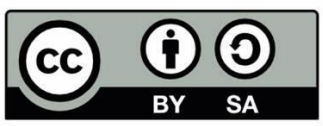
STM32C0 Register-Level Guide

Frank Bauernöppel

Draft

28 January 2025



1 TABLE OF CONTENTS

2 LIST OF FIGURES

3 INTRODUCTION

The main goal of this document is to provide introductory guidance for programming a STM32 microcontroller unit (MCU) at the register level in the C programming language. It does so by providing many ready-to-use code snippets showing selected use-cases for various hardware components in the microcontroller. The code uses the CMSIS core library [1] headers, adapted and provided by the MCU vendor. They have a very systematic and concise naming for the registers, bits and bit fields matching the naming in the reference manual [2].

Another goal was to keep this document concise. Therefore, it should always be used in conjunction with the original ST Microelectronics documentation. Of course, not every detail is reproduced or discussed here.

The STM32C0 series of microcontrollers was chosen, because it is as of today (2024) the most recent entry-level mainstream microcontroller series in the STM32 family. The series uses an Arm Cortex-M0+ processor core plus a number of representative ST peripherals in recent versions. Most examples can be easily translated to other STM32 series.

The first chapters containing the basics should be read in linear order and well understood before starting with the chapters on specific peripherals. Those chapters are relatively independent and can be read on demand in any order. Concepts like GPIO, clocks, interrupts and DMA will be used in several chapters however.

The author highly values your feedback on this guide, error reports and improvement suggestions. Simply open an issue in the git repository: https://github.com/FrankBau/stm32c0.

3.1 DISCLAIMER

The information in this document is based on the author's knowledge, experience and opinions. The methods described in this book are not intended to be a definitive set of instructions. You may discover other methods and materials to accomplish the same end result. Your results may differ.

There are no representations or warranties, express or implied, about the completeness, accuracy, or reliability of the information, products, services, or related materials contained in this book. The information is provided "as is," to be used at your own risk.

All brand names, trademarks, and referenced materials are the property of their respective owners and are used for descriptive purposes only.

4 GETTING STARTED

The primary entry point for all official ST Microelectronics documentation and tools is the website https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html. For downloading material, you need a freely available account on that website.

Reading and **understanding** the official documentation, especially the **reference manual** [2] and the **data sheet** [3] of the microcontroller are ***absolutely required*** for a successful software and hardware development process on the long run.

Understanding is the difficult part here, because the technical documents are terse and packed with acronyms. This guide is aimed to help you getting started in one single document.

4.1 HARDWARE

4.1.1 STM32C0116-DK Development Kit

The STM32C0116-DK Development Kit is primarily used throughout this guide:

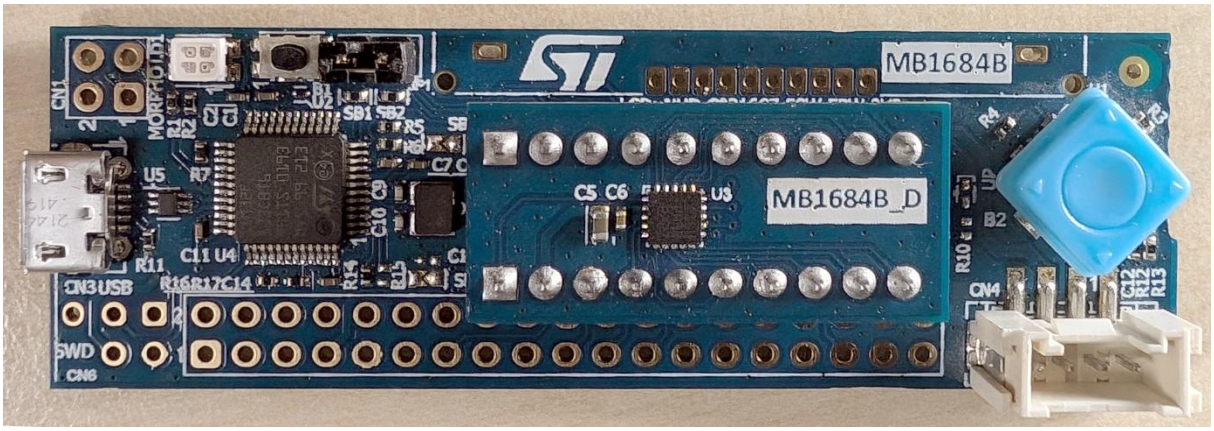


Figure 1: STM32C0116-DK with STM32C0116 MCU (center) and ST-LINK debugger (left).

Using a known-good low-cost development kit (DK) or Nucleo board helps to separate software and hardware issues during development. The **board user manual** [4] and **board schematics** [5] are the primary references for those boards.

Besides the MCU itself (chip U3 in Figure 1), these boards already contain a ST-LINK debugging interface (chip U4 on the left, close to the USB connector) which is essential for programming (flashing) and debugging the MCU from a host computer.

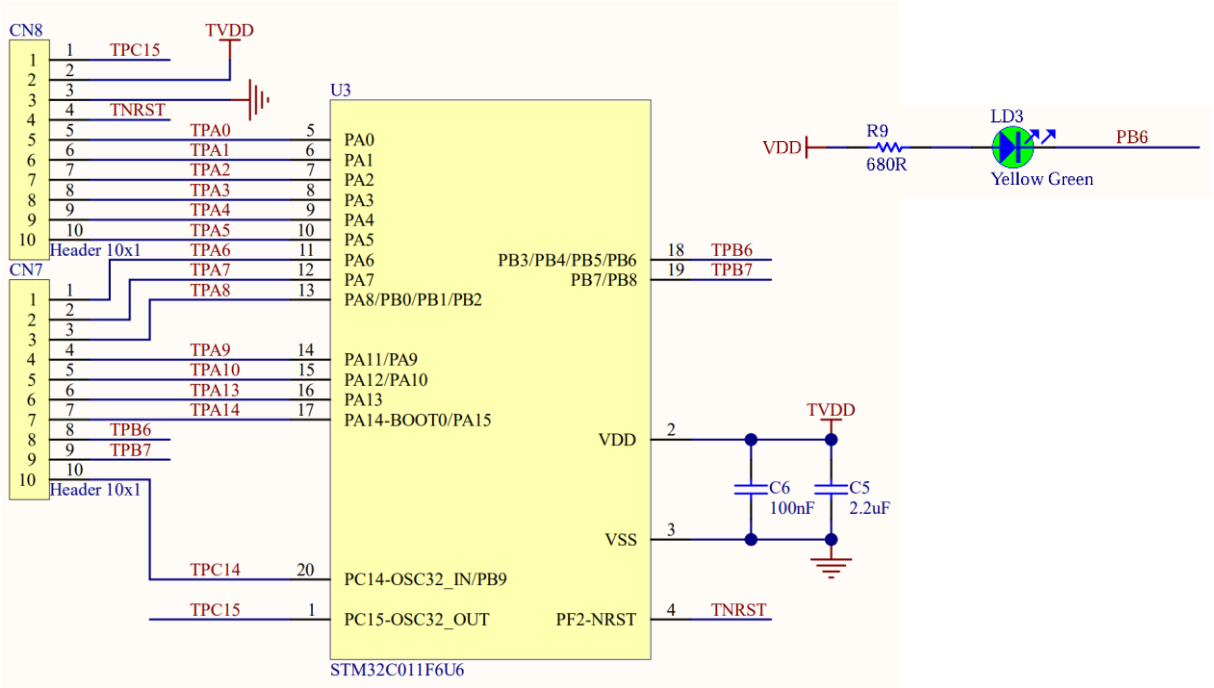


Figure 2: STM32C0116-DK board schematic cutouts, source: [5].

4.1.2 STM32C011F6 Microcontroller

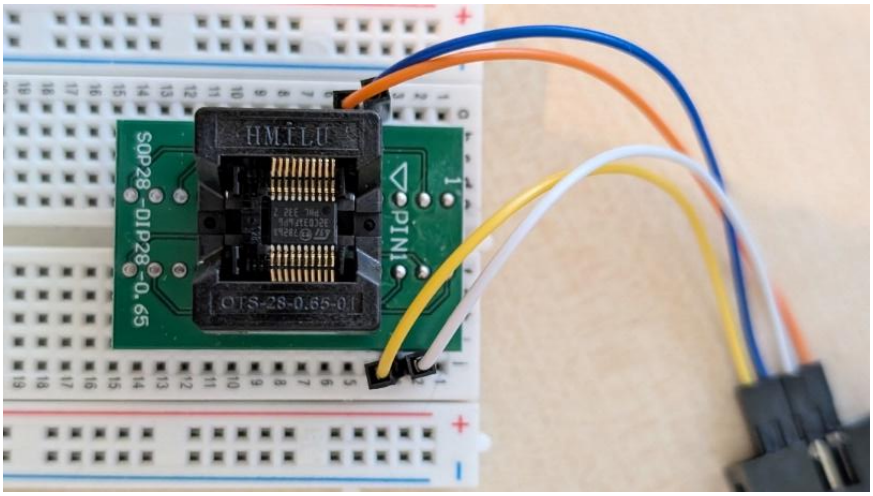


Figure 3: STM32C011F6 with connected VSS, VDD, and SWD debug interface in a test fixture.

When using a microcontroller unit (MCU) stand-alone, at least the following connections must be made:

- Power supply: **VDD** (min 2.0V; typ. +3.3 V; max. 3.6V) and **VSS** (GND). The power supply should be stabilized and filtered by two capacitors (100nF + 4.7µF) close to the chip pins. All VDD and VSS pins on the package must be connected for proper power supply.
- Single Wire Debug (**SWD)** interface (**SWCLK** and **SWDIO** signals, connected to an external **ST-LINK debug probe** for programming (flashing) the chip and debugging.

Strictly speaking, the debug interface is not necessary for MCU operation, but strongly recommended for the entire development process and ideally still accessible in the field.

Other special pins (depending on option byte settings and the STM32 series used):

- **NRST**: reset (bidirectional). Only needed if an external reset shall be implemented or the internal reset shall be monitored. MCU option bytes must be programmed for using NRST.
- **BOOT0** boot mode pin. Only accessible, when the MCU option bytes are programmed accordingly. Used for switching the boot mode for firmware updates in the field via external interfaces (USART, I2C, ...) when SWD is not to be used, see chapter 6.1.1.
- **External oscillators** for high-speed (HSE) and low-speed (LSE) clocks. These pins are not available for all chip packages. These are only needed when the on-chip **internal oscillators** (HSI, LSI) are not to be used. Good PCB routing and component selection for external oscillators are critical.

For reference schematics, PCB guidelines etc. always see the "**Getting started with STM32C0 Series hardware development**" application note [6].

4.1.3 Lab Equipment

We try to keep the entry threshold low. Most code snippets only require the on-board **LED** (see chapter) or the **virtual COM port** connection that comes with the ST-LINK USB connection and connects the serial **USART1** transmit (TX) and receive (RX) lines to the debug host PC, see chapter 9.1..

Recommended terminal programs for serial communication with the board are **TeraTerm** on Windows and **tio** on Linux.

A **logic analyzer** or a **scope** come in handy for observing the output of some code snippets, even if you are using only entry-level lab equipment.

4.2 Software Tools

Most of the underlying command line tools are open-source software and you will find other distributions ready for download and use. There may be minor differences, as ST Microelectronics has applied some specific patches, see STMicroelectronics/gnu-tools-for-stm32 (github.com).

4.2.1 Visual Studio Code with STM32 Extension

Visual Studio Code with the STM32 VS Code Extension is highly recommended and used throughout this guide as the Integrated Development Environment (**IDE**).

Version 2.1.1. of the extension was used for testing. Earlier releases had issues with generating broken startup files.

Check out the extension's website for installing the prerequisites:

- STM32CubeCLT,
- STM32CubeMX, and
- ST-MCU-FINDER.

Short videos explain the first steps with the STM32 VS Code Extension: [7], [8], and [9].

4.2.2 STM32CubeIDE

STM32CubeIDE is a mature and complex freely available IDE, based on the Eclipse IDE. STM32CuebIDE has the most recent and complete support for all STM32 MCUs including various advanced debugging aids. But, complexity has its price and nowadays many users prefer either professional (non-free) or lightweight free IDEs like Visual Studio Code. The

interested reader will find STM32CubeIDE documentation along with the software download on the STM32CubeIDE website. However, STM32CuebIDE is not used in this guide.

4.2.3 STM32CubeMX

STM32CubeMX is a GUI tool for project skeleton code generation. It can be very useful for pin and clock tree configuration when using HAL or low-level (LL) libraries. As we focus on register-level code here, STM32CubeMX is not used, but worth to be considered.

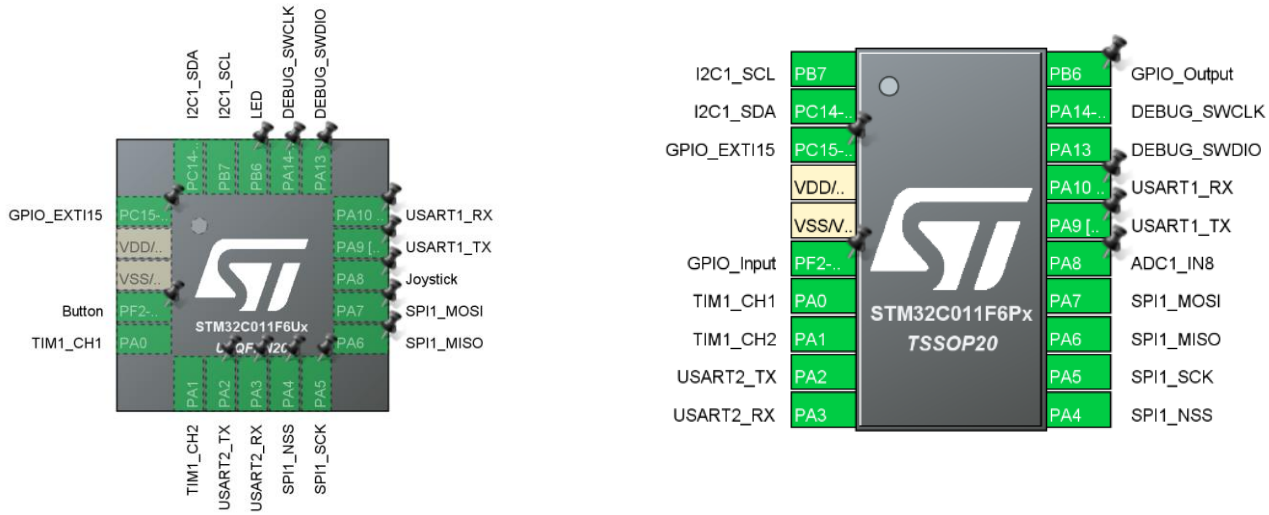


Figure 4: Pin planning with STM32CubeMX for STM32C011 MCUs.

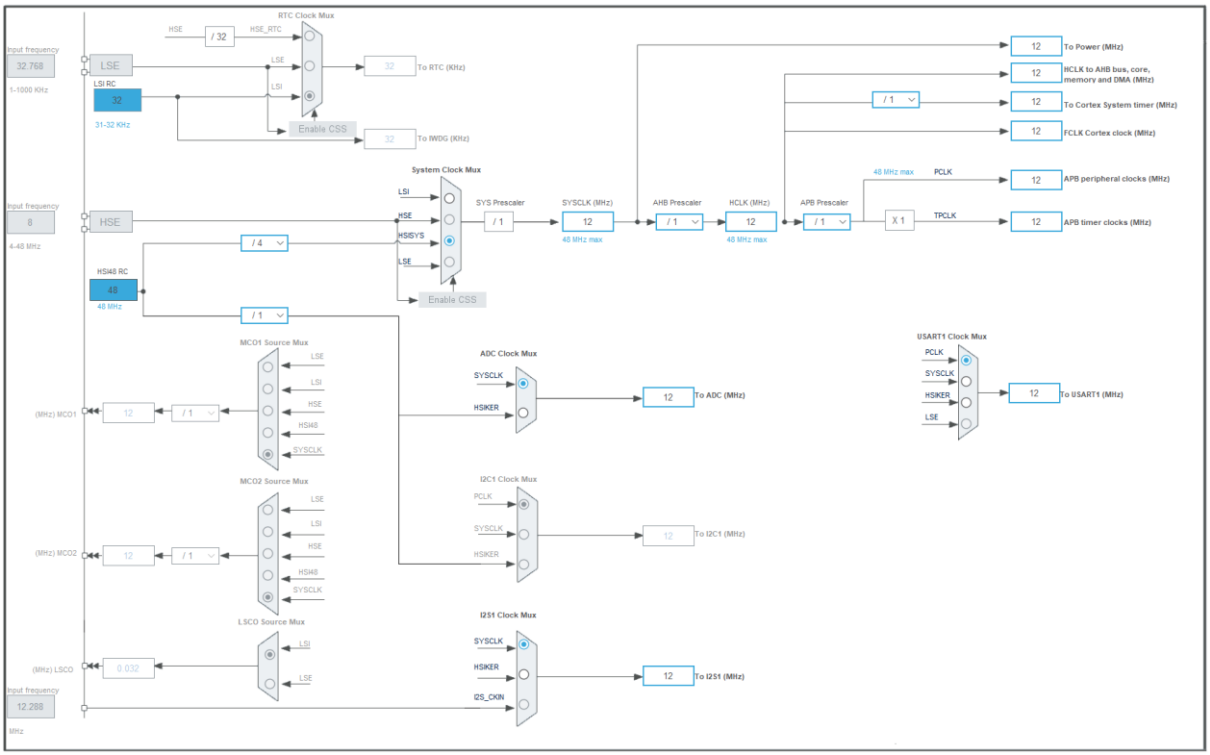


Figure 5: STM32CubeMX Clock Configuration View for a STM32C011 MCU.

4.2.4 STM32CubeProgrammer

This is a rather lightweight stand-alone GUI tool for programming the flash, the flash option bytes (chip configuration before the firmware takes control) and related tasks.

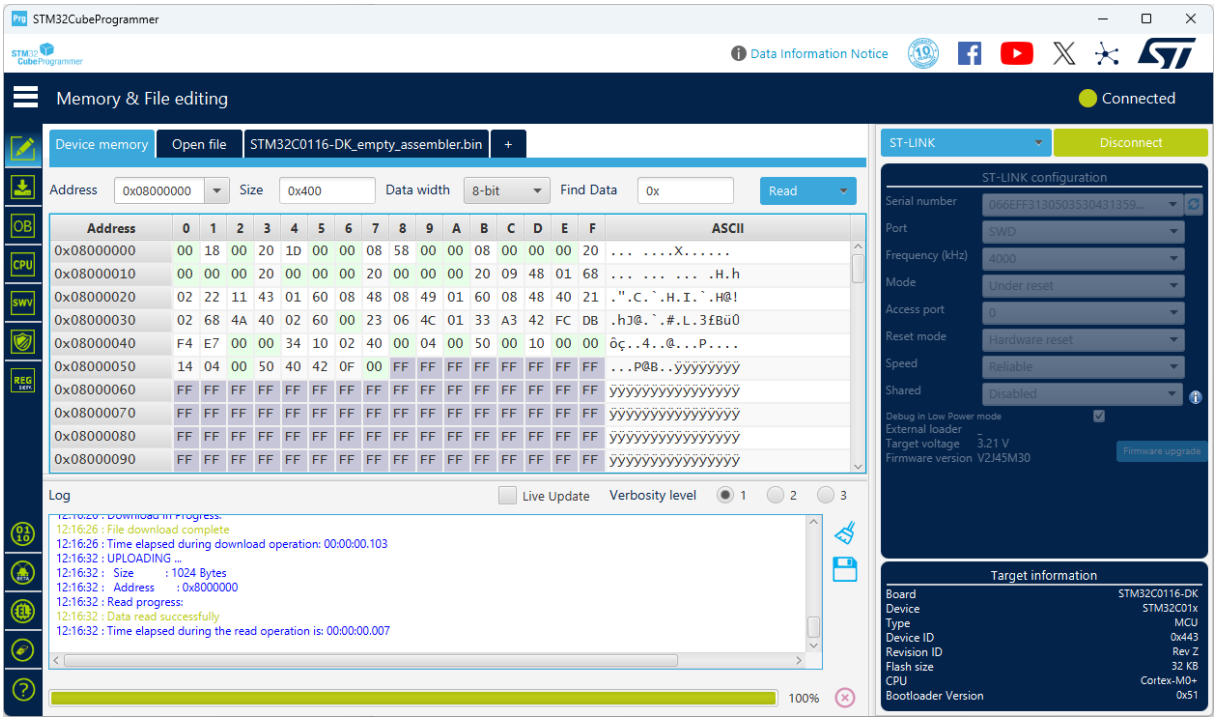


Figure 6: STM32CubeProgrammer with the flash image of a bare-metal blinky program.

Note that the concept of **flash programming** means uploading a binary image (binary machine code and data) to the flash of the MCU.

4.3 Developing Software

The code examples shown in the following chapters will use the **CMSIS Core Library** [1], which is extended for STM32C0 devices and bundled with more drivers, documentation, and example projects in form of the **STM32CubeC0 MCU Firmware Package** [10]. Similar repositories do exist for other MCU series.

Only the CMSIS **header files** (.h) will be used for the examples in this document, as they are a very thin layer of C defines and macros with a systematic naming scheme corresponding to the names used in the programming and reference manuals.

4.3.1 STM32CubeC0 MCU Firmware Package

STM32 firmware packages will be managed by STM32CubeMX when creating new projects.

The STM32C0 firmware package can be cloned directly from **github** to one new, central folder used by all projects:

```
# git clone https://github.com/STMicroelectronics/STM32CubeC0.git -branch v1.3.0 -recursive -depth=1
```

The overall folder structure of this repository is

```
STM32CubeC0/
├── CODE_OF_CONDUCT.md
├── CONTRIBUTING.md
├── Documentations                # STM32CubeC0GettingStarted.pdf
├── Drivers
├── LICENSE.md
├── Middlewares
├── Projects                      # Sample Projects (see STM32CubeProjectsList.html)
├── README.md
├── Release_Notes.html
├── SECURITY.md
├── Utilities
├── _htmresc
└── package.xml
```

all driver packages are in

```
STM32CubeC0/Drivers/
├── BSP                   # Board Support Packages for Nucleo... boards, not used here
├── CMSIS                 # CMSIS headers for register-level access
└── STM32C0xx_HAL_Driver  # HAL and LL (low-level) drivers, not used here
```

and among them the relevant CMSIS headers in

```
STM32CubeC0/Drivers/CMSIS/
├── ARM.CMSIS.pdsc
├── Core                  # CMSIS core headers (Cortex-M CPU core related)
├── Core_A
├── DAP
├── DSP
├── Device                # CMSIS device specific headers (for peripherals)
├── Documentation
├── Include
├── LICENSE.txt
├── NN
├── RTOS
└── RTOS2
```

A stripped down version is already included in the Examples repository:
https://github.com/FrankBau/stm32c0

Two subfolders **must be on the include path** for all examples in this guide:

- `STM32CubeC0/Drivers/CMSIS/`**`Core/Include`**
- `STM32CubeC0/Drivers/CMSIS/`**`Device/ST/STM32C0xx/Include/`**

4.3.2 Blinky – The Embedded "Hello World"

We start our journey with a first project here, but postpone the details of the register-level coding to the following chapters. The focus in this chapter is the overall project structure, not the details in the code.

First, start Visual Studio Code with the STM32 extension installed (chapter 4.2.1) and use the extension feature "**Create empty project**" to create a new project called "blinky". Choose the

STM32C0116-DK as the evaluation board and the **debug** configuration for the project. The project structure should now look like this:

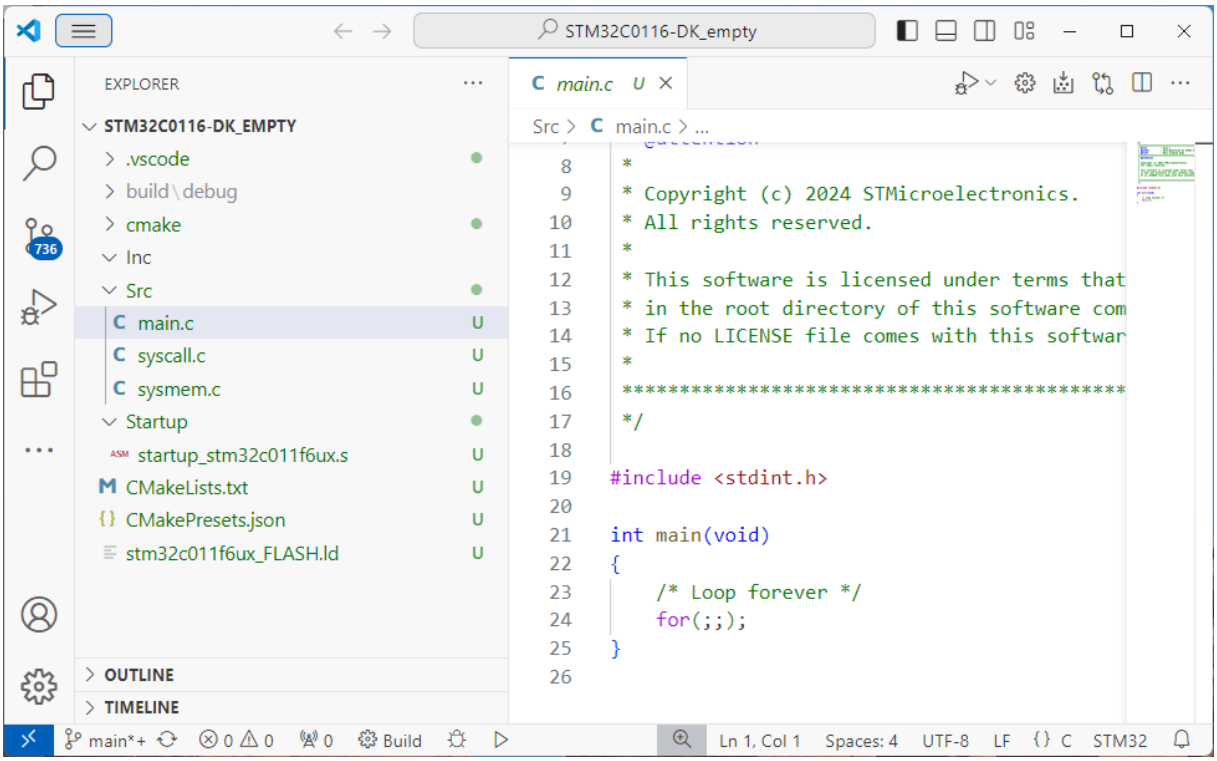


Figure 7: Visual Studio Code with a STM32 empty project.

The following table explains the created files and folders in greater detail.

File / Folder	Usage
.vscode/	VS Code specific configuration files
.vscode/tasks.json	Configuration for building (compile, link, download) the project. Do not edit in the beginning.
.vscode/launch.json	Configuration for debugging the project. Do not edit in the beginning.
build/	Folder with build artefacts like object and executable files. Can be cleared and re-built at any time.
cmake/	Folder with configuration files for the CMake build system Do not edit.
Inc/	Folder for include (.h header) files
Src/	Folder for source code (.c) files
Src/main.c	The main C entry code (main function) start writing code here.
Src/syscall.c	Stub functions which may be implemented to connect the C runtime to the system (system calls) for file IO, date&time, etc.. Do not edit in the beginning.
Src/sysmem.c	Implementation of the _sbrk function which will eventually be called my malloc and other heap functions when more heap memory is requested. Do not edit in the beginning. May need adaptions when using an RTOS etc..
Startup/startup_stm32c011f6ux.s	The interrupt vector table (see chapter) and the very first assembly instructions before the C function main() is called and executed. Note: The **interrupt vector table is wrongly generated** in versions 2.0.0 to 2.1.0 of the STM32 VS Code Extension, **do not use interrupts without correcting it!** See chapter 20.1.
stm32c011f6ux_FLASH.ld	The linker description file. It tells the linker where and in what order to place code and data. May be changed for special requirements, like using parts of the RAM for code or flash pages for persistent parameter storage. Do not edit in the beginning.
CMakeLists.txt	Contains the configuration for CMake: • additional include folders • additional source files **To be edited.**
CMakePresets.json	Choices of different build configurations. Do not edit.

Note, that the folder structure might be slightly different when you let STM32CubeMX create a local copy of the STM32CubeC0 package in your new project.

Edit **CMakeLists.txt** as shown below for including the relevant CMSIS headers from the STM32CubeC0 MCU Firmware Package. We are using the folder ../Drivers/CMSIS relative to the project root folder. Adapt this to your needs.

```
# Include directories for all compilers
set(include_DIRS
    ${CMAKE_CURRENT_SOURCE_DIR}/../Drivers/CMSIS/Core/Include
    ${CMAKE_CURRENT_SOURCE_DIR}/../Drivers/CMSIS/Device/ST/STM32C0xx/Include
)
```

Two CMSIS folders must be added to the include path: one for the Cortex®-M0+ **Core**, and the other for the specific microcontroller **Device**. Note that the CMIS core library is a header-only library and there is no need to add any source code .c files from there.

Here is the same project with the MCU specific header file `stm32c011xx.h` included, and after adding some blinky code to main.c:

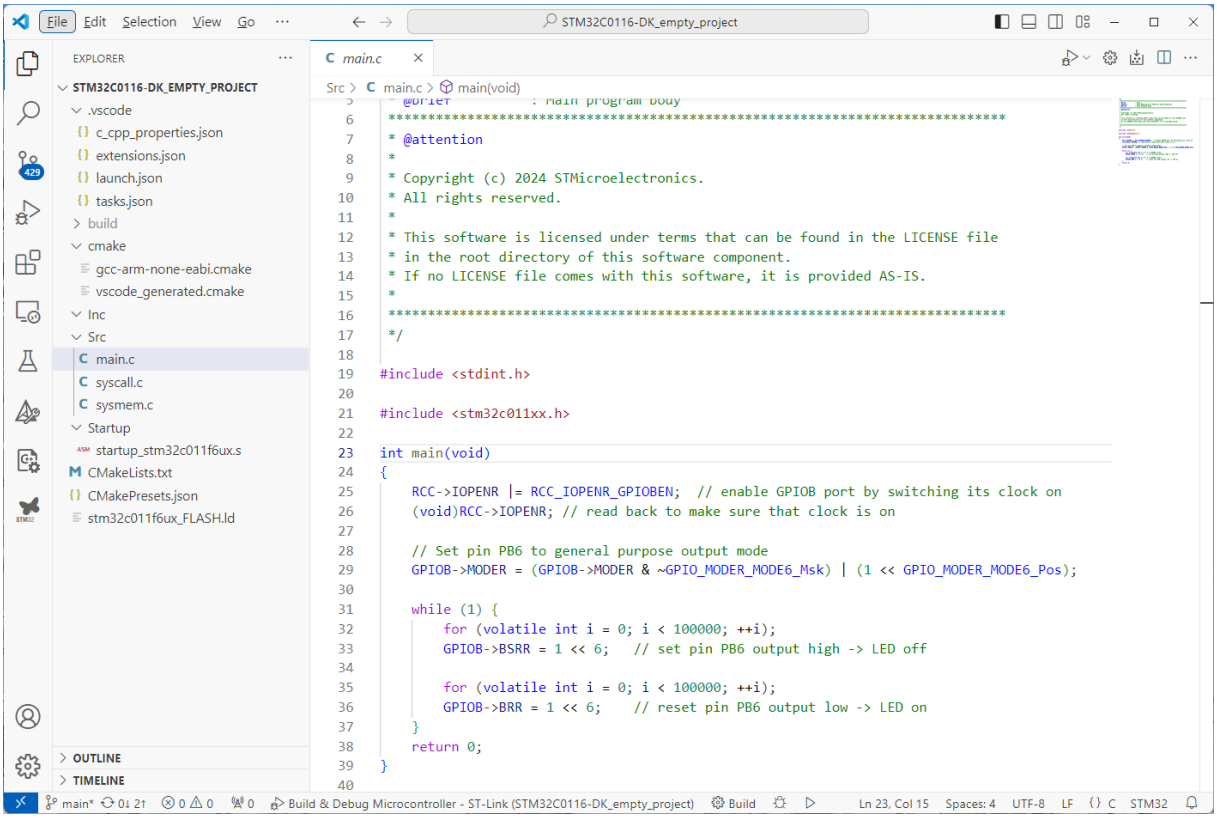


Figure 8: Visual Studio Code with a STM32 blinky project.

As said before, you are not expected to understand all details in that code right now, but should recognize the **main function** with **initialization code** (setup) followed by the **endless main loop** which does the blinking. Consult Figure 3 for the relevant board schematics with the MCU and the LED.

For easier copy & paste the complete content of main.c follows:

```c
#include <stm32c011xx.h>
int main(void)
{
    // LED pin setup
    RCC->IOPENR |= RCC_IOPENR_GPIOBEN;  // enable GPIOB port by switching its clock on
    (void)RCC->IOPENR; // read back the register to make sure that the clock is now on

    // set the pin PB6 to general purpose **output** mode (which is mode 1)
    GPIOB->MODER = (GPIOB->MODER & ~GPIO_MODER_MODE6_Msk) | (1 << GPIO_MODER_MODE6_Pos);

    // loop forever
    for(;;) {
        for (volatile int i = 0; i < 100000; ++i);      // some delay
        GPIOB->BSRR = 1 << 6;                           // set pin PB6 output high -> LED off

        for (volatile int i = 0; i < 100000; ++i);      // some delay
        GPIOB->BRR = 1 << 6;                            // reset pin PB6 output low -> LED on
    }
    return 0;   // unreachable code, main shall never return in embedded software
}
```

You should be able to **build** this project now. During the build process, several files are created. The **final output** in the build\debug folder consists of the following files (not all are needed):

File Type	File Name (example)	Purpose
.elf	blinky.elf	Binary file with all machine code, data, and auxiliary information which is used by the debugger or the programmer for programming the flash memory in the MCU chip.
.bin	blinky.bin	Binary image (copy) of the MCU flash content. Stripped-down and relocated version of the .elf file. Not used for debugging, but can be used for programming the MCU using the ST-LINK file drop feature or in a production line.
.hex	blinky.hex	ASCII text version of the .bin file which is safer for file exchange, e.g. when sending the image file to a sub-contractor for programming.
.map	blinky.map	Memory layout map, mapping of source code and data items to MCU memory addresses in text form for reference.

When using an Integrated Development Environment (IDE), you don't have to touch these files directly.

Ideally, you have a STM32C0116-DK board attached to a USB port of your development host and are now able to start **debugging** or **running** the program and watch the LED blinking before continue reading.

If you prefer command line builds, clone the repository https://github.com/FrankBau/stm32c0, navigate to a project, and issue from a command prompt:

```
    $ cmake -G Ninja -B build

    $ cmake --build build
```

The final `build/*.bin` binary image file can be simply flashed into the board by dragged it onto the new USB drive that appears when you connect the board via USB to the build host. But, of course, interactive debugging can give you much deeper insights.

4.4 RECOMMENDED DOCUMENTATION

Document	What is it good for?	Version used
Reference Manual	Reference of all STM32 peripheral components in the MCU with all registers, bits and bytes. Primary reference for programming the peripherals at register level.	RM0490 Reference manual „STM32C0x1 advanced Arm®-based 32-bit MCUs” RM0490 Rev 3 December 2022 825 pages [2]
Programming Manual	Description of programming model, instruction set, and core peripherals of the Cortex®-M0+ processor core and core peripherals (NVIC, STK, SCB, MPU).	PM0223 programming manual for Cortex®-M0+ core PM0223 - Rev 9 - June 2024 80 pages [11]
Data Sheet	Chip specific information: functional overview, pinout, pin functions and assignments, electrical characteristics and package information	STM32C011x4/x6 Data Sheet DS13866 Rev 4 January 2024 92 pages [3]
Errata Sheet	Description of the known device errata, with respect to the device datasheet and reference manual. Remember the errata sheet when desperately hunting bugs.	STM32C011x4/x6 Errata Sheet ES0569 - Rev 4 - June 2023 17 pages [12]
“Hello and welcome …” training material	Training presentations for many peripherals explaining main use cases, slides with short explanations	Various documents, see [13]
STM32CubeMX User Manual	Learn how to use STM32CubeMX for generating project skeletons.	UM1718 STM32CubeMX for STM32 configuration and initialization C code generation Rev 45 - June 2024 522 pages [14]
STM32CubeCLT Installation Guide	Command-line toolset for use by the VS Code STM32 Extension: GNU-tools-for-STM32, CMake, Ninja, STM32CubeProgrammer, STMicroelectronics_CMSIS_SVD	UM3089 User manual STM32CubeCLT installation guide Rev 3 - March 2024 25 pages
Board Schematics	Understand the MCU pin wiring, connectors and additional board components in detail	MB1684-C011F6-B01 Board schematic v1 20/10/2021, 5 pages [5]
Board User Manual	Board features, connectors, solder bridges, options, power supply and external connections	UM2970 User manual Discovery kit with STM32C011F6 MCU Rev 2 - November 2022 26 pages [4]
Getting Started with Hardware Development	Systems design principles: power supply, clock management, reset control, boot mode settings and debug management, Reference schematics and PCB layout guidelines	AN5673 Application note Getting started with STM32C0 Series hardware development Rev 2 - December 2022 32 pages [6]

4.5 HELPFUL RESOURCES

- STM32 Arm® Cortex® MCU wiki https://wiki.st.com/stm32mcu/
- STMicroelectronics Community Forum https://community.st.com/
- STM32 Education Landing Page https://www.st.com/content/st_com/en/support/learning/stm32-education.html
- STM32 gotchas by efton http://efton.sk/STM32/gotcha/index.html

5 STM32C0 SYSTEM ARCHITECTURE

The system architecture of the STM32C0 series is relatively simple compared to the other STM32 MCU series. This makes this series especially well-suited for this introductory guide.

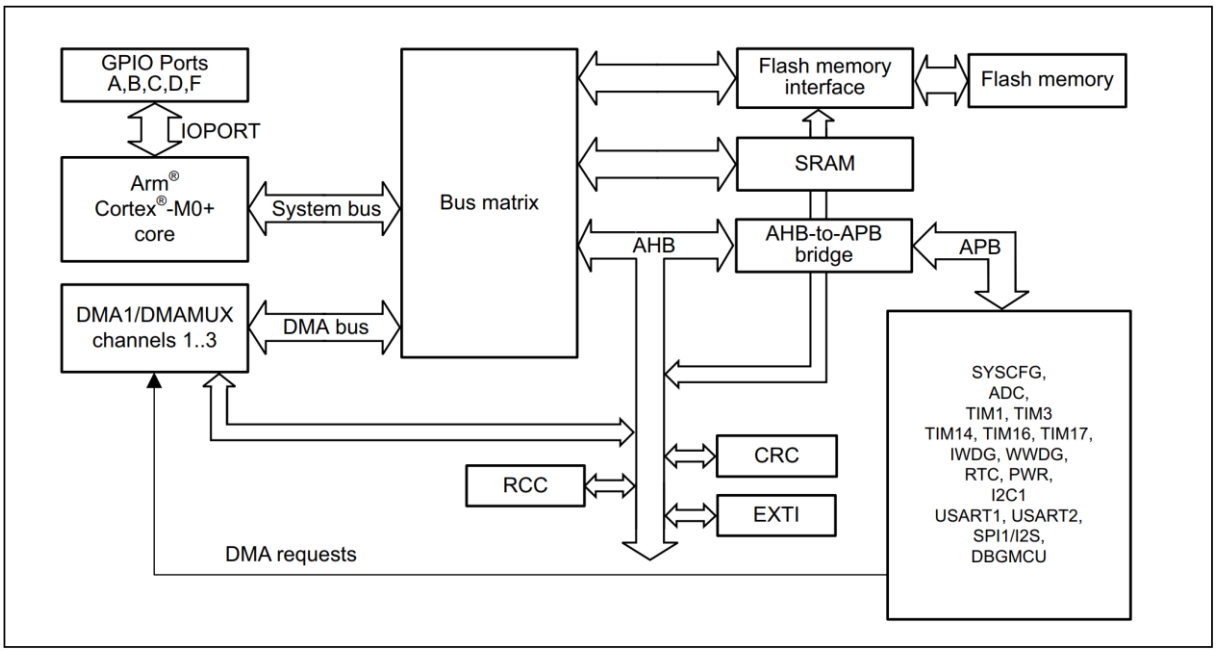


Figure 9: STM32C0 System Architecture, source: reference manual [2].

There are two active components (**bus masters**) in the MCU that can initiate data read and write transfers:

- the **Arm Cortex-M0+ core** fetching instructions and data IO,
- one **Direct Memory Access (DMA) unit**, see chapter 10.

There are three passive components (**slaves**) serving the read / write requests of the bus masters:

- the **SRAM**: mainly the program data storage, **volatile** (holds content as long as power is supplied)
- the **Flash**: mainly the program code storage, **persistent** (holds content while powered off)
- the **AHB** (Advanced High-Performance Bus) with some high-bandwidth demanding **peripherals** attached. One of them is the **APB** (Advanced Peripheral Bus) bridge to the APB bus which by itself has several less bandwidth demanding **peripherals** attached.

Note that the GPIO ports controlling the MCU pins are directly connected to the CPU core, and not to the bus matrix. This is typical for Arm Cortex-M0+ designs, allowing faster GPIO access. As a drawback, the GPIO registers cannot be reached by DMA. This is easily overlooked and may then cause serious headaches.

The MCU internal **peripherals** (like I2C, SPI,...) need to have their peripheral **clock switched on** before they can be used. Default state is “clock off” for power saving. Switching the clocks, resetting peripherals and related stuff is configured in the Reset and Clock Control (RCC) unit, see chapter 17.

Here is another, more detailed block diagram from the data sheet [3]:

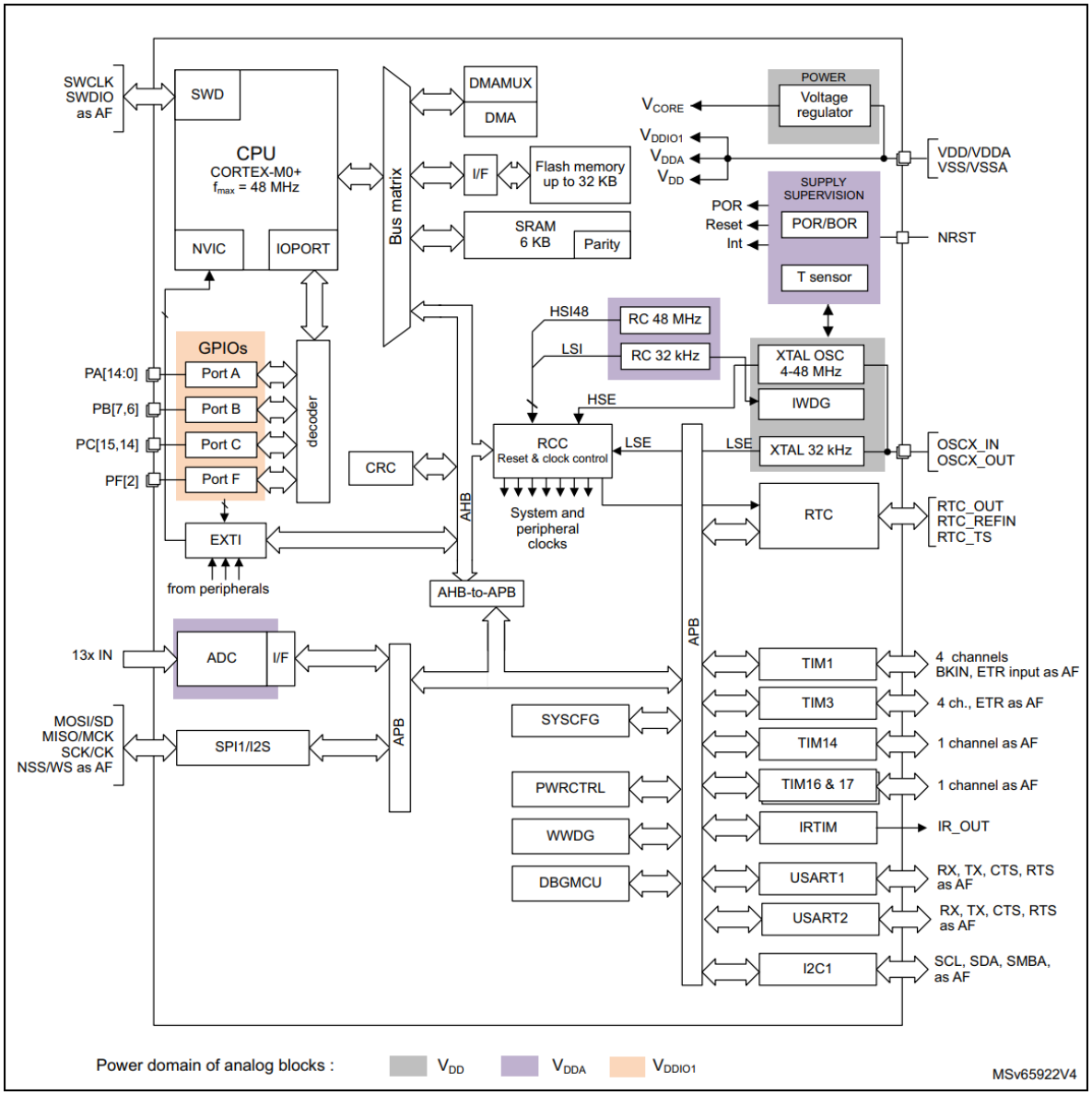


Figure 10: STM32C011 Block Diagram, source: data sheet [3].

6 STM32C0 MEMORY MAP

The 32-bit MCU architecture allows for a very regular and systematic addressing scheme. All memories (flash, SRAM) and peripheral registers are organized in a single address space:

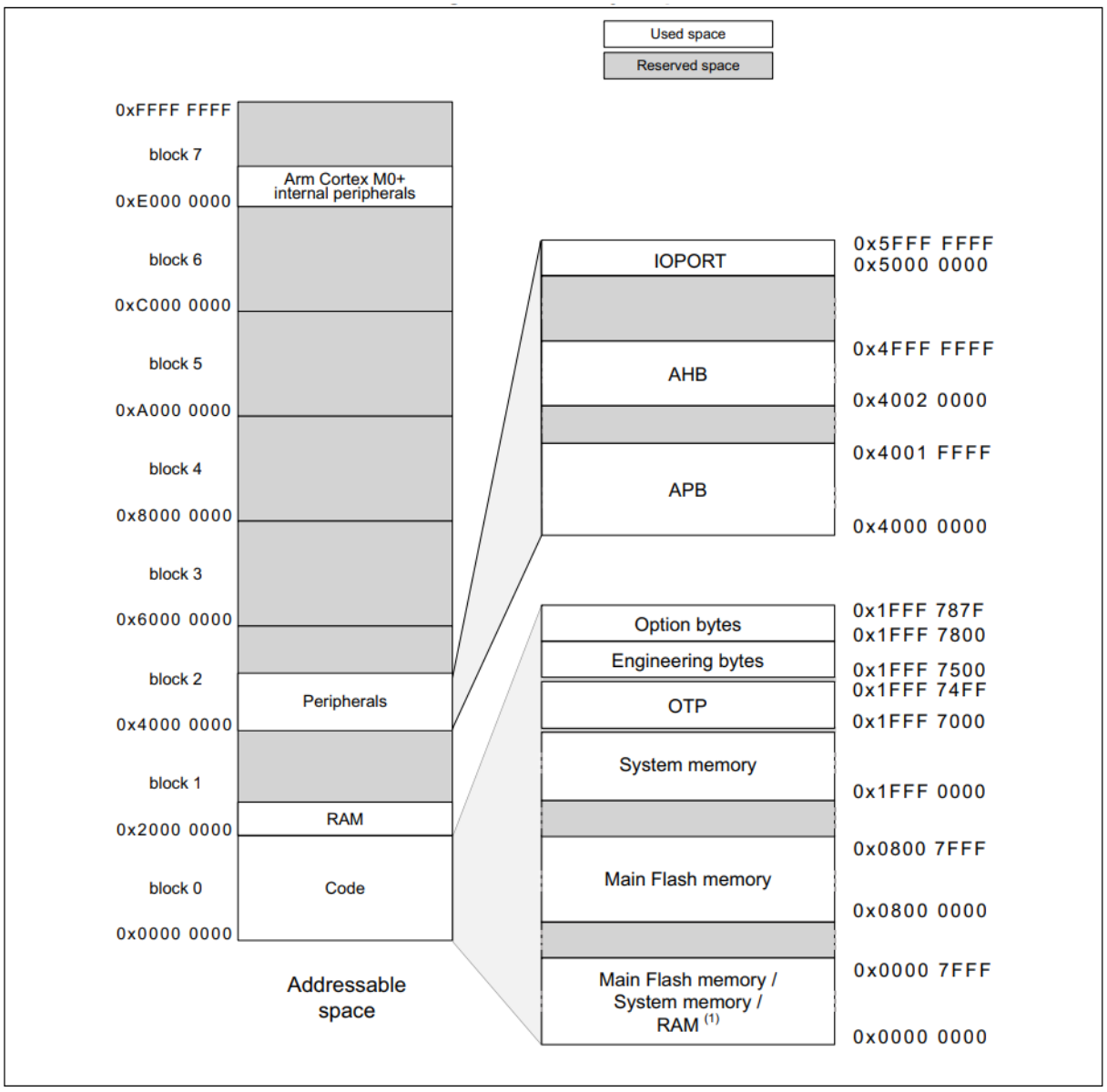


Figure 11: STM32C011 Memory Map, source: reference manual [2].

Caution: read or write attempts to **unmapped** (grayed-out) addresses, or peripheral addresses while the peripheral clock is not switched on, may lead to unwanted behavior like bus faults, hard faults, or processor lockups.

6.1 MEMORY BLOCKS

The most-significant (top-most) address bits select one of the following address blocks:

- Code (starting at 0x0000 0000),
- SRAM (starting 0x2000 0000),
- STM peripherals (0x4000 0000), and
- Arm Cortex-M0+ Core peripherals (0xE000 0000).

Within each block, one or more hardware components of the MCU can be addressed.

Therefore, during a debug session, it is common to see

- 0x2000.... addresses for variables (data) in the SRAM,
- 0x0800.... addresses for code in the flash (jump targets, function calls, ...), and
- 0x4000.... and 0xE000.... addresses when accessing peripheral registers (MMIO).

6.1.1 The Code Block

The Arm Cortex-M0+ core (CPU) always starts reading from memory address 0x0000 0000 and 0x0000 0004 as the very first steps after a reset, like when powering on. There are three options, which hardware block the CPU will really see when accessing this address range:

1. the **Main Flash Memory**, or
2. some read-only code containing a boot loader called the **System Memory**, or
3. the **SRAM**.

This is called **aliasing**, which means redirecting the read/write accesses from the address range starting at 0x00000000 to the configured hardware block.

Usually, the code block 0x00000000 .. 0x00007FFF is aliased to the **main flash memory**. Then, after a reset, the code in the flash is executed as expected.

For firmware upgrades or other special purposes, the internal boot-loader (**System Memory**) or the **SRAM** can be aliased at address 0x00000000. In older STM32 series a boot mode pin was required to select this **boot mode**. This is now optional and can be programmed internally using the **option bytes**, for example with the STM321CubeProgrammer (chapter 4.2.4). The reference manual [2] and application note AN2606 [15] have all details on that.

Besides of aliasing, these hardware blocks are **always** visible in their native address ranges.

Main flash programming is discussed in chapter 17.1. Other parts in the code block are:

- **One-time Programmable Fuses** (**OTP**) for storing one-time programmable user specific data like serial numbers, production data, personalized IDs, cryptographic keys and certificates,
- **Engineering Bytes** - factory-programmed calibration data, e.g. for the ADC (chapter 14) and other (mostly undocumented) chip specific data, and
- **Option Bytes** –user programmable MCU configuration data, see chapter 17.2.

6.1.2 The SRAM Block

The SRAM block is used for the Static RAM (Random Access Memory) holding the **data** while the power supply is on. Similar to the other blocks, only a part of the block is **mapped** to existing RAM cells, starting at the base (lowest) address 0x20000000. For the STM32C011x4/x6 devices with 6 kB of RAM (0x1800 bytes), the highest mapped address is 0x200017FF. All higher addresses are **unmapped** and must never be accessed.

The Arm Cortex-M0+ Core Peripherals Block

The Arm Cortex-M0+ Core peripherals belong to the Arm Cortex-M0+ core design and are briefly described in chapter 20 and in full detail in the **programming manual** [11].

The Arm Cotrex-M0+ core peripherals are:

- Nested Vector Interrupt Controller (**NVIC**), see chapter 20.1
- SysTick timer (**STK**), see chapter 20.2
- System Control Block (**SCB**), and
- Memory Protection Unit (**MPU**)

6.1.3 The STM32 Peripherals Block

The STM32 Peripherals (like I2C, SPI, …) make the MCU a true STM32 MCU. These peripherals are described in the following chapters and in full detail in the **reference manual** [2].

Every peripheral (STM32 and Core) is exposed in the 32-bit address space through a group of **memory-mapped registers**. Writing to or reading from these registers affects the peripheral and often causes **side effects** like transferring data on a peripheral bus, configuring some clock signal, or starting a timer.

Let's for example study the GPIO Port B (**GPIOB**) with its Mode Register (**MODER**) and Output Data Register (**ODR**). By writing the appropriate values to the GPIOB_MODER and GPIOB_ODR registers, one can, on the STM32C0116-DK board, switch the on-board LED on and off. This on-board LED is connected to pin PB6, see chapter 4.1.2.

Extracts from the reference manual [2]:

Table 2. STM32C0x1 peripheral register boundary addresses

Bus	Boundary address	Size	Peripheral	Peripheral register map
-	0xE000 0000 - 0xE00F FFFF	1MB	Cortex®-M0+ internal peripherals	-
IOPORT	0x5000 1800 - 0x5FFF 17FF	~256 MB	Reserved	-
	0x5000 1400 - 0x5000 17FF	1 KB	GPIOF	*Section 6.4.12 on page 160*
	0x5000 1000 - 0x5000 13FF	1 KB	Reserved	-
	0x5000 0C00 - 0x5000 0FFF	1 KB	GPIOD	*Section 6.4.12 on page 160*
	0x5000 0800 - 0x5000 0BFF	1 KB	GPIOC	*Section 6.4.12 on page 160*
	0x5000 0400 - 0x5000 07FF	1 KB	GPIOB	*Section 6.4.12 on page 160*
	0x5000 0000 - 0x5000 03FF	1 KB	GPIOA	*Section 6.4.12 on page 160*

6.4.1 GPIO port mode register (GPIOx_MODER) (x =A, B, C, D, F)

Address offset:0x00

Reset value: 0xEBFF FFFF (port A)

Reset value: 0xFFFF FFFF (ports other than A)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[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
MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **MODEy[1:0]:** Port x configuration for I/O y (y = 15 to 0)
These bits are written by software to set the I/O to one of four operating modes.
00: Input
01: Output
10: Alternate function
11: Analog

6.4.6 GPIO port output data register (GPIOx_ODR) (x = A, B, C, D, F)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OD15	OD14	OD13	OD12	OD11	OD10	OD9	OD8	OD7	OD6	OD5	OD4	OD3	OD2	OD1	OD0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODy:** Port output data I/O y (y = 15 to 0)
These bits can be read and written by software.
Note: For atomic bit set/reset, the OD bits can be individually set and/or reset by writing to the GPIOx_BSRR register (x = A, B, C, D, F).

6.2 Memory-Mapped Registers and the CMSIS Header Files

In the CMSIS header files, the memory mapped registers for each type of **peripheral** (say GPIO) are modelled in the C programming language by a **struct** type.

Then, for each **instance** of a peripheral (say GPIO Port B) a **pointer** is defined, pointing to the lowest ("base") address of that peripheral.

```
typedef struct
{
  __IO uint32_t MODER;       /*!< GPIO port mode register,               Address offset: 0x00      */
  __IO uint32_t OTYPER;      /*!< GPIO port output type register,        Address offset: 0x04      */
  __IO uint32_t OSPEEDR;     /*!< GPIO port output speed register,       Address offset: 0x08      */
  __IO uint32_t PUPDR;       /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C      */
  __IO uint32_t IDR;         /*!< GPIO port input data register,         Address offset: 0x10      */
  __IO uint32_t ODR;         /*!< GPIO port output data register,        Address offset: 0x14      */
  __IO uint32_t BSRR;        /*!< GPIO port bit set/reset  register,     Address offset: 0x18      */
  __IO uint32_t LCKR;        /*!< GPIO port configuration lock register, Address offset: 0x1C      */
  __IO uint32_t AFR[2];      /*!< GPIO alternate function registers,     Address offset: 0x20-0x24 */
  __IO uint32_t BRR;         /*!< GPIO Bit Reset register,               Address offset: 0x28      */
} GPIO_TypeDef;
…
#define IOPORT_BASE           (0x50000000UL)  /*!< IOPORT base address */
…
#define GPIOB_BASE            (IOPORT_BASE + 0x00000400UL)
…
#define GPIOB                 ((GPIO_TypeDef *) GPIOB_BASE)
```

The __IO prefix is a macro which expands to the C keyword **volatile**. Volatile indicates to the compiler that reading or writing will have side effects unknown to the compiler and therefore read and write accesses must not be optimized.

Using the above struct, a register, say register ODR of GPIOB, can be accessed in C code like

```
GPIOB->ODR = 0x00000000; // set all bits in the ODR register of GPIO port B to zero (low level)
```

However, it is more common to set only parts of a register, say bit 6 of ODR register while leaving all others at their current value which will be shown next.

We use pin PB6 (which is bit 6 of GPIOB) as an example because on the STM32C011-DK board this pin is connected to the on-board LED via a MOSFET (driver).

6.3 Accessing Register Bits and Bit Ranges

C macros in the CMSIS core library for register bits have the following form:

```
#define GPIO_ODR_OD6_Pos              (6U)
#define GPIO_ODR_OD6_Msk              (0x1UL << GPIO_ODR_OD6_Pos)            /*!< 0x00000040 */
#define GPIO_ODR_OD6                  GPIO_ODR_OD6_Msk
```

Only bit 6 (for pin 6) of GPIO register ODR is given here. Compare these macros to the GPIO ODR register definition in the reference manual [2] which are shown above.

Similar for a range of consecutive bits. Definitions for the two bits controlling pin 6 in the GPIO register MODER are shown. Again, compare this to the GPIO MODER register definition above from the reference manual [2].

```
#define GPIO_MODER_MODE6_Pos            (12U)
#define GPIO_MODER_MODE6_Msk            (0x3UL << GPIO_MODER_MODE6_Pos)          /*!< 0x00003000 */
#define GPIO_MODER_MODE6                GPIO_MODER_MODE6_Msk
```

Note that these macros are the same for all instances (GPIOA, GPIOB, ...) of GPIO. The peripheral name in the macros is the type name of a peripheral, not the instance name.

Using these macros, we can set (to 1) or clear (reset to 0) a single bit:

```
#include "stm32c011xx.h"
…
GPIOB->ODR |= GPIO_ODR_OD6; // set bit 6 to 1 → switch LED off (LED is low-active)
…
GPIOB->ODR &= ~GPIO_ODR_OD6; // reset bit 6 to 0 → switch LED on (LED is low-active)
…
GPIOB->ODR ^= GPIO_ODR_OD6; // toggle (flip) bit 6, change state from 0 to 1 or from 1 to 0
```

and assign a continuous range of bits a new value:

```
#include "stm32c011xx.h"
…
// Set PB6 pin to some mode (mode = 0 to 3):
GPIOB->MODER = (GPIOB->MODER & ~GPIO_MODER_MODE6_Msk) | (mode << GPIO_MODER_MODE6_Pos);
```

Note: The above code snippets do not show the part of **switching the clock on** for GPIOB, which must be done once, before the first access to a GPIOB register. For the sake of completeness:

```
RCC->IOPENR |= RCC_IOPENR_GPIOBEN; // enable GPIOB port by switching its peripheral clock on
(void)RCC->IOPENR; // read the RCC_IOPENR register again, see ref.man. 5.2.14
```

RCC is another peripheral located on the AHB bus (base address 0x40020000), peripheral offset 0x1000. The IOPENR register has the offset 0x34 within the RCC peripheral. Thus, RCC->IOPENR is the systematic name for the MMIO register at address 0x40021034.

In the above code, the first line shows a **read-modify-write** access to the RCC->IOPENR register. The second line shows a read access to the same register. The read result is never used (cast to void), but this read access is required to ensure that the RCC->IOPENR write access is completed before the peripheral is used. The volatile definition of the register ensures that the compiler will generate machine code for that. This idiom is regularly used when enabling RCC clocks.

Admittedly, these CMSIS defines and macros are cumbersome and prone to copy and paste errors. But, for the rest of this document it is essential to thoroughly **understand all of the above C idioms**.

7 GENERAL-PURPOSE I/OS (GPIO)

Almost all MCU pins are General Purpose Input/Output (GPIO) pins. The GPIO pins are organized in ports. Port names are A, B, C,.... Each port can have up to 16 pins numbered 0..15. The name PB6 denotes pin 6 of port B (which is connected to the LED on the STM32C011-DK board), and so on.

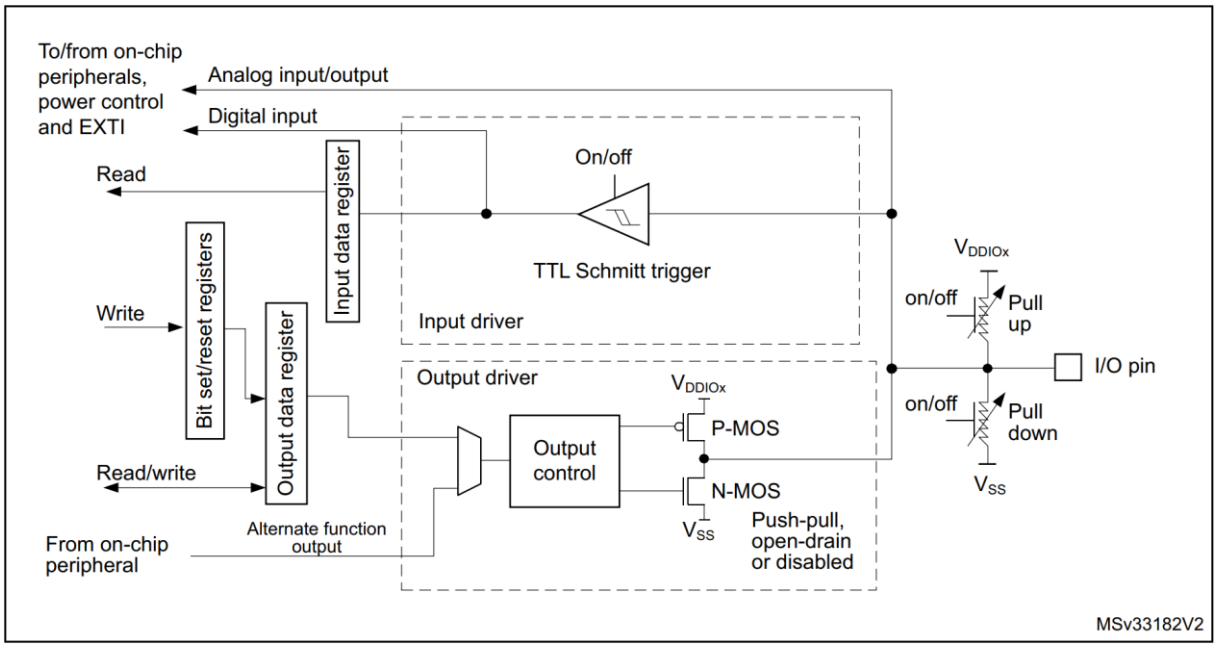


Figure 12: Basic structure of an I/O port bit, source: reference manual [2].

During and after a reset, the mode of almost all GPIO pins is set by the hardware to **analog mode**, which is a passive mode: the pin neither driven high nor low.

Exceptions are **PA13** and **PA14** which are initially configured as **SWD debugging interface**. More exceptions are possible for the BOOT and NRST pins depending on the programmable **option bytes**, see the data sheet [3].

The firmware is then responsible for configuring the GPIO pins according to the MCU interfaces and board signal connections actually used.

For the STM32C011 all signal pins are **5V tolerant**. Beware when using other MCUs that may have some pins which are not 5V tolerant and might need external level conversion.

It might be necessary to add external pull-up or pull-down resistors to a board design for enforcing deterministic output levels on some pins while the MCU is powered-off, in some low-power mode, during reset, and startup before the firmware has started and can reconfigure those pins. The I2C interface (see chapter 12) is well-known for needing external pull-ups.

Depending on the chip package size, not all pins are available on all packages. Notable, pin **PA9 and PA10** are not accessible by default on the STM32C011 (all packages). If needed, they must be **remapped** by the firmware to replace PA10 and PA11 pin functions as shown in chapter 9 for the UART1 communication to the host.

7.1 GPIO MODE REGISTER (MODER)

The pin **mode** determines the basic role of a GPIO pin: input vs. output, and digital vs. analog.

Pin Function	What it is used for	MODER Register Bits
digital input	Digital input signal, the level (high/low) can be queried by a GPIO IDR register bit	00
digital output	Digital output signal, the level (high/low) is set by a GPIO ODR, BSRR, or BRR register bit	01
alternate function	Digital input or output which is routed to an internal peripheral block like UART, I2C, SPI, Timer,... and not controlled by the GPIO port.	10
analog	All digital input and output drivers are switched off, lowest power consumption. Some pins have **additional functions** in this mode like serving as an ADC input channel	11

Note that STM32C0 series does not have analog output functions, but other series do.

Digital Output and Input modes are described first, followed by Alternate Function Mode. Analog Mode is used in chapter 14 for ADC and is otherwise the default choice for **unused pins**.

7.2 GPIO Output Type Register (OTYPER)

There are two **types** of GPIO output which can be used:

GPIO Output Type	OTYPER register bit	Comment
push-pull	0	Normal type for driving external digital inputs. Both MOSFETs are used for actively driving the output low (pull) or high (push).
open-drain	1	Special type for wired-or / tri-state outputs. Only the low-side MOSFET is used for actively pulling the output low. Used in I2C and other protocols.

7.3 GPIO Output Speed Register (OSPEEDR)

The maximum **speed** of a GPIO output pin can be **limited**. Speed limiting lowers Electro-magnetic Interference (EMI) and enhances reliable operation when higher speed is not necessary. Four speed grades are available on STM32C0 (values may differ for other STM32 series):

Name	OSPEEDR register bits setting	max. frequency (see data sheet [3])
very low speed	00	2 MHz
low speed	01	10 MHz
high speed	10	30 MHz
very high speed	11	48 MHz (max. system frequency)

7.4 GPIO Pull-Up Pull-Down Register (PUPDR)

For each GPIO pin, an **internal pull-up or pull-down resistor** may be configured. These internal resistors help keeping deterministic logic levels for digital **inputs** and **open-drain outputs**. The internal resistors are relatively **weak** (typ. 40 kOhm) and may not be sufficient for every purpose, say for a reliable I2C communication. Additional external resistors may be required in such cases.

Pull-up / pull-down use	PUPDR bits setting	Comment
none	00	Common for push-pull outputs and also good for inputs driven by an external push-pull output.
pull-up	01	For some inputs or open-drain outputs
pull-down	10	For some inputs
reserved	11	**Do not use**

Note that pull-up / pull-down resistors don't make sense for push-pull outputs which are always actively driven (high or low) by the output buffer.

7.5 GPIO Output Data Register (ODR)

The ODR register is used to set a digital output pin (or a whole set of pins) high:

```
GPIOB->ODR |= GPIO_ODR_OD6; // set bit 6 in GPIOB ODR → pin PB6 output level high
```

or low:

```
GPIOB->ODR &= ~GPIO_ODR_OD6; // reset bit 6 in GPIOB ODR → pin PB6 output level low
```

The following code will set PB6 high and PB7 low simultaneously:

```
GPIOB->ODR = (GPIOB->ODR & ~GPIO_ODR_OD6) | GPIO_ODR_OD7; // PB6 low, PB7 high
```

When testing this code, don't forget to switch the GPIOB clock on using RCC->IOPENR as explained before and demonstrated in chapter 7.7.

This will work for the **push-pull output** type and at any speed setting. For the **open-drain output** type, a pull-up resistor is required for observing the high level. Without a pull-up, an open-drain output pin will not be pulled high and is in an high-impedance (Hi-Z) state, also called tri-stated.

The above C statements are compiled to at least three machine instructions:

- one for reading the current ODR value (load),
- one for modifying this value (and, or, ...), and
- one for writing back the modified value to the ODR register (store).

This is called a **read-modify-write** access and has some drawbacks, see below.

7.6 GPIO Bit Set/Reset Register (BSRR) and Bit Reset Register (BRR)

The BRR and BSRR registers are **write-only** and can be understood as optimized variants of ODR avoiding the read-modify-write access.

BRR can be used for setting bits low (sometimes called resetting) with a single machine instruction:

```
GPIOB->BRR = GPIO_BRR_BR6;  // set bit 6 in GPIOB BRR → pin PB6 output level low
```

BSRR can be used for setting bits **high** with a single machine instruction:

```
GPIOB->BSRR = GPIO_BSRR_BS6; // set bit 6 in GPIOB BSRR → pin PB6 output level high
```

BSRR and can be used to **set and reset** some pins **at the same time**:

```
GPIOB->BSRR = GPIO_BSRR_BS6 | GPIO_BSRR_BR7; // set PB6 high, reset PB7 low, leave others
```

Watch out for the subtle differences in the bit naming: BS (bit set) vs. BR (bit reset).

When using BRR and BSRR, you are always specifying a **bit mask**:

- if a bit set (1) in the bit mask, the corresponding GPIO pin will be affected (set or reset),
- if a bit is not set (0) in the bit mask, the corresponding pin is unaffected by the operation.

When changing GPIO bits by writing to BRR or BSRR, the resulting pin states (as seen at the input stage of the output control) can be observed by reading the ODR register.

7.7 GPIO Output Mode Example: Blinking a LED

The following blinky code is for the STM32C0116-DK where LED LD3 is connected to pin PB6 (Port GPIOB, Bit 6). Change to PA5 (Port GPIOA, Bit 5) for NUCLEO-C031C6 board.

```
#include <stm32c011xx.h>

void init_LED(void) {
    RCC->IOPENR |= RCC_IOPENR_GPIOBEN; // enable the peripheral clock for GPIOB block
    (void)RCC->IOPENR; // read back the register to make sure that the clock is now on
    // Setup pin PB6 to push-pull type, very low speed, no pull-up/pull-down, and high level:
    GPIOB->OTYPER = (GPIOB->OTYPER &~GPIO_OTYPER_OT6_Msk) | (0 << GPIO_OTYPER_OT6_Pos);
    GPIOB->OSPEEDR = (GPIOB->OSPEEDR &~GPIO_OSPEEDR_OSPEED6_Msk) | (0 << GPIO_OSPEEDR_OSPEED6_Pos);
    GPIOB->PUPDR = (GPIOB->PUPDR &~GPIO_PUPDR_PUPD6_Msk) | (0 << GPIO_PUPDR_PUPD6_Pos);
    GPIOB->BSRR = GPIO_BSRR_BS6; // set GPIOB pin PB6 high -> LED off (LED is low active)
    // GPIO pin mode is set last to avoid unwanted glitches at the output during setup
    GPIOB->MODER = (GPIOB->MODER & ~GPIO_MODER_MODE6_Msk) | (1 << GPIO_MODER_MODE6_Pos);
}

void blink(void) {

    GPIOB->BRR = 1 << 6;    // set GPIOB pin 6 low -> LED (low active) on
    for (volatile int i = 0; i < 200000; ++i)
        ; // busy loop, spending some time to achieve a visible delay

    GPIOB->BSRR = 1 << 6;   // set GPIOB pin 6 high -> LED (low active) off
    for (volatile int i = 0; i < 500000; ++i)
        ; // busy loop, spending some time to achieve a visible delay
}

int main(void)
{
    init_LED();
    for(;;) {
        blink();
    }
    return 0;
}
```

This above code extensively uses the _Msk and _Pos defines from CMSIS as explained in chapters 6.1.5 ff. Although left shifting a zero (0) has no effect, it is demonstrated here as a blueprint for different parameters settings.

The init_LED and blink functions come in handy for giving visual feedback in many examples to follow.

7.8 GPIO Input Data Register (IDR) - Debouncing

While a GPIO pin is not in Analog Mode, its digital level (high or low) as observable at the external pin itself, can always be read from the **IDR** register. Comparing ODR and IDR bits may help diagnosing external short-cuts and other electrical anomalies. In addition, it is also possible to use the ADC for measuring the analog voltage level at a pin.

Pressing a real button often causes **bouncing**: For a reliable detection of button presses, de-bouncing should be implemented (which is in fact easier done by polling than by interrupt).

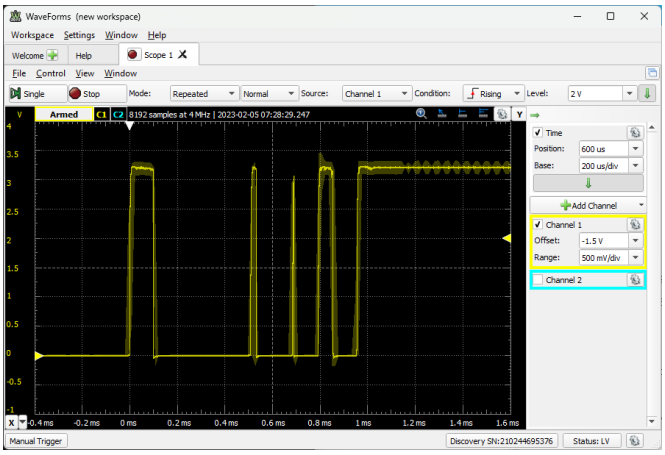


Figure 13: Oscillogram demonstrating bouncing (mechanical button pressed)

The next example shows debouncing a mechanical button in software:

```
// init the joystick button at pin PA8
// when this button is pressed down, PA8 is connected to GND
// we activate the internal pull-up resistor of that GPIO pin
// therefore, the button would also work without any external pull-up resistors
void init_Button(void)
{
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN;  // enable clock for peripheral component GPIOA
    (void)RCC->IOPENR;                  // make sure that the clock is on by now
    // set pin mode to input (0).
    GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE8_Msk) | (0 << GPIO_MODER_MODE8_Pos);
    // enable internal pull-up resistor (1).
    GPIOA->PUPDR = (GPIOA->PUPDR & ~GPIO_PUPDR_PUPD8_Msk) | (1 << GPIO_PUPDR_PUPD8_Pos);
    // Reset defaults for other GPIO registers are okay here
}
```

```
int main(void)
{
    init_LED();         // initialize the LED pin, see blinky
    init_Button();      // initialize the button pin

    int down_count = 0;
    // loop forever
    for(;;) {
        if(!(GPIOA->IDR & GPIO_IDR_ID8)) {
            // button down detected
            down_count++;
        } else {
            // button up detected
            if(down_count > 50) {
                // long button press detected
                GPIOB->BSRR = 1 << 6;   // set GPIOB pin 6 high -> LED (low active) off
            } else if(down_count > 5) {
                // short button press detected
                GPIOB->BRR = 1 << 6;    // set GPIOB pin 6 low -> LED (low active) on
            }
            down_count = 0;
        }
        for(volatile int i=0; i<10000; ++i); // some delay. better use precise 10ms delay here
    }

    return 0;   // unreachable code, main shall never return in embedded software
}
```

7.9 GPIO Alternate Function Mode Registers (AFR)

In Alternate Function (AF) mode, a GPIO pin is no longer controlled by the ODR register, but by a MCU peripheral block, say USART, SPI, I2C and so on; see Figure 12.

Even when a pin is in AF mode, the GPIO OTYPER, OSPEEDR, and PUPDR register settings for that pin still apply and should match the intended application, like open-drain for I2C signal pins or internal pull-up resistors for others.

Only a fixed set of specific pin-peripheral combinations can be configured in the Alternate Function Mode Registers (AFR). The data sheet [3] has the tables of alternate function mappings. In addition, STM32XCubeMX can help with graphical pin planning and peripheral assignment.

For each pin is an AFSEL field of length 4 for choosing the AF in the GPIO port. Therefore, two 32-bit registers are needed to support 16 pins per port: **AFRL** (lower part) and **AFRH** (higher part), sometimes named **AFR[0]** and **AFR[1]** respectively.

7.10 Further Notes on GPIO

Being able to set/reset pins in one machine instruction has clearly some advantages. One is faster **bit-banging**, i.e. implementing signal protocols in software, and another one **atomic** (uninterruptable, thread-safe) operations.
The values of the pins can be locked by the lock register (**LCKR**) as a safety measure. See the reference manual [2] for details.

Application note AN4899 “STM32 GPIO configuration for hardware settings and low-power consumption” [16] discussed GPIO in-depth, like **electrical characteristics** and guidelines for hardware and software developers.

The MCU has a feature **system memory boot mode** which probes and sets various pins in search for firmware updates over external buses shortly after reset. Sometimes, this causes “unexpected” pin behavior, e.g. on a brand new MCU when the flash is still empty and the MCU is switching to system memory boot mode by the so called **empty check**.

Application note AN2606 [15] has many details on system memory boot mode and the pins and protocols involved.

8 EXTENDED INTERRUPT AND EVENT CONTROLLER (EXTI)

The Extended interrupt and event controller (EXTI) is used for **external interrupts** (signal edges on GPIO pins) and **low-power wakeup** events caused by pins or some wake-up capable internal peripherals.

The **EXTI lines** (event inputs) are coming from

- **GPIO** (EXTI line 0-15, configurable)
- **RTC** (EXTI line 19, direct)
- **I2C1** wakeup (EXTI line, 23 direct)
- **USART1** wakeup (EXTI line, 25 direct)
- **LSE_CSS** (low speed external clock security system; EXTI line 31, direct)

The GPIO input lines to EXTI are **configurable**, the **rising edge** and/or the **falling edge** observed on a pin can be selected as event trigger. The other (**direct**) input lines are to be configured in the peripheral.

Wakeup events can wakeup the MCU without generating an interrupt and don't need a handler.

8.1 EXTI INTERRUPT BY A PIN

The following code snippet implements an interrupt handler for pin PA8 which is connected to the (analog) joystick of the STM323C0116-DK, but used as a digital input here:

```
void EXTI4_15_IRQHandler(void) {
    if(EXTI->FPR1 & EXTI_FPR1_FPIF8) {  // falling edge on PA8 -> IRQ pending
        EXTI->FPR1 |= EXTI_FPR1_FPIF8;  // clear pending bit
        blink();                        // blink the LED (must be initialized, not shown)
    }
}

// make PA8 EXTI interrupt
void init_EXTI() {
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN; // enable peripheral clock
    (void)RCC->IOPENR; // read back to make sure that clock is on
    GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE8_Msk) | (0 << GPIO_MODER_MODE8_Pos); // input mode

    // let m be the pin number (0..15)
    // and x be the port number (A=0, B=1,...)
    // for PA8:
    int m = 8;  // pin name m corresponds to reference manual
    int x = 0;  // port name x corresponds to reference manual

    EXTI->FTSR1 |= 1 << m; // enable falling edge for pin m (0..15)

    // there is 1 byte for each port number
    // that is 4 bytes per 32-bit register (layout chosen by CMSIS)
    EXTI->EXTICR[m / 4] &= ~(0xF << (m % 4)); // clear port bits for pin m
    EXTI->EXTICR[m / 4] |=     x << (m % 4);  // set port x for pin m

    EXTI->IMR1 |= EXTI_IMR1_IM8;    // EXTI CPU wakeup with interrupt mask register
    //EXTI->EMR1 |= EXTI_EMR1_EM8;    // EXTI CPU wakeup with event mask register

    NVIC_EnableIRQ(EXTI4_15_IRQn);
}
```

The interrupt must be enabled in the Nested Vector Interrupt Controller NVIC, see chapter 20.1.

When using a mechanical button, bouncing may occur. Debouncing is more reliably done by polling than by EXTI interrupt. EXTI interrupts are to be used when external peripherals like a MPU6050 sensor generates an interrupt that must be handled by the MCU; or if the MCU should wake up from some low power mode.

9 UNIVERSAL SYNCHRONOUS / ASYNCHRONOUS RECEIVER TRANSMITTER (USART)

Only the asynchronous communication mode is described here, which is generally called **UART** (Universal Asynchronous Receiver Transmitter).

Typically, there is one USART interface routed from the MCU to the PC via the ST-LINK debugger chip and the USB connection (virtual com port, **VCP**). For the STM32C0116-DK board, this is **USART1** with the TX (transmit data) on pin PA9 and RX (receive data) on pin PA10. On the smaller chip packages, PA9 and PA10 are not accessible by default, but must be explicitly remapped to PA11 and PA12 as shown in the code below.

Note, that the common UART parameters **115200 8N1** are used (115200 baud rate, 8 data bits, no parity, 1 stop bit) which must exactly match the settings on the other side of the serial line.

Here is the initialization code:

```
#include <stm32c011xx.h>

void init_UART1(void) {
    RCC->APBENR2 |= RCC_APBENR2_SYSCFGEN;   // enable clock for peripheral component
    (void)RCC->APBENR2; // ensure that the last instruction finished and the clock is now on

    SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA11_RMP; // remap PA9 instead of PA11
    SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA12_RMP; // remap PA10 instead of PA12

    // PA9, PA10 = USART1 TX, RX, routed to ST-LINK VCP, see STM32C011-DK board schematics
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN;
    (void)RCC->IOPENR; // ensure that the last write command finished and the clock is on

    GPIOA->AFR[1] = (GPIOA->AFR[1] & ~GPIO_AFRH_AFSEL9_Msk) | (1 << GPIO_AFRH_AFSEL9_Pos); // AF1
    GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE9_Msk) | (2 << GPIO_MODER_MODE9_Pos);   // AF mode

    GPIOA->AFR[1] = (GPIOA->AFR[1] & ~GPIO_AFRH_AFSEL10_Msk) | (1 << GPIO_AFRH_AFSEL10_Pos); // AF1
    GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE10_Msk) | (2 << GPIO_MODER_MODE10_Pos);   // AF mode

    RCC->APBENR2 |= RCC_APBENR2_USART1EN;
    (void)RCC->APBENR2; // ensure that the last instruction finished and the clock is on
    USART1->BRR = 12000000 / 115200; // for SYSCLK = 12MHz and baud rate 115200
    USART1->CR1 = USART_CR1_UE | USART_CR1_RE | USART_CR1_TE;   // enable UART, RX, TX
}
```

9.1 REDIRECTING STDOUT TO UART1 FOR LOGGING

It is often desirable using **printf** for output of diagnostic messages. In a **hosted** implementation (like on a PC), the C library would send the printf output to stdout. But a MCU is using a **freestanding** implementation of the C runtime library, where there is no stdout unless you implement one.

For your convenience, there are already dummy implementations for various system calls provided in the syscall.c file. The printf function will call the **_write** system call to make an output. _write is, by default, implemented as a series of calls to **int __io_putchar(int ch)**.

The following code implements the __io_putchar function such that it redirects every output produced by printf to USART1:

```c
int __io_putchar(int ch) {
    // redirect all file output to USART1
    while (!(USART1->ISR & USART_ISR_TXE_TXFNF))
        ;   // loop while the TX register is not empty (last transmission not completed)
    USART1->TDR = (uint8_t)ch;  // write the char to the transmit data register
    return ch;  // indicate success to the caller
}
```

Notes:

1. USART1 must be **initialized** before it can be used for data transmission,
2. stdout and therefore printf is by default **line-buffered**. End your output always with a newline '\n' to make it immediately visible in the terminal program,
3. the printf implementation used (unfortunately) **malloc** to allocate a line buffer. The malloc function calls in turn _sbrk to get more memory. _sbrk is implemented in sysmem.c,

To avoid the hassle with line-buffering, malloc, and _sbrk, one can make stdout unbuffered:

```c
int main(void)
{
    setbuf(stdout, NULL);           // make stdout unbuffered
    init_UART1();                   // setup USART1
    printf("Hello, world.\n");      // no line buffering, no call to malloc or _sbrk
    /* Loop forever */
    for(;;);
}
```

and you will see "Hello, world." on a connected terminal program. For the curious: Set a breakpoint in __io_putchar and inspect the call stack.

The details of redirecting stdout depend on the C runtime library used and, closely related, the choice of your C compiler. The above works for **gcc** with the **newlib-nano C runtime** as used by the STM32CubeIDE and STM32 VS Code plugin. The newlib-nano printf may need some extra linker settings for supporting floating point parameters. There is a 3rd party printf alternative available: **nanoprintf** https://github.com/charlesnicholson/nanoprintf.

An in-depth analysis of using printf for debugging is given in [17].

9.2 UART Receive and Transmit by Polling: rot13

Rot13 is a traditional method for scrambling ASCII text. It is by no way a cryptographically safe encoding, but simply prevents humans from understanding rot13 encoded text on-the-fly, like this one:

HFNEG fgnaqf sbe Havirefny Flapuebabhf Nflapuebabhf Erprvire naq Genafzvggre

```c
#include <stm32c011xx.h>

// read bytes on UART1 and echo them back, but scramble all alphanumeric chars with rot13
char rot13(char ch)
{
    if ('0' <= ch && ch <= '9')
        ch = '0' + (ch - '0' + 5) % 10;
    else if ('A' <= ch && ch <= 'Z')
        ch = 'A' + (ch - 'A' + 13) % 26;
    else if ('a' <= ch && ch <= 'z')
        ch = 'a' + (ch - 'a' + 13) % 26;
    return ch;
}

int main(void)
{
    init_UART1();
    USART1->TDR = '>'; // == ASCII code 62 == 0x2E; greet the other side with a prompt
    while (1) {
        while (!(USART1->ISR & USART_ISR_RXNE_RXFNE))
            ;   // loop while the RX register is empty (nothing received)
        char ch = (char)USART1->RDR;     // read the byte received
        ch = rot13(ch);                  // apply the rot13 scrambling algorithm
        while (!(USART1->ISR & USART_ISR_TXE_TXFNF))
            ;   // loop while the TX register is not empty (last transmission not completed)
        USART1->TDR = (uint8_t)ch;       // write the char to transmit
    }
    return 0;
}
```

Note that rot13 is a symmetric chiffre which can be used for scrambling and descrambling.

As the UART interface is relatively slow compared to the Arm Cortex-M0+ core clock, the MCU wastes many cycles by waiting for the UART receive/transmission.

Possible enhancements are discussed further below: USART can be used with **interrupts** and/or direct memory access (**DMA**, see also chapter 10) which will be discussed next.

9.3 UART Receive with RXNE Interrupt for each single char received

```c
#include <stm32c011xx.h>

char rx_buffer[64];
int rx_index = 0;

void USART1_IRQHandler(void) {
    if (USART1->ISR & USART_ISR_RXNE_RXFNE) {    // RXNE flag set?
        rx_buffer[rx_index] = USART1->RDR;       // reading RDR automagically clears the RXNE flag
        rx_index = (rx_index+1) % sizeof(rx_buffer);
    } // else: handle more flags here like TXE, overrun error, ...
}

int main(void) {
    // setup PA9, PA10 for USART1 TX, RX (routed to ST-LINK VCP, see STM32C011-DK board schematics)
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN;
    (void)RCC->IOPENR; // ensure that the last write command finished and the clock is on

    GPIOA->AFR[1] = (GPIOA->AFR[1] & ~GPIO_AFRH_AFSEL9_Msk) | (1 << GPIO_AFRH_AFSEL9_Pos); // AF1
    GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE9_Msk) | (2 << GPIO_MODER_MODE9_Pos);   // AF mode

    GPIOA->AFR[1] = (GPIOA->AFR[1] & ~GPIO_AFRH_AFSEL10_Msk) | (1 << GPIO_AFRH_AFSEL10_Pos); // AF1
    GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE10_Msk) | (2 << GPIO_MODER_MODE10_Pos);   // AF mode

    // remap PA9 and PA10
    RCC->APBENR2 |= RCC_APBENR2_SYSCFGEN;   // enable clock for peripheral component
    (void)RCC->APBENR2; // ensure that the last instruction finished and the clock is on

    SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA11_RMP; // remap PA9 instead of PA11
    SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA12_RMP; // remap PA10 instead of PA12

    // setup UART
    RCC->APBENR2 |= RCC_APBENR2_USART1EN;
    (void)RCC->APBENR2; // ensure that the last instruction finished and the clock is on

    uint32_t baud_rate = 115200;
    USART1->BRR = 12000000 / baud_rate; // == 104 == 0x68; assuming SYSCLK = 12MHz
    // enable UART including RXNE interrupt generation in USART peripheral
    USART1->CR1 = USART_CR1_UE | USART_CR1_RE | USART_CR1_TE | USART_CR1_RXNEIE_RXFNEIE;

    NVIC_EnableIRQ(USART1_IRQn);        // enable interrupt handling in NVIC
    USART1->TDR = '>'; // == ASCII code 62 == 0x2E; greet the other side with a prompt
    /* Loop forever */
    for(;;);
}
```

9.4 UART Receive with DMA and Idle Interrupt

For reference, see chapter 24.5.19 “Continuous communication using USART and DMA” in the reference manual [2].

The idea is to let the UART receive incoming chars automatically with the help of DMA. Here, when an important “event” occurs, an interrupt is triggered and the core can take care of the received input. Here, the idle interrupt is used which is triggered as soon as the UART RX line is high (idle) for an extend period of time. This is a common choice for line-buffered inputs with a limited line length where a complete line is transmitted to the receiving UART as one continuous block of characters.

The UART is initialized as above. A linear peripheral to memory DMA is configured with the UART read data register as source and a global rx_buffer as destination:

```
char rx_buffer[80];     // buffer for incoming chars, max. 80

void uart1_rx_dma() {
    RCC->AHBENR |= RCC_AHBENR_DMA1EN;   // this is good for DMA and DMAMUX
    (void)RCC->AHBENR;

    if( DMA1_Channel1->CCR & DMA_CCR_EN) {      // channel was in use before
        DMA1_Channel1->CCR &= ~DMA_CCR_EN;      // disable DMA channel for setup
    }

    // route peripheral DMA request to DMA channel
    // Table 34: DMAMUX usart1_rx_dma == 50
    // caution: DMAMUX1_Channel0 is for DMA1_Channel1 and so on!
    DMAMUX1_Channel0->CCR = 50 << DMAMUX_CxCR_DMAREQ_ID_Pos;

    DMA1->IFCR = DMA_IFCR_CGIF1;    // clear all (HT, TC, TE) flags for DMA channel 1

    DMA1_Channel1->CPAR = (uint32_t)&(USART1->RDR);
    DMA1_Channel1->CMAR = (uint32_t)rx_buffer;
    DMA1_Channel1->CNDTR = sizeof(rx_buffer);
    DMA1_Channel1->CCR =
        0 << DMA_CCR_MEM2MEM_Pos    // MEM2MEM 0: no memory-to-memory mode
    |   0 << DMA_CCR_PL_Pos         // PL priority level 0: low.. 3: very high
    |   0 << DMA_CCR_MSIZE_Pos      // MSIZE 0: 8-bit 1: 16-bit 2: 32-bit
    |   0 << DMA_CCR_PSIZE_Pos      // PSIZE 0: 8-bit 1: 16-bit 2: 32-bit
    |   1 << DMA_CCR_MINC_Pos       // MINC memory increment mode on (1)
    |   0 << DMA_CCR_PINC_Pos       // PINC peripheral increment mode off (0)
    |   0 << DMA_CCR_CIRC_Pos       // CIRC 1: circular mode
    |   0 << DMA_CCR_DIR_Pos        // DIR 0: read from peripheral, 1: memory
    |   0 << DMA_CCR_TEIE_Pos       // TEIE transfer error interrupt 1: enable
    |   0 << DMA_CCR_HTIE_Pos       // HTIE half transfer interrupt 1: enable
    |   0 << DMA_CCR_TCIE_Pos       // TCIE transfer complete interrupt 1: enable
    ;
    DMA1_Channel1->CCR |= DMA_CCR_EN; // enable DMA channel

    // A channel, as soon as enabled, may serve any DMA request from the peripheral
    // connected to this channel, or may start a memory-to-memory block transfer.
}
```

A UART1 interrupt handler is defined handling the IDLE interrupt. The interrupt handler determines how many characters were receive in the rx_buffer, before the idle condition appeared. For demonstration, it parses the received string for a valid command (“LED on” and LED off”) and switches the LED accordingly. The LED GPIO pin PB6 must be configured a GPIO output.

```
void USART1_IRQHandler(void) {
    if (USART1->ISR & USART_ISR_IDLE) { // idle line flag set ?
        USART1->ICR |= USART_ICR_IDLECF; // writing 1 *clears* the idle line detected flag
        uint32_t received = sizeof(rx_buffer) - DMA1_Channel1->CNDTR;
        if(received < sizeof(rx_buffer))
            rx_buffer[received] = '\0'; // set terminating '\0' char if enough space

        // evaluation of rx_buffer content
        if (strncmp(rx_buffer, "LED on", 6) == 0) {
            GPIOB->BRR = 1 << 6; // set GPIOB pin 6 low -> LED on (low active)
        } else if (strncmp(rx_buffer, "LED off", 7) == 0) {
            GPIOB->BSRR = 1 << 6; // set GPIOB pin 6 high -> LED off (low active)
        } else {
            // unknown command
        }

        if (USART1->ISR & USART_ISR_ORE) {  // overrun detected
            // this happens when too many chars were received resp. the DMA buffer was too short
            USART1->ICR |= USART_ICR_ORECF; // clear overrun flag
        }
        uart1_rx_dma();                 // init next DMA
    }
}
```

In the main setup, UART DMA and an UART Idle Interrupt Handler are configured:

```
int main(void) {
    init_UART1();
    uart1_rx_dma();
    USART1->CR3 |= USART_CR3_DMAR;      // enable reveiver DMA
    USART1->CR1 |= USART_CR1_IDLEIE;    // enable uart idle interrupt
    NVIC_EnableIRQ(USART1_IRQn);        // enable this interrupt in NVIC
    /* Loop forever */
    for(;;);
}
```

Now, when the code is running and you enter “LED on” in a connected serial terminal, the LED is switched on. Entering “LED off” switches it off again.

When many chars is received and the rx_buffer is full before an idle event, the DMA terminates and the UART is *overrun*: `USART1->RDR` is no longer read before the next char is received. This situation is caught by the ORE (OverRun Error) flag. Alternatively, an overrun error could be handled directly in the interrupt handler or by a DMA completion interrupt handler.

Note: Instead of handling the idle condition, it is possible to define a matching char (typically a newline ‘\n’ or some other “magic” char) and configure the UART to raise an Character Match Interrupt when that magic char was received.

10 DIRECT MEMORY ACCESS (DMA)

The DMA controller is used for doing memory transfers, called DMA requests, from a source address to a destination address.

The addresses are typically addresses of global buffers (C arrays) in SRAM or peripheral registers.

The Arm Cortex-M0+ core only configures the DMA request and triggers it by setting the EN (enable) bit of the channel's configuration register (CCR).

Then, the Arm Cortex-M0+ core is not involved in the actual data transfer which is offloaded to the DMA controller acting as a bus master. The core and DMA transfers are using the chip internal busses concurrently as a shared resource.

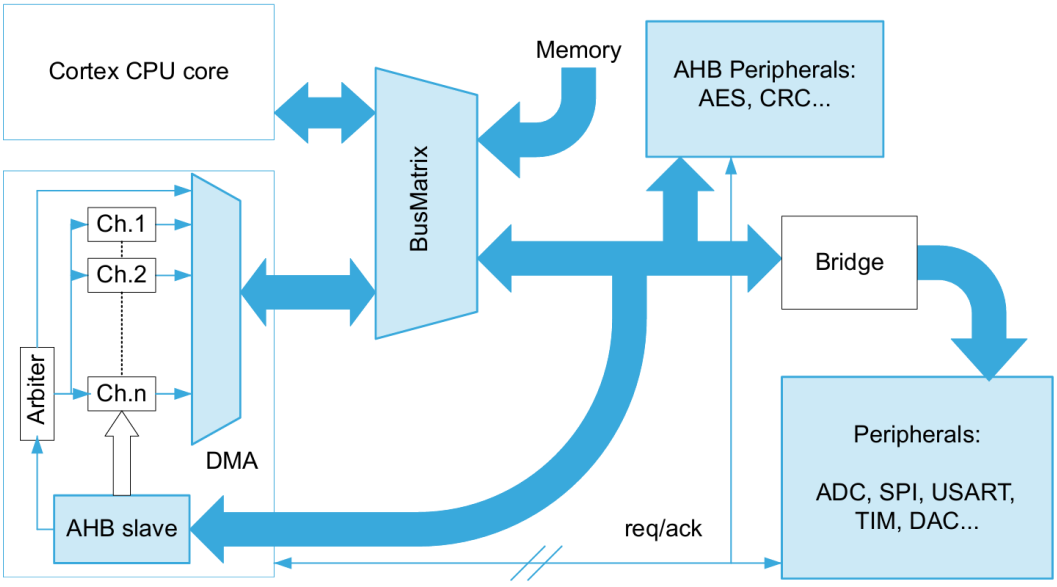


Figure 14: DMA Block Diagram. source: ST AN2548 Application note [18].

The data items transferred can be 8-bit (byte), 16-bit (halfword), or 32-bit (word) wide. Wider transfers are more efficient, but for peripheral registers the appropriate data type is determined by the register, i.e. 8-bit for UART transmission, 16-bit for ADC measurements, and so on.

Caution: In the STM32C0 series and most others with Cortex-M0+, **DMA cannot access the GPIO** port registers because GPIO is tightly coupled to the Arm Cortex-M0+ core and not attached to the AHB/APB bus structure, see chapter 0.

10.1 DMA: MEMORY TO MEMORY TRANSFER

Memory to Memory transfer simply copies a memory array to another array similar to the **memcpy** function. There is a variant of DMA which can copy a single value to an entire array thus imitating **memset**. The advantage of DMA lies in freeing the CPU from dumb data shuffling tasks.

```
void mem2mem_dma() {

    static char src[64] = "The quick brown fox jumped over the lazy dog.";
    char dst[64] = {0}; // gcc will clear it with memset or equivalent

    RCC->AHBENR |= RCC_AHBENR_DMA1EN; // enable peripheral clock
    (void)RCC->AHBENR; // read back to make sure that clock is on

    DMA1_Channel1->CCR &= ~DMA_CCR_EN;   // disable DMA channel for setup
    DMA1->IFCR = DMA_IFCR_CGIF1;        // clear all (HT, TC, TE) flags for DMA channel 1

    DMA1_Channel1->CPAR = (uint32_t)src; // source address for the transfer
    DMA1_Channel1->CMAR = (uint32_t)dst; // destination address for the transfer
    DMA1_Channel1->CNDTR = sizeof(src);  // number of data items to be transferred
    DMA1_Channel1->CCR =
        1 << DMA_CCR_MEM2MEM_Pos    // MEM2MEM 1: memory-to-memory mode
    |   0 << DMA_CCR_PL_Pos         // PL priority level 0: low.. 3: very high
    |   0 << DMA_CCR_MSIZE_Pos      // MSIZE 0: 8-bit 1: 16-bit 2: 32-bit
    |   0 << DMA_CCR_PSIZE_Pos      // PSIZE 0: 8-bit 1: 16-bit 2: 32-bit
    |   1 << DMA_CCR_MINC_Pos       // MINC memory increment mode       1: enable
    |   1 << DMA_CCR_PINC_Pos       // PINC peripheral increment mode   1: enable
    |   0 << DMA_CCR_CIRC_Pos       // CIRC 0 : normal (linear) DMA     1: circular DMA
    |   0 << DMA_CCR_DIR_Pos        // DIR 0: read from peripheral,     1: memory
    |   0 << DMA_CCR_TEIE_Pos       // TEIE transfer error interrupt    1: enable
    |   0 << DMA_CCR_HTIE_Pos       // HTIE half transfer interrupt     1: enable
    |   0 << DMA_CCR_TCIE_Pos       // TCIE transfer complete interrupt 1: enable
    |   1 << DMA_CCR_EN_Pos         // EN : set 1 to enable DMA channel
    ;

    // this works only because UART transmission by CPU core is slower than DMA
    puts(dst);  // output string. Caution: DMA transfer puts are running concurrently !!!
}
```

10.2 DMA: Memory to Peripheral Transfer

In this type of DMA transfer, a peripheral register is the destination of the DMA. During setup, the DMA request source must be configured in the DMAMUX. In the demo code below, this will be DMA request ID 51 which stands for USART1 TX DMA, see the reference manual [2] Table 34 for all possible DMA request sources.

At the end of setup, the DMA request is triggered in the peripheral register. In the example, this is USART1 CR3 DMAT.

Once triggered, the DMA block transfer is done completely without intervention of the Arm Cortex-M0+ core. Note, that the speed of the DMA is determined by the peripheral. In the example, the next byte is transferred as soon as the USART TXD register becomes empty. Therefore, the DMA transfer speed depends on the USART baud rate.

```c
#include <stm32c011xx.h>

void mem2uart_dma() {
    static char src[64] = "The quick brown fox jumped over the lazy dog.\n";

    USART1->TDR = '!'; // send one char for connection testing (without DMA))
    while (!(USART1->ISR & USART_ISR_TXE_TXFNF));   // busy wait for TDR empty

    RCC->AHBENR |= RCC_AHBENR_DMA1EN;   // this is good for DMA and DMAMUX
    (void)RCC->AHBENR;

    if( DMA1_Channel1->CCR & DMA_CCR_EN) {      // channel was in use before
        while(!(DMA1->ISR & DMA_ISR_TCIF1));    // wait for transfer complete (TC) channel flag
        DMA1_Channel1->CCR &= ~DMA_CCR_EN;      // disable DMA channel for setup
    }

    // route peripheral DMA request to DMA channel
    // Table 34: DMAMUX usart1_tx_dma == 51
    // caution: DMAMUX1_Channel0 is for DMA1_Channel1 and so on!
    DMAMUX1_Channel0->CCR = 51 << DMAMUX_CxCR_DMAREQ_ID_Pos;

    DMA1->IFCR = DMA_IFCR_CGIF1;    // clear all (HT, TC, TE) flags for DMA channel 1

    DMA1_Channel1->CPAR = (uint32_t)&(USART1->TDR);
    DMA1_Channel1->CMAR = (uint32_t)src;
    DMA1_Channel1->CNDTR = sizeof(src);
    DMA1_Channel1->CCR =
        0 << DMA_CCR_MEM2MEM_Pos    // MEM2MEM 0: no memory-to-memory mode
    |   0 << DMA_CCR_PL_Pos         // PL priority level 0: low.. 3: very high
    |   0 << DMA_CCR_MSIZE_Pos      // MSIZE 0: 8-bit 1: 16-bit 2: 32-bit
    |   0 << DMA_CCR_PSIZE_Pos      // PSIZE 0: 8-bit 1: 16-bit 2: 32-bit
    |   1 << DMA_CCR_MINC_Pos       // MINC memory increment mode on (1)
    |   0 << DMA_CCR_PINC_Pos       // PINC peripheral increment mode off (0)
    |   0 << DMA_CCR_CIRC_Pos       // CIRC 1: circular mode
    |   1 << DMA_CCR_DIR_Pos        // DIR 0: read from peripheral, 1: memory
    |   0 << DMA_CCR_TEIE_Pos       // TEIE transfer error interrupt 1: enable
    |   0 << DMA_CCR_HTIE_Pos       // HTIE half transfer interrupt 1: enable
    |   0 << DMA_CCR_TCIE_Pos       // TCIE transfer complete interrupt 1: enable
    ;
    DMA1_Channel1->CCR |= DMA_CCR_EN; // enable DMA channel

    // A channel, as soon as enabled, may serve any DMA request from the peripheral
    // connected to this channel, or may start a memory-to-memory block transfer.

    USART1->CR3 |= USART_CR3_DMAT; // trigger usart1_tx_dma request
}
```

10.3 DMA: Peripheral to Memory Transfer

An example of DMA peripheral to memory transfer is multi-channel ADC measurement, where several analog voltages are to be measured and the results are copied to an array indexed by the ADC channel.

10.4 DMA: Circular Transfer

By setting the CIRC flag in the DMA Channel Control Register (CCR), the transfer will not terminate when the configured number of data items (CNDTR) is exceeded, but the transfer will be started again in an endless sequence until disabled. This is very useful, but only in certain circumstances.

Examples:

- Configure a timer channel in PWM output mode and setup a circular DMA from an array of pulse width values to the timer channels capture/compare register (CCRx). The DMA transfer will modulate the PWM pulse width by the values stored in the array. Can be used to

generate PWM ramp-ups / ramp-downs, creating a “breathing” LED and much more. Often combined with a timer trigger for the DMA resulting in cycle-exact updates.
- Receive data from an UART by a cyclic DMA placing the RX data in a large ring buffer acting as a FIFO queue in main memory. The idle loop or a background task can query the DMAS channel CNDTR register to find the current head of the queue and read the received data.
- Permanent transfer of ADC measurement results to a buffer (array). This is described in the ADC chapter 14.2..

10.5 DMA: Circular Transfer with HT and TC Interrupts

This is an extension of the circular DMA with two interrupts: a **half-transfer** (HT) interrupt and a **transfer-complete** (TC) interrupt. One address is typically a buffer in SRAM while the other is a peripheral register (set to non-incrementing).

Handling these interrupts allows for changing the data dynamically while the circular DMA transfer keeps running. The buffer then acts as a ring buffer.

Examples:

- Read a continuous stream of analog audio data by a timer-triggered ADC measurement (audio sampling) and put the 16-bit results into a global memory buffer. Each half of the buffer should be able to store, say 10ms of audio samples. In the HT interrupt handler, process the first half of the buffer while the DMA fills the second half. In the TC handler, process the second half of the buffer while the first half is filled with new ADC data by the DMA. Processing could mean anything, like audio signal presence detection, audio filtering, Fast Fourier Transform, DTMF detection and so on. In fact, transformed audio samples could be placed in another buffer which is fed to a DAC (not present in STM32C0) or another output using another cyclic DMA. The only requirement is, that the processing is fast enough, before the next interrupt occurs.

10.6 Further Reading

- AN2548 Application note “Introduction to DMA controller for STM32 MCUs” [18],
- AN5224 Application note “Introduction to DMAMUX for STM32 MCUs” [19].

11 TIMER (TIM)

Timers are used for implementing counters and time-triggered events and IO completely in hardware, independent of the Arm-Cortex-M0+ core and interrupt handlers.

The timer behavior is deterministic in time, even at high speed up to the core clock frequency. Timer can trigger other peripheral components for applications like periodic sampling of external signals, driving motors, counting or measuring external pulses, and so on.

There are different types of timers. TIM1 is an **advanced timer** with special features for motor control like PWM dead time generation and hardware safety (break) features. On the other hand, TIM14 is the **general purpose timer** with the least number of features and registers in a STM32C0. So we concentrate on TIM14 first.

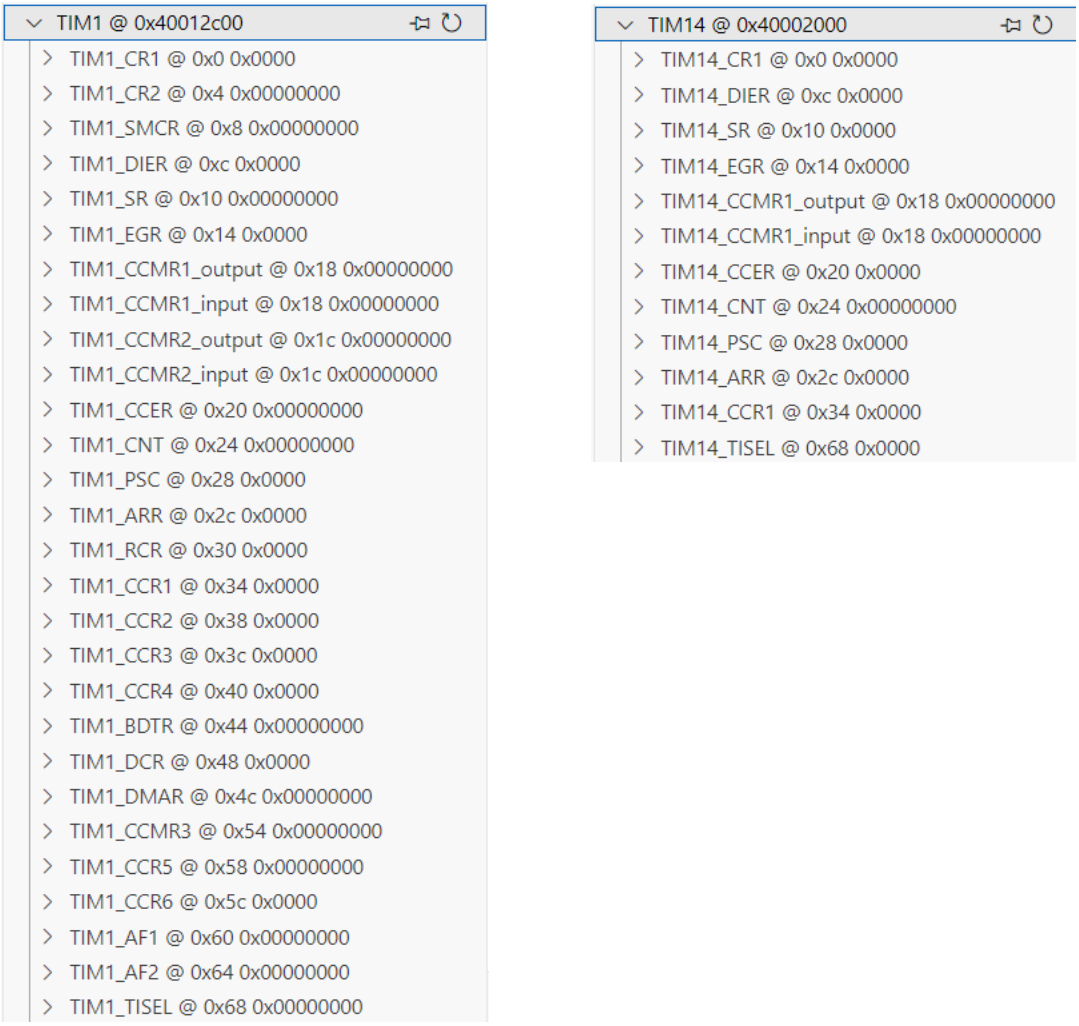


Figure 15: TIM1 and TIM14 registers in comparison (from cortex-debug XPERIPHERALS view)

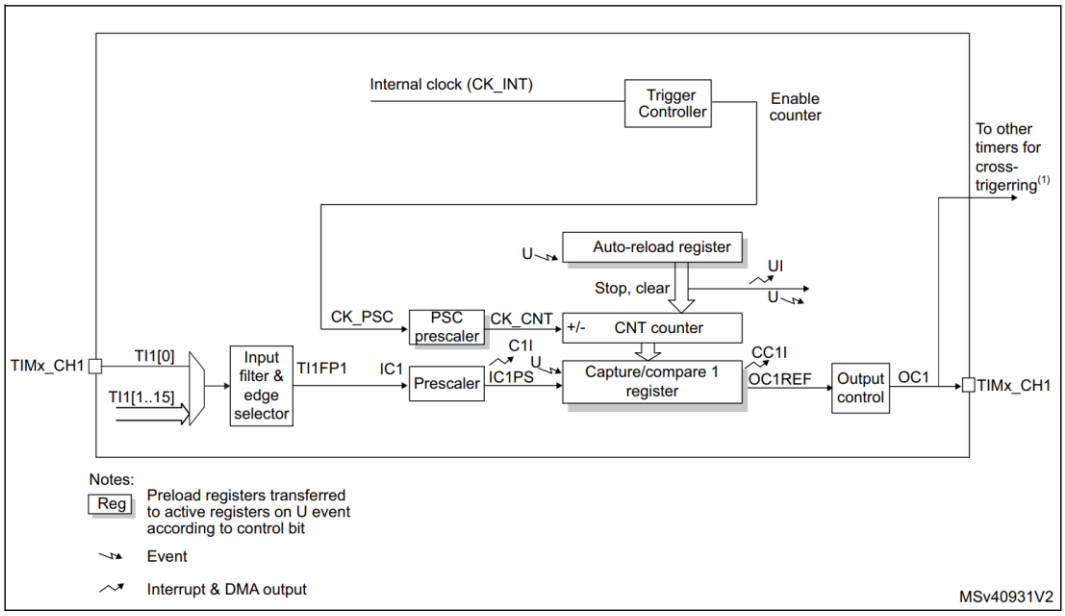


Figure 16: TIM14 Block Diagram, source: reference manual [2].

11.1 Counter Mode (Blinky with Polling)

```
// assume that all relevant clocks (SYSCLK, HCLK, PCLK, TPCLK are 12 MHz)
RCC->APBENR2 |= RCC_APBENR2_TIM14EN;  // enable TIM14 clock
(void)RCC->APBENR2; // ensure that the last write command finished and the clock is on

// config TIM14 in up-counter mode. With 12 MHz clock input (CK_INT),
// the timer will, after pre-scaling, count milliseconds 0, 1, 2,...
// ARR[15:0] Auto-reload register - Counter counts 0..ARR-1 (reset value: 0xFFFF)
//TIM14->ARR = 1000-1;        // no need to change reset value: count a full (2^16) period

// PSC[15:0] Pre-scaler register - divides the counter clock by factor PSC+1. (reset value: 0x0000)
TIM14->PSC = 12000-1;         // set PSC such that CNT will increment each millisecond

// CNT[15:0] Counter register - current value of the counter (reset value: 0x0000)
//TIM14->CNT = 0;             // no need to set / change the default
TIM14->CR1 |= TIM_CR1_CEN;    // enable the timer, this starts counting

// we avoid setting CNT to a specific start value which allows
// peaceful coexistence of several counting intervals
// assume that LED GPIO init was already done elsewhere
while(1) {
  LED_GPIO_Port->BRR = LED_Pin;          // LED (low active) on
  uint16_t start_cnt = TIM14->CNT;
  while(TIM14->CNT - start_cnt < 500u);  // busy wait 500ms

  LED_GPIO_Port->BSRR = LED_Pin;         // LED (low active) off
  start_cnt = TIM14->CNT;
  while(TIM14->CNT - start_cnt < 500u);  // busy wait 500ms
}
```

Note: It is important that, for 16-bit timers, 16-bit **unsigned arithmetic** is used in the busy loop. When properly used , the calculation is **overflow safe** and no additional if statements are required.

Tip: When debugging, the timers keep running by default, even if the debugger is paused at a breakpoint or when stepping through the code. If this is not desired, set once during setup:

```
DBG->APBFZ2 |= DBG_APB_FZ2_DBG_TIM14_STOP;
```

Similar flags are available for many other peripherals.

11.2 Counter Mode (Blinky with Interrupt)

TIM14 setup code is similar to above, but

- we must set the ARR register to the desired period-1 because the update interrupt is triggered when the period is over (CNT==ARR)
- we must enable the TIM14 update interrupt (UIE bit in DIER register)
- we must enable the interrupt in the NVIC (Nested Vector Interrupt Controller)
- we must implement an interrupt handler

TIM14 Update Interrupt Handler:

```
void TIM14_IRQHandler() {               // name must match startup code g_pfnVectors entry
  if(TIM14->SR & TIM_SR_UIF) {          // TIM14 update interrupt flag set ?
    TIM14->SR &= ~TIM_SR_UIF;           // clear TIM14 update interrupt flag
    LED_GPIO_Port->ODR ^= LED_Pin;      // yes: toggle pin
  }
}
```

TIM14 setup:

```
  RCC->APBENR2 |= RCC_APBENR2_TIM14EN;  // enable TIM14 clock
  (void)RCC->APBENR2; // ensure that the clock is on

  // config TIM14 in up-counter mode. With 12 MHz clock input (CK_INT),
  // the timer will, after prescaling, count milliseconds 0, 1, 2,...
  // PSC[15:0] Prescaler register - divides the counter clock by factor PSC+1. (reset value: 0x0000)
  TIM14->PSC = 12000-1;     // set PSC such that CNT will increment each millisecond
  // ARR[15:0] Auto-reload register - Counter counts 0..ARR-1 (reset value: 0xFFFF)
  TIM14->ARR = 500-1;       // set ARR to blinky period
  // CNT[15:0] Counter register - current value of the counter (reset value: 0x0000)
  //TIM14->CNT = 0;         // no need to set / change the default

  TIM14->DIER |= TIM_DIER_UIE;  // enable TIM14 update interrupt

  // NVIC_SetPriority(TIM14_IRQn, 0); // set highest priority, which is the smallest: 0 (default)
  NVIC_EnableIRQ(TIM14_IRQn);   // enable this interrupt in NVIC

  TIM14->CR1 |= TIM_CR1_CEN;    // enable the timer (start counting)
```

There is no blinky loop at the end. The MCU is available for more adventures, while keeping the LED blinking. Blinking can later be paused by `NVIC_DisableIRQ(TIM14_IRQn)` and resumed by `NVIC_EnableIRQ(TIM14_IRQn)`.

11.3 Output Compare Modes

We now use not only the time base, but in addition a **timer channel** of TIM14: Channel 1 (CH1 or TIM14_CH1). A timer channel is basically a register plus a bunch of digital circuitry optionally connecting the channel to the outer world by an alternate function pin (or, sometimes, two).

The reference manual [2] has a number of block diagrams explaining the input stage, main circuit, and output stage of timer channels. However, these descriptions are quite complex, especially for advanced mode timers (TIM1) which have an extended feature set.

Timer channels allows for automating common timer tasks completely in hardware. The TIM14 Block Diagram shows that the timer channel can be connected to an external pin. In that block diagram, TIMx_CH1 pin is drawn as an input pin on the left side and as an output pin on the right side, but both are **same physical pin** which can either be used as input or as output.

Only specific pins can be used. Details are found in the data sheet [3] or when configuring the pins in STM32CubeMX. We will connect TIM14_CH1 to Pin PA4 on the STM32C011-DK. The idea of **output compare mode** is, that the channel register (here: CCR1) is permanently compared to the CNT register by the timer hardware, and some hardware event is triggered when the contents of the two registers match.

The following output compare modes are generally available:

Mode	Name	What is it good for
0	Frozen	Freeze the output at its current level
1	Active level on match	Turn output active synchronized with the next match
2	Inactive level on match	Turn output inactive synchronized with the next match
3	Toggle	Toggle the output level on the next match
4	Forced inactive	Turn output active immediately
5	Forced active	Turn output inactive immediately
6	PWM mode 1	Pulse width level modulation, active phase first
7	PWM mode 2	Pulse width level modulation, inactive phase first

Active level means high level after reset and inactive level means low level, but this can be flipped by setting the CCER.CCxP polarity bit. If a timer channel has a second, complementary output pin, active level means low for that pin after reset and that can be flipped by setting the CCER.CCxNP bit.

In order to simplify the descriptions, an internal signal OCxREF is defined before any polarity bits are applied. OCxREF cannot be observed externally.

Some channels do have more modes (8..15) which will not be considered here.

11.3.1 PWM Output Mode (Blinky with Output Compare)

In the blinky example below, PWM mode 1 will be used. The OC1REF signal is generated from the comparison “CNT < CCR1 ?”. OC1REF is high as long as this condition is true. As a consequence, a falling edge will be triggered, when the counter register CNT reaches the value CCR1, and a rising edge will be triggered when CNT is reset to 0 after reaching the period ARR.

This generates a periodic PWM signal with an active phase of length CCR1 timer ticks and a period of ARR timer ticks. See the figures “Edge-aligned PWM waveforms” in the reference manual [2] for various examples.

For observation, we connect PA4 to the LED pin PB6 with a fly wire. Take care, that the original LED pin PB6 is still in reset state to avoid connecting two outputs which is hazardous. If the counter frequency is low, say 1 Hz, we can observe a blinking LED. If the frequency is higher, say 300 Hz, the human eye observes a dimmed LED, but a logic analyzer still shows the high and low phases of PA4.

If we keep ARR constant, the timer period remains constant. We can now change the active (high) phase of the output by simply changing CC1R. This is called PWM (pulse width modulation).

```
  // TIM14 CH1 PWM on PA4

  // assume that all relevant clocks (SYSCLK, HCLK, PCLK, TPCLK are 12 MHz)
  RCC->APBENR2 |= RCC_APBENR2_TIM14EN;  // enable TIM14 clock
  (void)RCC->APBENR2; // ensure that the last write command finished and clock is on

  TIM14->PSC = 12000-1;         // set PSC such that CNT will increment each millisecond (ms)
  TIM14->ARR = 1000-1;          // set period to 1000 (ms)

  // 0b00 CH1  channel is configured as output
  TIM14->CCMR1 = (TIM14->CCMR1 &~TIM_CCMR1_CC1S_Msk) | (0 << TIM_CCMR1_CC1S_Pos);
  // 0b0110 CH1  PWM mode 1: In up-counting, channel 1 is active as long as CNT < CCR1
  TIM14->CCMR1 = (TIM14->CCMR1 &~TIM_CCMR1_OC1M_Msk) | (6<<TIM_CCMR1_OC1M_Pos);
  TIM14->CCMR1 |= TIM_CCMR1_OC1PE;      // OC preload enable (see reference manual)
  TIM14->CCER |= TIM_CCER_CC1E;         // enable CH1 and channel output pin
  TIM14->CCR1 = 200-1;                  // set length of active (high) phase of PWM pulse

  // set PA4 to TIM14_CH1 output see data sheet and STM32C011-DK board schematics
  RCC->IOPENR |= RCC_IOPENR_GPIOAEN; // enable clock for peripheral
  (void)RCC->IOPENR; // ensure that the last write command finished and clock is on
  // AF4 for PA4 is TIM14_CH1 see data sheet table "alternate function mapping"
  GPIOA->AFR[0] = (GPIOA->AFR[0] & ~GPIO_AFRL_AFSEL4_Msk) | (4 << GPIO_AFRL_AFSEL4_Pos); // AF4
  GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE4_Msk) | (2 << GPIO_MODER_MODE4_Pos); // 0b10=AF mode

  TIM14->CR1 |= TIM_CR1_CEN;    // enable the timer
```

11.4 Input Capture Mode

This mode can be used for time-stamping external signal pulse edges. When a timer channel is configured for input capture mode, the current value of the timer's counter register CNT is copied to the Capture/Compare Register (CCR) of that channel, when some external signal change (edge) is detected on the channels IO pin. Typical uses are pulse width and frequency measurement by measuring the time between two consecutive edges of an external clock.

```
void init_TIM14_IC(void) {
    // assume that all relevant clocks (SYSCLK, HCLK, PCLK, TPCLK are 12 MHz)
    RCC->APBENR2 |= RCC_APBENR2_TIM14EN;  // enable TIM14 clock
    (void)RCC->APBENR2; // ensure that the last write command finished and clock is on

    TIM14->PSC = 12-1;      // CNT will increment each microsecond (µs)
    TIM14->ARR = 0xFFFF;    // use full period for a 16-bit timer

    // follow ref.man 17.3.5 Input capture mode
    // TIM14->TISEL = 0 << TIM_TISEL_TI1SEL_Pos;   // TIM14_CH1 external input
    TIM14->TISEL = 3 << TIM_TISEL_TI1SEL_Pos;      // MCO internal input

    TIM14->CCMR1 = 1 << TIM_CCMR1_CC1S_Pos; // CC1 channel is configured as input, IC1 is mapped on TI
    // we don't use input filter or prescaler

    TIM14->CCER = (0 << TIM_CCER_CC1NP_Pos) | (0 << TIM_CCER_CC1P_Pos); // capture on rising edge
    TIM14->CCER |= TIM_CCER_CC1E;           // enable input capture mode

    TIM14->DIER |= TIM_DIER_CC1IE;  // TIM14 capture compare channel 1 interrupt enable
    NVIC_EnableIRQ(TIM14_IRQn);     // enable this interrupt in NVIC
    // Note: some timers have different interrupts for different events, e.g. TIM1

    TIM14->CR1 |= TIM_CR1_CEN;    // enable the timer
}
```

```
volatile uint16_t measured_period;
void TIM14_IRQHandler() {       // function name must match startup code g_pfnVectors entry
  if(TIM14->SR & TIM_SR_CC1IF) {    // TIM14 capture compre channel 1 interrupt flag set ?
    TIM14->SR &= ~TIM_SR_CC1IF;     // clear TIM14 capture compare channel 1 interrupt flag
    static uint16_t last;           // last capture value, initially 0
    uint16_t curr = TIM14->CCR1;    // get new capture value
    measured_period = curr - last;  // this is correct mod 65636, even if overflow occured
    last = curr;                    // save capture value for next interrupt
    GPIOA->ODR ^= 1<<7;             // toggle PA7
  }
}
```

For testing, we configure LSI as clock source, having a nominal frequency of 32 kHz, subdivide it by a factor of 64 and route that clock signal to the MCO (Master Clock Output). The expected subdivided clock frequency is 500 Hz. Since Timer TIM14 runs at 1 MHz (1 µs increments), we expect a measured period of about 2000 TIM14 timer ticks which correspond to 2000µs.

Notes:

- MCO can be observed on pin PA8 with a scope or logic analyzer by setting PA8 to alternate function output mode 0
- Since the timer is clocked by HSI which is not very accurate (see data sheet [3]), the measurement is also not very accurate. Better results could be achieved if the MCU uses an accurate external crystal/ceramic resonator (HSE crystal) or external clock source (HSE bypass).
- When PA8 is configured in alternate function mode AF13 (data sheet [3]) as TIM14_CH1 pin and TISEL is set to 0 the timer will measure any externally applied clock.
- The filter in the input stage of the timer channel can be used to suppress short spikes in the input signal, but is not used here.
- The divider in the input stage of the timer channel can be used as a prescaler of the input signal.

```
    RCC->CSR2 |= RCC_CSR2_LSION;    // low speed internal oscillator (LSI) on
    while((RCC->CSR2 & RCC_CSR2_LSIRDY) != RCC_CSR2_LSIRDY); // wait for LSI ready

    // select LSI (32 kHz) as MCO
    RCC->CFGR = (RCC->CFGR & ~RCC_CFGR_MCOSEL_Msk) | (6 << RCC_CFGR_MCOSEL_Pos);

    // set MCO divider to 1:64 (2^6) which yields 500 Hz
    RCC->CFGR = (RCC->CFGR & ~RCC_CFGR_MCOPRE_Msk) | (6 << RCC_CFGR_MCOPRE_Pos);
```

11.4.1 PWM Input Mode

This mode is an extension of Input Capture Mode using a pair of channels for simultaneous measurement of some clock period and active phase, e.g. for measuring a PWM duty cycle. Only some channels of some timers can be paired, for example TIM3_CH1 and TIM3_CH2.

We use TIM14_CH1 as a PWM generator on pin PA4 in alternate function mode AF4. A PWM frequerncy of 20 ms and duty cycle of 1.5 ms is configured.

We use TIM3_CH1 for PWM capture input on pin PB6 in AF12 mode.

```c
void init_TIM14_PWM(void)
{
  // TIM14 CH1 PWM on PA4

  // assume that all relevant clocks (SYSCLK, HCLK, PCLK, TPCLK are 12 MHz)
  RCC->APBENR2 |= RCC_APBENR2_TIM14EN;  // enable TIM14 clock
  (void)RCC->APBENR2; // ensure that the last write command finished and clock is on

  TIM14->PSC = 12-1;        // set PSC such that CNT will increment each microsecond (µs)
  TIM14->ARR = 20000-1;         // set period to 20 ms

  // 0b00 CH1  channel is configured as output
  TIM14->CCMR1 = (TIM14->CCMR1 &~TIM_CCMR1_CC1S_Msk) | (0 << TIM_CCMR1_CC1S_Pos);
  // 0b0110 CH1  PWM mode 1: In upcounting, channel 1 is active as long as TIMx_CNT<TIMx_CCR1
  TIM14->CCMR1 = (TIM14->CCMR1 &~TIM_CCMR1_OC1M_Msk) | (6<<TIM_CCMR1_OC1M_Pos);
  TIM14->CCMR1 |= TIM_CCMR1_OC1PE;  // OC preload enable (see reference manual)
  TIM14->CCER |= TIM_CCER_CC1E;         // enable CH1

  TIM14->CCR1 = 1500-1;             // set active (high) phase of PWM pulse to 1.5 ms

  // set PA4 to TIM14_CH1 output see data sheet and STM32C011-DK board schematics
  RCC->IOPENR |= RCC_IOPENR_GPIOAEN; // enable clock for peripheral
  (void)RCC->IOPENR; // ensure that the last write command finished and clock is on
  // AF4 for PA4 is TIM14_CH1 see data sheet table "alternate function mapping"
  GPIOA->AFR[0] = (GPIOA->AFR[0] & ~GPIO_AFRL_AFSEL4_Msk) | (4 << GPIO_AFRL_AFSEL4_Pos); // AF4
  GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE4_Msk) | (2 << GPIO_MODER_MODE4_Pos); // 0b10=AF mode

  TIM14->CR1 |= TIM_CR1_CEN;    // enable the timer
}
```

```c
void init_TIM3_PWM_IC(void) {
    // assume that all relevant clocks (SYSCLK, HCLK, PCLK, TPCLK are 12 MHz)
    RCC->APBENR1 |= RCC_APBENR1_TIM3EN;  // enable peripheral clock
    (void)RCC->APBENR1; // ensure that the last write command finished and clock is on

    TIM3->PSC = 12-1;      // CNT will increment each microsecond (µs)
    TIM3->ARR = 0xFFFF;    // use full period for a 16-bit timer

    // follow ref.man  16.3.6 PWM input mode
    TIM3->TISEL = 0 << TIM_TISEL_TI1SEL_Pos;   // TIM3_CH1 external input

    TIM3->CCMR1 =
        1 << TIM_CCMR1_CC1S_Pos      // CC1 channel is configured as input, IC1 is mapped on TI
    |   2 << TIM_CCMR1_CC2S_Pos;     // CC2 channel is configured as input, IC1 is mapped on TI

    TIM3->CCER  = (0 << TIM_CCER_CC1P_Pos) | (0 << TIM_CCER_CC1NP_Pos); // CC1 capture on rising edge
    TIM3->CCER |= (1 << TIM_CCER_CC2P_Pos) | (0 << TIM_CCER_CC2NP_Pos); // CC2 capture on falling edge

    //  0b00101 TI1FP1 selected as trigger input
    TIM3->SMCR = (TIM3->SMCR &~TIM_SMCR_TS_Msk) | (5 << TIM_SMCR_TS_Pos);
    //  0b100 slave mode controller in reset mode
    TIM3->SMCR = (TIM3->SMCR &~TIM_SMCR_SMS_Msk) | (4 << TIM_SMCR_SMS_Pos);

    TIM3->CCER |= TIM_CCER_CC1E;        // enable channel 1 input capture mode
    TIM3->CCER |= TIM_CCER_CC2E;        // enable channel 2 input capture mode
    TIM3->CR1 |= TIM_CR1_CEN;           // enable the timer

    // PB6 AF12 as TIM3_CH1
    // enable GPIOB port by switching its clock on
    RCC->IOPENR = (RCC->IOPENR & ~RCC_IOPENR_GPIOBEN_Msk) | (1 << RCC_IOPENR_GPIOBEN_Pos);
    (void)RCC->IOPENR; // ensure that the last write command finished and the clock is on

    GPIOB->MODER = (GPIOB->MODER & ~GPIO_MODER_MODE6_Msk) | (2 << GPIO_MODER_MODE6_Pos);
    GPIOB->AFR[0] = (GPIOB->AFR[0] & ~GPIO_AFRL_AFSEL6_Msk) | (12 << GPIO_AFRL_AFSEL6_Pos);
}
```

11.5 Timer Synchronization

Timers can generate triggers for other peripherals. A prominent example is the timely periodic ADC measurement for audio (chapter 12.3) or general input signal sampling. Some timers have

a dedicated internal trigger output (TRGO) or even two of them (TRGO2). These signals can be connected to other peripherals, and, in fact other timers.

The following example shows TIM1 in up-counting mode generating a trigger once per second. This trigger is fed into TIM3 as a TRGI clock signal. Therefore, TIM3 increments with a 1Hz clock.

```
    // setup TIM3 as the slave timer
    RCC->APBENR1 |= RCC_APBENR1_TIM3EN;  // enable TIM3 clock
    (void)RCC->APBENR1; // ensure that the last write command finished and the clock is on

    // External Clock Mode 1 - Rising edges of the selected trigger (TRGI) clock the counter.
    TIM3->SMCR = (TIM3->SMCR &~TIM_SMCR_SMS_Msk) | (7 << TIM_SMCR_SMS_Pos);
    TIM3->SMCR |= TIM_SMCR_MSM; // delay TRGI to perfectly sync master and slave timers
    // use ITR0 as trigger which is TIM1. caution: TIM_SMCR_TS_Msk is not as continous range of bits!
    TIM3->SMCR &= ~TIM_SMCR_TS_Msk;

    TIM3->CR1 |= TIM_CR1_CEN;   // enable the timer (start counting)
```

```
   // setup TIM1 as the master timer
    RCC->APBENR2 |= RCC_APBENR2_TIM1EN;  // enable TIM1 clock
    (void)RCC->APBENR2; // ensure that the last write command finished and the clock is on

    // ARR[15:0] Auto-reload register - (reset value: 0xFFFF)
    TIM1->ARR = 1000-1;

    // PSC[15:0] Prescaler register - divides the counter clock by factor PSC+1. (reset value: 0x0000)
    TIM1->PSC = 12000-1;        // set PSC such that CNT will increment each millisecond

    // the timer update event is selected as trigger output (TRGO)
    TIM1->CR2 = (TIM1->CR2 & ~TIM_CR2_MMS_Msk) | (2 << TIM_CR2_MMS_Pos);

    TIM1->CR1 |= TIM_CR1_CEN;   // enable the timer (start counting)
```

You can watch TIM3 counting in Visual Studio Code using the **Cortex Live Watch** pane when you permanently copy the counter to an otherwise useless global variable:

```
uint32_t tim3_cnt;  // used for live watching

int main(void)
{
    // assume that all relevant clocks (SYSCLK, HCLK, PCLK, TPCLK are 12 MHz)

    // setup and start the slave timer
…
    // setup and start the master timer
…
    /* Loop forever */
    for(;;) {
        tim3_cnt = TIM3->CNT;
    }
}
```

Further modes of synchronization are described in the reference manual: reset mode, gated mode, and trigger mode.

12 ANALOG-TO-DIGITAL CONVERTER (ADC)

The Analog-Digital-Converter (ADC) is an advanced peripheral for measuring analog voltages on many MCU pins with 12-bit digital conversion results in the range 0..4095. Those analog input voltages must be between Vss (digital read: 4095) and Vdd (digital read: 0). For other voltages or high impedance measurements, some external circuitry is needed in addition to the ADC like a voltage divider, operational amplifier, and so on.

The ADC has two **internal channels** for measuring a precise **reference voltage** V_refint generated internally and, indirectly, the chip **temperature** by measuring the voltage V_sense of an internal temperature sensitive element.

The first channel can be used to estimate the chips actual Vdd which in turn can be used to convert other voltage measurement from digital numbers to physical units (mV). See below and the reference manual [2] section “Converting a supply-relative ADC measurement to an absolute voltage value”.

The ADC has many features for fast, timely measurements of multiple channels (inputs) which is essential to motor control and other advanced topics. Measurements can be triggered by software or hardware triggers, often provided by timers. The measurement results can be read from the data register (DR) by software or stored in arrays by using DMA.

Other advanced ADC features include analog watchdogs which can trigger interrupts when a configured target voltage range is missed.

12.1 ADC Single Channel Software Triggered Measurement

```c
#include <stm32c011xx.h>
// global variables can be read by cortex-debug live watch or STM32CubeMonitor
uint32_t adc_data_raw;  // raw 12-bit ADC result
uint32_t adc_data_mV;   // in millivolt

int main(void)
{
    // the GPIO pins used for ADC are already in analog mode after reset
    // no need to re-config GPIO here (unless used before)

    // let ADC (digital block) be clocked by: SYSCLK
    RCC->CCIPR = (RCC->CCIPR &~RCC_CCIPR_ADCSEL_Msk) | (0<<RCC_CCIPR_ADCSEL_Pos);
    RCC->APBENR2 |= RCC_APBENR2_ADCEN;  // turn ADC clock on
    (void)RCC->APBENR2; // read back to make sure that clock is on
    ADC1->CR |= ADC_CR_ADVREGEN; // power up ADC voltage regulator
    // wait  t_ADCVREG_STUP (ADC voltage regulator start-up time),
    for(volatile int i=0; i<12*20; ++i); // min 20 µs see data sheet

    // do self-calibration
    ADC1->CR |= ADC_CR_ADCAL;
    while(ADC1->CR & ADC_CR_ADCAL); // wait for calibration to finish
    uint8_t calibration_factor = ADC1->DR;

    ADC1->CFGR1 = 0;    // default config after reset
    ADC1->CFGR2 = 0;    // default config after reset
    ADC1->SMPR  = 0;    // sampling time register, default after reset
    // "enable the ADC" procedure from RM0490 Rev 3:
    ADC1->ISR |= ADC_ISR_ADRDY; //  Clear the ADRDY bit in ADC_ISR register
    ADC1->CR |= ADC_CR_ADEN;    //  Set ADEN = 1 in the ADC_CR register.
    while(!(ADC1->ISR & ADC_ISR_ADRDY)); //  Wait until ADRDY = 1 in the ADC_ISR register

    ADC1->CALFACT = calibration_factor;

    // above: CHSELRMOD = 0 in ADC_CFGR1, so every channel has a bit. set bit to activate that channel
    ADC1->CHSELR = ADC_CHSELR_CHSEL8; // select channel ADC_IN8 which is PA8 connected to joystick
    while(!(ADC1->ISR & ADC_ISR_CCRDY)); // wait until channel configuration update is applied

    uint32_t Vdda_mV = 3300;    // there are more precise ways to estimate Vdda

    while(1) {
        ADC1->CR |= ADC_CR_ADSTART;         // start ADC conversion
        while(!(ADC1->ISR & ADC_ISR_EOC));  // wait for end of conversion
        adc_data_raw = ADC1->DR;            // conversion done. store result
        adc_data_mV = (adc_data_raw * Vdda_mV) / 4095;  // Vdda == 4095 digital reading
    }
}
```

12.2 ADC Temperature and Vdda Measurements with DMA

We want to measure 6 ADC channels, the last 4 of them a special internal channels.

ADC Channel Nr.	ADC CHSELR Bit Name	Comment
7	ADC_CHSELR_CHSEL7	PA7 (freely available)
8	ADC_CHSELR_CHSEL8	PA8 connected to joystick
9	ADC_CHSELR_CHSEL9	internal temperature sensor V_sense
10	ADC_CHSELR_CHSEL10	internal reference voltage V_refint
15	ADC_CHSELR_CHSEL15	internal connection to Vdda
16	ADC_CHSELR_CHSEL16	internal connection to Vssa

We start with two global structs. The first for the raw ADC data (0..4095 for 12-bit ADC) and the second for the same data, converted to physical units (mV respectively °C for the temperature):

```
#pragma pack(1)  // ensure a tight memory layout for the struct, without any padding bytes

volatile struct {     // the selected ADC channels by increasing channel number, raw values
  uint16_t pa7;
  uint16_t joy;
  uint16_t temp;
  uint16_t vref;
  uint16_t vdda;
  uint16_t vssa;
} adc_raw_data;

volatile struct {     // derived (calculated) values in physical units
  uint16_t pa7_mV;
  uint16_t joy_mV;
  uint16_t temp_degC;
 // uint16_t vref_mV; // vref is the reference input, not a calculated result
  uint16_t vdda_mV;
  uint16_t vssa_mV;
} adc_data;
```

The ADC initialization is similar to before, but has more channels:

```
void init_ADC(void) {
  // the GPIO pins used for ADC are already in analog mode after reset
  // no need to re-config GPIO here (unless used before)

  // let ADC (digital block) be clocked by: SYSCLK
  RCC->CCIPR = (RCC->CCIPR &~RCC_CCIPR_ADCSEL_Msk) | (0<<RCC_CCIPR_ADCSEL_Pos);
  RCC->APBENR2 |= RCC_APBENR2_ADCEN;  // turn ADC clock on
  (void)RCC->APBENR2; // read back to make sure that clock is on
  ADC1->CR |= ADC_CR_ADVREGEN; // power up ADC voltage regulator
  // wait  t_ADCVREG_STUP (ADC voltage regulator start-up time),
  for(volatile int i=0; i<12*20; ++i); // min 20 µs see data sheet

  // do self calibration
  ADC1->CR |= ADC_CR_ADCAL;
  while(ADC1->CR & ADC_CR_ADCAL); // wait for calibration to finish
  uint8_t calibration_factor = ADC1->DR;

  ADC1->CFGR1 = 0;  // default config after reset
  ADC1->CFGR2 = 0;  // default config after reset
  ADC1->SMPR = 0;   // sampling time register, default after reset
  // "enable the ADC" procedure from RM0490 Rev 3:
  ADC1->ISR |= ADC_ISR_ADRDY; //  Clear the ADRDY bit in ADC_ISR register
  ADC1->CR |= ADC_CR_ADEN;    //  Set ADEN = 1 in the ADC_CR register.
  while(!(ADC1->ISR & ADC_ISR_ADRDY)); //  Wait until ADRDY = 1 in the ADC_ISR register

  ADC1->CALFACT = calibration_factor;

  // above: CHSELRMOD = 0 in ADC_CFGR1, so every channel has a bit. set bit to activate that channel
  // ADC1->CHSELR = ADC_CHSELR_CHSEL8; // select channel ADC_IN8 which is PA8 connected to joystick
  ADC1->CHSELR =
      ADC_CHSELR_CHSEL7   // select channel ADC_IN7 which is PA7 freely available
    | ADC_CHSELR_CHSEL8   // select channel ADC_IN8 which is PA8 connected to joystick
    | ADC_CHSELR_CHSEL9   // temperature sensor V_sense
    | ADC_CHSELR_CHSEL10  // internal reference voltage V_refint
    | ADC_CHSELR_CHSEL15  // Vdda
    | ADC_CHSELR_CHSEL16  // Vssa
  ;
  while(!(ADC1->ISR & ADC_ISR_CCRDY)); // wait until channel configuration update is applied

  // At the end of ADC initialization, the internal temp. sensor must be woken up from power-down:

  ADC->CCR |= ADC_CCR_TSEN | ADC_CCR_VREFEN; // wake up temp. + vrefint blocks from power down mode
  for(volatile int i=0; i<12*15; ++i); // 15µs wait for stabilization, see data sheet
}
```

Data transfer will be done by DMA channel 1. This is a circular, endless DMA from ADC1->DR to the global array. 6 16-bit values are to be transferred. In DMAMUX->CCR the adc_dma request (no. 5, see reference manual Table 34) is configured

DMA initialization:

```
void init_DMA_ADC(void) {
    RCC->AHBENR |= RCC_AHBENR_DMA1EN; // enable peripheral clock
    (void)RCC->AHBENR; // read back to make sure that clock is on

    // route peripheral DMA request to DMA channel
    // Table 34: DMAMUX adc_dma == 5
    // caution: DMAMUX1_Channel0 is for DMA1_Channel1 and so on!
    DMAMUX1_Channel0->CCR = 5 << DMAMUX_CxCR_DMAREQ_ID_Pos;

    DMA1_Channel1->CCR &= ~DMA_CCR_EN;  // disable DMA channel for setup
    DMA1->IFCR = DMA_IFCR_CGIF1;          // clear all (HT, TC, TE) flags for DMA channel 1

    DMA1_Channel1->CPAR = (uint32_t)(&ADC1->DR);
    DMA1_Channel1->CMAR = (uint32_t)&adc_raw_data;
    DMA1_Channel1->CNDTR = sizeof(adc_raw_data) / sizeof(uint16_t);
    DMA1_Channel1->CCR =
        0 << DMA_CCR_MEM2MEM_Pos    // MEM2MEM 1: memory-to-memory mode
    |   0 << DMA_CCR_PL_Pos         // PL priority level 0: low.. 3: very high
    |   1 << DMA_CCR_MSIZE_Pos      // MSIZE 0: 8-bit 1: 16-bit 2: 32-bit
    |   1 << DMA_CCR_PSIZE_Pos      // PSIZE 0: 8-bit 1: 16-bit 2: 32-bit
    |   1 << DMA_CCR_MINC_Pos       // MINC memory increment mode   1: enable
    |   0 << DMA_CCR_PINC_Pos       // PINC peripheral increment mode   1: enable
    |   1 << DMA_CCR_CIRC_Pos       // CIRC 0 : normal mode 1: circular mode
    |   0 << DMA_CCR_DIR_Pos        // DIR 0: read from peripheral,     1: memory
    |   0 << DMA_CCR_TEIE_Pos       // TEIE transfer error interrupt    1: enable
    |   0 << DMA_CCR_HTIE_Pos       // HTIE half transfer interrupt     1: enable
    |   0 << DMA_CCR_TCIE_Pos       // TCIE transfer complete interrupt     1: enable
    |   1 << DMA_CCR_EN_Pos         // EN enable DMA channel
    ;
}
```

Finally, in the main loop the ADSTART flag is set, triggering a sequence of measurements for all configured channels in increasing order of the channel numbers. The ADC DMA request must be explicitly enabled beforehand by setting DMAEN. The while within the main loop waits for the EOS (End of Sequence) flag to be set.

Main Loop:

```c
int main(void)
{
  init_ADC();
  init_DMA_ADC();

  // sampling time set to 7 (max): 160.5 ADC clock cycles.
  // internal temperature sensor needs at least 5 µs sampling time, see data sheet
  ADC1->SMPR = (ADC1->SMPR &~ADC_SMPR_SMP1_Msk) | (7 << ADC_SMPR_SMP1_Pos);

  ADC1->CFGR1 |= ADC_CFGR1_CONT;    // 1: continuos conversion mode
  ADC1->CFGR1 |= ADC_CFGR1_DMACFG;  // 1: DMA circular mode selected
  ADC1->CFGR1 |= ADC_CFGR1_DMAEN;   // enable DMA use in ADC
  ADC1->CR |= ADC_CR_ADSTART;       // start ADC conversion sequence(s)

  for(;;) {
    // see reference manual: Calculating the actual VDDA
    int VREFINT_CAL = *VREFINT_CAL_ADDR;  // calibration value (engineering bytes)
    adc_data.vdda_mV = (VREFINT_CAL_VREF * VREFINT_CAL) / adc_raw_data.vref;

    // other voltage measurements are proportional to Vdda:
    adc_data.pa7_mV   = (adc_data.vdda_mV * adc_raw_data.pa7 ) / 4095;
    adc_data.joy_mV   = (adc_data.vdda_mV * adc_raw_data.joy ) / 4095;
    adc_data.vssa_mV  = (adc_data.vdda_mV * adc_raw_data.vssa) / 4095;

    // Reading the temperature
    int TS_CAL1 = *TEMPSENSOR_CAL1_ADDR;  // calibration value (engineering bytes)
    int t_meas_mV = adc_raw_data.temp * adc_data.vdda_mV / 4095;
    int t_calib_mV = TS_CAL1 * VREFINT_CAL_VREF / 4095;
    int Avg_Slope = 2530;   // data sheet: average slope from VSENSE voltage, but in µV/°C

    adc_data.temp_degC = TEMPSENSOR_CAL1_TEMP + (t_meas_mV-t_calib_mV) * 1000 / Avg_Slope;
  }
```

The values in the global structs can be monitored in cortex-debug live watch, STM32CubeMonitor, other debug tools, or send to the UART for logging.

It is important to note, that the ADC measurements are continuous and the for loop at the end of main is only used for result conversion and does not use the ADC in any way. There is a **data race hazard** as it is not guaranteed that the structs are not overwritten by DMA while reading in an ongoing conversion. This could be fixed by using some synchronization mechanism or by not using the CONT flag in ADC CFGR1 and starting each single sequence of 6 ADC measurements at the beginning of the for loop explicitly by setting the ADSTART flag in ADC register CR.

Notes:

1. the reference manual explains in chapter 14.10 “Temperature sensor and internal reference voltage” how to use V_sense and V_refint to calculate the chip temperature and absolute voltages (independent of Vdda)
2. Channels 15 (Vdda) and 16 (Vssa) shall produce values very close to 4095 respectively 0. This can be used for plausibility checks in your code
3. ADC allows two setting sampling times SMP1 and SMP2 in SMPR and selecting one of those for each channel independently by the SMPSEL bits

With only few configuration steps, It is straightforward to monitor global variables like `adc_data.joy_mV` live with the STM32CubeMonitor software:

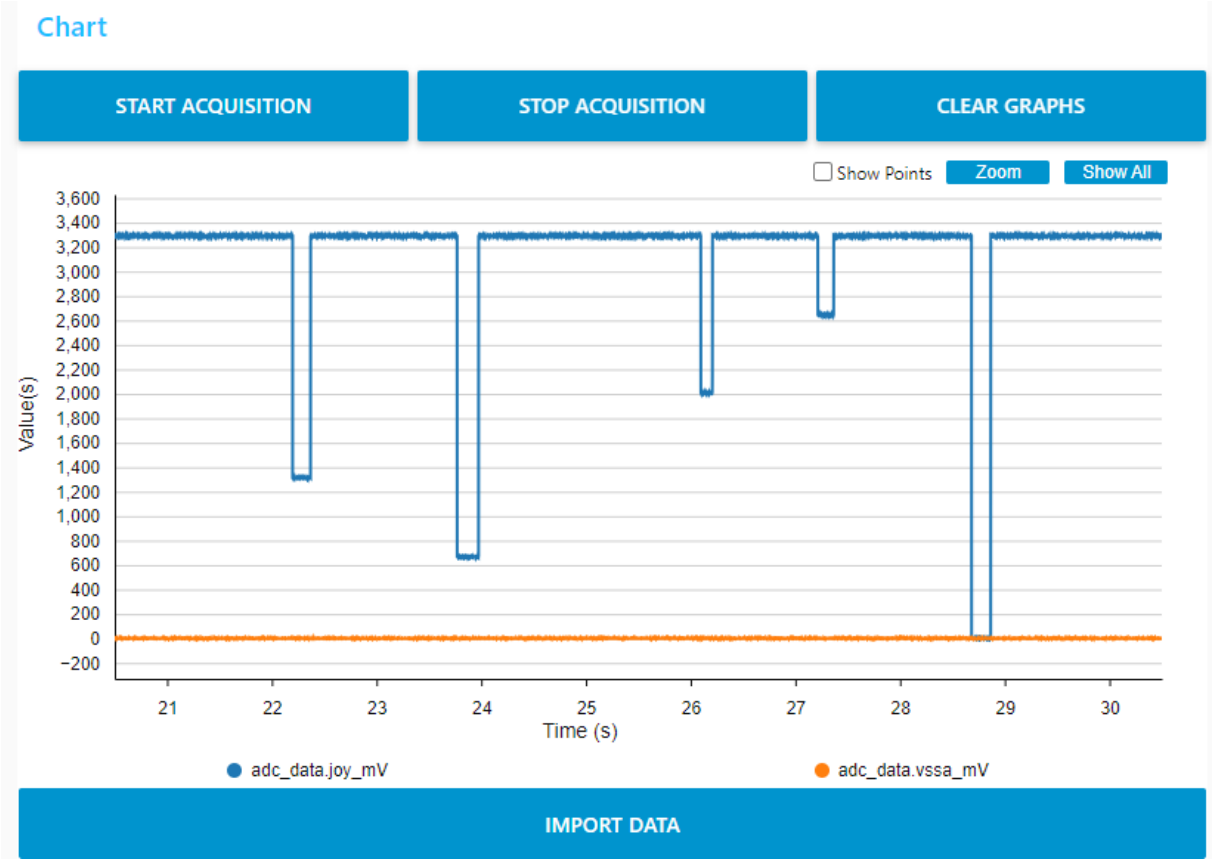


Figure 17: STM32CubeMonitor monitoring the analog voltage in mV at the joystick input

12.3 ADC Timer Triggered Multi Channel Measurement with DMA

The ADC is no longer triggered by software in the main loop, but by a periodic timer.

Table 53. „External triggers“ in the ADC chapter of the reference manual shows which triggers are possible. We opt for TRG3 (trigger 3) which is TIM3_TRGO, the trigger output from timer TIM3.

We start by configuring TIM3 as an up-counter:

```
    // assume that all relevant clocks (SYSCLK, HCLK, PCLK, TPCLK are 12 MHz)
    RCC->APBENR1 |= RCC_APBENR1_TIM3EN;  // enable TIM14 clock
    (void)RCC->APBENR1; // ensure that the last write command finished and the clock is on

    // PSC[15:0] Prescaler register - divides the counter clock by factor PSC+1. (reset value: 0x0000)
    TIM3->PSC = 12000-1;        // set PSC such that CNT will increment each millisecond
    TIM3->ARR = 10-1;           // update event generated after 10ms period, (resets TIM3->CNT to 0)
    // The update event is selected as trigger output (TRGO):
    TIM3->CR2 = (TIM3->CR2 &~TIM_CR2_MMS_Msk) | (2 << TIM_CR2_MMS_Pos);
    TIM3->CR1 |= TIM_CR1_CEN;   // counter enable: start timer
```

ADC1 initialization must additionally enable DMA continuously and select hardware trigger TRG3.

```
    ADC1->CFGR1 |= ADC_CFGR1_DMACFG;     // select DMA circular mode for repeated DMA
    ADC1->CFGR1 |= ADC_CFGR1_DMAEN;      // enable generation of DMA requests by ADC

    // enable hardware trigger detection on the rising edge
    ADC1->CFGR1 = (ADC1->CFGR1 &~ADC_CFGR1_EXTEN_Msk) | (1 << ADC_CFGR1_EXTEN_Pos);
    ADC1->CFGR1 = (ADC1->CFGR1 &~ADC_CFGR1_EXTSEL_Msk) | (3 << ADC_CFGR1_EXTSEL_Pos); // set TRG3

    ADC1->CR |= ADC_CR_ADSTART;          // start ADC conversion once a hardware trigger event occurs
```

The converted values can be stored in a ring buffer by circular DMA and further processed in the DMA half-transfer (HT) rsp. transfer-complete (TC) interrupt handlers.

Application: Digital Signal Processing, like audio frequency analysis by performing a Discrete Fourier Transform. See Arm CMSIS DSP library for ready-to-go DFT / Fast Fourier Transform code.

13 SERIAL PERIPHERAL INTERFACE (SPI)

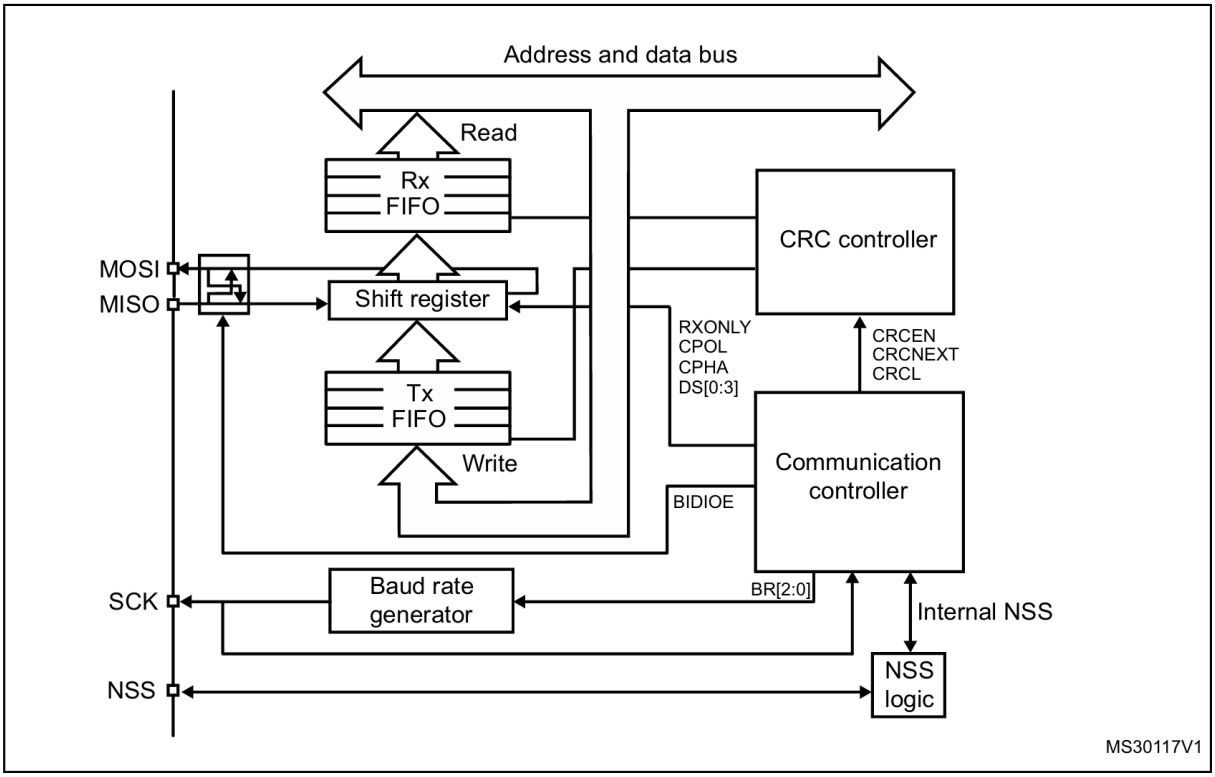


Figure 18: SPI block diagram, source: reference manual [2].

13.1 SPI FULL DUPLEX SINGLE MASTER MODE

We configure the SPI peripheral for single-master mode and the **NSS** pin is used as output: **MSTR=1, SSOE = 1, SSM=0.**

We will transmit 8-bit values (**DS=7**) on MOSI and simultaneously receive 8-bit values on MISO. The clock will be low when idle (**CPOL=0**) and the data bits will be valid and sampled at the first edge of the clock (**CPHA=0**).

The low-active SPI Slave Select (NSS) signal will be pulsed after each byte (**NSSP=1**).

The baud rate is chosen to be 1/16 of the peripheral clock (**BR=3**), see the table in the reference manual. With the default settings of SYSCLK = 12 MHz after reset, the SPI clock frequency will be 750 kHz. The transfer of 1 byte will last about 10 µs.

An array of 4 bytes will be sent. Using a logic analyzer, the following output was observed:

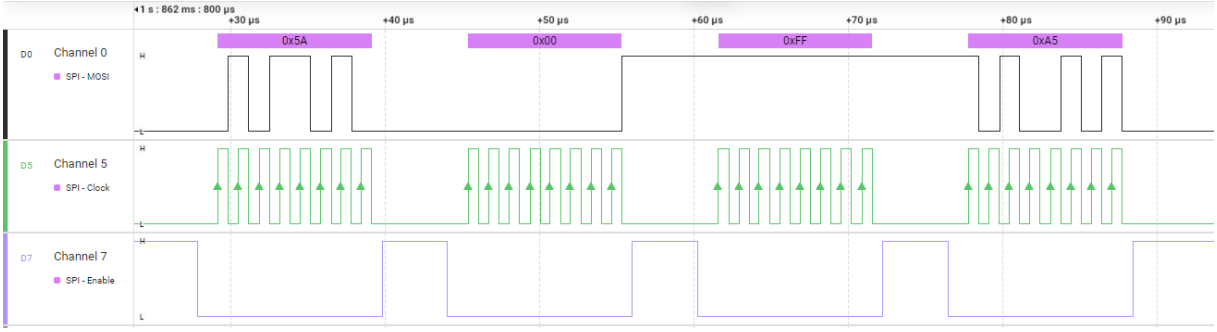


Figure 19: SPI Master Mode Output with CPHA=0 and CPOL=0 on a logic analyzer.

```c
#include <stm32c011xx.h>
void init_SPI(void)
{   // 25.5.7 Configuration of SPI (RM490)
    // 1. Write proper GPIO registers
    // PA7 -> MOSI
    // PA6 -> MISO
    // PA5 -> SCK
    // PA4 -> NSS
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN; // enable clock for this peripheral
    (void)RCC->IOPENR;                 // ensure that enable instruction completed
    // we use some shortcuts here for denser code
    // we must take into account PA13 and PA14 which are
    // used for SWD debugging and must not be changed
    GPIOA->MODER = 0xEBFFAAFF;   // PA4..PA7: alternate function
    GPIOA->OTYPER = 0x0000;      // PA4..PA7 push-pull type
    GPIOA->OSPEEDR = 0x0C000000; // PA4..PA7 low-speed
    GPIOA->PUPDR = 0x24000000;   // PA4..PA7 no pull-up, no pull-down
    GPIOA->AFR[0] = 0x00000000;  // PA4..PA7: AF0 = SPI (data sheet)

    RCC->APBENR2 |= RCC_APBENR2_SPI1EN; // enable clock for this peripheral
    (void)RCC->APBENR2;                 // ensure that enable instruction completed and the clock is on
    // 2.  Write to the SPI_CR1 register:
    SPI1->CR1 =
        3 << SPI_CR1_BR_Pos         //  BR[2:0] baud rate. 3 (0b011): f_PCLK/16
        | 0 << SPI_CR1_CPHA_Pos     // clock phase. data bit sampled at: 0=first clock edge; 1=second
        | 0 << SPI_CR1_CPOL_Pos     // clock polarity: 0: clock low when idle; 1=high when idle
        | 0 << SPI_CR1_LSBFIRST_Pos // least significant bit (LSB): 0=LSB last, 1=LSB first
        | 0 << SPI_CR1_SSM_Pos      // software slave mgmt: 0: disabled (NSS pin used) 1: enabled (SSI
bit used)
        | 1 << SPI_CR1_SSI_Pos      // internal slave select bit (when NSS pin is not used, SSM=0)
        | 1 << SPI_CR1_MSTR_Pos     // 0=slave configuration; 1=master configuration
        ;
    // 3.  Write to the SPI_CR2 register:
    SPI1->CR2 =
        (8 - 1) << SPI_CR2_DS_Pos // DS: data size (x+1 bit). Here: 8 bit (1 byte)
        | 1 << SPI_CR2_SSOE_Pos   // 1: NSS pin is output, managed by the hardware, for single-master
        | 0 << SPI_CR2_FRF_Pos    // FRF frame format: Motorola (default) mode, not TI (special) mode
        | 1 << SPI_CR2_NSSP_Pos   // 1: generate NSS pulse after each byte. (only when CPHA==0)
        | 1 << SPI_CR2_FRXTH_Pos  // FRXTH (RX FIFO threshold) must be 1 for 8-bit transfers
        ;
    // 4. Write to SPI_CRCPR register: Configure the CRC polynomial if needed. (not used)
    // 5. Write proper DMA registers (not used)
    SPI1->CR1 |= SPI_CR1_SPE; // SPI enable
}
```

```c
int main(void)
{
    init_SPI();

    uint8_t tx[4] = {0x5A, 0x00, 0xFF, 0xA5};
    uint8_t rx[4]; // variable 'rx' set but not used

    for (unsigned i = 0; i < sizeof(tx) / sizeof(tx[0]); ++i)
    {
        // the ugly type casts below are needed to enforce
        // 8-bit store/load instructions to/from DR

        while (!(SPI1->SR & SPI_SR_TXE))
            ; // wait for TXE (transmit buffer empty)

        // put next byte to send into data register
        *(__IO uint8_t *)(&SPI1->DR) = (uint8_t)tx[i];

        // hardware writes value of DR to MOSI and simultaneously reads new DR value from MISO

        while (!(SPI1->SR & SPI_SR_RXNE))
            ; // wait for RXNE (receive buffer not empty)

        // get next byte received from data register
        rx[i] = *(__IO uint8_t *)(&SPI1->DR);
    }

    /* Loop forever */
    for (;;)
        ;
}
```

If MOSI and MISO are connected (external loopback), then at the end rx will contain the same values as tx. Alternatively, if MISO is connect to Vdd or GND, e.g by setting an internal pull-up/pull-down, rx will be filled to all 0xFF respectively all 0x00.

14 INTER-INTEGRATED CIRCUIT (I2C) INTERFACE

We use I2C1 with SCL (serial clock) signal on pin PB7 and SDA (serial data) signal on pin PC14. The MCU is used as I2C controller (formerly known as master). A BH1750 light sensor is used as I2C target (formerly known as slave). When using another target, change the I2C bus address of the target and access the I2C registers of the target according to its data sheet.

14.1 I2C TIMING

The reference manual [2] recommends using STM32CubeMX for computing the timing constants for the I2C_TIMINGR register:

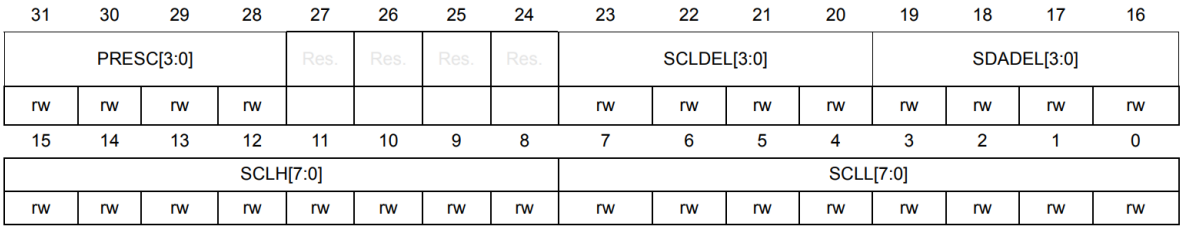


Figure 20: I2C timing register, source: reference manual [2].

Example: for I2C1 clock = 12 MHz and I2C Standard Mode @ 100kHz, STM32CubeMX calculates I2C_TIMINGR= 0x40000A0B:

	I2C1 clock	12	MHz	
	t_clock	83	ns	
PRESC	4	417	ns	$t_{PRESC} = (PRESC + 1) \times t_{I2CCLK}$
	0			reserved
SCLDEL	0	417	ns	$t_{SCLDEL} = (SCLDEL + 1) \times t_{PRESC}$
SDADEL	0	0	ns	$t_{SDADEL} = SDADEL \times t_{PRESC}$
SCLH	0x0A = 10	4583	ns	$t_{SCLH} = (SCLH + 1) \times t_{PRESC}$
SCLL	0x0B = 11	5000	ns	$t_{SCLL} = (SCLL + 1) \times t_{PRESC}$

Which makes a clock period of SCLH+SCLL = 10 µs (100 kHz). See reference manual [2] Table 94 ff. for more examples of timing settings.

14.2 I2C Master Initialization

```
// SCL: PB7 SDA: PC14
void init_i2c(void) {

    // see ref.man.  23.4.9 I2C master mode

    RCC->APBENR1 |= RCC_APBENR1_I2C1EN; // enable peripheral clock
    (void)RCC->APBENR1; // ensure that the last write command finished and the clock is on

    I2C1->CR1 &=~I2C_CR1_PE; // disable I2C peripheral for setup

    // default filter settings
    I2C1->CR1 &= ~I2C_CR1_ANFOFF;
    I2C1->CR1 &= ~I2C_CR1_DNF;

    I2C1->TIMINGR = 0x40000A0B; // for I2C1 clock = 12 MHz and I2C Standard Mode @ 100kHz

    I2C1->CR1 &= ~I2C_CR1_NOSTRETCH; // enable clock stretching

    I2C1->CR1 |= I2C_CR1_PE; // enable I2C peripheral

    // Configure the Pins for I2C
    RCC->IOPENR |= RCC_IOPENR_GPIOBEN | RCC_IOPENR_GPIOCEN; // enable GPIOB and GPIOC clocks
    (void)RCC->IOPENR; // ensure that the last write command finished and the clock is on

    // Set PB7 to I2C1_SCL AF mode
    GPIOB->OTYPER = (GPIOB->OTYPER &~GPIO_OTYPER_OT7_Msk) | (1 << GPIO_OTYPER_OT7_Pos); // open-drain
    GPIOB->OSPEEDR = (GPIOB->OSPEEDR &~GPIO_OSPEEDR_OSPEED7_Msk)|(3<<GPIO_OSPEEDR_OSPEED7_Pos);//v.high
    GPIOB->PUPDR = (GPIOB->PUPDR &~GPIO_PUPDR_PUPD6_Msk) | (1<<GPIO_PUPDR_PUPD6_Pos); // pull-up
    //GPIOB->BSRR = GPIO_BSRR_BS6; // GPIO output level - not used
    GPIOB->AFR[0] = (GPIOB->AFR[0] & ~GPIO_AFRL_AFSEL7_Msk) | (14 << GPIO_AFRL_AFSEL7_Pos); // AF14
    GPIOB->MODER = (GPIOB->MODER & ~GPIO_MODER_MODE7_Msk) | (2 << GPIO_MODER_MODE7_Pos);   // AF mode

    // Set PC14 to I2C1_SDA AF mode
    GPIOC->OTYPER = (GPIOC->OTYPER &~GPIO_OTYPER_OT14_Msk) | (1 << GPIO_OTYPER_OT14_Pos); // open-drain
    GPIOC->OSPEEDR = (GPIOC->OSPEEDR &~GPIO_OSPEEDR_OSPEED14_Msk)|(3<<GPIO_OSPEEDR_OSPEED14_Pos);
    GPIOC->PUPDR = (GPIOC->PUPDR &~GPIO_PUPDR_PUPD14_Msk) | (1<<GPIO_PUPDR_PUPD14_Pos); // pull-up
    //GPIOC->BSRR = GPIO_BSRR_BS6; // GPIO output level - not used
    GPIOC->AFR[1] = (GPIOC->AFR[1] & ~GPIO_AFRH_AFSEL14_Msk) | (14 << GPIO_AFRH_AFSEL14_Pos); // AF14
    GPIOC->MODER = (GPIOC->MODER & ~GPIO_MODER_MODE14_Msk) | (2 << GPIO_MODER_MODE14_Pos);   // AF mode
}
```

14.3 I2C Master Transmit

```
// nbytes must be 1..255
int i2c_write(uint8_t addr, uint8_t nbytes, uint8_t *data) {

    while(I2C1->ISR & I2C_ISR_BUSY);    // wait for bus free

    // ref.man.  Figure 228. Transfer sequence flow for I2C master transmitter for N ≤ 255 byte

    I2C1->CR2 = nbytes << I2C_CR2_NBYTES_Pos; // length of the data transfer
    I2C1->CR2 |= ((addr<<1) << I2C_CR2_SADD_Pos); // beware: 7-bit slave addr expected here
    // I2C1->CR2 &= ~I2C_CR2_AUTOEND;   // 0: for no stop generation (restart)
    I2C1->CR2 |= I2C_CR2_AUTOEND;       // 1: for automatic stop generation after the last byte
    I2C1->CR2 &=~I2C_CR2_RD_WRN;        // 0: Master requests a write transfer
    I2C1->CR2 |= I2C_CR2_START;         // send start condition followed by the address sequence

    for(;;) {
        if(I2C1->ISR & I2C_ISR_TXE) {           // transmit data register empty?
            I2C1->TXDR = *data++;               // transmit next byte, this clears TXE
            if(--nbytes==0)                     // last byte transmitted?
                return 1;                       // normal end of transfer
        } else if(I2C1->ISR & I2C_ISR_NACKF) {
            return 0;                           // no ACK seen on I2C bus, abort transfer
        }
    }
}
```

14.4 I2C MASTER RECEIVE

```
// nbytes must be 1..255
int i2c_read(uint8_t addr, uint8_t nbytes, uint8_t *data) {

    while(I2C1->ISR & I2C_ISR_BUSY);    // wait for bus free

    // ref.man.  Figure 231. Transfer sequence flow for I2C master receiver for N ≤ 255 bytes

    I2C1->CR2 = nbytes << I2C_CR2_NBYTES_Pos; // length of the data transfer
    I2C1->CR2 |= ((addr<<1) << I2C_CR2_SADD_Pos); // beware: 7-bit slave addr expected here
    // I2C1->CR2 &= ~I2C_CR2_AUTOEND;   // 0: for no stop generation (restart)
    I2C1->CR2 |= I2C_CR2_AUTOEND;       // 1: for automatic stop generation
    I2C1->CR2 |= I2C_CR2_RD_WRN;        // 1: Master requests a read transfer
    I2C1->CR2 |= I2C_CR2_START;         // send start condition followed by the address sequence

    for(;;) {
        if(I2C1->ISR & I2C_ISR_RXNE) {          // receive data register not empty?
            *data++ = I2C1->RXDR;               // read next byte, this clears RXNE
            if(--nbytes==0)                     // last byte received?
                return 1;                       // normal end of transfer
        } else if(I2C1->ISR & I2C_ISR_NACKF) {
            return 0;                           // no ACK seen on I2C bus, abort transfer
        }
    }
}
```

15 INDEPENDENT WATCHDOG (IWDG)

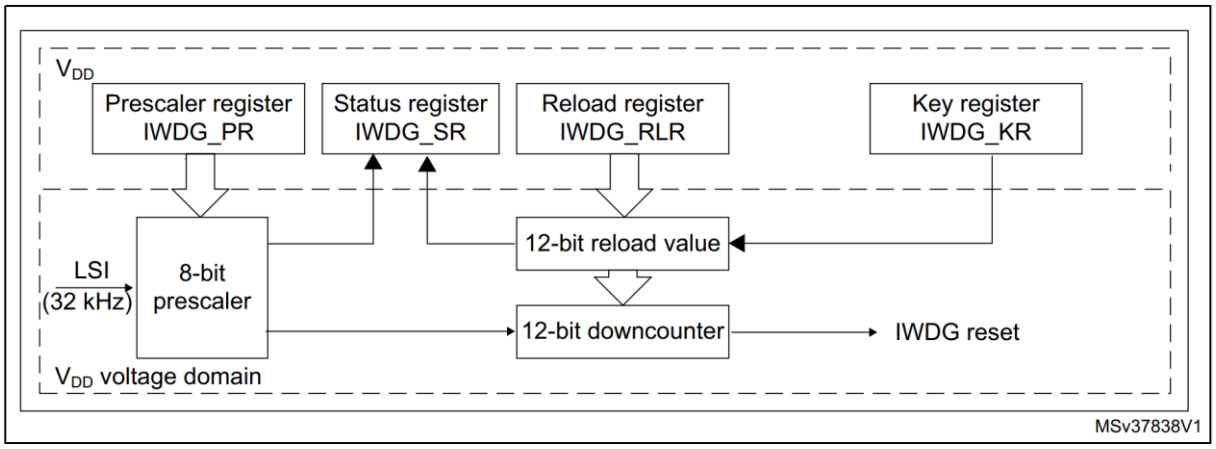


Figure 21: Independent watchdog block diagram, source: reference manual [2].

Like the RTC, the independent watchdog is clocked by the Low Speed Internal (LSI) 32 kHz clock. Therefore, it operates even when the main clock fails and in low power modes (standby mode, stop mode) when the HSIU and HSE clocks are switched off.

```c
#include <stm32c011xx.h>
int main(void) {
    for(volatile int i=0; i<1000000; ++i);   // wait some time (LED still off)

    RCC->IOPENR = (RCC->IOPENR & ~RCC_IOPENR_GPIOBEN_Msk) | (1 << RCC_IOPENR_GPIOBEN_Pos);
    (void)RCC->IOPENR; // ensure that the last write command finished and the clock is on

    GPIOB->BRR = 1 << 6;    // writing to BRR sets GPIOB bit 6 to low. LED on PB6 is low-active -> on.

    // Set PB6 to general purpose output mode
    // other defaults are okay after reset (push-pull, low speed, no pull-up, no pull-down)
    GPIOB->MODER = (GPIOB->MODER & ~GPIO_MODER_MODE6_Msk) | (1 << GPIO_MODER_MODE6_Pos);
    // LED is now on

    // IWDG is independently clocked by 32 kHz LSI clock
    IWDG->KR = 0x5555;      // key register: unprotect access to IWDG registers. must be done first

    while(IWDG->SR & IWDG_SR_PVU); // must wait until hardware is ready for register update
    IWDG->PR = 3;           // Prescale Register: set clock divider to 32. IWDG counts with 1 kHz

    while(IWDG->SR & IWDG_SR_RVU); // must wait until hardware is ready for register update
    IWDG->RLR = 1000;       // Reload Register: IWDG must be refreshed every second (1000 ms)

    IWDG->KR = 0xAAAA;      // Key Register: reload the watchdog down counter to RLR value
    IWDG->KR = 0xCCCC;      // Key Register: start the watchdog

    /* Loop forever */
    for(;;) {
        for(volatile int i=0; i<100000; ++i);   // wait some time, but not too long...
        IWDG->KR = 0xAAAA;      // key register: reload the watchdog down counter, keeps IWDG calm
    }
}
```

After the reset, the most recent reset cause (IWDG_RESET) can be read from a register, see the RCC chapter 18.2. This can be used for logging or error handling.

The IWDG can be put in **hardware mode** by programming the option bytes accordingly, see the chapter on Flash. In hardware mode, the IWDG is automatically started by hardware after reset. This can be used for supervising early boot processes etc..

The IWDG reset can be used purposedly for periodic self-wakeup from low power modes, see chapter 19.

There is another watchdog, the WWDG (windowed watchdog), which has similar features, but runs on a faster clock. The WWDG can trigger an interrupt shortly before the countdown expired and a reset is triggered. The interrupt handler may do emergency things or put the system into a safe state.

A in-depth discussion of using watchdogs in general is given in [20].

16 REAL-TIME CLOCK (RTC)

The Real Time Clock (RTC) contains registers for date and time keeping even in sleep and stop power savings modes, see chapter 19. RTC interrupts will exit power saving modes and return to run mode.

In contrast to low-power STM32 series, the RTC is powered down and must be re-initialized after exiting Standby or Shutdown mode.

For long term “wall clock precision”, an external 32.768 kHz oscillator (LSE - low speed external) is needed. RTC can alternatively run on HSE (high speed external) or LSI (low speed internal) clocks, but LSI precision is limited, see the data sheet [3]. Running RTC on LSI clock is still useful for low-cost applications (no external components) spending most time in stop mode and wakeup, say about every 10 seconds for doing some short work in run mode.

16.1 CLOCK INITIALIZATION

Besides the usual peripheral clock (RTC is on APB1 bus) for accessing the RTC registers, RTC needs a separate clock RTCCLK which is, after pre-scaling, input to the time and date keeping circuitry (**calendar** clock).

```
void rtc_init(void) {
    // RTC peripheral clock init
    RCC->APBENR1 |= RCC_APBENR1_RTCAPBEN; // enable clock for peripheral
    (void)RCC->APBENR1;   // ensure that the last write finished and the clock is now on

    // RTCCLK clock init, the clock for date&time, this is not the clock for the peripheral
    RCC->CSR2 |= RCC_CSR2_LSION;    // low speed internal oscillator (LSI) on
    while((RCC->CSR2 & RCC_CSR2_LSIRDY) != RCC_CSR2_LSIRDY); // wait for LSI ready
    // set LSI as RTCCLK clock source
    RCC->CSR1 = (RCC->CSR1 &~RCC_CSR1_RTCSEL_Msk) | (2<<RCC_CSR1_RTCSEL_Pos);
    RCC->CSR1 |= RCC_CSR1_RTCEN;    // enable RTCCLK (clock for timing)
}
```

16.2 Setting Date and Time

```
void rtc_set_time_date(void) {

    // After a system reset, the application can read the INITS flag in the RTC_ICSR
    // register to check if the calendar has been initialized or not.
    if(RTC->ICSR & RTC_ICSR_INITS)
        return; // calendar initialized, don't redo.
    //  Note: counters are stopped during init. Do not re-init without need.
    RTC->WPR = 0xCA;  // disable RTC register write protection, step 1
    RTC->WPR = 0x53;  // disable RTC register write protection, step 2
    // "Calendar initialization and configuration" (see ref.man.)
    RTC->ICSR |= RTC_ICSR_INIT;             // stop RTC and enter init mode
    while(!(RTC->ICSR & RTC_ICSR_INITF));   // wait until update is allowed

    // set prescalers. The initialization must be performed in two separate write accesses
    // RTC is clocked from LSI, this is a nominal 32 kHz clock
    // STM32C011 data sheet: min. 31.04 typ. 32 max. 32.96 kHz (Vdd=3.3V, Ta=25°C)
    RTC->PRER = (32-1) << RTC_PRER_PREDIV_A_Pos;         // PREDIV_A
    // f_CK_APRE about 1 kHz. used to feed the sub second register
    RTC->PRER |= (1000-1) << RTC_PRER_PREDIV_S_Pos;      // PREDIV_S
    //  f_CK_SPRE about 1 Hz. The main clock for updating date & time

    RTC->TR = 0x132500;      // load time register with current date
    RTC->DR = 0x240630;      // load date register with current time

    RTC->ICSR &= ~RTC_ICSR_INIT; // leave the init mode
    // When the initialization sequence is complete, the calendar starts counting.

    // enable RTC register write protection
    RTC->WPR = 0xFF;
}
```

16.3 Getting Date and Time

```
        while(!(RTC->ICSR & RTC_ICSR_RSF)); // wait for shadow register synchronization
        rtc_subseconds = RTC->SSR;   // SSR or TR must be read first. This locks the shadow registers
        rtc_time = RTC->TR;
        rtc_date = RTC->DR;    // DR must be read last. unlocks the shadow registers
        // time and date are in BCD format!
        uint8_t ht = (rtc_time & RTC_TR_HT_Msk) >> RTC_TR_HT_Pos;       // hours tens digit
        uint8_t hu = (rtc_time & RTC_TR_HU_Msk) >> RTC_TR_HU_Pos;       // hours unit digit
        uint8_t mnt = (rtc_time & RTC_TR_MNT_Msk) >> RTC_TR_MNT_Pos;    // minutes tens digit
        uint8_t mnu = (rtc_time & RTC_TR_MNU_Msk) >> RTC_TR_MNU_Pos;    // minutes unit digit
        // display on a 4-digit dispaly as "ht hu  :  mnt mnu"
```

16.4 RTC Alarm

We set a periodic RTC alarm which will trigger a RTC interrupt. For the alarm, one can select which parts of the calendar must match preconfigured values. So you can set an alarm once per minute, once per hour, and so on. It is, of course, also possible to change the matching conditions after the first alarm to achieve arbitrary alarm rhythms.

For demonstration, we set an alarm once per minute:

date & time field	set to	mask
date / day	don't care (masked out)	RTC_ALRMAR_MSK4
hours	don't care (masked out)	RTC_ALRMAR_MSK3
minutes	don't care (masked out)	RTC_ALRMAR_MSK2
seconds	42	(RTC_ALRMAR_MSK1)
sub-seconds	don't care (masked out)	RTC_ALRMASSR_MASKSS

```
void rtc_set_alarm_it(void) {
    RTC->WPR = 0xCA;    // disable RTC register write protection, step 1
    RTC->WPR = 0x53;    // disable RTC register write protection, step 2

    RTC->CR &=~RTC_CR_ALRAIE;   // Alarm A interrupt disabled
    RTC->CR &=~RTC_CR_ALRAE;    // Alarm A disabled

    RTC->SCR |= RTC_SCR_CALRAF; // Writing a 1 clears the ALRAF bit in the RTC_SR register.

    while(!(RTC->ICSR & RTC_ICSR_ALRAWF)); // wait until alarm registers can be changed
    // set RTC alarm A every minute, when seconds are 42
    RTC->ALRMAR =
            1 << RTC_ALRMAR_MSK4_Pos    // Date/day don't care in alarm A comparison
        |   1 << RTC_ALRMAR_MSK3_Pos    // Hours don't care in alarm A comparison
        |   1 << RTC_ALRMAR_MSK2_Pos    // Minutes don't care in alarm A comparison
        |   0 << RTC_ALRMAR_MSK1_Pos    // Seconds *do* care in alarm A comparison
        |   4 << RTC_ALRMAR_ST_Pos      // Second tens in BCD format.
        |   2 << RTC_ALRMAR_SU_Pos      // Second units in BCD format.
        ;
    RTC->ALRMASSR =
        0 << RTC_ALRMASSR_MASKSS_Pos    // no subsecond bits are used for matching
    |   0 << RTC_ALRMASSR_SS_Pos;       // subseconds (SS) match value set to 0

    RTC->CR |= RTC_CR_ALRAE;    // Alarm A enabled
    RTC->CR |= RTC_CR_ALRAIE;   // Alarm A interrupt enabled

    EXTI->IMR1 |= EXTI_IMR1_IM19;   // enable RTC interrupt on RTC EXTI line (==19)
    NVIC_EnableIRQ(RTC_IRQn);       // enable RTC interrupt on NVIC

    // enable RTC register write protection
    RTC->WPR = 0xFF;
}
```

The RTC interrupt is routed via the Extended interrupt and event controller (EXTI) for extended low power wakeup capabilities, see chapter 8.

The interrupt handler:

```
void RTC_IRQHandler(void) {
    if(RTC->CR &  RTC_CR_ALRAIE) {
        if(RTC->SR & RTC_SR_ALRAF) {
            RTC->SCR = RTC_SCR_CALRAF; // writing 1 clears the ALRAF bit in RTC_SR
            // do something on RTC alarm: toggle LED (LED pin must be initialized before)
            GPIOB->ODR ^= GPIO_ODR_OD6;
        }
    }
}
```

and the main function:

```
#include <stm32c011xx.h>
…
int main(void)
{
    led_init();

    rtc_init();
    rtc_set_time_date();
    rtc_set_alarm_it();

    /* Loop forever */
    for(;;) {
        // enter_stop_mode_wfi();
    }
}
```

That the MCU could be put in a low-power mode when there is nothing else to do.

```
void enter_stop_mode_wfi() {
    RCC->APBENR1 |= RCC_APBENR1_PWREN;
    (void)RCC->APBENR1;
    PWR->CR1 = 0;   //  LPMS[2:0]
    SCB->SCR |= (1 << SCB_SCR_SLEEPDEEP_Pos); // enable MCU deep sleep
    __WFI();                                  // wait for interrupt, zzz…
    SCB->SCR &=~(1 << SCB_SCR_SLEEPDEEP_Pos); // disable MCU deep sleep again
}
```

For testing, move the LED toggle line from the interrupt into the for loop. The for loop body which will be executed once after each alarm.

Advanced: it is also possible to use a wakeup event instead of an interrupt.

16.5 Further reading

Application Note AN4759 „Introduction to using the hardware real-time clock (RTC) and the tamper management unit (TAMP) with STM32 MCUs”.

17 EMBEDDED FLASH MEMORY (FLASH)

The flash is normally used as a read-only memory. It consists of different memory areas for different purposes. For an overview, see the STM32C0 Memory Map in chapter 6 and the reference manual [2].

17.1 MAIN FLASH MEMORY – IMPLEMENTING A BOOT COUNTER

The **Main Flash Memory** is used for permanently storing the code, initialized global data, and read-only data. It can also be used as a persistent store for user data like program parameters, calibration values, a boot counter or an operating hour meter.

Writing to the flash is a two-step process: First a so called flash **page** must be **erased**. On a STM32C011, the main flash memory has a size of 32 kB and each page a size of 2 kB. Page 0 starts at the **FLASH_BASE** address 0x08000000, page 1 at address 0x08000800, and so on up to page 15 starting at address 0x08007800.

After erasing, the entire page is filled with 0xFF bytes. Then, the page can be **programmed**. Programming is done on a **doubleword** (64-bit) basis. Each doubleword can be programmed **only once**. As the MCU busses are 32-bit wide, writing a doubleword is not atomic and consist actually of writing two 32-bit words.

Flash wears out, and the total number of **erase cycles** is **limited**. There are libraries available for mitigating this limitation by implementing “wear levelling” like the X-CUBE-EEPROM library [21].

Flash erase time is typ. 22 ms and programming time for a doubleword about 85 µs, see the data sheet [3].

The following example assumes that the last page (15) is not used by the program and uses it for implementing a persistent boot counter. Each time when the firmware starts, the boot counter is read, incremented, and stored in the flash again. This will wear out the flash slowly (especially if a misbehaved firmware reboots ad inifinitum). However, implementing a boot counter and also an operating hour meter in flash greatly helps diagnosing devices sent in for repair or replacement.

For more advanced use cases take a look at ST Microelectronics X-CUBE-EEPROM EEPROM Emulation Library.

Before the very first firmware start, the flash memory should be erased, e.g. by using STM32CubeProgrammer, or using brand-new MCUs.

In production code, a checksum should be added to find out if the flash contains valid data.

```
// the flash memory is write protected, to write to the flash memory, the flash memory must be unlocked
void flash_unlock(void) {
    if(FLASH->CR & FLASH_CR_LOCK) {
        FLASH->KEYR = 0x45670123;
        FLASH->KEYR = 0xCDEF89AB;
    }
}

// the flash memory should be locked again after writing to it
void flash_lock(void) {
    FLASH->CR |= FLASH_CR_LOCK;
}
```

```
// Flash memory page erase, flash must be unlocked
void erase_page(uint32_t page_nr) {
    while(FLASH->SR & FLASH_SR_BSY1);   // wait until flash is not busy
    // Check and clear all error programming flags, If not, PGSERR is set.
    FLASH->CR |= FLASH_CR_PER;              // begin page erase
    FLASH->CR = (FLASH->CR &~FLASH_CR_PNB_Msk) | (page_nr << FLASH_CR_PNB_Pos);
    FLASH->CR |= FLASH_CR_STRT;
    while(FLASH->SR & FLASH_SR_BSY1);       // wait until flash is not busy
    FLASH->CR &= ~FLASH_CR_PER;             // end page erase
}
```

```
// Standard programming procedure, flash must be unlocked
void programn_doubleword(volatile uint64_t *address, uint64_t data) {
    while(FLASH->SR & FLASH_SR_BSY1);       // wait until flash is not busy
    // Check and clear all error programming flags, If not, PGSERR is set.
    FLASH->CR |= FLASH_CR_PG;               // begin programming
    *address = data;
    while(FLASH->SR & FLASH_SR_BSY1);       // wait until flash is not busy
    // Check that the EOP flag is set in FLASH->SR
    // clear the PG flag after the last 64-bit doubleword has been programmed
    FLASH->CR &=~FLASH_CR_PG;               // end programming
}
```

```
#define STM32C011xx
#include <stm32c0xx.h>

// the flash memory is divided into pages, each page has a size of 2KB
#define PAGE_SIZE   2048ul

#define MY_PAGE_NR     ((FLASH_SIZE_DEFAULT)/(PAGE_SIZE)-1u)      // last page of the flash memory

const uint32_t address_of_counter = FLASH_BASE + MY_PAGE_NR * PAGE_SIZE;

// step thru the code and use cortex-debug variables/watch to watch the counter variable
int main(void)
{
    // reading from flash memory works like reading from RAM
    uint64_t counter = *(volatile uint64_t*)address_of_counter;
    counter++;

    // writing to flash memory is a bit more complicated
    flash_unlock();
    flash_erase_page(MY_PAGE_NR);
    flash_programn_doubleword(address_of_counter, counter);
    flash_lock();

    /* Loop forever */
    for(;;);
}
```

The assumption that the last page is not used by the program should be ensured by changing the **linker description file** (.ld) of the project.

The most simple way is reducing the flash size visible to the linker from 32K to 30K bytes:

```
MEMORY
{
  RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH =  6K
  FLASH   (rx)    : ORIGIN =  0x8000000,   LENGTH = 32K - 2K
}
```

Then, the linker doesn't know about and cannot use the last page of the main flash memory.

There are various advanced methods of read protection (**RDP**), write protection (**WRP**), and more available for protecting the flash use.

17.2 Option Bytes

The Option Bytes allow configuration of some special MCU properties like **read-out protection, boot mode, and security**. These settings will be enforced by the MCU **hardware** immediately during and after reset, independent of any firmware configuration.

Caution: Programming the Option Bytes in a wrong way can irreversibly brick the MCU or make it at least difficult to recover.

The Option Bytes can be easily programmed using **STM32CubeProgrammer**.

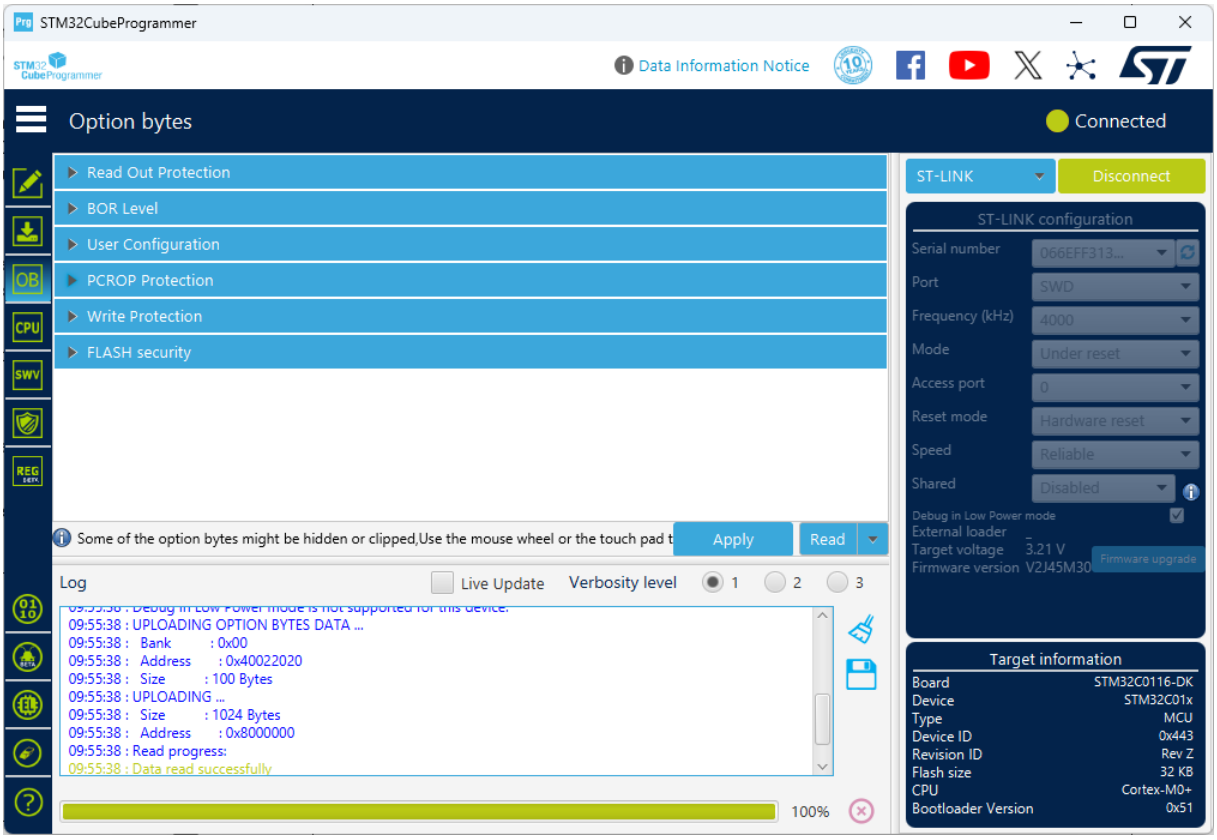


Figure 22: STM32CubeProgrammer showing the Option Bytes sections.

18 RESET AND CLOCK CONTROL (RCC)

A general overview over the MCU clock tree is shown in chapter 9, Figure 5.

After power-on or any other system reset, the MCU runs with an internally generated clock (SYSCLK) of 12 MHz, derived from the on-chip high speed oscillator (HSI48) by the HSIDIV clock divider which defaults to a value of 4.

It is possible to increase SYSCLK to 24 MHz or 48 MHz by changing clock divider HSIDIV value

```
        // set HSI clock divider to 2^1==2, the default after reset is 2^2==4
        RCC->CR = (RCC->CR & ~RCC_CR_HSIDIV_Msk) | (1 << RCC_CR_HSIDIV_Pos);
        // HCLK is now 24 MHz
```

Note: If HCLK is higher than 24MHz, the **flash latency must be increased** beforehand to ensure proper instruction fetch and therefore operation. The details are in the reference manual [2].

18.1 SWITCHING PERIPHERAL CLOCKS ON AND OFF

The peripheral blocks need a clock to operate. In order to keep the power consumption low, all clock not necessary for booting after rest are by default off and must be explicitly switched on. This is exercised in nearly every example:

```
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN;
    (void)RCC->IOPENR; // ensure that the last write command finished and the clock is on
…
    RCC->APBENR2 |= RCC_APBENR2_USART1EN;
    (void)RCC->APBENR2; // ensure that the last write command finished and the clock is on
…
```

For power saving, it might be useful to switch some clocks off before entering a power saving mode.

For some peripherals, it is also possible to choose among different clocks or changing the clock speed and therefore the power consumption. The clock tree in Figure 5 shows the possibilities, but the details are not discussed here.

18.2 EXAMINING RESET FLAGS

It is possible determining the reason why the MCU was reset. This should be done early in main after the reset when the firmware boots for the next time.

```
    if(RCC->CSR2 & RCC_CSR2_SFTRSTF) {
        printf("reset by software detected\n");
    }
    if(RCC->CSR2 & RCC_CSR2_PINRSTF) {
        printf("reset by NRST pin detected\n");
    }
    if(RCC->CSR2 & RCC_CSR2_IWDGRSTF) {
        printf("reset by IWDG watchdog detected\n");
    }
    if(RCC->CSR2 & RCC_CSR2_WWDGRSTF) {
        printf("reset by WWDG watchdog detected\n");
    }
    if(RCC->CSR2 & RCC_CSR2_PWRRSTF) {
        printf("reset by power-on (POR) or brown-out (BOR) detected\n");
    }
…
    RCC->CSR2 &= ~RCC_CSR2_RMVF; // remove reset flags before next reset
```

18.3 Monitoring a Clock

The SYSCLK clock (or other clocks) can be output on pin PA8 when configured as master clock output (MCO):

```
        // check SYSCLK freq. after reset. expected: 12 MHz
        // set MCO divider to 1:32 (2^5)
        RCC->CFGR = (RCC->CFGR & ~RCC_CFGR_MCOPRE_Msk) | (5 << RCC_CFGR_MCOPRE_Pos);
        // select SYSCLK as MCO
        RCC->CFGR = (RCC->CFGR & ~RCC_CFGR_MCOSEL_Msk) | (1 << RCC_CFGR_MCOSEL_Pos);

        // enable GPIOA port by switching its clock on
        RCC->IOPENR = (RCC->IOPENR & ~RCC_IOPENR_GPIOAEN_Msk) | (1 << RCC_IOPENR_GPIOAEN_Pos);
        (void)RCC->IOPENR; // ensure that the last write command finished and the clock is on

        // set PA8 to AF0 which is MCO output (data sheet)
        GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE8_Msk) | (2 << GPIO_MODER_MODE8_Pos);
        GPIOA->AFR[1] = (GPIOA->AFR[1] & ~GPIO_AFRH_AFSEL8_Msk) | (0 << GPIO_AFRH_AFSEL8_Pos);
        // now observe SYSCLK/32 as MCO on pin PA8 with a logic analyzer…
```

19 Power Control (PWR)

Dealing with low-power modes can get sophisticated easily and requires careful reading and understanding of the reference manual, the errata sheet, the board schematics, and ideally some reference code. This is especially true for the (ultra-)low-lower STM32 series. For an advanced discussion, see: How to enter Standby or Shutdown mode on STM32 - STMicroelectronics Community.

Concerning power saving options, the STM32C0 MCU series is clearly entry-level, but still features a variety of **low-power modes** which are summarized in the following table:

	Run mode (default)	Low-power modes			
		Sleep mode	Stop mode	Standby mode	Shutdown mode
CPU Core	active	paused	paused	-	-
SRAM	active	active	active	-	-
Flash	active	active	-	-	-
Clocks	any	any	LSE/LSI	LSI only	-
Wake-up possible by	- (awake)	IRQ or event by all peripherals and pins	RTC USART, I2C BOR IWDG all GPIO pins NRST pin	BOR IWDG WKUPx pins NRST pin	WKUPx pins NRST pin
Power consumption	58 µA / MHz	20 µA / MHz	80 µA	7.45 µA	0.019 µA
Wake-up time	-	10 cycles	5.9 µs	23 µs	385 µs

Data source: Reference Manual [2] and STM32C0 PWR training slides [13].

Compared to the reference manual, the table is slightly simplified, as many details can be fine-tuned in the firmware.

Low-power mode **entry** must configured and initiated by the firmware. **Exit** from a low-power mode is done by the hardware. After the exit, the MCU will be in run mode again.

There are other STM32 series dedicated to low (**L4, L4+, L5**) and ultra-low (**U0, U5**) power consumption. Those have special features like lower operating voltages, a battery powered backup domain and many more.

The **STM32CubeMX** software includes a **power consumption calculator** (PCC) tool, estimating and visualizing power consumption and battery life for user definable scenarios [14].

19.1 Run Mode

Run mode is the **default** operating mode after reset. The Arm Cortex-M0+ core is clocked and running. Flash and SRAM are operating. The firmware is executed **only in run mode**. Most peripherals must be explicitly enabled by setting appropriate RCC registers before they can be used.

The option bytes can be programmed to **prohibit** entering stop, standby, and shutdown modes as a safety measure for keeping the MCU awake and responsive.

Power consumption in run mode highly depends on the **clock speeds** and setting appropriate clock speeds is a viable **power-saving measure** even if no low-power mode is used.

19.2 Low-Power Modes

19.2.1 Sleep Mode

In sleep mode, the Arm Cortex core clock is stopped, but the core is still powered. The complete state of the core (registers, flags) is retained such that the core can continue execution after wakeup.

Sleep mode does not automatically influence flash, SRAM, or other peripherals. This may be configured by software to further reduce power consumption.

Sleep mode is entered when the core executes a **WFI** (wait for interrupt) or a **WFE** (wait for event) machine instruction. The core wakes up from sleep mode when an interrupt respectively a wakeup event occurs.

Sleep mode is often used in the main loop or the idle task of an operating system when nothing else is to do. In the following example, the Arm Cortex-M0+ core sleeps until the next interrupt is pending:

```
int sleepy_main(void) {
    SCB->SCR &= ~(1 << SCB_SCR_SLEEPDEEP_Pos); // clear sleep deep bit

    while(1) {
        GPIOB->ODR ^= 1 << 6;   // toggle GPIOB pin 6 (LED) as a sign of life
        __WFI();        // wait for interrupt.
    }
    return 0;
}
```

Note that in many firmware frameworks, the periodic **SysTick** interrupt (see chapter 20.2) is already activated in the startup code. Each SysTick interrupt will then wakeup the core which may lead to unexpected results. The SysTick should be suspended before and resumed after low power modes.

There is an interesting option for the core to automatically re-enter sleep mode after all interrupt handlers have finished:

```
SCB->SCR |= 1 << SCB_SCR_SLEEPONEXIT_Pos;
```

When this option is set, there is no main loop needed at all (interrupt driven software design).

19.2.2 Stop Mode (wakeup on USART1 receive)

In stop mode, the **flash is powered down** and the high speed clocks (HSI48, HSE) are shut down. Most peripherals are off. SRAM and low speed clocks (LSI, LSE) remain active.

All configured **EXTI** interrupts or wakeup events can cause the device to exit the stop mode. Therefore, it is possible to wake up the MCU at a specific date and time or with a **periodic alarm** with the Real-Time-Clock (**RTC**) , see chapter 16.4, on **UART1** receive, and on **I2C1** slave receive.

The following example sets USART1 in interrupt based receive mode enabled for stop mode. When a char is received, the MCU wakes up to run mode, handles the interrupt, and goes back to stop mode.

```
void enter_stop_mode_wfi() {
    RCC->APBENR1 |= RCC_APBENR1_PWREN;
    (void)RCC->APBENR1;
    PWR->CR1 = 0;   //  LPMS[2:0]
    SCB->SCR |= (1 << SCB_SCR_SLEEPDEEP_Pos); // enable MCU deep sleep
    __DSB();
    __WFI();                                  // wait for interrupt, zzz…
    SCB->SCR &=~(1 << SCB_SCR_SLEEPDEEP_Pos); // disable MCU deep sleep again
}
```

```c
void init_UART1(void) {
    RCC->APBENR2 |= RCC_APBENR2_SYSCFGEN;   // enable clock for peripheral component
    (void)RCC->APBENR2; // ensure that the last instruction finished and the clock is now on

    SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA11_RMP; // remap: use  PA9 instead of PA11
    SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA12_RMP; // remap: use PA10 instead of PA12

    // PA9, PA10 = USART1 TX, RX, routed to ST-LINK VCP, see STM32C011-DK board schematics
    RCC->IOPENR |= RCC_IOPENR_GPIOAEN;
    (void)RCC->IOPENR; // ensure that the last write command finished and the clock is on

    GPIOA->AFR[1] = (GPIOA->AFR[1] & ~GPIO_AFRH_AFSEL9_Msk) | (1 << GPIO_AFRH_AFSEL9_Pos); // AF1
    GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE9_Msk) | (2 << GPIO_MODER_MODE9_Pos);   // AF mode

    GPIOA->AFR[1] = (GPIOA->AFR[1] & ~GPIO_AFRH_AFSEL10_Msk) | (1 << GPIO_AFRH_AFSEL10_Pos); // AF1
    GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODE10_Msk) | (2 << GPIO_MODER_MODE10_Pos);   // AF mode

    RCC->APBENR2 |= RCC_APBENR2_USART1EN;
    (void)RCC->APBENR2; // ensure that the last instruction finished and the clock is on
    uint32_t baud_rate = 115200;
    USART1->BRR = (12000000 + (baud_rate/2)) / baud_rate;    // rounded, SYSCLK after reset
    USART1->CR1 =
        USART_CR1_UE |              // enable USART
        USART_CR1_RE |              // enable receiver
        USART_CR1_RXNEIE_RXFNEIE |  // enable RXNE interrupt            <---
        USART_CR1_UESM |            // enable USART in stop mode        <---
        USART_CR1_TE;               // enable transmitter
    NVIC_EnableIRQ(USART1_IRQn);    // enable interrupt                 <---
}
```

```c
int main(void)
{
    RCC->CR = (RCC->CR & ~RCC_CR_HSIDIV_Msk) | (2 << RCC_CR_HSIDIV_Pos); // HIS48/4 == 12 MHz (default)
    // set HSIKER HIS48/4 = 12 MHz, yes the constant is 3 here, not 2
    RCC->CR = (RCC->CR & ~RCC_CR_HSIKERDIV_Msk) | (3 << RCC_CR_HSIKERDIV_Pos);
    //RCC->CR |= RCC_CR_HSIKERON; // enable HSI in run and stop modes - (more power + faster wakeup)
    while(!(RCC->CR & RCC_CR_HSIRDY)); // wait for HSI ready

    RCC->APBSMENR2 |= RCC_APBSMENR2_USART1SMEN; // enable USART1 in sleep and stop modes
    // select HSIKER as USART1 clock source:
    RCC->CCIPR = (RCC_CCIPR_USART1SEL &~RCC_CCIPR_USART1SEL_Msk) | (2 << RCC_CCIPR_USART1SEL_Pos);
    DBG->CR &= ~DBG_CR_DBG_STOP;        // disable debug in stop mode

    init_UART1();
    /* Loop forever */
    SCB->SCR |= SCB_SCR_SLEEPONEXIT; // optional: re-enter sleep mode after ISR exit
    for(;;) {
        USART1->TDR = '$'; // send a prompt
        while(!(USART1->ISR & USART_ISR_TC));   // wait for UART transmission completed
        enter_stop_mode_wfi();                  // never returns when SCB_SCR_SLEEPONEXIT is set
    }
}
```

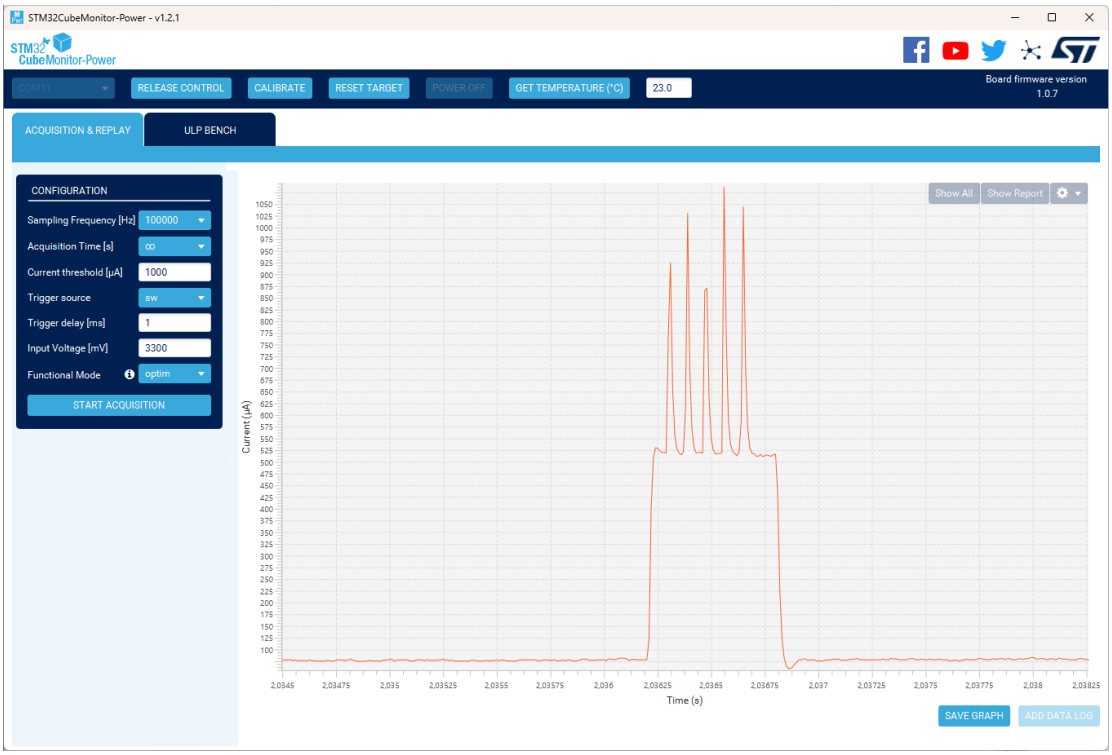


Figure 23: Stop Mode current measured for input "Hello" with X-NUCLEO-LPM01A

19.2.3 Standby Mode (periodic wakeup on IWDG timeout)

The **SRAM** is powered down and its **content is lost**.

Therefore, the code structure is different when standby mode is used: code after entering standby mode is **unreachable code**. Instead, after wakeup the Arm Cortex-M0+ core starts from the beginning as in a regular power-on or NRST pin reset. If desired, the **wakeup reason** can be examined by evaluating flags in the RCC CSR2 register.

There are few **PWR backup registers** which are kept powered and retain information while the device is in standby mode.

The **GPIO** pins are **no longer driven** unless they are configured with an internal pull-up or pull-down to **retain logic levels** needed for some external components.

A **Brown Out Reset** (BOR) is available in standby mode which allows detecting low power supply situations, e.g. sounding an alarm when the **battery goes low** in a smoke detector.

The **RTC is powered down** in standby and shutdown modes. It cannot be used and must be re-initialized after exiting these low power modes.

Periodic self-wakeup of the MCU is still possible by using the **IWDG** (for IWDG see chapter 15).

This is shown in the next example which also demonstrates how a bit **PWR backup register** is used to keep a counter variable in the MCU during standby mode.

```c
int main(void) {
    // when experimenting with low-power modes, we add some safety period
    // after reset to be able to attach a debugger, pause execution ...
    // before we possibly lose control over the MCU
    for(volatile int i=0; i<1000000; ++i);

    RCC->APBENR1 |= RCC_APBENR1_PWREN;  // enable PWR peripheral clock
    (void)RCC->APBENR1;     // read back to make sure clock is running now

    uint16_t count = PWR->BKP0R;    // read counter value which survives standby mode
    PWR->BKP0R = (count + 1) % 10;  // and increment counter mod 10 for next round

    init_LED();
    for(int i=0; i<count+1; ++i)    // indicate the counter value by blinking
        blink();

    // IWDG is independently clocked by the 32 kHz LSI clock, no need to switch a clock on
    IWDG->KR = 0x5555;      // key register: unprotect access to IWDG. must be done first

    while(IWDG->SR & IWDG_SR_PVU); // must wait until hardware is ready for register update
    IWDG->PR = 7;           // maximum pre-scaler of 256. -> IWDG counts at 125 Hz

    while(IWDG->SR & IWDG_SR_RVU); // must wait until hardware is ready for register update
    IWDG->RLR = 4095;       // maximum reload register. IWDG expires after 32,752 seconds

    IWDG->KR = 0xAAAA;      // key register: reload the watchdog down counter
    IWDG->KR = 0xCCCC;      // key register: start the watchdog

    __disable_irq();        // disable any IRQ (not used in this example, but keep in mind)

    // enter standby mode
    SCB->SCR |= (1 << SCB_SCR_SLEEPDEEP_Pos);   // enable MCU deep sleep
    PWR->CR1 = 3;                               // LPMS[2:0] – select standby mode
    (void)PWR->CR1;                             // ensure completion of previous write
    // DBGMCU->CR = 0;                          // optional: disable debug clocks running
    __DSB();                                    // data synchronization barrier
    __WFI();                                    // wait for interrupt, zzz…
    // unreachable code: Upon waking up from Standby and Shutdown mode,
    // the program execution restarts in the same way as upon a reset
}
```

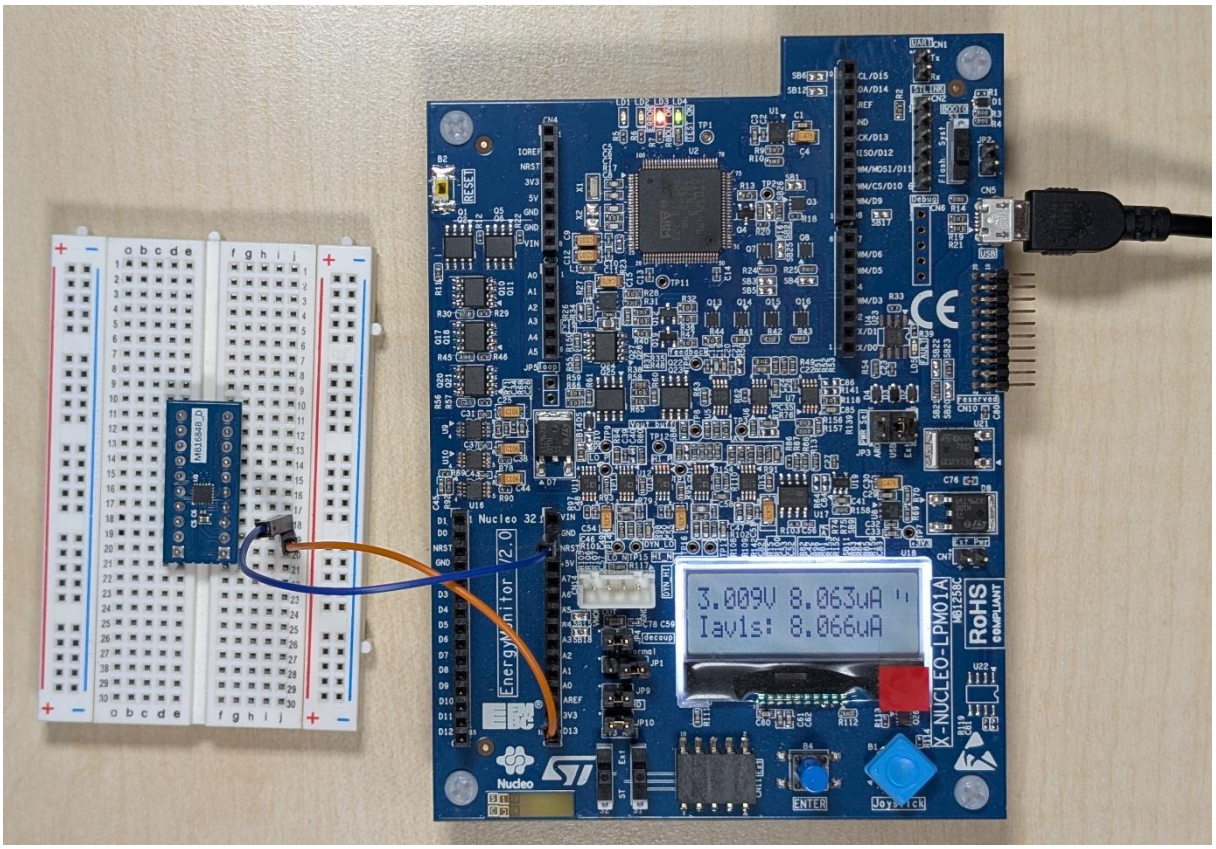


Figure 24: Detached STM32C011 MCU for current measurement

The above figure shows the STM32C011F6U6 MCU detached from the STM32C011-DK and connected to a X-NUCLEO-LPM01A Power Shield for low-power current measurement during standby mode. The STM32C011-DK base board can still serve as SWD programmer / debugger.

19.2.4 Shutdown Mode

The only active wakeup sources in this mode are some MCU pins: the **NRST** input and up to 5 (depending on the package) **dedicated wakeup pins**. There is no MCU self-wakeup from this mode possible.

Shutdown mode is useful for **smart power-on solutions** like powering the MCU on by one or more push buttons, a proximity sensor, accelerometer or any other external device which can trigger a digital level change on one of those wakeup pins.

20 CORTEX-M0+ CORE PERIPHERALS

These peripherals belong to the Arm Cortex-M0+ core and are described in the programming manual [11]. The CMSIS core library [1] provides not only register-level definitions for that, but in many cases also access functions which are mainly implemented as macros for your convenience and implementation efficiency.

20.1 NESTED VECTOR INTERRUPT CONTROLLER (NVIC)

In Arm Cortex-M parlance, **interrupts** are a subset of the set of **exceptions**. To cite Joseph You from the famous "definitive guide" book: "Exceptions are events that cause change to program control: instead of continuing program execution, the processor suspends the current executing task and executes a part of the program code called the exception handler. After the exception handler is completed, it will then resume the normal program execution." [22].

The other kind of exceptions are **faults** which occur when the Arm Cortex-M0+ core hits a faulty condition like accessing unmapped (non-existent) memory or when finding some illegal instruction in the executed code stream.

The **NVIC** is the central resource, where interrupts can be enabled, disabled and priorized, whereas the fault handlers are dealt with in the **System Control Block** (SCB).

Chris Colemans article "A Practical guide to ARM Cortex-M Exception Handling" [23] is a recommended introduction to the topic.

In most cases, you will encounter the following aspects of the NVIC:

20.1.1 Implementing an Interrupt Handler

When an interrupt handler needs to be executed, the Arm Cortex-M0+ core will fetch the interrupt handler address from a specific memory location in the **interrupt vector table**. The **linker script** takes care of placing this table at the correct memory address. This table is defined in the startup code like in the following code taken from a STM32CubeIDE generated STMC011 project:

```
g_pfnVectors:
  .word  _estack                          /* initial value loaded to stack pointer (SP) at reset */
  .word  Reset_Handler                    /* initial value loaded to program counter (PC) at reset */
  .word  NMI_Handler                      /* non maskable interrupt, last resort */
  .word  HardFault_Handler                /* triggered when a fault was detected */
  .word  0
  .word  0
  .word  0
  .word  0
  .word  0
  .word  0
  .word  0
  .word  SVC_Handler                       /* for RTOS use */
  .word  0
  .word  0
  .word  PendSV_Handler                    /* for RTOS use */
  .word  SysTick_Handler                   /* SysTick timer interrupt      */
  .word  WWDG_IRQHandler                   /* Window WatchDog              */ /* <-- NVIC related */
  .word  0                                 /* reserved                     */
  .word  RTC_IRQHandler                    /* RTC through the EXTI line    */
  .word  FLASH_IRQHandler                  /* FLASH                        */
  .word  RCC_IRQHandler                    /* RCC                          */
  .word  EXTI0_1_IRQHandler                /* EXTI Line 0 and 1            */
  .word  EXTI2_3_IRQHandler                /* EXTI Line 2 and 3            */
  .word  EXTI4_15_IRQHandler               /* EXTI Line 4 to 15            */
  .word  0                                 /* reserved                     */
  .word  DMA1_Channel1_IRQHandler          /* DMA1 Channel 1               */
  .word  DMA1_Channel2_3_IRQHandler        /* DMA1 Channel 2 and Channel 3 */
  .word  DMAMUX1_IRQHandler                /* DMAMUX1                      */
  .word  ADC1_IRQHandler                   /* ADC1                         */
  .word  TIM1_BRK_UP_TRG_COM_IRQHandler    /* TIM1 Break, Update, Trigger and Commutation */
  .word  TIM1_CC_IRQHandler                /* TIM1 Capture Compare         */
  .word  0                                 /* reserved                     */
  .word  TIM3_IRQHandler                   /* TIM3                         */
  .word  0                                 /* reserved                     */
  .word  0                                 /* reserved                     */
  .word  TIM14_IRQHandler                  /* TIM14                        */
  .word  0                                 /* reserved                     */
  .word  TIM16_IRQHandler                  /* TIM16                        */
  .word  TIM17_IRQHandler                  /* TIM17                        */
  .word  I2C1_IRQHandler                   /* I2C1                         */
  .word  0                                 /* reserved                     */
  .word  SPI1_IRQHandler                   /* SPI1                         */
  .word  0                                 /* reserved                     */
  .word  USART1_IRQHandler                 /* USART1                       */
  .word  USART2_IRQHandler                 /* USART2                       */
  .word  0                                 /* reserved                     */
  .word  0                                 /* reserved                     */
  .word  0                                 /* reserved                     */
```

The names of these handlers are arbitrary, but follow a well-established logical scheme.

Note: The startup file generation was broken. It was fixed in VS Code STM32 Extension 2.1.1.

To implement an interrupt handler, you have to define a C function with exact the same name, having no parameters and a return type of void. Examples in this document are

- the SysTick_Handler in chapter 20.2,
- the EXTI4_15_IRQHandler in chapter 8,
- the USART1_IRQHandler in chapter 9.2

and more. These examples show, that for peripheral interrupts not only the NVIC needs to be configured, but also special **interrupt enable bits** in the peripheral register block. It is often necessary to inspect the peripheral registers for the **interrupt flags** that have caused the interrupt request and to **clear the interrupt condition** in the peripheral. Failing to do so may result in an **interrupt storm**, i.e. the handler will immediately be called again in an endless sequence.

All handlers that are not explicitly implemented are usually aliased to a **Default_Handler** in the startup code for space saving reasons. The Default_Handler should never be called. If so, it indicates a missing interrupt handler (or a misspelled interrupt handler name).

20.1.2 Enabling and Disabling Interrupts

Individual interrupts can be easily enabled and disabled by calling the CMSIS core functions (macros) **NVIC_EnableIRQ** and **NVIC_DisableIRQ** respectively.

For temporarily disabling and enabling all interrupts, e.g. in a **critical section** of code, there are the **__disable_irq()** and **__enable_irq()** intrinsics. The time span with disabled interrupts should however be as short as possible to minimize the impact of all interrupts being disabled.

More advanced cores like Arm Cortex-M4 allow for enabling and disabling interrupts by priority.

20.1.3 Setting Interrupt Priorities

The STM32C0 microcontrollers implement 2 bits for interrupt priority encoding which allows for 4 interrupt priorities 0, 1, 2, and 3. In Cortex-M cores, priority 0 is always the highest priority, 1 the next highest, and so on. Note that the number of interrupt priority bits may vary for different microcontrollers. CMSIS provides a generic macro for that: **`__NVIC_PRIO_BITS`**.

On Cortex-M4 and other cores you'll find interrupt subpriorities which are not discussed here and are only relevant in some advanced scenarios.

An example of setting an interrupt priority and enabling an interrupt is given in the SysTick chapter.

20.1.4 Handling Faults

Even if you don't intend using interrupts, the Arm Cortex core may hit a faulty condition which triggers a fault handler. For example, the following code will cause a Hard Fault:

```
    *(uint32_t*)0xdeadbeef = 42;
```

because there is no memory mapped at address 0xdeadbeef. Other faults are caused by decoding an illegal instructions, misaligned memory access, and more.

Diagnosing the root cause of a fault can get complicated and is not discussed here. Some IDEs like STM32CubeIDE support this type of diagnosis by a Fault Analyzer.

Further reading:

- SEGGER's "**Analyzing HardFaults on Cortex-M CPU**" [24], and
- Chris Coleman's **How to debug a HardFault on an ARM Cortex-M MCU** [25]

20.1.5 Software Reset

NVIC can be used to trigger a system reset in software by simply calling

```
NVIC_SystemReset();
```

20.1.6 Dealing with Pending Interrupts

NVIC support several functions (macros) dealing with pending interrupt requests (IRQs). These functions are only needed in advanced scenarios and are not discussed here. The programming manual [11] has more information on those functions.

20.2 SysTick (STK)

The SysTick is a simple timer which can trigger a periodic interrupt. This is often used as a global millisecond tick counter, but the period can be set to a different value if really needed. The SysTick is often initialized in firmware libraries and encapsulated by API functions like HAL_Delay. Access to the SysTick registers is **privileged**. Therefore, in RTOS (Real-Time Operating Systems) applications, the SysTick is often used for the RTOS scheduler and not available to user tasks.

The following code configures a 1 ms periodic SysTick interrupt, assuming a 12 MHz core clock. The interrupt handler increments a global variable ticks

```
#include <stm32c011xx.h>

volatile uint32_t ticks;

void SysTick_Handler(void) {
    ticks++;
}

int main(void) {
    NVIC_SetPriority(SysTick_IRQn, 1);  // set priority 1, the second highest after 0
    NVIC_EnableIRQ(SysTick_IRQn);       // enable SysTick interrupt in NVIC

    // assume 12 MHz core clock
    SysTick->LOAD = 12000000/1000 - 1;  // 1 ms counter for 1 kHz interrupt freq.
    SysTick->VAL = 0;
    SysTick->CTRL =
        1 << SysTick_CTRL_CLKSOURCE_Pos // use processor clock for SysTick input clock
    |   1 << SysTick_CTRL_TICKINT_Pos   // enable SysTick interrupt generation
    |   1 << SysTick_CTRL_ENABLE_Pos    // enable SysTick counter
    ;
    /* Loop forever */
    for(;;);
}
```

Instead of register-level programming, one may simply use the CMSIS **SysTick_Config** function.

20.3 Memory Protection Unit (MPU)

The Memory Protection Unit can be used to restrict firmware access rights (read / write). This can be used to protect parts of the Flash or SRAM.

The following code shows how to protect, starting at address 0x00000000, the first 256 Bytes of the address space. This can be used to catch NULL pointer access by the Hard Fault Handler.

```
void HardFault_Handler(void) {
    for(;;);    // set a breakpoint here or indicate otherwise that a hard fault occured
}

int main(void) {
    MPU->RBAR = 0x0U                            // base address
              | MPU_RBAR_VALID_Msk              // valid region
              | (7U << MPU_RBAR_REGION_Pos);    // region #7
    MPU->RASR = (7U << MPU_RASR_SIZE_Pos)       // 2^(7+1) bytes size
              | (0x0U << MPU_RASR_AP_Pos)       // no-access region
              | MPU_RASR_ENABLE_Msk;            // enable region
    MPU->CTRL = MPU_CTRL_PRIVDEFENA_Msk         // enable background region
              | MPU_CTRL_ENABLE_Msk;            // enable the MPU
    __ISB();
    __DSB();
    // ...
    *(int *)0 = 0x1234;         // illegal NULL pointer access triggers the hard fault handler
    for (;;);
}
```

Read more about hard faults and hard fault handling:

- https://interrupt.memfault.com/blog/cortex-m-hardfault-debug
- https://mcuoneclipse.com/2012/11/24/debugging-hard-faults-on-arm-cortex-m/
- https://kb.segger.com/Cortex-M_Fault

Read more on the MPU:

- https://interrupt.memfault.com/blog/fix-bugs-and-secure-firmware-with-the-mpu
- AN4838 Application note "Introduction to memory protection unit management on STM32 MCUs"

21 MISCELLANEOUS

21.1 DEBUG SUPPORT (DBG)

The MCU has two dedicated pins PA13 (SWDIO) and PA14 (SWCLK) for the Single Wire Debug (SWD) interface. These pins are usable by a debug adaptor during and right after a power-on or system reset. The firmware may later repurpose those pins like any other GPIO pin.

Important DBG registers:

DBG Register	Purpose
`IDCODE`	MCU device ID (IDCODE[11:0]) and revision ID (IDCODE[31:16]) for chip identification by debuggers
`CR`	Debug configuration register (for enabling debugging in standby and stop power modes)
`APBFZ1`, `APBFZ2`	"freeze" bits for various peripherals: When a freeze bit for a peripheral is set, that peripheral clock is halted when the core is halted. This can be useful for step by step debugging or when hitting breakpoints. example: `DBG->APBFZ2 |= DBG_APB_FZ2_DBG_TIM14_STOP;`

A great resource for various debugging strategies and tools is the **STM32 microcontroller debug toolbox** [26].

21.2 CYCLIC REDUNDANCY CHECK CALCULATION UNIT (CRC)

The Cyclic Redundancy Calculation (CRC) Unit is used for computing a checksum over an array of data. Several standards and de-facto standards for CRC checksums do exist with different checksum sizes, typically 8, 16, or 32 bit and different computing algorithms with different capabilities of error detection and error correction. Since these algorithms are originally defined at the bit-level, hardware support for CRC accelerates the computation and off-loads it from the CPU core.

```
// size in bytes, must be a multiple of 4
uint32_t crc32(const uint32_t *data, size_t size)
{

  RCC->AHB1ENR |= RCC_AHB1ENR_CRCEN; // enable peripheral clock
  (void)RCC->AHB1ENR;                // read back to ensure that the clock is now running

  // settings for standard CRC-32 polynomial
  // see also https://m0agx.eu/2021/04/09/matching-stm32-hardware-crc-with-standard-crc-32/

  CRC->INIT = 0xFFFFFFFF; // initial crc value (reset value)
  CRC->POL = 0x04C11DB7;  // CRC-32 polynomial (reset value) - normal representation
  CRC->CR = CRC_CR_RESET; // reset CRC peripheral
  CRC->CR =
      1 << CRC_CR_REV_OUT_Pos    // 1: Bit-reversed output format
      | 3 << CRC_CR_REV_IN_Pos   // 11: Bit reversal done by word
      | 0 << CRC_CR_POLYSIZE_Pos // 00: 32 bit polynomial
      ;

  for (size_t i = 0; i < size; i+=sizeof(uint32_t))
  {
    CRC->DR = *data;
    data++;
  }
  return ~CRC->DR; // final bit reversal
}
```

```
uint32_t test[1] = {0x00010203};
uint32_t crc = crc32(test, 4);
assert(crc == 0x296E95DD);
```

The CRC unit works with linear DMA to off-load CRC calculations from the core. This can be used as a safety feature, for integrity checking of a SRAM or flash area.

21.3 Device Electronic Signature

These registers contain factory-programmed chip data which is read-only for the MCU user.

The 96-Bit Unique Device ID can be used as serial number or device specific seed for pseudorandom or cryptographic algorithms.

Register	Purpose
PACKAGE_BASE	Package Data Register (chip package type)
UID_BASE	96-Bit Unique ID Register (wafer coordinates, wafer num, lot num)
FLASHSIZE_BASE	Flash Memory Size Data Register (in kB)

22 REFERENCES

[1] Arm Limited, "CMSIS core library," [Online]. Available: https://arm-software.github.io/CMSIS_5/latest/Core/html/index.html. [Accessed 12 Aug 2024].

[2] ST Microelectronics, "RM0490 Reference manual STM32C0x1 advanced Arm®-based 32-bit MCUs," Rev. 3 2022.

[3] ST Microelectronics, "DS13866 STM32C011x4/x6 Data Sheet," Rev 4 2024.

[4] ST Microelectronics, "UM2970 User manual Discovery kit with STM32C011F6 MCU," Rev. 2, 2022.

[5] ST Microelectronics, "STM32C0116-DK Board Schematics - MB1684".

[6] ST Microelectronics, "AN5673 Application note Getting started with STM32C0 Series hardware development," Rev. 2 2022.

[7] ST Microelectronics, "How to install the STM32 VS Code extension," [Online]. Available: http://st.com/content/dam/videos-cf/pub/2024/q1/How-to-install-the-STM32-VS-Code-extension.mp4. [Accessed 12 Aug 2024].

[8] ST Microelectronics, "How to create projects using STM32 VS Code Extension," [Online]. Available: http://st.com/content/dam/videos-cf/pub/2024/q1/How-to-create-projects-using-STM32-VS-Code-Extension.mp4. [Accessed 12 Aug 2024].

[9] ST Microelectronics, "How to debug using STM32 VS Code Extension," [Online]. Available: http://st.com/content/dam/videos-cf/pub/2024/q1/How-to-debug-using-STM32-VS-Code-Extension.mp4. [Accessed 12 Aug 2024].

[10] ST Microelectronics, "STM32CubeC0 MCU Firmware Package," [Online]. Available: https://github.com/STMicroelectronics/STM32CubeC0. [Accessed 12 Aug 2024].

[11] ST Microelectronics, "PM0223 STM32 Cortex®-M0+ MCUs programming manual," Rev. 9 2024.

[12] ST Microelectronics, "ES0569 STM32C011x4/x6 Errata sheet," Rev. 4 2023.

[13] ST Microelectronics, "STM32C0 Online Training," [Online]. Available: https://www.st.com/content/st_com/en/support/learning/stm32-education/stm32-online-training/stm32c0-online-training.html. [Accessed 12 Aug 2024].

[14] ST Microelectronics, "UM1718 STM32CubeMX for STM32 configuration and initialization C code generation," Rev. 45 June 2024.

[15] ST Microelectronics, "AN2606 Application note STM32 microcontroller system memory boot mode," Rev. 653, May 2023.

[16] ST Microelectronics, "AN4899 STM32 Application note GPIO configuration for hardware settings and low-power consumption," Rev. 3 2022.

[17] N. Pendleton, "Exploring printf on Cortex-M," Interrupt by Memfault, 20 Sep 2023. [Online]. Available: https://interrupt.memfault.com/blog/printf-on-embedded. [Accessed 13 Aug 2024].

[18] ST Microelectronics, "AN2548 Application note Introduction to DMA controller for STM32 MCUs," Rev. 9 2024.

[19] ST Microelectronics, "AN5224 Application note Introduction to DMAMUX for STM32 MCUs," Rev. 6 2024.

[20] C. Coleman, "A Guide to Watchdog Timers for Embedded Systems," Interrupt by Memfault, 18 Feb 2020. [Online]. Available: https://interrupt.memfault.com/blog/firmware-watchdog-best-practices. [Accessed 13 Aug 2024].

[21] ST Microelectronics, "AN4894 Application note How to use EEPROM emulation on STM32 MCUs," Rev. 9 2024.

[22] J. Yiu, "The Definitive Guide to Arm® Cortex®-M0 and Cortex-M0+ Processors," Elsevier, 2016.

[23] C. Coleman, "A Practical guide to ARM Cortex-M Exception Handling," Interrupt by Memfault, 04 Sep 2019. [Online]. Available: https://interrupt.memfault.com/blog/arm-cortex-m-exceptions-and-nvic. [Accessed 12 Aug 2024].

[24] SEGGER, "AN00016 Application Note Analyzing HardFaults on Cortex-M CPU," Rev. 10, 2017.

[25] C. Coleman, "How to debug a HardFault on an ARM Cortex-M MCU," Interrupt by Memfault, 20 Nov 2019. [Online]. Available: https://interrupt.memfault.com/blog/cortex-m-hardfault-debug. [Accessed 13 Aug 2024].

[26] ST Microelectronics, "AN4989 Application note STM32 microcontroller debug toolbox," Rev 3, 2021.

[27] J. Yiu, "ARM Cortex-M for Beginners - ARM White paper," ARM Limited, March 2017.

[28] ST Microelectronics, "UM2237 STM32CubeProgrammer software description," Rev. 25 2024.

--- The End ---