: if your CAD allows you to place constraints on nets, it will simplify the routing process, avoiding mistakes that would appear only during test phase.

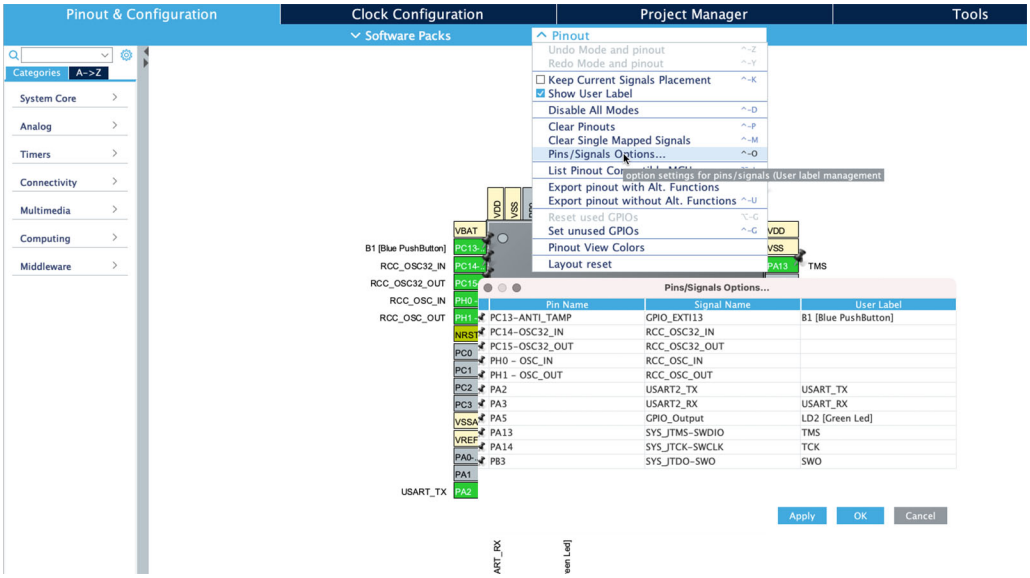


Figure 28.12: How to assign custom names to PINs

28.1.11 Board Layout Strategies

The layout of the final board is a sort of "art", a complex task that involves a deep knowledge of all modules used in your design. This is the reason why in large organizations this work is accomplished by specific engineers.

Here, I would like to provide a brief introduction to the whole process based on my personal experience.

- **A good layout is all about component placing**: if you are new to this task, remember that all starts from placing components on the final board. Every board can be logically and physically divided in sub-modules: power part, MCU and digital part, analog part and so no. Don't start routing signals before you have placed all components on the final board. Moreover, a good subdivision in sub-modules allows you to reuse design for different boards.
- **Follow these steps when doing the layout of an STM32 MCU**:
  - start placing the MCU;
  - if your board need external clock sources, place them immediately close to the MCU pins;
  - next place all decoupling capacitors needed;
  - connect power sources to the corresponding power lines or power planes if your layer stackup allows them;
  - **never forget to tie to the ground BOOT0 pin if needed, and to decouple NRST pin**;
  - if your design need an external SRAM or a fast flash memory, start placing them and route differential pair first;
  - route all high speed signals;
  - route remaining signals;

– avoid using too many vias during the signal routing and use CubeMX looking for better alternatives (that is, use other equivalent signal I/Os if possible).

28.2 Software Design

Once you have completed the hardware design, you can start developing the firmware part. If you have used CubeMX to design the MCU section of your custom board, you should be able to start coding the firmware immediately. If the CubeMX project observes faithfully the actual board design, you can simply generate the project as we have done for the Nucleo development board, then you can import it inside a new STM32CubeIDE project and start working on your application. Nothing different from what described in Chapter 3.

If you have already developed the firmware using a development board, and you need to adapt it to your custom design, you may proceed in this way:

- Generate a fresh new CubeMX project both for your development board (e.g., the Nucleo-F030), enabling the needed peripherals, and for the custom board you have designed.
- Do a comparison between the initialization routines for the used peripherals: if they differ, start replacing them one by one in the project made for the development board, and do a complete project compilation before to continue with the next peripheral. This will allow you to keep the control of what is changing in your firmware.
- To simplify the porting process, never change the peripheral initialization code generated by CubeMX, but use CubeMX to change peripheral settings.
- Try to use macros to wrap peripheral handlers. Once you change them, you only need to redefine the macros (for example, if your firmware developed with the Nucleo uses the USART2 peripheral, define a global macro in this way: `#define USART_PERIPHERAL huart2` and base your code on that macro; if your new design uses the USART1, then you have to redefine only that macro accordingly).

Remember that CubeMX essentially generates 5 or 6 files. If you reduce the modification to these files at minimum, it will be easy to rearrange the code.

Having a minimum viable firmware made with a development kit helps a lot during the debugging of your custom board. It happens often that, during the testing of a new board, you are in doubt if your issues arise from the hardware or the software. Knowing that the firmware works simplifies the hardware debugging stage.

28.2.1 Generating the binary image for production

In large organizations, who effectively loads the binary image of the firmware on the final board is a completely different person. As as engineer, you may be asked to generate an image of the final firmware in *release mode*. This is a way to indicate a binary image of the firmware compiled with

the highest possible optimization level, in order to reduce the final size of the image, and without including any debug information. This last requirement is needed both to reduce the size of binary image and to protect the intellectual property (the ELF file of a firmware compiled with debug symbols usually contains the whole firmware source code, so that GDB can show you the original source code while debugging).

From the Eclipse/GCC point of view, generating a binary image in *release mode* is nothing more than to configure the project accordingly. You might have already noticed that every new Eclipse project comes with two *Build Configurations* (go to **Project->Build Configurations->Manage** menu if you have never used this feature before): one named *Debug* and one *Release*. A build configuration is nothing more than a project configuration, and you can have as many separated configurations as you want in a single project.

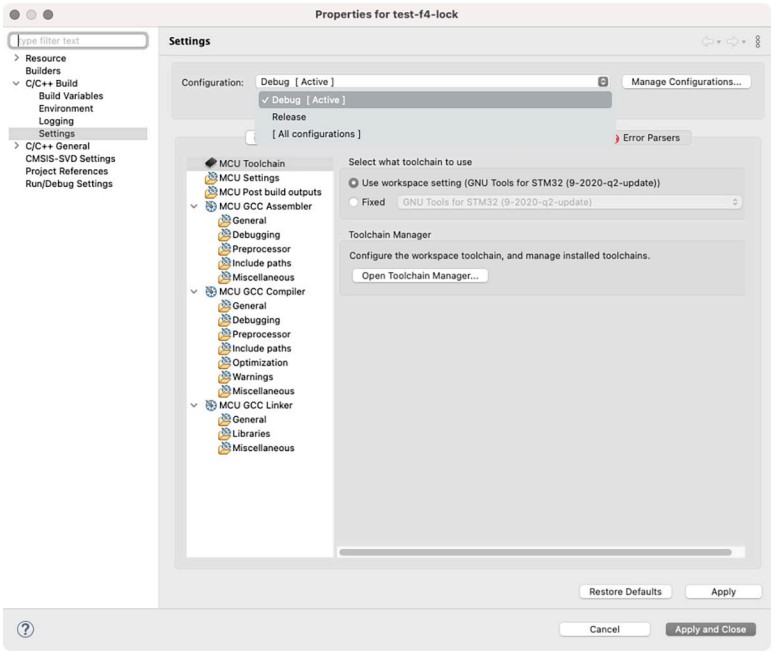


Figure 28.13: The Eclipse project settings dialog allows to switch to another *build configuration* easily

Figure 28.13 shows the project settings dialog (go to **Project->Properties** menu to open it). The **C/C++ Build->Settings** pane allows to configure the build options. Moreover, as you can see in **Figure 28.13**, you can quickly move to another build configuration using the **Configuration** combo-box. In the **Cross ARM C++ Compiler->Optimization** section we can setup the GCC optimization levels. GCC provides 6 optimization levels. Let us briefly introduce them:

- **-O0**: this corresponds to the *no optimization* level. It generates unoptimized code but usually has the fastest compilation time. Note that other compilers do extensive optimizations even if *no optimization* is specified. With GCC, it is very unusual to use -O0 for production if execution time is of any concern, since -O0 does mean no optimization at all. This difference between GCC and other compilers should be kept in mind when doing performance comparisons.

- **-O1**: this corresponds to a *moderate optimization*. It optimizes reasonably well but does not degrade compilation time significantly.
- **-O2**: this corresponds to *full optimization*. It generates highly optimized code and has the slowest compilation time.
- **-O3**: this also corresponds to *full optimization* as in "-O2", but it also uses more aggressive automatic inlining of subprograms within a unit and attempts to vectorize loops.
- **-Os**: this corresponds to *optimization for space*. It optimizes space usage (both code and data) of resulting program.
- **-Ofast**: this corresponds to *optimization for speed*. It optimizes code to increase its speed at expense fo the final binary size.
- **-Og**: this corresponds to *optimization for debug*. It enables optimizations that do not interfere with debugging. It should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience.

By default, the GCC optimization level for the *Release* configuration is *-Os*. Higher optimization levels perform more global transformations on the program and apply more expensive analysis algorithms in order to generate faster and more compact code. However, in embedded programming is usually suggested to start the development using the *no optimization* (-O0) level. This because more aggressive optimizations my lead to different behavior of time-constrained routines. As a rule of thumb, develop your firmware with the -O0 or the -Og levels, and start increasing it as long as you test all its features. Sometimes, it also happens that a firmware working perfectly when compiled with the -O0 level stops working at all when a more aggressive optimization is chosen. This often happens we have not correctly declared shared and global variables as `volatile`, and they are optimized to the compilers causing wrong behavior of ISR routines or different threads if we are using an RTOS.

Another important configuration parameter for the *Release* configuration is related to *Debug level*. This feature is configured inside the **Cross ARM C++ Compiler->Debugging** view, and GCC offers four increasing levels: **None**, **-g1**, **-g** (the default in *Release* configuration) and **-g3**. If you want to generate a binary image without debug information, select the **None** level.

Appendix

A. Miscellaneous HAL functions and STM32 features

This appendix chapter contains an overview of some HAL functions and STM32 features that makes little sense to treat in a separate chapter.

Force MCU reset from the firmware

Sometimes, when all is lost and we no longer have control of what is happening, the only salvation is to reset the microcontroller. The function

```
void HAL_NVIC_SystemReset(void);
```

initiates a system reset of the MCU. It uses the `void NVIC_SystemReset(void)` provided by the CMSIS package.

STM32 96-bit Unique CPU ID

The most of STM32 microcontrollers provides a unique CPU ID, which is factory-programmed. It is read only, and it cannot be changed.

This ID can be useful in several contexts. For example, it can be used:

- as unique USB device serial number;
- to generate custom license keys;
- for use as security keys in order to increase the security of code in Flash memory while using and combining this unique ID with software cryptography primitives and protocols before programming the internal Flash memory;
- to activate secure boot processes, etc.

Unfortunately, the position in memory of this ID is not common to all STM32 microcontrollers, but its memory mapped address changes between each STM32-series. **Table 1** shows the memory-mapped address of the Unique MCU ID for the MCUs equipping the Nucleo boards.

| Nucleo P/N | Factory-programmed 96-bit Unique-ID base address| |————-|————————————-| | NUCLEO-G474RE | `0x1FFF 7590` | | NUCLEO-F446RE | `0x1FFF 7A10` | | NUCLEO-F401RE | `0x1FFF 7A10` | | NUCLEO-F303RE | `0x1FFF F7AC` | | NUCLEO-F103RB | `0x1FFF F7E8` | | NUCLEO-F072RB | `0x1FFF F7AC` | | NUCLEO-L476RG | `0x1FFF 7590` | | NUCLEO-L152RE | `0x1FF8 00CC` | | NUCLEO-L073RZ | `0x1FF8 007C` | For example, in an STM32F401xE MCU it is mapped at `0x1FFF 7A10`. To access to the unique ID we can use the following code fragment:

```
...
uint32_t *uniqueID = (uint32_t*)0x1FFF7A10;

for(uint8_t i = 0; i < 12; i++)
    i < 11 ? printf("%x:", (uint8_t)uniqueID[i]) : printf("%d\n", (uint8_t)uniqueID[i]);
...
```

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.

STM32CubeIDE Issues

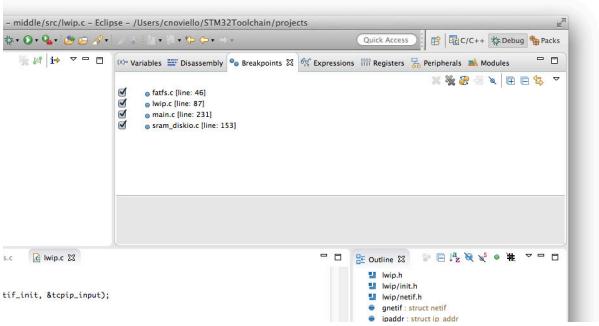
Debugging Continuously Breaks at Every Instruction During a Debug Session

If you have not enabled the *instruction stepping mode*, this happens because you have defined too many hardware breakpoints. Please, consider that the number of hardware breakpoints is limited for every Cortex-M family, as shown in the following table:

Available breakpoints/watchpoints in Cortex-M cores

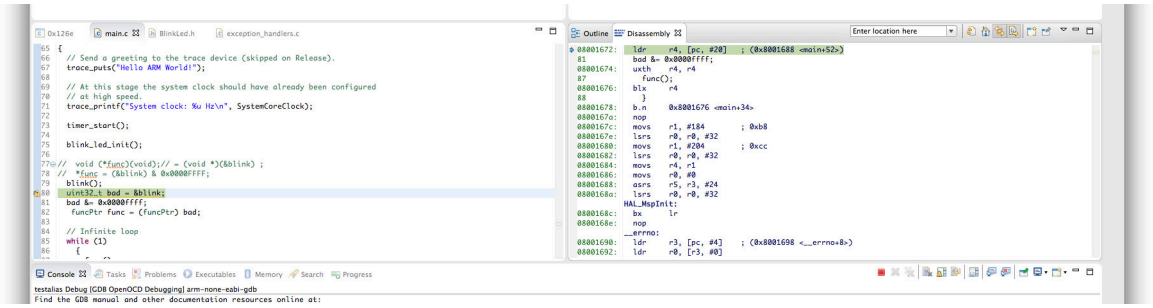
Cortex-M	Breakpoints	Watchpoints
M0/0+	4	2
M3/4/7	6	4

To check the used breakpoints in your application, go to the *Debug perspective*, then in the *Breakpoints* pane (see figure below) and disable or delete unneeded breakpoints.



The Step-by-Step Debugging is Really Slow

This happens when the *Disassembly view* is enabled, as shown below.



Eclipse needs to reload ARM assembly instructions at every step (one C instruction can correspond to a lot of assembly instructions), and this really slows down the debugging session. It is not an issue related to OpenOCD or the ST-LINK interface, but instead is just an overhead connected with Eclipse. Switch to another view (or simply close the *Disassembly view*) to resolve the issue.

The Firmware Works Only Under a Debug Session

This mostly like happens because you have the *semihosting* support enabled. As described in Chapter 24, ARM *semihosting* can rely on the ARM assembly `BKPT` instruction, which halts the CPU execution waiting for an action of the debugger.

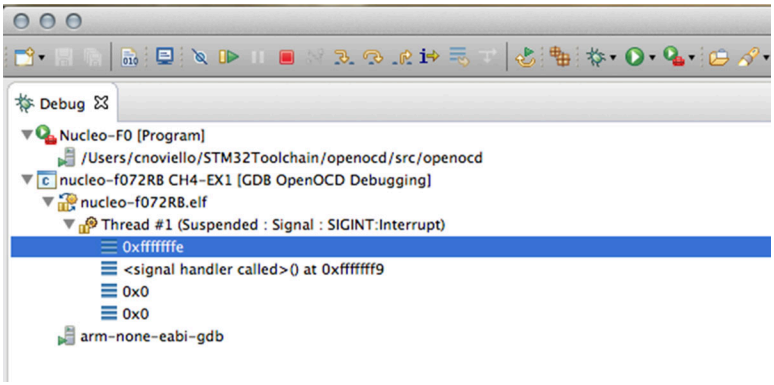
STM32 Related Issues

This section contains a list of frequently issues related with the programming of STM32 microcontrollers.

The Microcontroller Does Not Boot Correctly

Although this might seem strange, there is a quite long list of reasons why an STM32 refuses to boot properly. This issue usually has the following symptoms:

- the firmware does not start.
- the Program Counter points to a completely invalid address (usually `0xfffffffd` or `0xfffffffe`, but other addresses of the 4GB memory space are possible too), as shown by Eclipse during the debug session.



To resolve this issue, we need to distinguish between two cases: if you are developing the firmware for a development board like the Nucleo or for a custom designed board (this difference is just to simplify the analysis).

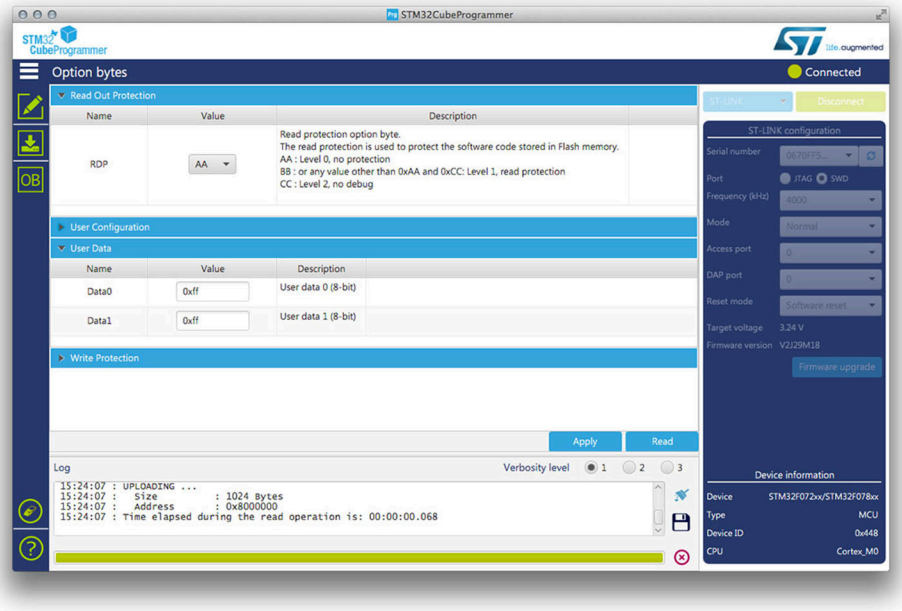
If you are developing the firmware using a development board then, especially if you are new to this platform (but tiredness can play nasty tricks even to experienced users…), probably two things may be wrong:

- The definition of memory sections inside the linker script `STM32XXXXXX_FLASH.ld` file is wrong, either for the flash region or the SRAM region (usually, the flash region simply does not start from `0x08000000`).
- The startup file is corrupted.

If, instead, you are developing the firmware for a custom board, then besides controlling the previous two points you must also check that:

- The configuration of BOOT pins is right (at least BOOT0 pin tight to ground, BOOT1 floating).
- The NRST pin is correctly decoupled using a 100nF capacitor.

Sometimes it happens that, even if all the previous points are correct, the micro still refuses to boot. This often suddenly happens after a debug session, or after you have tested a buggy firmware designed to access in write mode to the internal flash memory. Another recognizable symptom in that neither OpenOCD is able to flash the MCU. If so, probably you have a corrupted *Option bytes* memory region. The STM32CubeProgrammer can help you a lot to debug this situation. Once you have connected the ST-LINK debugger, go inside the **Option bytes** section and check that BOOT configuration (inside the **User configuration** section) correctly matches your MCU.



Finally, sometimes a full chip erase may also help in solving obscure booting issues ;-)

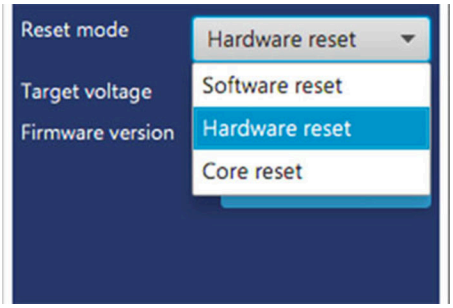
It is Not Possible to Flash or to Debug the MCU

Sometimes it happens that it is not possible to flash the MCU or to debug it using the debugger. Another recognizable symptom in that the ST-LINK LD1 LED (the one that blinks red and green alternatively while the board is under debugging) stops blinking and remains frozen with both the LEDs ON.

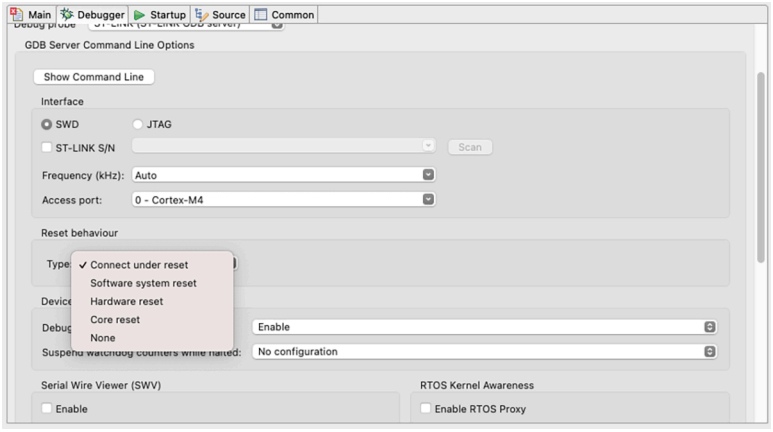
When this happens, it means that the ST-LINK debugger cannot access to the debug port (through SWD interface) of the target MCU, or the flash is locked preventing its access to the debugger. There are usually two reasons that leads to this faulty condition:

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

To address this issue, we have to force ST-LINK debugger to connect to the target MCU while keeping its nRST pin low. This operation is called *connection under reset*, and it can be performed by using the STM32CubeProgrammer tool, by selection the **Hardware reset** mode in the **Reset mode** combo box, as shown below.



The same operation can be performed in the *Debug Configuration*, as shown next.



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[16].

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

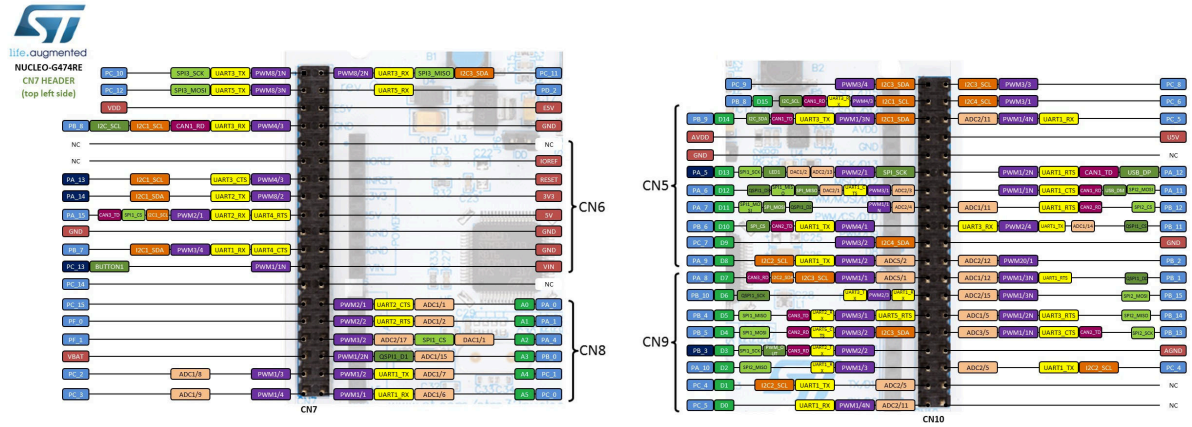
[16] https://developer.mbed.org/platforms/?tvend=10

Nucleo-G474RE

Arduino compatible headers

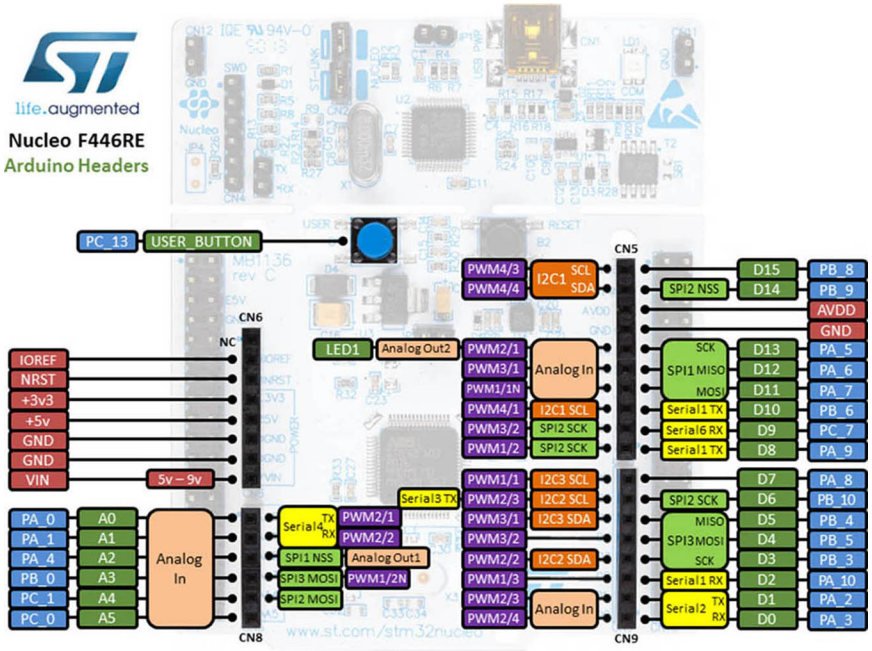


Morpho headers

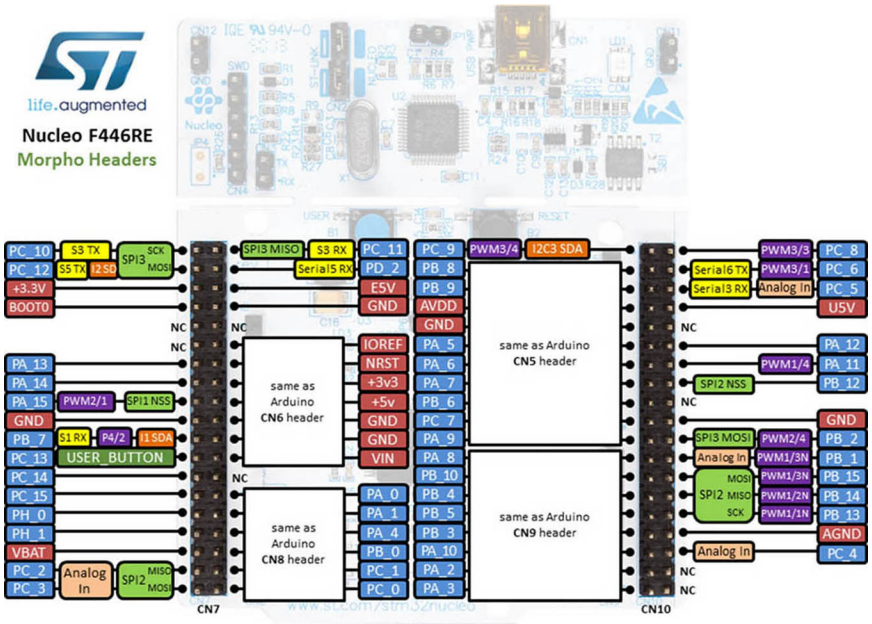


Nucleo-F446RE

Arduino compatible headers

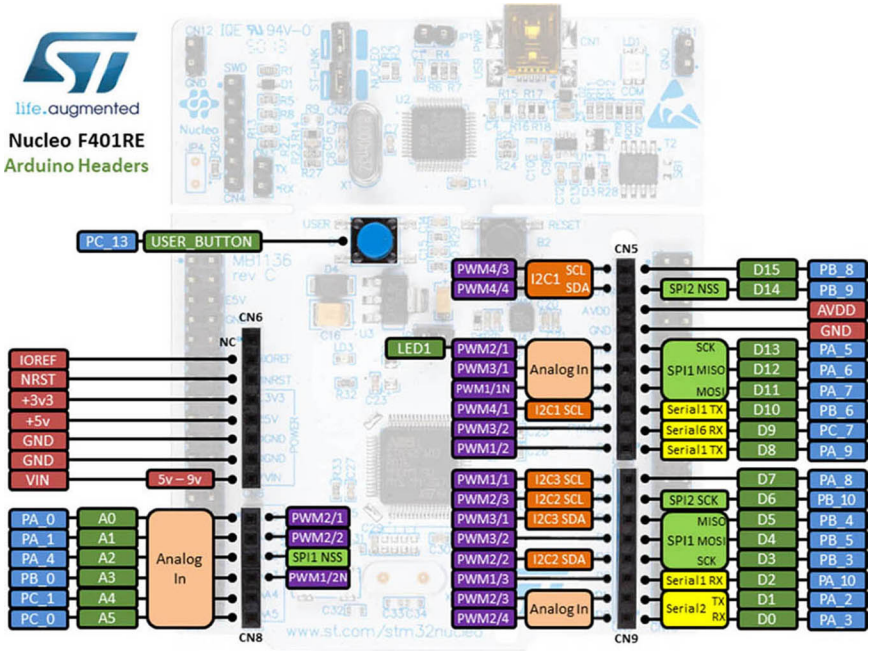


Morpho headers

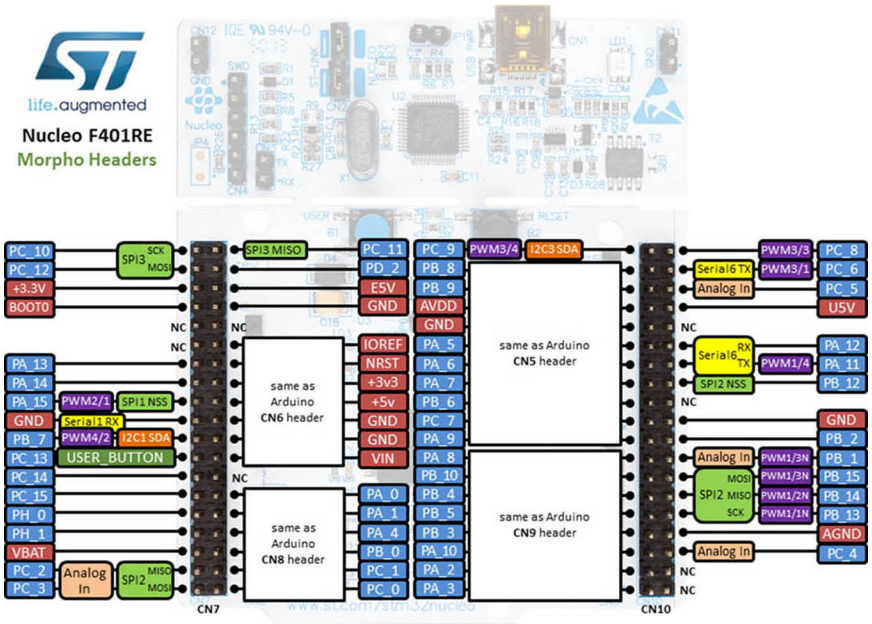


Nucleo-F401RE

Arduino compatible headers

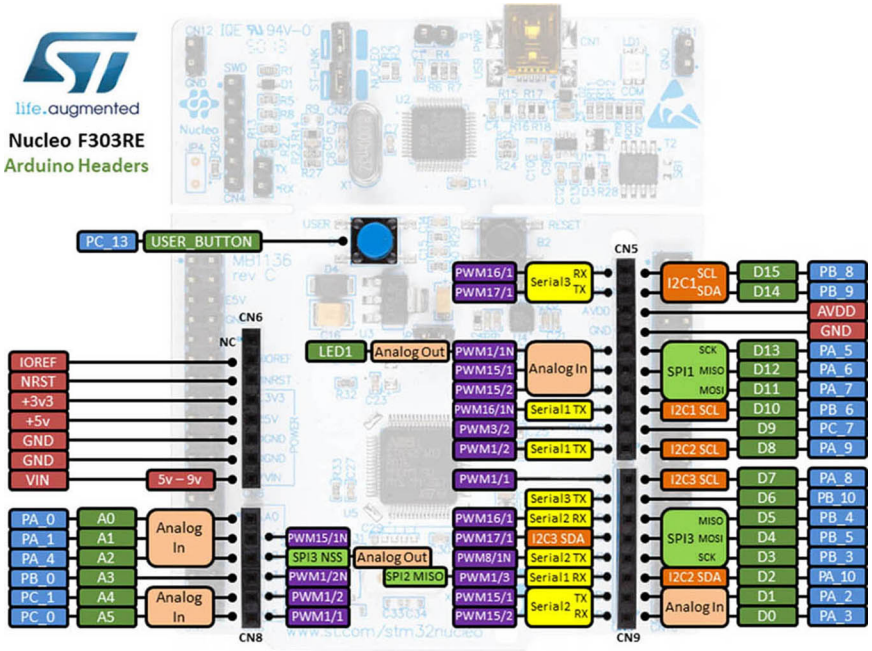


Morpho headers

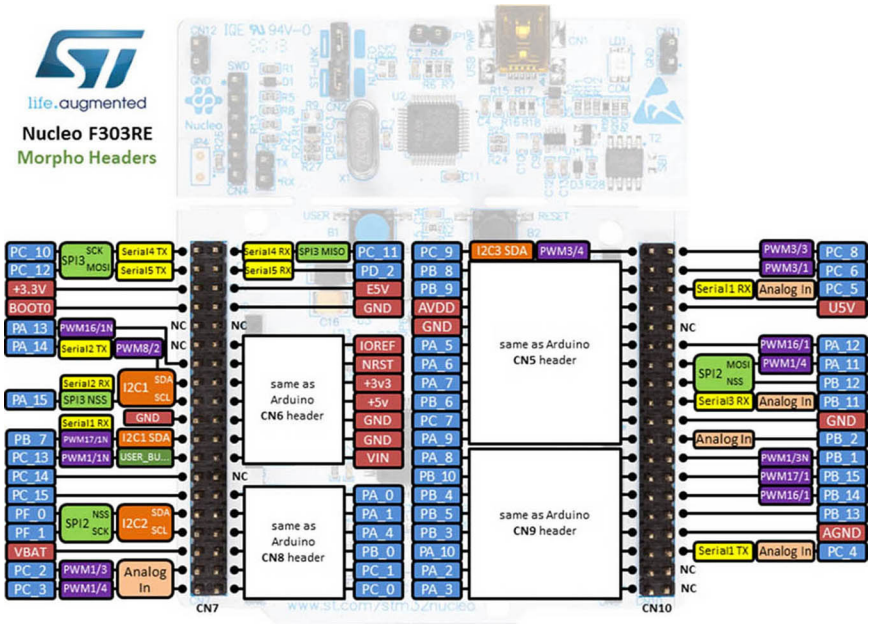


Nucleo-F303RE

Arduino compatible headers

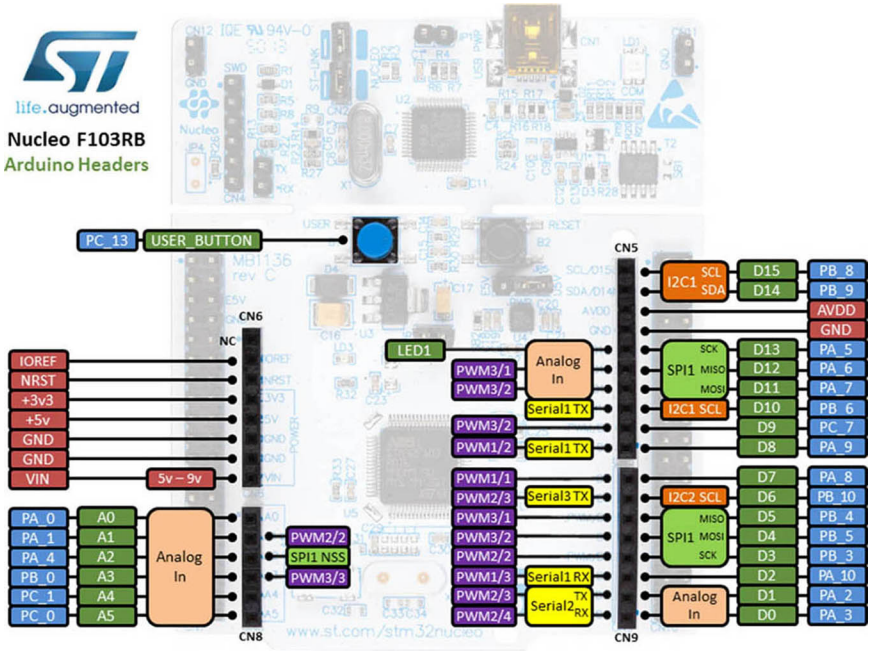


Morpho headers

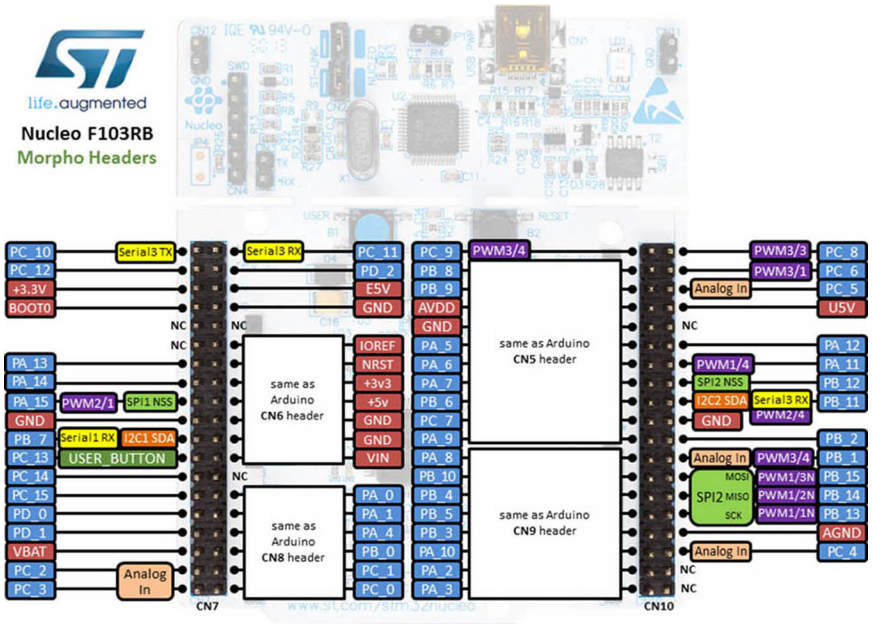


Nucleo-F103RB

Arduino compatible headers

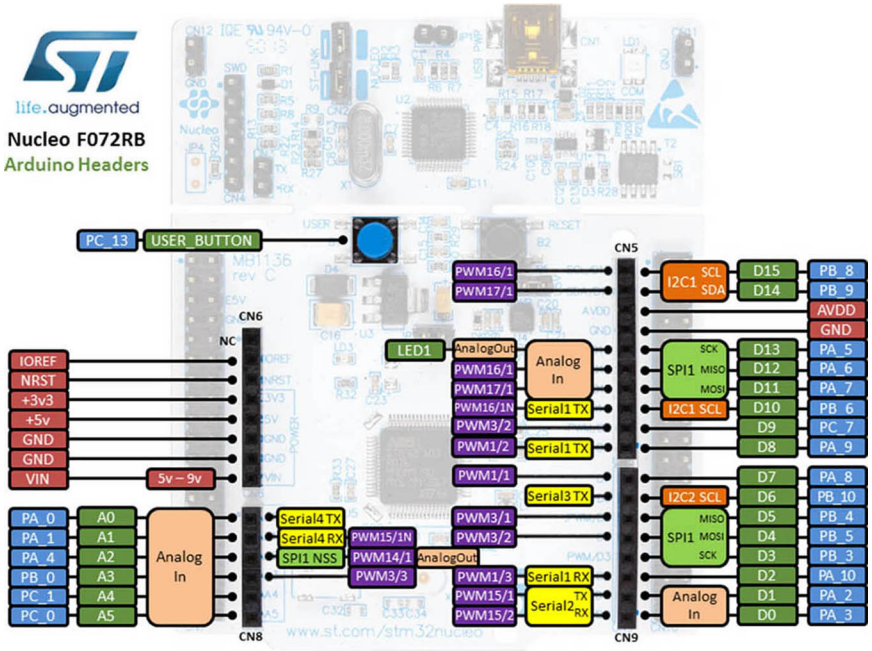


Morpho headers

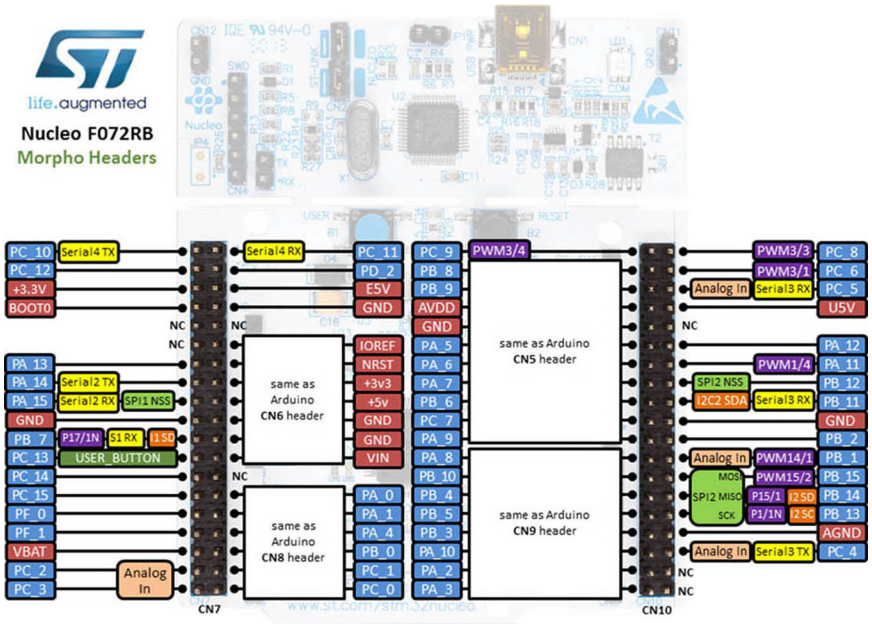


Nucleo-F072RB

Arduino compatible headers

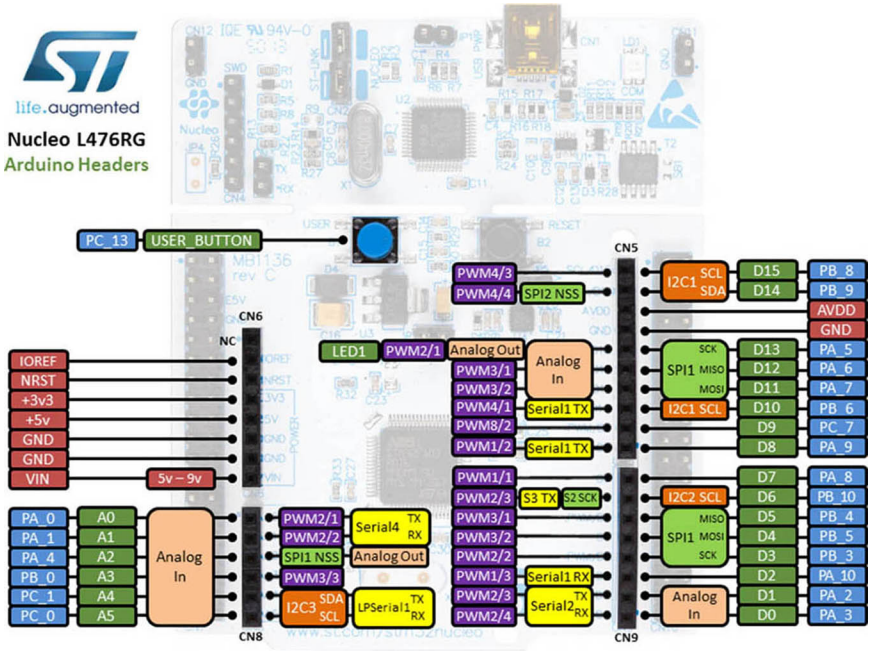


Morpho headers

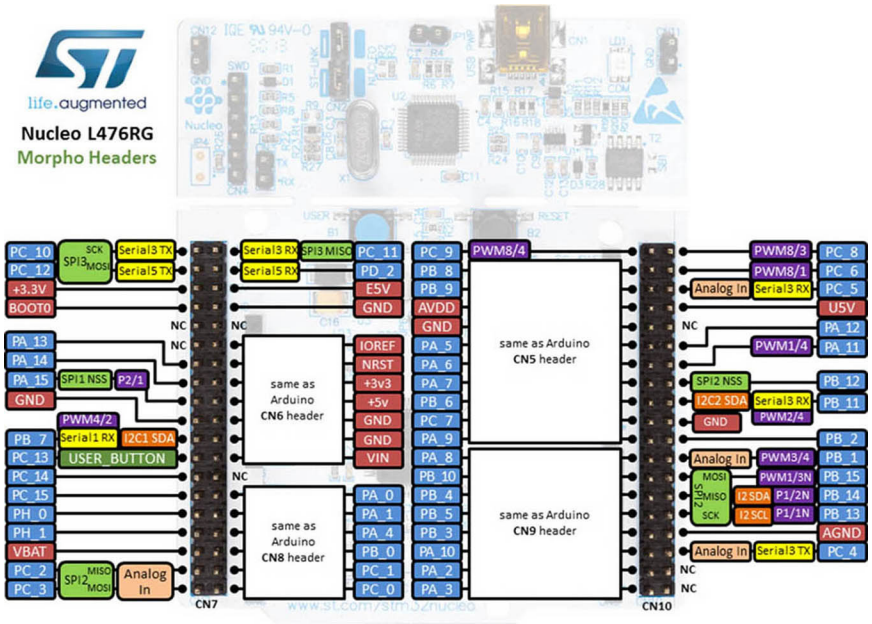


Nucleo-L476RG

Arduino compatible headers

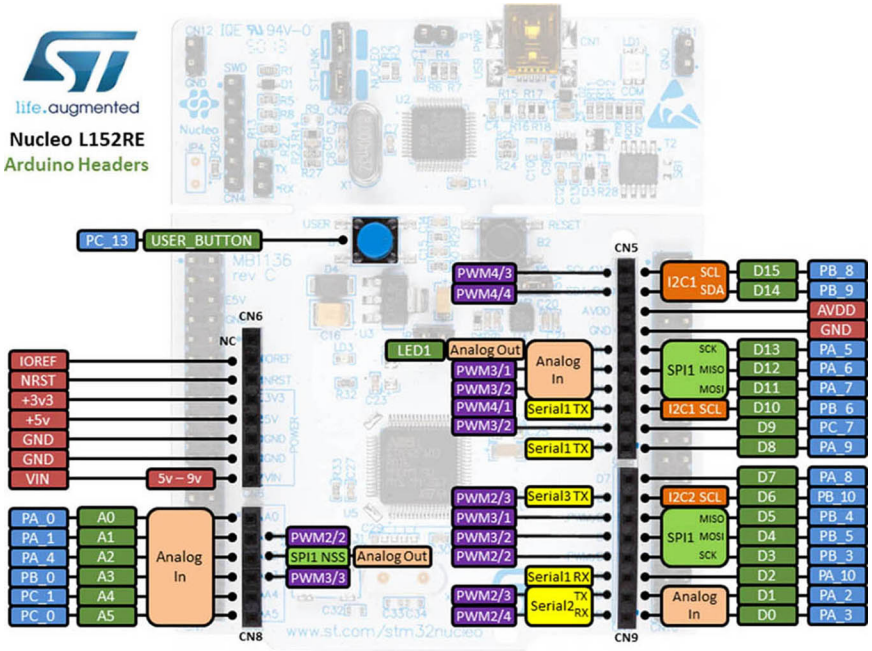


Morpho headers

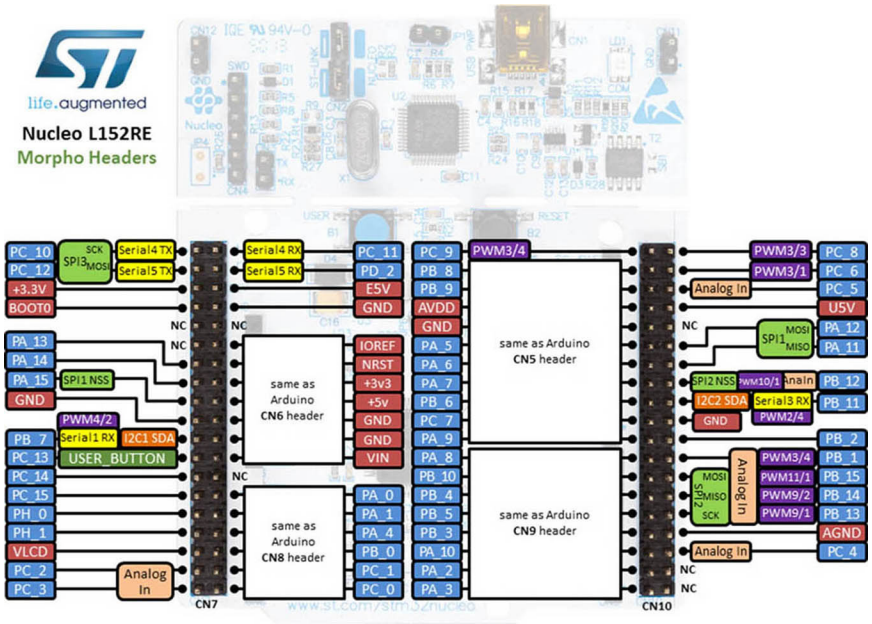


Nucleo-L152RE

Arduino compatible headers

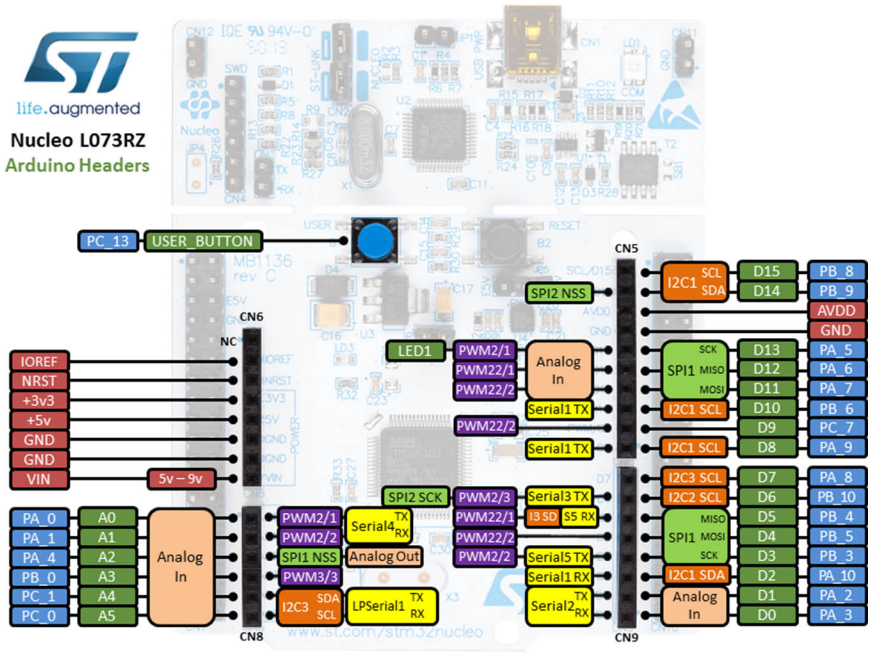


Morpho headers

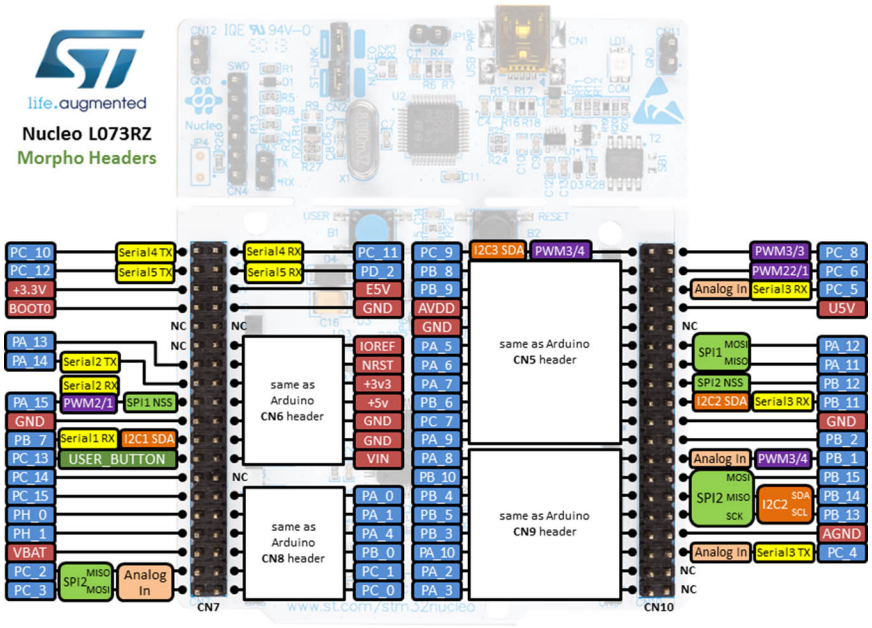


Nucleo-L073R8

Arduino compatible headers



Morpho headers



D. Differences with the 1st edition

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

Chapter 1

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

Chapter 2

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

Chapter 3 and 4

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

Chapter 5

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

Chapter 6

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

Chapter 7

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

Chapter 8

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

Chapter 9

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

Chapter 10

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

Chapter 11

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

Chapter 12-22

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

Chapter 23

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

Chapter 24

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

Chapter 25-26

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

Chapter 27

Chapters 27 is totally new.

Chapter 28

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