UM2303
User manual

Getting started with STM32CubeG0 for STM32G0 Series

Introduction

STM32Cube is an STMicroelectronics original initiative to significantly improve designer's 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 the conception to the realization, among which:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards.
  - STM32CubeProgrammer (STM32CubeProg), a programming tool available in graphical and command-line versions.
  - STM32CubeMonitor-Power (STM32CubeMonPwr), a monitoring tool to measure and help in the optimization of the power consumption of the MCU.
- STM32Cube MCU Packages, comprehensive embedded-software platforms specific to each microcontroller series (such as STM32CubeG0 for STM32G0 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 the HW.
  - A consistent set of middleware components such as RTOS, USB PD, and FatFS.
  - All embedded software utilities with full sets of peripherals and applicative examples.

This user manual describes how to get started with the STM32CubeG0 MCU Package.

Section 1 describes the main features of the STM32CubeG0 MCU Package.

Section 2 and *Section 3* provide an overview of the STM32CubeG0 architecture and MCU Package structure.

Contents

List of tables

List of figures

1 STM32CubeG0 main features

The STM32CubeG0 MCU Package runs on STM32 32-bit microcontrollers based on the Arm®(a) Cortex®-M processor.

STM32CubeG0 gathers, in a single package, all the generic embedded software components required to develop an application on STM32G0 microcontrollers. In line with the STM32Cube initiative, this set of components is highly portable, not only within STM32G0 Series but also to other STM32 Series.

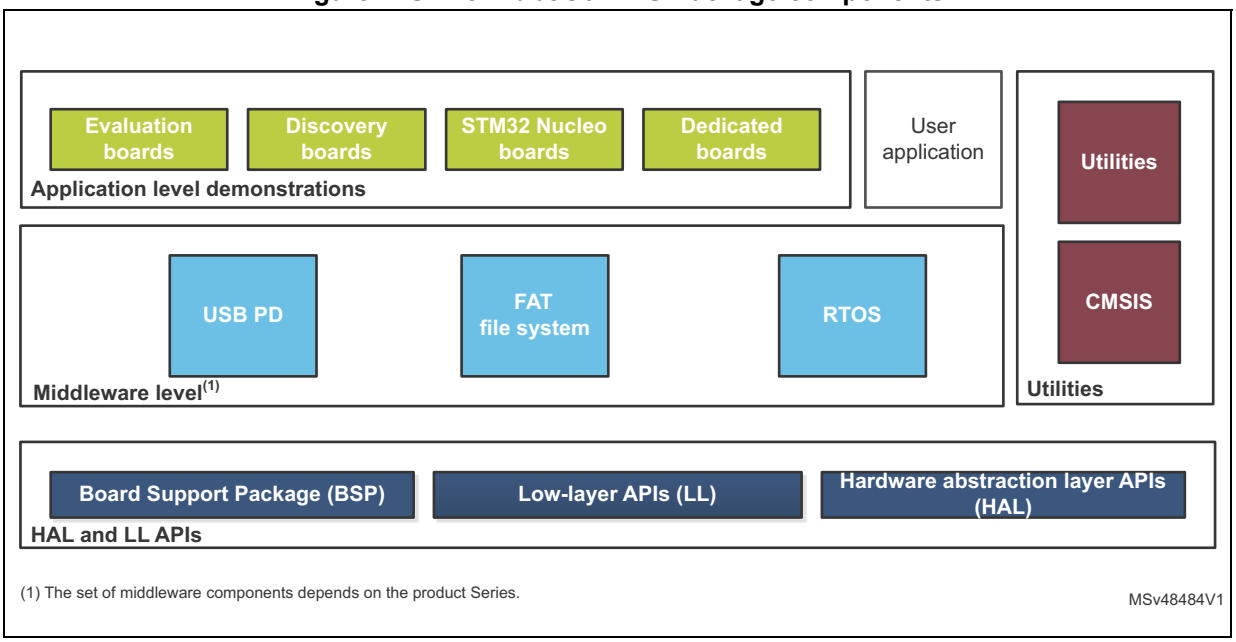
STM32CubeG0 is fully compatible with STM32CubeMX code generator that allows generating initialization code. The package includes low-layer (LL) and hardware abstraction layer (HAL) APIs that cover the microcontroller hardware, together with an extensive set of examples running on STMicroelectronics boards. The HAL and LL APIs are available in open-source BSD license for user convenience.

STM32CubeG0 MCU Package also contains a set of middleware components with the corresponding examples. They come in free user-friendly license terms:

- CMSIS-RTOS implementation with FreeRTOS™ open source solution
- USB PD library
- FAT file system based on open source FatFS solution

Several applications and demonstrations implementing all these middleware components are also provided in the STM32CubeG0 MCU Package.

Figure 1. STM32CubeG0 MCU Package components

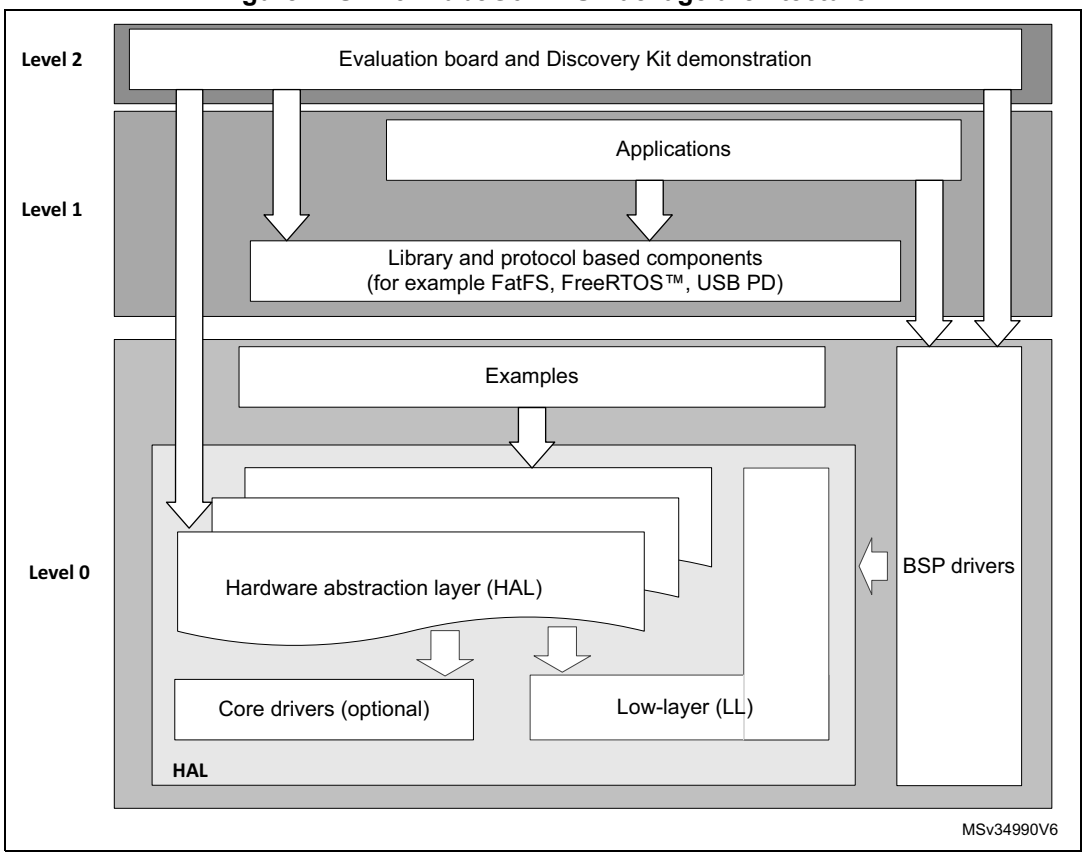


a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and or elsewhere.

2 STM32CubeG0 architecture overview

The STM32CubeG0 MCU Package solution is built around three independent levels that easily interact as described in *Figure 2*.

Figure 2. STM32CubeG0 MCU Package architecture



2.1 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

2.1.1 Board support package (BSP)

This layer offers a set of APIs relative to the hardware components in the hardware boards (such as LCD, Audio, microSD™ and MEMS drivers). Audio and MEMS drivers are not supported. It is composed of two parts:

- Component

  This is the driver relative to the external device on the board and not to the STM32. The component driver provide specific APIs to the BSP driver external components and could be portable on any other board.

- BSP driver

  It allows linking the component driver 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()

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

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

The STM32CubeG0 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 end user.

  The HAL drivers provide generic multi-instance feature-oriented APIs which simplify user application implementation by providing ready to use process. As 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, interrupt or DMA process, and handling communication errors that may raise during communication. The HAL driver APIs are split in two categories:

  – Generic APIs which provides common and generic functions to all the STM32 Series

  – Extension APIs which provides specific and customized functions for a specific family or a specific part number.

- The low-layer APIs provide low-level APIs at 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 light-weight expert-oriented layer which 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 and/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 used 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.

2.1.3 Basic peripheral usage examples

This layer encloses the examples build over the STM32 peripheral using only the HAL and BSP resources.

2.2 Level 1

This level is divided into two sub-layers:

- Middleware components
- Examples based on the middleware components.

2.2.1 Middleware components

The middleware is a set of libraries covering USB PD library, FreeRTOS™ and FatFS. Horizontal interactions between the components of this layer is done directly by calling the feature 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. For example, the FatFS implements the disk I/O driver to access microSD™ drive. USB PD provides the new USB Type C Power Delivery service. Implementing a dedicated protocol for the management of power management in this evolution of the USB.org specification. Please refer to http://www.usb.org/developers/powerdelivery/ for more details

The main features of each middleware component are as follows:

- USB PD library
  – PD2 and PD3 specifications (support of Source / Sink / Dual role)
  – Fast Role Swap
  – Dead Battery
  – Use of configuration files to change the core and the library configuration without changing the library code (Read Only)
  – RTOS and Standalone operation.
  – Link with low-level driver through an abstraction layer using the configuration file to avoid any dependency between the Library and the low-level drivers.
- FreeRTOS™
  – Open source standard

  - CMSIS compatibility layer
  - Tickless operation during low-power mode
  - Integration with all STM32Cube middleware modules
- FAT file system
  - FatFS FAT open source library
  - Long file name support
  - Dynamic multi-drive support
  - RTOS and standalone operation
  - Examples with microSD™

A dedicated application (demo) will propose the full capabilities of the STM32G0 with its USB PD library.

2.2.2 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.

2.3 Level 2

This level is composed of a single layer which consist in a global real-time and graphical demonstration based on the middleware service layer, the low-level abstraction layer and the basic peripheral usage applications for board based features.

3 STM32CubeG0 MCU Package overview

3.1 Supported STM32G0 devices and hardware

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

In addition, thanks to its layered architecture, the STM32CubeG0 offers full support of all STM32G0 Series. The user has only to define the right macro in *stm32g0xx.h*.

Table 1 shows the macro to define depending on the STM32G0 device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32G0 Series

Macro defined in *stm32g0xx.h*	STM32G0 devices
STM32G070xx	STM32G070KB, STM32G070CB, STM32G070RB, STM32G070K8, STM32G070C8, STM32G070R8, STM32G070K6, STM32G070C6, STM32G070R6, STM32G070GB, STM32G070KB, STM32G070CB, STM32G070G8, STM32G070K8, STM32G070C8, STM32G070G6, STM32G070K6, STM32G070C6, STM32G070xBY6.
STM32G071xx	STM32G071KB, STM32G071CB, STM32G071RB, STM32G071K8, STM32G071C8, STM32G071R8, STM32G071K6, STM32G071C6, STM32G071R6, STM32G071GB, STM32G071KB, STM32G071CB, STM32G071G8, STM32G071K8, STM32G071C8, STM32G071G6, STM32G071K6, STM32G071C6, STM32G071xBY6.
STM32G081xx	STM32G081KB, STM32G081CB, STM32G081RB, STM32G081K8, STM32G081C8, STM32G081R8, STM32G081K6, STM32G081C6, STM32G081R6, STM32G081GB, STM32G081KB, STM32G081CB, STM32G081G8, STM32G081K8, STM32G081C8, STM32G081G6, STM32G081K6, STM32G081C6, STM32G081xBY6.
STM32G030xx	STM32G030C8, STM32G030K8, STM32G030C6, STM32G030K6, STM32G030F6, STM32G030J6.

Table 1. Macros for STM32G0 Series (continued)

Macro defined in *stm32g0xx.h*	STM32G0 devices
STM32G031xx	STM32G031C8, STM32G031K8, STM32G031G8, STM32G031F8, STM32G031Y8, STM32G031C6, STM32G031K6, STM32G031G6, STM32G031F6, STM32G031J6, STM32G031C4, STM32G031K4, STM32G031G4, STM32G031F4, STM32G031J4.
STM32G041xx	STM32G041C8, STM32G041K8, STM32G041G8, STM32G041F8, STM32G041Y8, STM32G041C6, STM32G041K6, STM32G041G6, STM32G041F6, STM32G041J6.

STM32CubeG0 features a rich set of examples and applications at all levels making it easy to understand and use any HAL driver and/or middleware components. These examples run on the STMicroelectronics boards listed in *Table 2*.

Table 2. Boards for STM32G0 Series

Board	Board STM32G0 supported devices
NUCLEO-G070RB	STM32G070xB
NUCLEO-G071RB	STM32G071xB
STM32G081B-EVAL	STM32G081xB
STM32G071B-DISCO	STM32G071xB
NUCLEO-G031K8	STM32G031x8
STM32G0316-DISCO	STM32G031x6

STM32CubeG0 supports the Nucleo-64 boards listed above.

Nucleo-64 boards are compatible with Adafruit LCD display Arduino™ Uno shields which embed a microSD™ connector and a joystick in addition to the LCD.

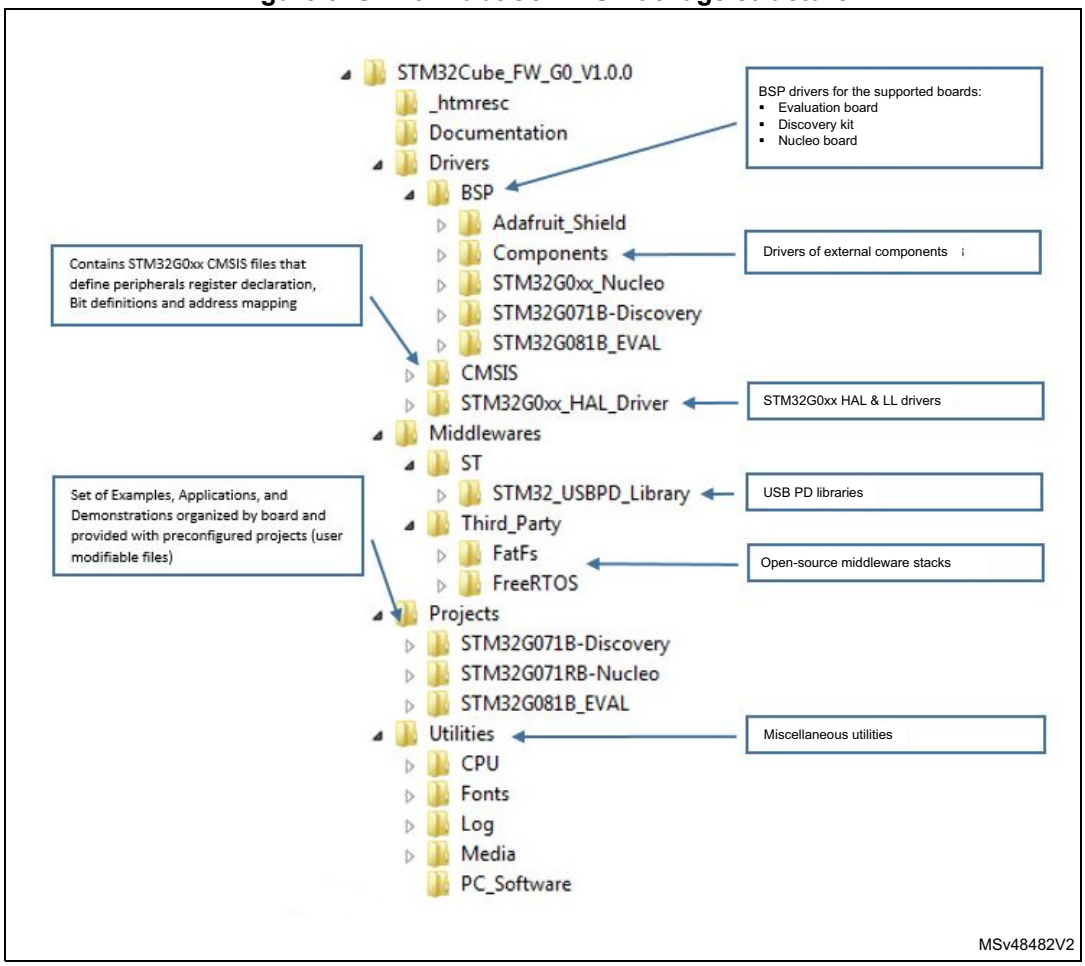
The Arduino™ shield drivers are provided within the BSP component. Their usage is illustrated by a demonstration firmware.

The STM32CubeG0 MCU Package is able to run on any compatible hardware. The user simply updates the BSP drivers to port the provided examples on his own board, if this latter has the same hardware features (LED, LCD display, buttons...).

3.2 MCU Package overview

The STM32CubeG0 MCU Package solution is provided in one single zip package having the structure shown in *Figure 3*.

Figure 3. STM32CubeG0 MCU Package structure

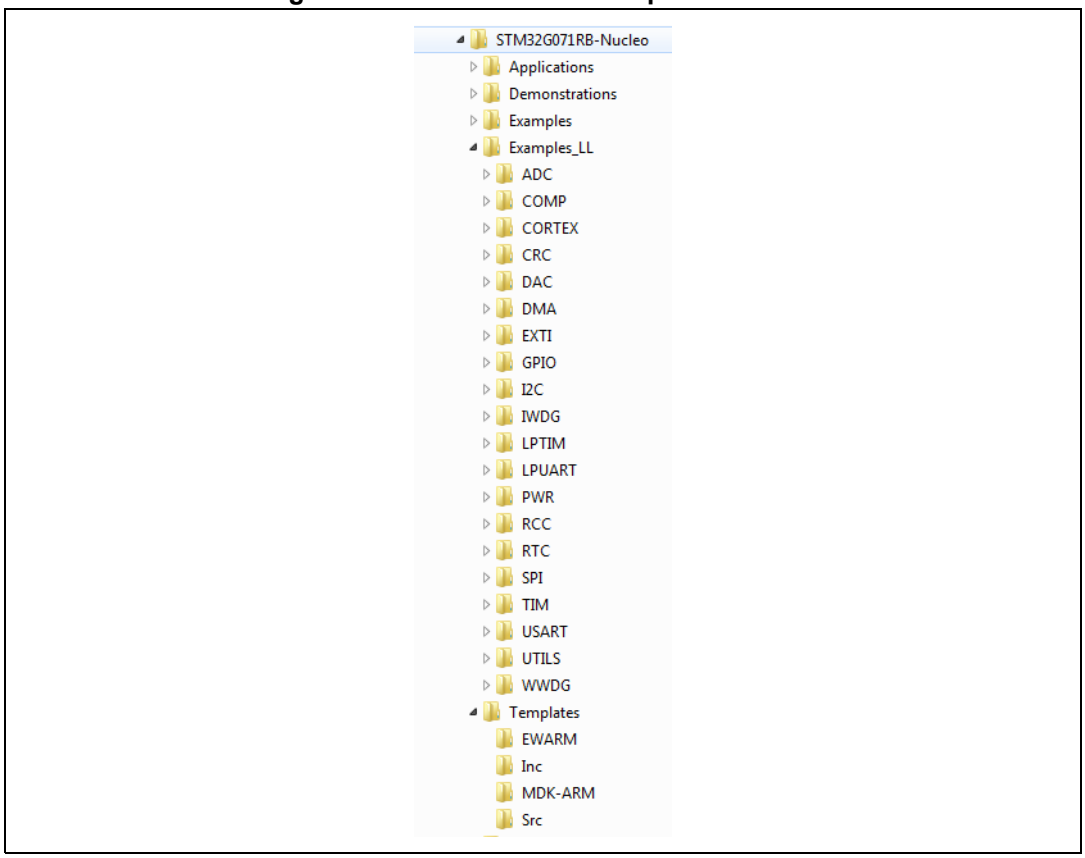


1. The components files must not be modified by the user. Only the *Projects* sources are eligible to changes by the user.

For each board, a set of examples are provided with pre-configured projects for EWARM, MDK-ARM, and SW4STM32 toolchains.

Figure 4 shows the project structure for the NUCLEO-G071RB board.

Figure 4. STM32CubeG0 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.

Templates projects available in the Templates and Templates_LL directories allow to quickly build any firmware application on a given board.

All examples have the same structure:

- *\Inc* folder that contains all header files.
- *\Src* folder for the sources code.
- *\EWARM*, *\MDK-ARM*, and *\SW4STM32* folders contain the pre-configured project for each toolchain.
- *readme.txt* describing the example behavior and needed environment to make it working
- **.ioc* file that allows users to open most of firmware examples within STM32CubeMX (starting from STM32CubeMX 5.0)

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

Table 3. Number of examples for each board

Level	NUCLEO-G070RB	NUCLEO-G071RB	STM32G081B-EVAL	STM32G071B-DISCO	NUCLEO-G031K8	STM32G0316-DISCO	Total
Templates_LL	1	1	1	1	1	1	6
Templates	1	1	1	1	1	1	6
Examples_MIX	9	9	0	2	6	0	26
Examples_LL	60	75	0	1	23	0	159
Examples	59	71	76	16	35	14	271
Demonstrations	1	1	3	1	1	1	8
Applications	10	10	12	0	2	2	36
Total	141	168	93	22	69	19	**512**

4 Getting started with STM32CubeG0

4.1 Running your first example

This section explains how simple is to run a first example within STM32CubeG0. It uses as illustration the generation of a simple LED toggle running on NUCLEO-G071RB board:

1. Download the STM32CubeG0 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 (e.g. C\Eval or G:\Tests) because some IDEs encounter problems when the path length is too long.
2. Browse to *\Projects\STM32G071RB-Nucleo\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 target memory.
6. Run the example: each time you press the USER pushbutton, the LED4 toggles (for more details, refer to the example readme file).

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

- EWARM
  a) Under the example folder, open *\EWARM* sub-folder
  b) Launch the Project.eww workspace[a]
  c) Rebuild all files: **Project->Rebuild all**
  d) Load project image: **Project->Debug**
  e) Run program: **Debug->Go(F5)**
- MDK-ARM
  a) Under the example folder, open *\MDK-ARM* sub-folder
  b) Launch the Project.uvprojx workspace[a]
  c) Rebuild all files: **Project->Rebuild all target files**
  d) Load project image: **Debug->Start/Stop Debug Session**
  e) Run program: **Debug->Run (F5)**.
- SW4STM32
  a) Open the SW4STM32 toolchain
  b) Click **File->Switch Workspace->Other** and browse to the SW4STM32 workspace directory
  c) Click **File->Import**, select **General->Existing Projects into Workspace** and then click **Next**
  d) Browse to the SW4STM32 workspace directory and select the project
  e) Rebuild all project files: select the project in the **Project explorer** window then click the **Project->build project** menu
  f) Run program: Run->Debug (F11)

a. The workspace name may change from one example to another.

4.2 Developing your own application

4.2.1 Using STM32CubeMX to develop or update your application

In the STM32CubeG0 MCU Package, all Example projects are generated with the STM32CubeMX tool to initialize the system, peripherals and middleware.

The direct use of an existing Example project from within STM32CubeMX requires STM32CubeMX 5.0 or higher:

- After the installation of STM32CubeMX, open and eventually 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 delimited by comments `USER CODE BEGIN` and `USER CODE END`. In case of a modification of the IP selection and setting, STM32CubeMX updates the initialization part of the code but preserves the main application source code.

For developing an own project in STM32CubeMX, follow the step-by-step process:

1. Select the STMicroelectronics STM32 microcontroller that matches the required set of peripherals.
2. Configure each required embedded software thanks to a pinout-conflict solver, a clock-tree setting helper, a power consumption calculator, and the utility performing MCU peripheral configuration (such as GPIO or USART) and middleware stacks (such as USB).
3. Generate the initialization C code based on the configuration selected. 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 STM32CubeMX user manual (UM1718).

For a list of the available Example projects in STM32CubeG0, refer to the *STM32Cube firmware examples for STM32G0 Series* application note (AN5110).

4.2.2 HAL application

This section describes the steps required to create your own HAL application using STM32CubeG0:

1. **Create your project**

   To create a new project, you either start 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 STM32G081B-EVAL).

   The *Template* project is providing empty main loop function, however it is a good starting point to get familiar with project settings for STM32CubeG0. The template has the following characteristics:

   – It contains the source code of HAL, CMSIS and BSP drivers which are the minimal components required to develop a code on a given board.

   – It contains the include paths for all the firmware components.

   – It defines the STM32G0 device supported, thus allowing to configure the CMSIS and HAL drivers accordingly.

   – It provides read-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 to update the include paths.

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

   The available middleware stacks are: USB PD library, FreeRTOS™, and FatFS. To know which source files must be added to the project file list, refer to the documentation provided for each middleware. It is possible to look at the applications available under *\Projects\STM32xxx_yyy\Applications\<MW_Stack>* (where *<MW_Stack>* refers to the middleware stack, such as USB_Device) to know which source files and which 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, it 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 know 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 do the following tasks:

   a) Configuration of the Flash prefetch and SysTick interrupt priority (through macros defined in *stm32g0xx_hal_conf.h*).

   b) Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority `TICK_INT_PRIO` defined in *stm32g0xx_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 *stm32g0xx_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:

a) `HAL_RCC_OscConfig()`: this API configures the internal and/or external oscillators, as well as the PLL source and factors. The user chooses to configure one oscillator or all oscillators. The PLL configuration can be skipped if there is no need to run the system at high frequency.

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

6. **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 you plan to use peripheral interrupt or DMA.

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

7. **Develop your application**

At this stage, your system is ready and you start developing your application code.

– The HAL provides intuitive and ready-to-use APIs to configure the peripheral. It supports polling, interrupts and DMA programming model, to accommodate any application requirements. For more details on how to use each peripheral, refer to the rich examples set provided in the STM32CubeG0 MCU Package.

– If your application has some real-time constraints, you find a large set of examples showing how to use FreeRTOS™ and integrate it with all middleware stacks provided within STM32CubeG0. This is a good starting point to develop your application.

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

4.2.3 LL application

This section describes the steps needed to create your own LL application using STM32CubeG0.

1. **Create your project**

   To create a new project you 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-G071RB).

   The *Template* project provides an empty main loop function, however it is a good starting point to get familiar with project settings for STM32CubeG0.

   Template main characteristics are the following:

   – It contains the source codes of the LL and CMSIS drivers which are the minimal components needed to develop code on a given board.
   – It contains the include paths for all the required firmware components.
   – It selects the supported STM32G0 device and allows to configure the CMSIS and LL drivers accordingly.
   – It provides ready-to-use user files, that are pre-configured as follows:

     main.h: LED & USER_BUTTON definition abstraction layer.

     main.c: System clock configuration for maximum frequency.

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

   To port an existing project to another target board, start from the *Templates_LL* project provided for each board and available under *\Projects\<STM32xxx_yyy>\Templates_LL*:

   a) Select a LL example

      To find the board on which LL examples are deployed, refer to the list of LL examples *STM32CubeProjectsList.html*, to *Table 3: Number of examples for each board* or to application note *"STM32Cube firmware examples for STM32G0 Series"* (AN5110)

   b) 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 have been flagged with specific tags:

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

   Thus the main porting steps are the following:

   – Replace the *stm32g0xx_it.h* file
   – Replace the *stm32g0xx_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 LED definition, replace each LEDx occurrence with another LEDy available in *main.h*.

Thanks to these adaptations, the example should be functional on the targeted board.

4.3 Getting STM32CubeG0 release updates

The STM32CubeG0 MCU Package comes with an updater utility, STM32CubeUpdater, also available as a menu within STM32CubeMX code generation tool.

The updater solution detects new firmware releases and patches available from *www.st.com* and proposes to download them to the user's computer.

4.3.1 Installing and running the STM32CubeUpdater program

Follow the sequence below to install and run the STM32CubeUpdater:

1. To launch the installation, double-click the *SetupSTM32CubeUpdater.exe* file.
2. Accept the license terms and follow the different installation steps.
3. Upon successful installation, STM32CubeUpdater becomes available as an STMicroelectronics program under *Program Files* and is automatically launched. The STM32CubeUpdater icon appears in the system tray. Right-click the updater icon and select **Updater Settings** to configure the Updater connection and whether to perform manual or automatic checks. For more details on Updater configuration, refer to section 3 of STM32CubeMX user manual (UM1718).

5 FAQ

5.1 What is the license scheme for the STM32CubeG0 MCU Package?

The HAL is distributed under a non-restrictive BSD (Berkeley Software Distribution) license.

The middleware stacks made by STMicroelectronics (USB Device Libraries, STemWin) come with a licensing model allowing easy reuse, provided it runs on an STMicroelectronics device.

The middleware based on well-known open-source solutions (FreeRTOS™ and FatFS) have user-friendly license terms. For more details, refer to the license agreement of each middleware.

5.2 What boards are supported by the STM32CubeG0 MCU Package?

The STM32CubeG0 MCU Package provides BSP drivers and ready-to-use examples for the following STM32G0 boards:

- NUCLEO-G070RB
- NUCLEO-G071RB
- STM32G081B-EVAL
- STM32G071B-DISCO
- NUCLEO-G031K8
- STM32G0316-DISCO

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

Yes. STM32CubeG0 provides a rich set of examples and applications. They come with the pre-configured projects for IAR™, Keil® and GCC-based toolchains.

5.4 Is there any link with Standard Peripheral Libraries?

The STM32Cube 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 peripheral common features rather than hardware. Their higher abstraction level allows defining a set of user-friendly APIs that are easily portable from one product to another.
- The LL drivers offer low-layer APIs at registers level. They are organized in a simpler and clearer way than 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 STM32Cube LL drivers, since each SPL API has its equivalent LL API(s).

5.5 Does the HAL layer take benefit from 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).

5.6 How are the product/peripheral specific features managed?

The HAL drivers offer extended APIs, i.e. specific functions as add-ons to the common API to support features available on some products/lines only.

5.7 How can STM32CubeMX generate code based on embedded software?

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

5.8 How can I get regular updates on the latest STM32CubeG0 MCU Package releases?

The STM32CubeG0 MCU Package comes with an updater utility, STM32CubeUpdater, that is configurable for automatic or on-demand checks for new firmware package updates (new releases or/and patches).

STM32CubeUpdater is integrated as well within the STM32CubeMX tool. When using this tool for STM32G0 configuration and initialization C code generation, the user benefits from STM32CubeMX self-updates as well as STM32CubeG0 MCU Package updates.

For more details, refer to *Section 4.3*.

5.9 When should I use HAL versus LL drivers?

HAL drivers offer high-level and function-oriented APIs, with a high level of portability. Product/IPs complexity is hidden for end users.

LL drivers offer low-layer APIs at registers level, with a better optimization but less portability. They require a deep knowledge of product/IPs specifications.

5.10 How can I include LL drivers in my environment? Is there any LL configuration file as for HAL?

There is no configuration file. Source code shall directly include the necessary *stm32g0xx_ll_ppp.h* file(s).

5.11 Can I use HAL and LL drivers together? If yes, what are the constraints?

It is possible to use both HAL and LL drivers. One handles the IP initialization phase with HAL and then manages 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 operates directly on peripheral registers. Mixing HAL and LL is illustrated in Examples_MIX example.

5.12 Are there any LL APIs which are not available with HAL?

Yes, there are.

A few Cortex® APIs have been added in *stm32g0xx_ll_cortex.h* , for instance for accessing SCB or SysTick registers.

5.13 Why are SysTick interrupts not enabled on LL drivers?

When using LL drivers in standalone mode, you do not need to enable SysTick interrupts because they are not used in LL APIs, while HAL functions requires SysTick interrupts to manage timeouts.

5.14 How are LL initialization APIs enabled?

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

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

6 Revision history

Table 4. Document revision history

Date	Revision	Changes
8-Dec-2017	1	Initial version
25-Oct-2018	2	Extended document scope to the NUCLEO-G070RB board: – Updated *Table 2: Boards for STM32G0 Series* and *Table 3: Number of examples for each board* – Updated *Section 5.2: What boards are supported by the STM32CubeG0 firmware package?* Detailed the use of STM32CubeG0 with STM32CubeMX: – Reorganized *Section 4.2: Developing your own application* – Updated *Section 4.2.1: Using STM32CubeMX to develop or update your application*
26-Feb-2019	3	Extended document scope to the STM32G071B-DISCO board: – Updated *Table 2: Boards for STM32G0 Series* and *Table 3: Number of examples for each board* – Updated *Section 5.2: What boards are supported by the STM32CubeG0 firmware package?* Updated STM32Cube™ description in *Introduction*.
1-Apr-2019	4	Extended document scope to the NUCLEO-G031K8 and STM32G0316-DISCO boards: – Updated *Table 1: Macros for STM32G0 Series*, *Table 2: Boards for STM32G0 Series* and *Table 3: Number of examples for each board* – Updated *Section 5.2: What boards are supported by the STM32CubeG0 MCU Package?* Replaced *STM32CubeG0 firmware package* by *STM32CubeG0 MCU Package* across the whole document.