Getting started with STM32CubeC0 for STM32C0 Series

Introduction

STM32Cube is an STMicroelectronics original initiative to significantly improve designer productivity by reducing development effort, time, and cost. STM32Cube covers the whole STM32 portfolio.

STM32Cube includes:

- A set of user-friendly software development tools to cover project development from conception to realization, among which are:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards
  - STM32CubeIDE, an all-in-one development tool with peripheral configuration, code generation, code compilation, and debug features
  - STM32CubeProgrammer (STM32CubeProg), a programming tool available in graphical and command-line versions
  - STM32CubeMonitor (STM32CubeMonitor, STM32CubeMonPwr, STM32CubeMonRF, STM32CubeMonUCPD) powerful monitoring tools to fine-tune the behavior and performance of STM32 applications in real-time
- STM32Cube MCU and MPU Packages, comprehensive embedded-software platforms specific to each microcontroller and microprocessor series (such as STM32CubeC0 for the STM32C0 Series), which include:
  - STM32Cube hardware abstraction layer (HAL), ensuring maximized portability across the STM32 portfolio
  - STM32Cube low-layer APIs, ensuring the best performance and footprints with a high degree of user control over hardware
  - A consistent set of middleware components such as ThreadX, FileX / LevelX, and OpenBL
  - All embedded software utilities with full sets of peripheral and applicative examples
- STM32Cube Expansion Packages, which contain embedded software components that complement the functionalities of the STM32Cube MCU and MPU Packages with:
  - Middleware extensions and applicative layers
  - Examples running on some specific STMicroelectronics development boards

This user manual describes how to get started with the STM32CubeC0 MCU Package.

STM32CubeC0 main features describes the main features of the STM32CubeC0 MCU Package. STM32CubeC0 architecture overview and STM32CubeC0 MCU Package overview provide an overview of the STM32CubeC0 architecture and MCU Package structure.

1 General information

The STM32CubeC0 MCU Package runs on STM32 32-bit microcontrollers based on the Arm® Cortex®-M0+ processor.

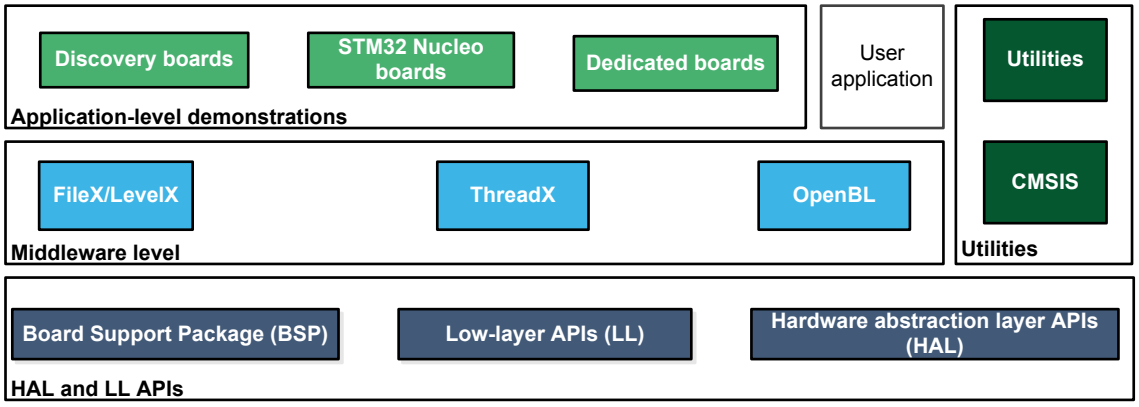
Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 STM32CubeC0 main features

- Consistent and complete embedded software that provides hardware abstraction to easily develop end-user firmware
- Maximized portability between all STM32 series supported by
- More than 100 examples and applications for easy understanding, all compatible with to facilitate the configuration through a graphical tool
- Production-ready HAL and LL API drivers, checked with static analysis tool and developed in compliance with MISRA C® guidelines, following a process certified according to IEC 61508 systematic capability 2 level (SC2)
- CMSIS CORE, DSP, and RTOS software components
- comprehensive middleware offer built around RTOS middleware, plus OpenBL
- Free-of-charge, user-friendly license terms
- Update mechanism with new-release notification capability
-

The STM32CubeC0 MCU Package component layout is illustrated in Figure 1.

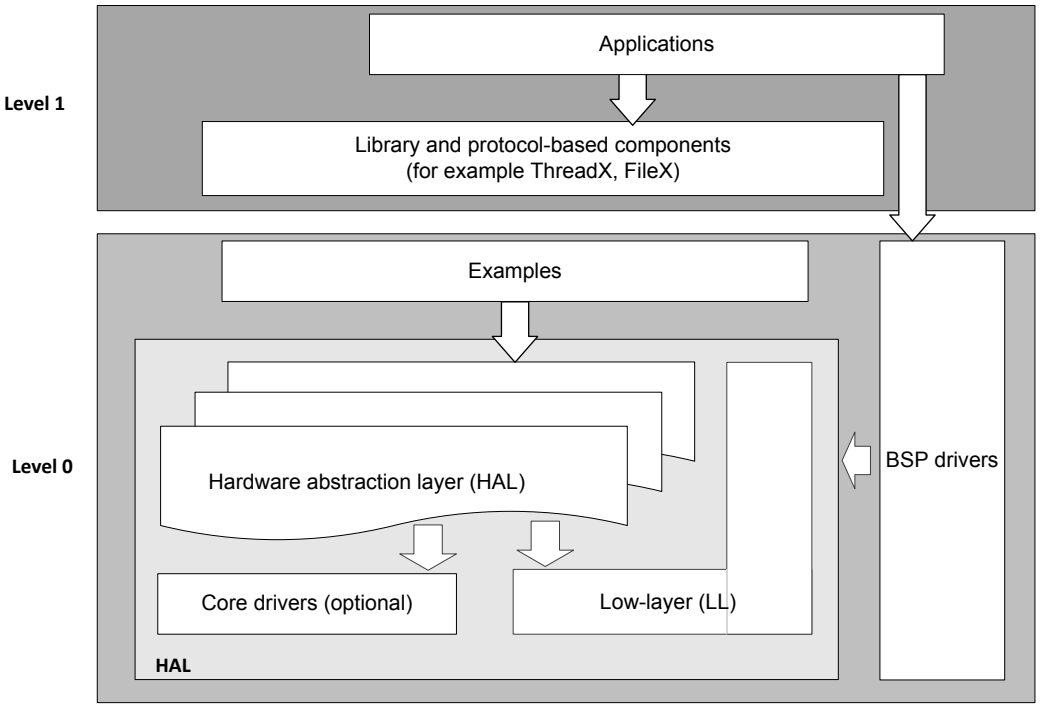
Figure 1. STM32CubeC0 MCU Package components



3 STM32CubeC0 architecture overview

The STM32CubeC0 MCU Package solution is built around three independent levels that easily interact as described in Figure 2.

Figure 2. STM32CubeC0 MCU Package architecture



3.1 Level 0

This level is divided into three sub-layers:

- Board support package (BSP)
- Hardware abstraction layer (HAL)
  - HAL peripheral drivers
  - Low-layer drivers
- Basic peripheral usage examples

3.1.1 Board support package (BSP)

This layer offers a set of APIs relative to the hardware components in the hardware boards. It is composed of:

- BSP driver

  It allows linking the component drivers to a specific board and provides a set of user-friendly APIs. The API naming rule is `BSP_FUNCT_Action()`.

  Example: `BSP_LED_Init()`, `BSP_LED_On()`

BSP is based on a modular architecture allowing easy porting on any hardware by just implementing the low-level routines.

3.1.2 Hardware abstraction layer (HAL) and low-layer (LL)

The STM32CubeC0 HAL and LL are complementary and cover a wide range of applications requirements:

- The HAL drivers offer high-level function-oriented highly-portable APIs. They hide the MCU and peripheral complexity to the end-user.

  The HAL drivers provide generic multi-instance feature-oriented APIs which simplify user application implementation by providing ready-to-use processes. For example, for the communication peripherals (I^2S, UART, and others), it provides APIs allowing initializing and configuring the peripheral, managing data transfer based on polling, interrupting, or DMA process, and handling communication errors that may arise during communication. The HAL driver APIs are split into two categories:

  1. Generic APIs which provide common and generic functions to all the STM32 series
  2. Extension APIs which provide specific and customized functions for a specific family or a specific part number

- The low-layer APIs provide low-level APIs at the register level, with better optimization but less portability. They require a deep knowledge of MCU and peripheral specifications.

  The LL drivers are designed to offer a fast lightweight expert-oriented layer that is closer to the hardware than the HAL. Contrary to the HAL, LL APIs are not provided for peripherals where optimized access is not a key feature, or for those requiring heavy software configuration or complex upper-level stack.

  The LL drivers feature:

  – A set of functions to initialize peripheral main features according to the parameters specified in data structures
  – A set of functions to fill initialization data structures with the reset values corresponding to each field
  – Function for peripheral de-initialization (peripheral registers restored to their default values)
  – A set of inline functions for direct and atomic register access
  – Full independence from HAL and capability to be used in standalone mode (without HAL drivers)
  – Full coverage of the supported peripheral features

3.1.3 Basic peripheral usage examples

This layer encloses the examples built over the STM32 peripherals using only the HAL and BSP resources.

3.2 Level 1

This level is divided into two sub-layers:

- Middleware components
- Examples based on the middleware components

3.2.1 Middleware components

The middleware is a set of libraries constructed around Microsoft® Azure® RTOS middleware and other in-house like OpenBL. All are integrated and customized for STM32 MCU devices and enriched with corresponding application examples based on STM32 evaluation boards. Horizontal interactions between the components of this layer are simply done by calling the featured APIs while the vertical interaction with the low-layer drivers is done through specific callbacks and static macros implemented in the library system call interface.

The main features of each middleware component are as follows:

- ThreadX:
  – A real-time operating system (RTOS), designed for embedded systems with two functional modes:
  – Common mode: Common RTOS functionalities such as thread management and synchronization, memory pool management, messaging, and event handling
  – Module mode: An advanced user mode that allows loading and unloading of pre-linked ThreadX modules on the fly through a module manager.
- FileX / levelX
  – Advanced Flash file system (FS) / Flash translation layer (FTL): fully-featured to support NAND/NOR Flash memories
- Open bootloader
  – provides an open-source bootloader with the same features as the STM32 system bootloader and with the same tools used for system bootloader

3.2.2 Examples based on the middleware components

Each middleware component comes with one or more examples (called also applications) showing how to use it. Integration examples that use several middleware components are provided as well.

3.3 Utilities

STM32CubeC0 provides in `Utilities` IAR Systems® IAR Embedded Workbench® (EWARM) and Keil® Microcontroller Development Kit for Arm®-based microcontrollers (MDK-ARM) patch that supports STM32C0xxxx devices.

4 STM32CubeC0 MCU Package overview

4.1 Supported STM32C0 Series devices and hardware

STM32Cube offers a highly portable hardware abstraction layer (HAL) built around a generic architecture. It allows the build-upon layers principle, such as using the middleware layer to implement their functions without knowing, in-depth, what MCU is used. This improves the library code re-usability and guarantees easy portability to other devices.

In addition, thanks to its layered architecture, the STM32CubeC0 offers full support of all STM32C0 Series. The user has only to define the right macro in `stm32c0xx.h`.

Table 1 shows the macro to define depending on the STM32C0 Series device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32C0 Series

Macro defined in stm32c0xx.h	STM32C0 Series devices
STM32C011xxxx	STM32C011F6U6, STM32C011F6P6, STM32C011D6Y6TR, STM32C011J6M6, STM32C011F4U6, STM32C011F4P6, STM32C011J4M6, STM32C011F6U3, STM32C011F6P3, STM32C011D6Y3TR, STM32C011J6M3, STM32C011F4U3, STM32C011F4P3, STM32C011J4M3
STM32C031xxxx	STM32C031C6T6, STM32C031C6U6, STM32C031K6T6, STM32C031K6U6, STM32C031G6U6, STM32C031F6P6, STM32C031C4T6, STM32C031C4U6, STM32C031K4T6, STM32C031K4U6, STM32C031G4U6, STM32C031F4P6, STM32C031C6T3, STM32C031C6U3,STM32C031K6T3, STM32C031K6U3, STM32C031G6U3, STM32C031F6P3, STM32C031C4T3, STM32C031C4U3,STM32C031K4T3, STM32C031K4U3, STM32C031G4U3, STM32C031F4P3.

STM32CubeC0 features a rich set of examples and applications at all levels making it easy to understand and use any HAL driver or middleware components. These examples run on the STMicroelectronics boards listed in Table 2.

Table 2. Boards for STM32C0 Series

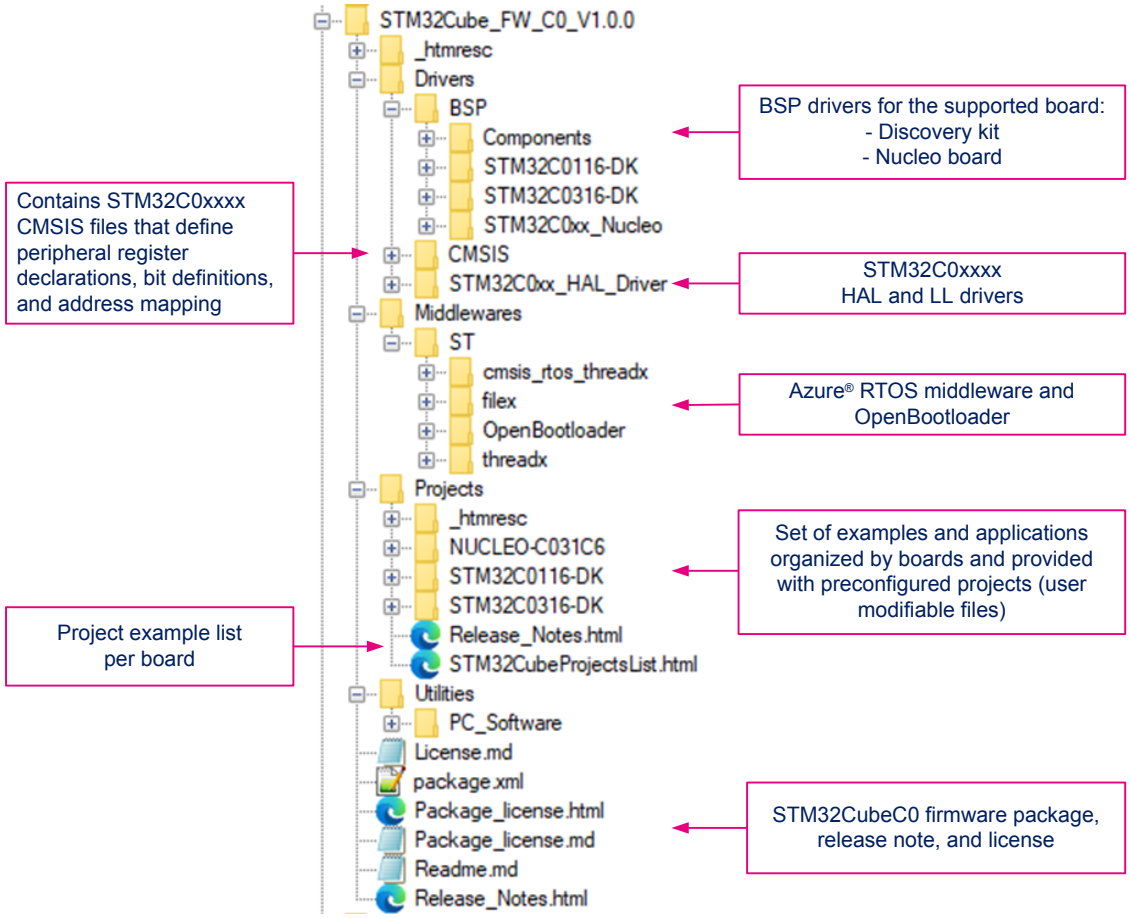
Board	Board STM32C0 supported devices
NUCLEO-C031C6	STM32C031C6T6
STM32C0116-DK	STM32C031C6T6
STM32C0316-DK	STM32C011F6U6

The STM32CubeC0 MCU Package can run on any compatible hardware. The user simply updates the BSP drivers to port the provided examples on his board, if the latter has the same hardware features, such as LEDs and buttons.

4.2 MCU Package overview

The STM32CubeC0 MCU Package solution is provided in one single zip package having the structure shown in Figure 3.

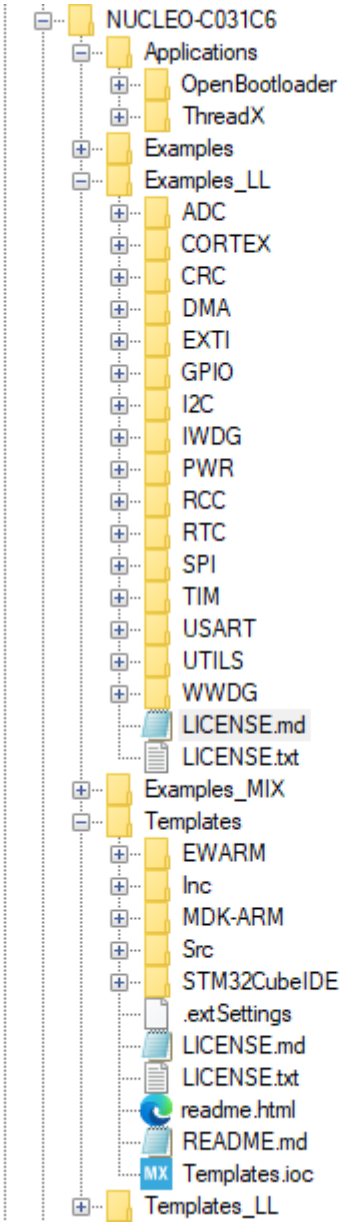
Figure 3. STM32CubeC0 MCU Package structure



For each board, a set of examples are provided with pre-configured projects for EWARM, MDK-ARM, and STM32CubeIDE toolchains.

Figure 4 shows the project structure for the NUCLEO-C031C6 board.

Figure 4. **STM32CubeC0 examples overview**



The examples are classified depending on the STM32Cube level they apply to, and are named as explained below:

- Level 0 examples are called *Examples*, *Examples_LL*, and *Examples_MIX*. They use respectively HAL drivers, LL drivers, and a mix of HAL and LL drivers without any middleware component.
- Level 1 examples are called Applications. They provide typical use cases of each middleware component.

Any firmware application for a given board can be quickly built thanks to template projects available in the `Templates` and `Templates_LL` directories.

The STM32CubeC0 firmware package provides default memory partitioning in the *partition_<device>.h* files available under:

`\Drivers\CMSIS\Device\ST\STM32C0xx\Include\Templates`.

All examples have the same structure:

- `\Inc` folder that contains all header files.
- `\Src` folder for the sources code.
- `\EWARM`, `\MDK-ARM`, and \STM32CubeIDE folders containing the pre-configured project for each toolchain.
- `readme.txt` describing the example behavior and needed environment to make it work
- `*.ioc` file that allows users to open most firmware examples within STM32CubeMX

Table 3 gives the number of projects available for each board.

Table 3. Number of examples for each board

Level	STM32C0316-DK	STM32C0116-DK	NUCLEO-C031C6	Total
Templates_LL	1	1	1	3
Templates	1	1	1	3
Examples_MIX	0	0	8	8
Examples_LL	7	8	43	58
Examples	8	8	16	32
Applications	3	1	4	8
Total	20	19	73	112

5 Getting started with STM32CubeC0

5.1 Running a first example

This section explains how simple is to run the first example within STM32CubeC0. It uses as an illustration the generation of a simple LED toggle running on the NUCLEO-C031C6 board:

1. Download the STM32CubeC0 MCU Package. Unzip it into a directory of your choice. Make sure not to modify the package structure shown in Figure 3. Note that it is also recommended to copy the package at a location close to your root volume, meaning `C\Eval` or `G:\Tests`, because some IDEs encounter problems when the path length is too long.
2. Browse to `\Projects\NUCLEO-C031C6\Examples`.
3. Open `\GPIO`, then `\GPIO_EXTI` folders.
4. Open the project with your preferred toolchain. A quick overview on how to open, build and run an example with the supported toolchains is given below.
5. Rebuild all files and load your image into the target memory.
6. Run the example: Each time the USER push-button is pressed, the LD4 LED toggles (for more details, refer to the readme file of the example).

To open, build and run an example with the supported toolchains, follow the steps below:

- EWARM:
  1. Under the example folder, open `\EWARM` sub-folder.
  2. Launch the Project.eww workspace (a).
  3. Rebuild all files: **Project** → **Rebuild all**.
  4. Load project image: **Project** → **Debug**.
  5. Run program: **Debug** → **Go (F5)**.
- MDK-ARM:
  1. Under the example folder, open the `\MDK-ARM` sub-folder.
  2. Launch the Project.uvprojx workspace (a).
  3. Rebuild all files: **Project** → **Rebuild all target files**.
  4. Load project image: **Debug** → **Start/Stop Debug Session**.
  5. Run program: **Debug** → **Run (F5)**.
- STM32CubeIDE:
  1. Open the STM32CubeIDE toolchain.
  2. Click **File** → **Switch Workspace** → **Other** and browse to the STM32CubeIDE workspace directory.
  3. Click **File** → **Import**, select **General** → **Existing Projects into Workspace** and then click **Next**.
  4. Browse to the STM32CubeIDE workspace directory and select the project.;
  5. Rebuild all project files: Select the project in the **Project Explorer** window then click the **Project** → **build project** menu.
  6. Run program: **Run** → **Debug (F11)**

5.2 Developing a custom application

5.2.1 Using STM32CubeMX to develop or update an application

In the STM32CubeC0 MCU Package, nearly all project examples are generated with the STM32CubeMX tool to initialize the system, peripherals, and middleware.

The direct use of an existing project example from the STM32CubeMX tool requires STM32CubeMX 6.5.0 or higher:

- After the installation of STM32CubeMX, open and if necessary update a proposed project. The simplest way to open an existing project is to double-click on the *.ioc file so that STM32CubeMX automatically opens the project and its source files.

- The initialization source code of such projects is generated by STM32CubeMX. The main application source code is contained by the comments “USER CODE BEGIN” and “USER CODE END”. In case the IP selection and setting are modified, STM32CubeMX updates the initialization part of the code but preserves the main application source code.

For developing a custom project in the STM32CubeMX, follow the step-by-step process:

1. Select the STM32 microcontroller that matches the required set of peripherals.
2. Configure all the required embedded software using a pinout-conflict solver, a clock-tree setting helper, a power consumption calculator, and the utility performing MCU peripheral configuration (such as GPIO or SPI) and middleware stacks (such as ThreadX).
3. Generate the initialization C code based on the selected configuration. This code is ready to use within several development environments. The user code is kept at the next code generation.

For more information about STM32CubeMX, refer to the user manual *STM32CubeMX for STM32 configuration and initialization C code generation* (UM1718).

For a list of the available project examples for STM32CubeC0, refer to the application note *STM32Cube firmware examples for STM32C0 Series*.

5.2.2 HAL application

This section describes the steps required to create a custom HAL application using STM32CubeC0:

1. Create a project

   To create a new project, start either from the `Template` project provided for each board under `\Projects\<STM32xxx_yyy>\Templates` or from any available project under `\Projects\<STM32xxy_yyy>\Examples` or `\Projects\<STM32xx_yyy>\Applications` (Where `<STM32xxx_yyy>` refers to the board name, such as STM32CubeC0).

   The Template project provides an empty main loop function, however, it is a good starting point to understand the STM32CubeC0 project settings. The template has the following characteristics:

   – It contains the HAL source code, CMSIS, and BSP drivers which are the minimum set of components required to develop a code on a given board.
   – It contains the included paths for all the firmware components.
   – It defines the supported STM32C0 Series devices, allowing the CMSIS and HAL drivers to be configured correctly.
   – It provides ready-to-use user files pre-configured as shown below:

     HAL initialized with default time base with Arm core SysTick.

     SysTick ISR implemented for `HAL_Delay()` purpose.

Note: When copying an existing project to another location, make sure all the included paths are updated.

2. **Add the necessary middleware to the user project (optional)**

   To identify the source files to be added to the project file list, refer to the documentation provided for each middleware. Refer to the applications under `\Projects\STM32xxx_yyy\Applications\<MW_Stack>` (Where *<MW_Stack>* refers to the middleware stack, such as ThreadX) to know which source files and include paths must be added.

3. **Configure the firmware components**

   The HAL and middleware components offer a set of build-time configuration options using macros `#define` declared in a header file. A template configuration file is provided within each component which has to be copied to the project folder (usually the configuration file is named `xxx_conf_template.h`, the word *_template* needs to be removed when copying it to the project folder). The configuration file provides enough information to understand the impact of each configuration option. More detailed information is available in the documentation provided for each component.

4. **Start the HAL library**

   After jumping to the main program, the application code must call `HAL_Init()` API to initialize the HAL library, which carries out the following tasks:

   a. Configuration of the flash memory prefetch and SysTick interrupt priority (through macros defined in `stm32c0xx_hal_conf.h`).

   b. Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority `TICK_INT_PRIO` defined in `stm32c0xx_hal_conf.h`, which is clocked by the HSI (at this stage, the clock is not yet configured and thus the system is running from the internal 16 MHz HSI).

   c. Setting of NVIC group priority to 0.

   d. Call of `HAL_MspInit()` callback function defined in `stm32c0xx_hal_msp.c` user file to perform global low-level hardware initializations.

5. **Configure the system clock**

   The system clock configuration is done by calling the two APIs described below:

   – `HAL_RCC_OscConfig()`: this API configures the internal and external oscillators. The user chooses to configure one or all oscillators.

   – `HAL_RCC_ClockConfig()`: this API configures the system clock source, the flash memory latency, and AHB and APB prescalers.

   Initialize the peripheral

   a. First write the peripheral `HAL_PPP_MspInit` function. Proceed as follows:

      ◦ Enable the peripheral clock.

      ◦ Configure the peripheral GPIOs.

      ◦ Configure the DMA channel and enable DMA interrupt (if needed).

      ◦ Enable peripheral interrupt (if needed).

   b. Edit the `stm32xxx_it.c` to call the required interrupt handlers (peripheral and DMA), if needed.

   c. Write process complete callback functions if peripheral interrupt or DMA is planned to be used.

   d. In user `main.c` file, initialize the peripheral handle structure then call the function `HAL_PPP_Init()` to initialize the peripheral.

6. **Develop an application**

   At this stage, the system is ready and user application code development can start.

   a. The HAL provides intuitive and ready-to-use APIs to configure the peripheral. It supports polling, interrupts, and a DMA programming model, to accommodate any application requirements. For more details on how to use each peripheral, refer to the rich example set provided in the STM32CubeC0 MCU Package.

Caution: In the default HAL implementation, the SysTick timer is used as a timebase: it generates interrupts at regular time intervals. If `HAL_Delay()` is called from the peripheral ISR process, make sure that the SysTick interrupt has higher priority (numerically lower) than the peripheral interrupt. Otherwise, the caller ISR process is blocked. Functions affecting timebase configurations are declared as `__weak` to make override possible in case of other implementations in the user file (Using a general-purpose timer for example or another time source). For more details, refer to the `HAL_TimeBase` example.

5.2.3 LL application

This section describes the steps needed to create a custom LL application using STM32CubeC0.

1. **Create a project**

   To create a new project, either start from the Templates_LL project provided for each board under `\Projects\<STM32xxx_yyy>\Templates_LL` or from any available project under `\Projects\<STM32xxy_yyy>\Examples_LL` (*<STM32xxx_yyy>* refers to the board name, such as NUCLEO-C031C6).

   The template project provides an empty main loop function, which is a good starting point to understand the project settings for STM32CubeC0. Template main characteristics are the following:

   – It contains the source codes of the LL and CMSIS drivers which are the minimum set of components needed to develop code on a given board.
   – It contains the included paths for all the required firmware components.
   – It selects the supported STM32C0 Series device and allows the correct configuration of the CMSIS and LL drivers.
   – It provides ready-to-use user files, that are pre-configured as follows:
     ◦ `main.h`: LED and USER_BUTTON definition abstraction layer.
     ◦ `main.c`: System clock configuration for maximum frequency.

2. **Port an existing project to another board**

   To support an existing project on another target board, start from the Templates_LL project provided for each board and available under `\Projects\<STM32xxx_yyy>\Templates_LL`.

   – Select an LL example: To find the board on which LL examples are deployed, refer to the list of LL examples `STM32CubeProjectsList.html`.

3. Port the LL example:

   – Copy/paste the `Templates_LL` folder - to keep the initial source - or directly update existing `Templates_LL` project.
   – Then porting consists principally in replacing `Templates_LL` files by the `Examples_LL` targeted project.
   – Keep all board specific parts. For reasons of clarity, board specific parts are flagged with specific tags:

   ```
   /* ============== BOARD SPECIFIC CONFIGURATION CODE BEGIN ============== */
   /* ============== BOARD SPECIFIC CONFIGURATION CODE END ============== */
   ```

   Thus the main porting steps are the following:

   ◦ Replace the `stm32c0xx_it.h` file
   ◦ Replace the `stm32c0xx_it.c` file
   ◦ Replace the `main.h` file and update it: Keep the LED and user button definition of the LL template under `BOARD SPECIFIC CONFIGURATION` tags.
   ◦ Replace the `main.c` file and update it:

     Keep the clock configuration of the `SystemClock_Config()` LL template function under `BOARD SPECIFIC CONFIGURATION` tags.

     Depending on the LED definition, replace each LDx occurrence with another LDy available in the `main.h` file.

With these modifications, the example now runs on the targeted board.

5.3 Getting STM32CubeC0 release updates

The latest STM32CubeC0 MCU Package releases and patches are available from STM32C0 Series. They may be retrieved from the `CHECK FOR UPDATE` button in STM32CubeMX. For more details, refer to *Section 3* of the user manual *STM32CubeMX for STM32 configuration and initialization C code generation* (UM1718).

6 FAQ

6.1 What is the licensing scheme for the STM32CubeC0 MCU Package?

HAL is distributed under a non-restrictive BSD (Berkeley software distribution) license.

6.2 What boards are supported by the STM32CubeC0 MCU Package?

The STM32CubeC0 MCU Package provides BSP drivers and ready-to-use examples for the following :

- NUCLEO-C031C6
- STM32C0116-DK
- STM32C0316-DK

6.3 Are any examples provided with the ready-to-use toolset projects?

Yes. STM32CubeC0 provides a rich set of examples and applications. They come with the pre-configured projects for IAR Embedded Workbench®, Keil®, and STM32CubeIDE.

6.4 Are there any links with standard peripheral libraries?

The STM32CubeC0 HAL and LL drivers are the replacement of the standard peripheral library:

- The HAL drivers offer a higher abstraction level compared to the standard peripheral APIs. They focus on the features that are common to the peripherals rather than hardware. A set of user-friendly APIs allows a higher abstraction level which in turn makes them easily portable from one product to another.
- The LL drivers offer low-layer registers level APIs. They are organized more simply and clearly, avoiding direct register accesses. LL drivers also include peripheral initialization APIs, which are more optimized compared to what is offered by the SPL, while being functionally similar. Compared to HAL drivers, these LL initialization APIs allows an easier migration from the SPL to the STM32CubeC0 LL drivers, since each SPL API has its equivalent LL API.

6.5 Does the HAL layer take advantage of interrupts or DMA? How can this be controlled?

Yes. The HAL layer supports three API programming models: Polling, interrupt, and DMA (With or without interrupt generation).

6.6 How are the product or peripheral specific features managed?

The HAL drivers offer extended APIs, which are specific functions provided as add-ons to the common API to support features only available on some products or lines.

6.7 Why should the LL be used versus HAL drivers?

Since the STM32C0 Series microcontroller has a flash memory size of up to 32 Kbytes and SRAM memory size of up to 12 Kbytes, it is recommended to use LL versus HAL to ensure a low memory footprint when developing applications to fit into the available RAM size.

6.8 When should the HAL be used versus LL drivers?

HAL drivers offer high-level and function-oriented APIs, with a high level of portability. The product or IP complexity is hidden for end users.

LL drivers offer low-layer register level APIs, with better optimization but less portable. They require in-depth knowledge of product or IP specifications.

6.9 How can LL drivers be included in an existing environment? Is there any LL configuration file as for HAL?

There is no configuration file. Source code must directly include the necessary `stm32c0xx_ll_ppp.h` files.

6.10 Can HAL and LL drivers be used together? If yes, what are the constraints?

It is possible to use both HAL and LL drivers. Use the HAL for the IP initialization phase and then manage the I/O operations with LL drivers.

The major difference between HAL and LL is that HAL drivers require to create and use handles for operation management while LL drivers operate directly on peripheral registers. Mixing HAL and LL is illustrated in the *Examples_MIX* example.

6.11 Are there any LL APIs that are not available with HAL?

Yes, there are. A few Cortex® APIs are added in `stm32c0xx_ll_cortex.h`, for instance, to access SCB or SysTick registers.

6.12 Why are SysTick interrupts not enabled on LL drivers?

When using LL drivers in standalone mode, there is no need to enable SysTick interrupts because they are not used in LL APIs, while HAL functions require SysTick interrupts to manage timeouts.

6.13 How are LL initialization APIs enabled?

The definition of LL initialization APIs and associated resources (Structures, literals, and prototypes) is conditioned by the `USE_FULL_LL_DRIVER` compilation switch.

To be able to use LL initialization APIs, add this switch in the toolchain compiler preprocessor.

6.14 How can STM32CubeMX generate code based on embedded software?

STM32CubeMX has built-in knowledge of STM32 microcontrollers, including their peripherals and software, that allows providing a graphical representation to the user and generating `*.h` or `*.c` files based on user configuration.

6.15 How to get regular updates on the latest STM32CubeC0 MCU Package releases?

Refer to Getting STM32CubeC0 release updates.

Revision history

Table 4. **Document revision history**

Date	Revision	Changes
3-Feb-2022	1	Initial release.

Contents

List of tables

List of figures