life.augmented

UM2656
User manual

Getting started with STM32CubeL5 for STM32L5 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 design to the production, 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 with both a graphical and command-line interfaces.
  - STM32CubeMonitor-Power (STM32CubeMonPwr), a monitoring tool to measure and help optimize the MCU power consumption.
- STM32Cube MCU Packages, comprehensive embedded-software platforms specific to each microcontroller series (such as STM32CubeL5 for STM32L5 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 Device, USB PD, FatFs, STMTouch™, TrustedFirmware (TF-M), mbed-TLS and mbed-crypto.
  - All embedded software utilities with a full set of peripherals and applicative examples.

This user manual describes how to get started with the STM32CubeL5 MCU Package.

Section 1 describes the main features of the STM32CubeL5 MCU Package.

Section 2 and *Section 3* provide an overview of the STM32CubeL5 architecture and MCU Package structure.

Contents

List of tables

List of figures

1 STM32CubeL5 main features

The STM32CubeL5 MCU Package runs on STM32 32-bit microcontrollers based on the Arm®(a) Cortex®-M33 processor with TrustZone® and FPU.

STM32CubeL5 gathers, in a single package, all the generic embedded software components required to develop an application for the STM32L5 Series microcontrollers. In line with the STM32Cube initiative, this set of components is highly portable, not only within STM32L5 Series microcontrollers but also to other STM32 series.

STM32CubeL5 is fully compatible with STM32CubeMX code generator for 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.

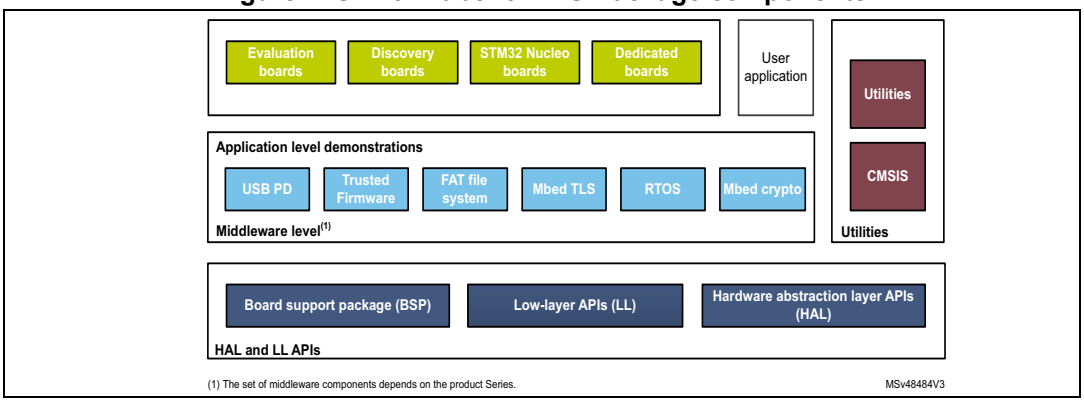
STM32CubeL5 MCU Package also contains a set of middleware components with the corresponding examples. They come with free user-friendly license terms:

- CMSIS-RTOS implementation with FreeRTOS™ open source solution
- Full USB Device stack supporting the following device classes: HID, MSC, CDC, Audio, DFU, LPM, BCD.
- USB PD library
- Arm Trusted Firmware-M (TF-M) integration solution
- Mbed TLS and Mbed Crypto libraries
- FAT file system based on open source FatFS solution
- STMTouch touch sensing library solution.

Several applications and demonstrations implementing all these middleware components are also provided in the STM32CubeL5 MCU Package.

The STM32CubeL5 MCU Package component layout is illustrated in *Figure 1*.

Figure 1. STM32CubeL5 MCU Package components

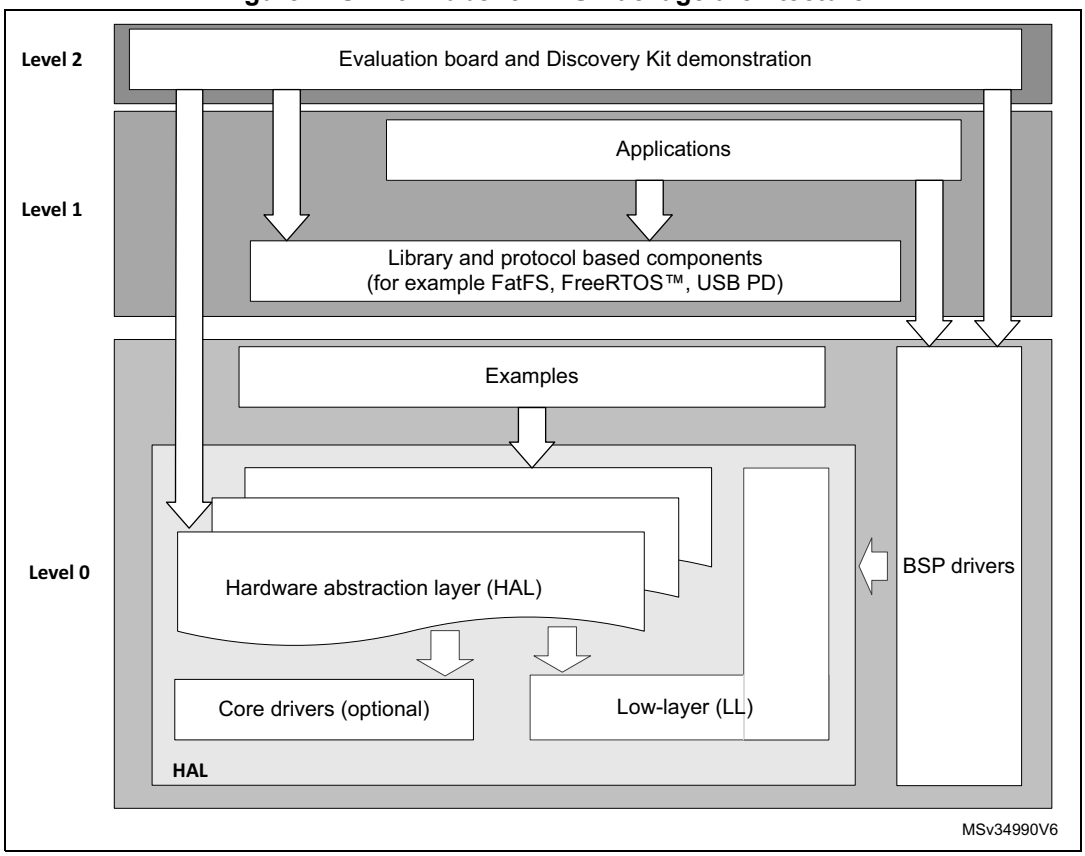


a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and or elsewhere.

2 STM32CubeL5 architecture overview

The STM32CubeL5 MCU Package solution is built around three independent levels that easily interact as described in *Figure 2*.

Figure 2. STM32CubeL5 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 that are specific to each of the hardware components implemented on the boards (such as LCD, Audio, microSD™ and MEMS drivers). Audio and MEMS drivers are not supported. It is composed of two parts:

- The component driver;

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

- BSP driver;

  This is where the component drivers are linked to a specific board to provide 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 STM32CubeL5 HAL and LL are complementary and cover a wide range of application requirements:

- The HAL drivers offer highly-portable, high-level function-oriented 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 processes.For example, in the case of communication peripherals (I^2S, UART, and others), it provides APIs that initialize and configure the peripheral, managing data transfer based on polling, interrupt or DMA process, and handling communication errors that may occur during communication. The HAL driver APIs are split in two categories:

  – Generic APIs which provide common and generic functions for all the STM32 Series microcontrollers

  – Extended 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 register level, which are more optipmized but less portable. They require a deeper 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 the data structures
– A set of functions used to fill initialization data structures with each of the fields corresponding reset values
– A set of functions to reset the peripherals (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 built 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 Device library, USB PD library, FreeRTOS™, FatFS, Arm Trusted Firmware-M (TF-M), Mbed TLS, Mbed Crypto and STMTouch touch sensing. Horizontal interactions between the components of this layer is simply done 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 a microSD drive or provides an implementation of a USB Mass Storage Class. 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 Device library:**
  - Several USB classes supported (Mass-Storage, HID, CDC, DFU, LPM and BCD).
  - Support of a multi-packet transfer feature that allows large amounts of data to be sent without splitting them into maximum packet size transfers.
  - Use of configuration files to change the core and the library configuration without changing the library code which can be kept as Read Only.
  - 32-bit aligned data structures to handle DMA-based transfer in high-speed modes.
  - RTOS and standalone operation.
  - Link with low-level drivers through an abstraction layer using the configuration file to avoid any dependency between the library and the low-level drivers.
- **USB PD Device and Core libraries**
  - 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
  - TrustZone support.
- **FAT file system**
  - FatFS FAT open source library
  - Long file name support
  - Dynamic multi-drive support
  - RTOS and standalone operation
  - Examples with microSD™
- **Arm Trusted Firmware-M (TF-M)**
  - Reference implementation of the Arm Platform Security Architecture (PSA) for TrustZone
  - Secure services are:
    - Secure Storage Service
    - Attestation
    - Crypto Service
    - TF-M Audit Log
    - Platform Service

An example of TF-M application is available in the STM32CubeL5 firmware package under *|Projects|STM32L562E-DK|Applications|TFM*.

- **Mbed TLS**
  - SSL/TLS secure layer based on open source
- **Mbed Crypto**
  - Open source cryptography library that supports a wide range of cryptographic operations, including:

    Key management

    Hashing

    Symmetric cryptography

    Asymmetric cryptography

    Message authentication (MAC)

    Key generation and derivation

    Authenticated encryption with associated data (AEAD).
- **STM32 Touch sensing library:**
  Robust STMTouch capacitive touch sensing solution supporting proximity, touchkey, linear and rotary touch sensors. It is based a proven surface charge transfer acquisition principle.

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 STM32CubeL5 MCU Package overview

3.1 Supported STM32L5 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 an easy portability to other devices.

In addition, thanks to its layered architecture, the STM32CubeL5 offers full support of all STM32L5 Series. The user has only to define the right macro in *stm32l5xx.h*.

Table 1 shows the macro to define depending on the STM32L5 device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32L5 Series

Macro defined in *stm32l5xx.h*	STM32L5 devices
STM32L552xx	STM32L552CC, STM32L552CE, STM32L552ME, STM32L552QC, STM32L552QE, STM32L552RC, STM32L552RE, STM32L552VC, STM32L552VE, STM32L552ZC, STM32L552ZE.
STM32L562xx	STM32L562CE, STM32L562ME, STM32L562QE, STM32L562RE, STM32L562VE, STM32L562ZE.

STM32CubeL5 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 STM32L5 Series

Board	Board STM32L5 supported devices
NUCLEO-L552ZE-Q	STM32L552xx
STM32L552E-EV	STM32L552xx
STM32L562E-DK	STM32L562xx

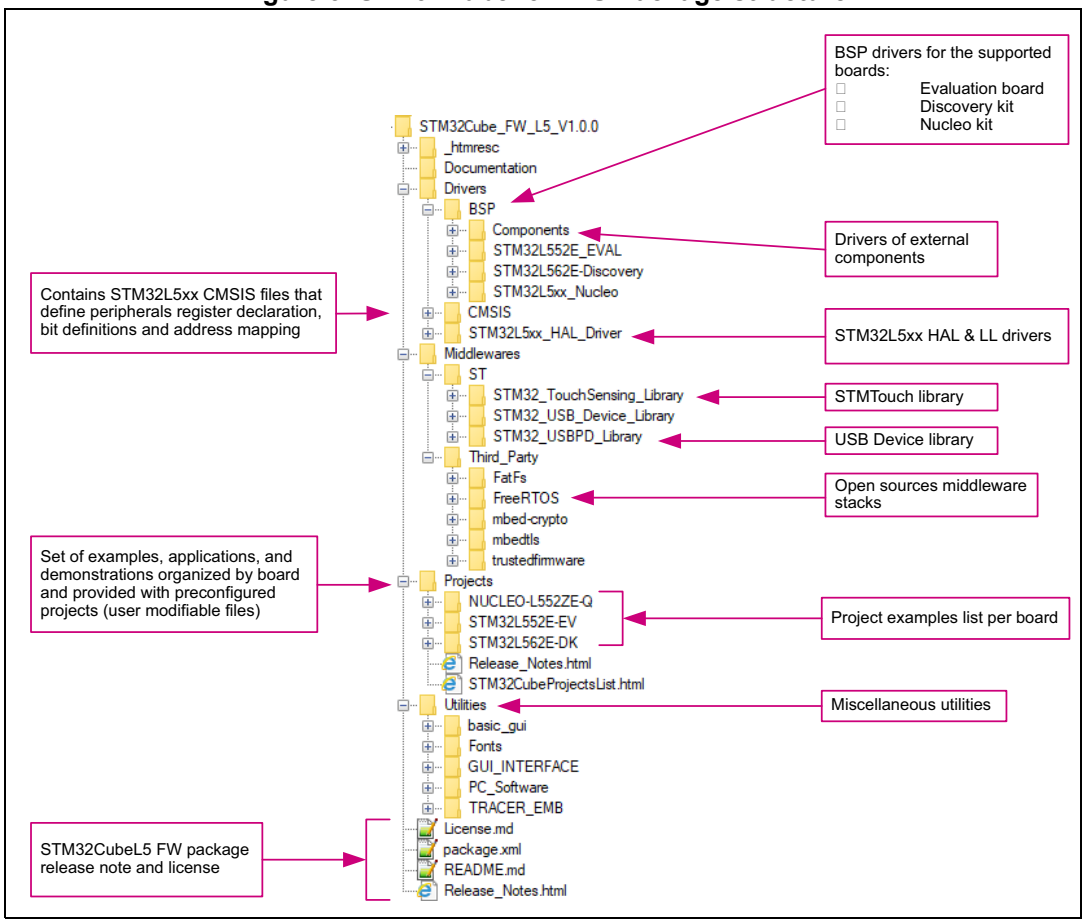
STM32CubeL5 supports the Nucleo-144 boards listed in *Table 2* above.

The STM32CubeL5 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 the latter has the same hardware features (LED, LCD display, buttons...).

3.2 MCU Package overview

The STM32CubeL5 MCU Package solution is provided in one single zip package having the structure shown in *Figure 3*.

Figure 3. STM32CubeL5 MCU Package structure

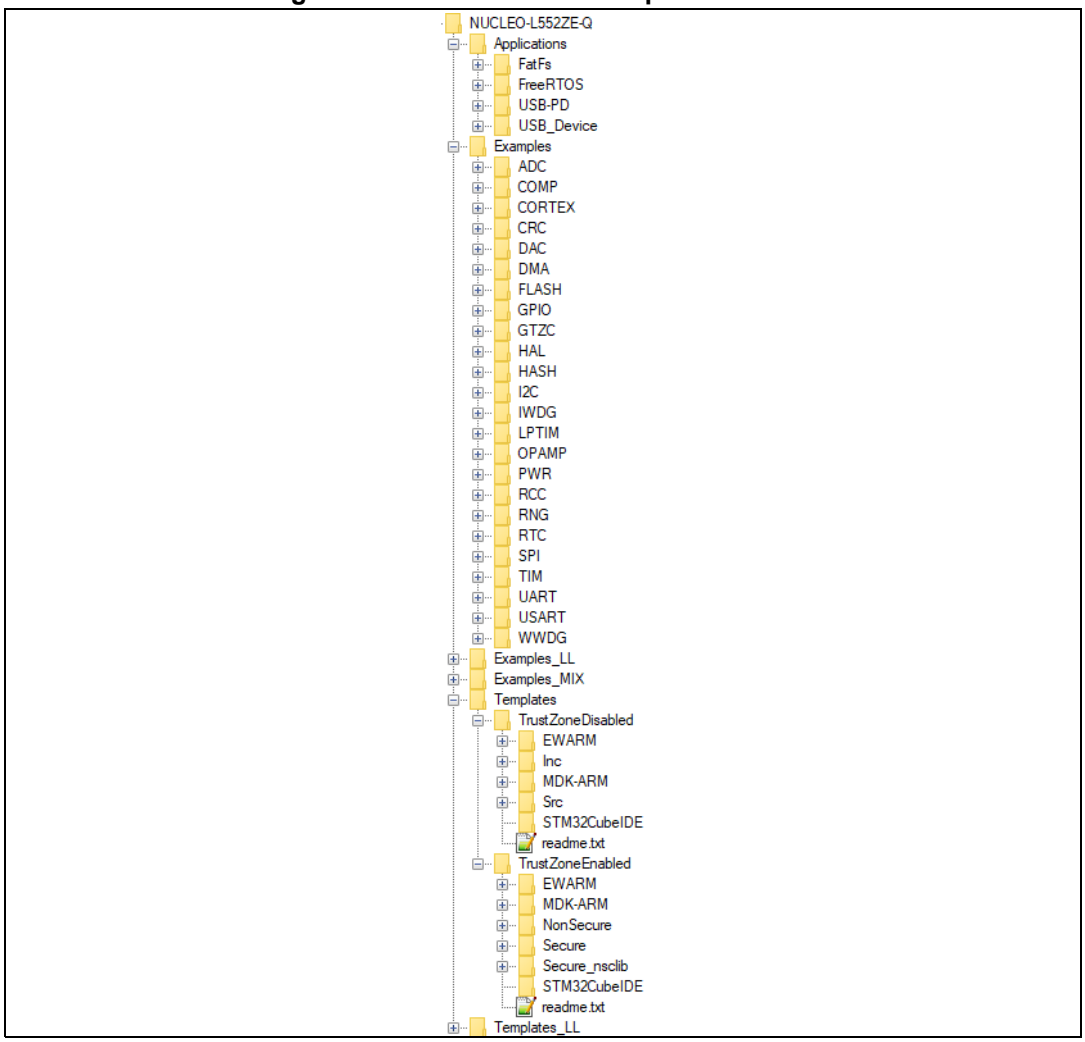


1. The components files must not be modified by the user. Only the *\Projects* sources are editable by the user.

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-G071RB board.

Figure 4. STM32CubeL5 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.

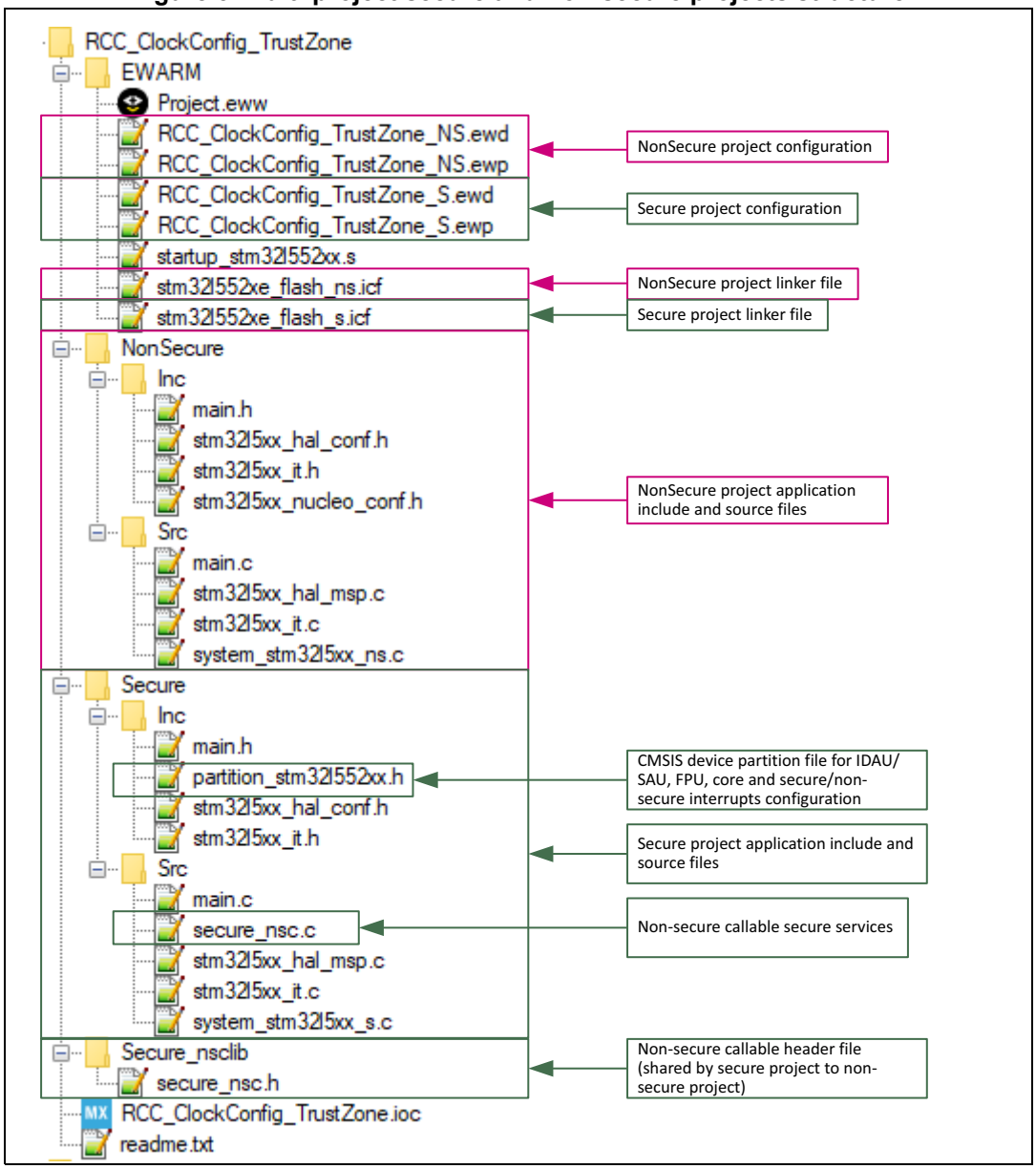
3.2.1 TrustZone-enabled projects

TrustZone-enabled “*Examples”* names are prefixed with "_TrustZone". The rule is applied also for *“Applications”* (except TFM which is natively for TrustZone).

TrustZone-enabled “Examples” and “Applications” are provided with a multi-project structure composed of secure and non-secure sub-projects as presented below in *Figure 5*.

TrustZone-enabled projects are developped according to CMSIS-5 device template extended to include the system partitioning hear file partition_<device>.h responsible for principally the setup of the secure attribute unit (SAU), the FPU and the secure/non-secure interrupts assignment in secure execution state. This setup is performed in secure CMSIS SystemInit() function called at startup before entering the secure application main() function (refer to Arm TrustZone-M documentation of software guidelines).

Figure 5. Multi-project secure and non-secure projects structure



The STM32CubeL5 package firmware package provides default memory partitioning in the *partition_ <device>.h* files available under:
`\Drivers\CMSIS\Device\STM32L5\Include\Templates.`
The default SAU regions definition is as follows:

- SAU region 0: 0x0C03E000-0x0C03FFFF (Secure, Non-Secure Callable)
- SAU region 1: 0x08040000-0x0807FFFF (Non-Secure FLASH Bank2 (256 Kbytes))
- SAU region 2: 0x20018000-0x2003FFFF (Non-Secure RAM (2nd half SRAM1 + SRAM2 (160 Kbytes)))
- SAU region 3: 0x40000000-0x4FFFFFFF (Non-Secure Peripheral mapped memory)
- SAU region 4: 0x60000000-0x9FFFFFFF (Non-Secure external memories)
- SAU region 5: 0x0BF90000-0x0BFA8FFF (Non-Secure System memory)

To match the default partitioning, the STM32L5xx Series devices must have the following user option bytes set:

- TZEN=1 (TrustZone-enabled device)
- DBANK=1 (Dual bank Flash configuration)
- SECWM1_PSTRT=0x0 SECWM1_PEND=0x7F (all 128 pages of Flash Bank1 set as secure)
- SECWM2_PSTRT=0x1 SECWM2_PEND=0x0 (no page of Flash Bank2 set as secure, hence Bank2 is non-secure).

Note: The internal Flash is fully secure by default in TZEN=1 and user option bytes SECWM1_PSTRT/SECWM1_PEND and SECWM2_PSTRT/SECWM2_PEND should be set according to the application memory configuration (SAU regions and secure/non-secure applications project linker files must be aligned too).

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 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 5.4.0)

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

Table 3. Number of examples for each board

Level	STM32L562E-DK	STM32L552E-EV	NUCLEO-L552ZE-Q	Total
Templates_LL	1	1	1	3
Templates	2	2	2	6
Examples_MIX	0	0	11	11
Examples_LL	2	0	67	69
Examples	61	50	93	204
Demonstrations	0	1	0	1
Applications	13	21	14	48
Total	79	75	188	**342**

4 Getting started with STM32CubeL5

4.1 Running a first example

This section explains how simple it is to run a first example on an STM32L5 board. The program simply toggles a LED on the NUCLEO-G071RB board:

Download the STM32CubeL5 MCU Package. Unzip it into an appropriate directory. Make sure the package structure shown in *Figure 3* is not modified. Note that it is also recommended to copy the package as close as possible to the root volume (for example C\ST or G:\Tests) because some IDEs encounter problems when the path length is too long.

4.1.1 Running a first TrustZone-enabled example

Prior to loading and running a TrustZone-enabled example, it is mandatory to read the example readme file for any specific configuration which insures that the security is enabled as described in section 3.2.1 (TZEN=1 (user option byte)).

1. Browse to *\Projects\NUCLEO-L552ZE-Q\Examples*.
2. Open *\GPIO*, then *\GPIO_IoToggle_TrustZone* folders.
3. Open the project with the preferred toolchain. A quick overview on how to open, build and run an example with the supported toolchains is given below.
4. Rebuild in sequence all secure and non-secure project files and load the secure and non-secure images into target memory.
5. Run the example: on a regular basis, the secure application toggles LED1 every second and non-secure application toggles LED2 twice as fast (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
  c) Rebuild the xxxxx_S secure project files: **Project->Rebuild all**
  d) Set the xxxxx_NS non-secure project as Active application (right click on xxxxx_NS project **Set as Active**)
  e) Rebuild the xxxxx_NS non-secure project files: **Project->Rebuild all**
  f) Flash the non-secure binary with Project->Download->Download active application
  g) Set the xxxxx_S as active application (right click on xxxxx_S project **Set as Active**)
  h) Flash the secure binary with **Download and Debug button (Ctrl+D)**
  i) Run program: **Debug->Go(F5)**
- MDK-ARM
  a) Open your MDK-ARM toolchain
  b) Open Multi-projects workspace file Project.uvmpw
  c) Select the xxxxx_s project as Active Project (**Set as Active Project**)
  d) Build xxxxx_s project
  e) Select the xxxxx_ns project as Active Project (**Set as Active Project**)
  f) Build xxxxx_ns project
  g) Load the non-secure binary (**F8**)
  (this shall download the \MDK-ARM\xxxxx_ns\Exe\xxxxx_ns.axf to flash memory)
  h) Select the Project_s project as Active Project (**Set as Active Project**)
  i) Load the secure binary (**F8**)
  (this shall download the \MDK-ARM\xxxxx_s\Exe\xxxxx_s.axf to flash memory)
  j) Run the example

- STM32CubeIDE
  a) Open your STM32CubeIDE toolchain
  b) Open Multi-projects workspace file .project
  c) Rebuild xxxxx_Secure project
  d) Rebuild xxxxx_NonSecure project
  e) Launch "Debug as STM32 Cortex-M C/C++ Application" for the secure project.
  On "Edit configuration" window, select "startup" panel and add load image an symbols of non secure project.
  Be carefull, the non-secure project has to be loaded before the secure project.
  Then click "OK".
  f) Run the example on debug perspective.

4.1.2 Running a first TrustZone-disabled example

Prior to loading and running a TrustZone-disabled example, it is mandatory to read the example readme file for any specific configuration or if nothing is mentioned, insure that the board device has the security disabled (TZEN=0 (user option byte)). See FAQ for doing the optional regression to TZEN=0.

1. Browse to *\Projects\NUCLEO-L552ZE-Q\Examples*.
2. Open *\GPIO*, then *\GPIO_EXTI* folders.
3. Open the project with a preferred toolchain. A quick overview on how to open, build and run an example with the supported toolchains is given below.
4. Rebuild all files and load the image into target memory.
5. Run the example: each time the USER pushbutton is pressed, LED1 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
  - g) Under the example folder, open *\EWARM* sub-folder
  - h) Launch the Project.eww workspace[(a)]
  - i) Rebuild all files: **Project->Rebuild all**
  - j) Load project image: **Project->Debug**
  - k) 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)**.
- STM32CubeIDE
  - a) Open the STM32CubeIDE toolchain
  - b) Click **File->Switch Workspace->Other** and browse to the STM32CubeIDE workspace directory
  - c) Click **File->Import**, select **General->Existing Projects into Workspace** and then click **Next**
  - d) Browse to the STM32CubeIDE 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 a custom application

4.2.1 Using STM32CubeMX to develop or update an application

In the STM32CubeL5 MCU Package, nearly 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 the STM32CubeMX tool requires STM32CubeMX 5.5.0 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 USART) and middleware stacks (such as USB).
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 *STM32CubeMX for STM32 configuration and initialization C code generation* (UM1718).

For a list of the available example projects for the STM32CubeL5, refer to the *STM32Cube firmware examples for STM32L5 Series application note* (AN5424).

4.2.2 HAL application

This section describes the steps required to create a custom HAL application using STM32CubeL5:

1. **Create a project**

   To create a new project, either starting 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 provides an empty main loop function, however it is a good starting point to understand the STM32CubeL5 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 include paths for all the firmware components.

   – It defines the STM32L5 device supported, allowing the CMSIS and HAL drivers to be configure 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 include paths are update.

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

   The available middleware stacks are: USB Device library, USB PD library, FreeRTOS™, FatFS and STMTouch sensing library. 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 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 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 the `HAL_Init()` API to initialize the HAL Library, which carries out the following tasks:

   a) Configuration of the Flash prefetch and SysTick interrupt priority (through macros defined in s*tm32l5xx_hal_conf.h*).

   b) Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority `TICK_INT_PRIO` defined in s*tm32l5xx_hal_conf.h*, which

is clocked by the MSI (at this stage, the clock is not yet configured and thus the system is running from the 4 MHz MSI).

c) Setting of NVIC Group Priority to 0.

d) Call of `HAL_MspInit()` callback function defined in *stm32l5xx_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 the DMA interrupt (if needed).
- Enable the 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 used.

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

7. Develop an application

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

– 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 examples set provided in the STM32CubeL5 MCU Package.

– If the application has real-time constraints, there is a large set of examples showing how to use FreeRTOS™ and how to integrate it with all middleware stacks provided within STM32CubeL5. This is a good starting point to develop an 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 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 allow an override in case of other implementations are needed 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 a custom LL application using the STM32CubeL5.

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-G071RB).

   The *Template* project provides an empty main loop function, which is a good starting point to understand the project settings for the STM32CubeL5.

   Template main characteristics are the following:

   – It contains the source codes of the LL and CMSIS drivers which are the minimun set of components needed to develop code on a given board.
   – It contains the include paths for all the required firmware components.
   – It selects the supported STM32L5 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 & 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 STM32L5 Series* (AN5424).

   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 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 *stm32l5xx_it.h* file
   – Replace the *stm32l5xx_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 LEDx occurrence with another LEDy available in *main.h*.

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

4.3 Getting STM32CubeL5 release updates

The new STM32CubeL5 MCU Package releases and patches are available from www.st.com/stm32l5. They may be retrieved from the "CHECK FOR UPDATE" button in STM32CubeMX. For more details, refer to section 3 of *STM32CubeMX for STM32 configuration and initialization C code generation* (UM1718).

5 FAQ

5.1 What is the license scheme for the STM32CubeL5 MCU Package?

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

The middleware stacks made by STMicroelectronics (ex: USB Device library) 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 appropriate middleware license agreement.

5.2 What boards are supported by the STM32CubeL5 MCU Package?

The STM32CubeL5 MCU Package provides BSP drivers and ready-to-use examples for the following STM32L5 boards:

- NUCLEO-G071RB
- STM32G071B-DISCO
- STM32G0316-DISCO

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

Yes. STM32CubeL5 provides a rich set of examples and applications. They come with the pre-configured projects for IAR™, Keil® and GCC-based toolchains.

5.4 How to enable TrustZone on STM32L5 Series devices?

All STM32L5 devices support TrustZone. Factory default state is TrustZone disabled. The TrustZone security is activated with the TZEN option bit in the FLASH_OPTR register. The user option bytes configuration may be done with STM32CubeProgrammer.

5.5 How to disable TrustZone on STM32L5 Series devices?

Unless the Readout protection (RDP) level is set to 2, it is possible to revert the TZEN option bit to 0 to disable the TrustZone option on all STM32L5 devices. The first step is to set the RDP level 1 protection and then reset TZEN bit to 0 and RDP level to 0 and apply the Option byte updates (all this may be done with STM32CubeProgrammer).

5.6 How to update the secure / non-secure memory mapping

In case of memory isolation for secure and non-secure applications, the secure and non-secure applications share the same internal Flash and embedded SRAMs.

The STM32CubeL5 firmware package provides default memory partitioning in the partition_<device>.h files available under:

|Drivers|CMSIS|Device|STM32L5|Include|Templates (see *Section 3.2.1*). Any memory map partitioning change between secure and non-secure applications requires the following updates and alignments (without overlap between secure and non-secure memory space and using secure and non-secure memory address aliases):

- SAU non-secure area update (internal Flash and SRAMs) (see partition_stm32l5xx.h file).
- Secure and non-secure linker files update to correctly locate the secure and non-secure code and data.
- Update the non-secure address to jump to (in secure main.c and non-secure reset handler in non-secure linker file)
- Update the Flash watermark option bytes (SEC_WMx_PSTRT/SEC_WMx_PEND) to define the secure/non-secure Flash areas (with STM32CubeProgrammer).

5.7 Why do I enter in SecureFault_Handler() ?

SecureFault_Handler() is reachable if the SecureFault handler is enabled by the secure code with SCB->SHCSR |= SCB_SHCSR_SECUREFAULTENA_Msk;

Any jump to SecureFault_Handler() during the application execution is the result of a security violation detected at IDAU/SAU level such as a fetch of a non-secure application to secure address.

If SecureFault handler is not enabled, the security violation is escalated to the HardFault handler.

5.8 Are there any links 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 the features that are common to the peripherals rather than hardware. a set of user-friendly APIs allow 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 in a simpler and clearer way 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 STM32Cube LL drivers, since each SPL API has its equivalent LL API(s).

5.9 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).

5.10 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.

5.11 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.

5.12 How to get regular updates on the latest STM32CubeL5 MCU Package releases?

Refer to *Section 4.3*.

5.13 When should the HAL be used 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 register level APIs, with a better optimization but less portable. They require in depth knowledge of product/IPs specifications.

5.14 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 shall directly include the necessary *stm32l5xx_ll_ppp.h* file(s).

5.15 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 operates directly on peripheral registers. Mixing HAL and LL is illustrated in Examples_MIX example.

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

Yes, there are. A few Cortex® APIs have been added in stm32l5xx_ll_cortex.h, for instance for accessing SCB or SysTick registers.

5.17 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 requires SysTick interrupts to manage timeouts.

5.18 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
12-Dec-2019	1	Initial version