Getting started with STM32CubeU3 for STM32U3 series

Introduction

STM32Cube is an STMicroelectronics original initiative to improve designer productivity significantly 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
  - STM32CubeCLT, an all-in-one command-line development toolset with code compilation, board programming, 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 STM32CubeU3 for the STM32U3 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, NetX Duo, USBX, touch library, Mbed TLS, 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 STM32CubeU3 MCU Package.

Section 2 describes the main features of the STM32CubeU3 MCU Package. Section 3 and Section 4 provide an overview of the STM32CubeU3 architecture and MCU Package structure.

1 General information

The STM32Cube MCU package runs on STM32 32-bit microcontrollers based on Arm® Cortex®-M33 processor with Arm® TrustZone® and FPU.

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

2 STM32CubeU3 main features

The STM32CubeU3 MCU Package gathers, in a single package, all the generic embedded software components required to develop an application for the STM32U3 series microcontrollers. In line with the STM32Cube initiative, this set of components is highly portable, not only within the STM32U3 series microcontrollers but also to other STM32 series.

STM32CubeU3 is fully compatible with the STM32CubeMX code generator, to generate 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 an open-source BSD license for user convenience.

The STM32CubeU3 MCU Package also contains a comprehensive middleware component constructed around Microsoft® Azure® RTOS middleware, and other in-house and open-source stacks, with the corresponding examples.

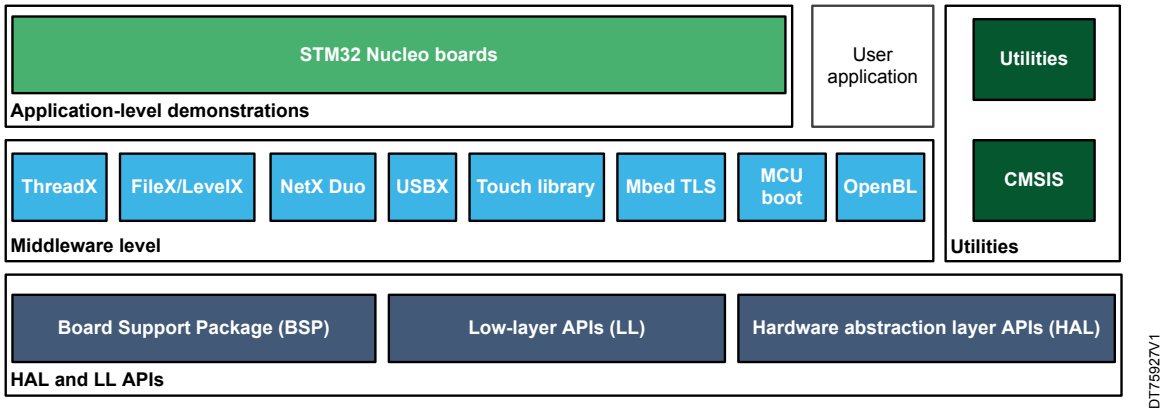
They come with free, user-friendly license terms:

- Integrated and full-featured Azure® RTOS: Azure® RTOS ThreadX
- CMSIS-RTOS implementation with Azure® RTOS ThreadX
- USB Host and Device stacks coming with many classes: Azure® RTOS USBX
- Advanced file system and flash translation layer: FileX/LevelX
- Industrial grade networking stack: optimized for performance coming with many IoT protocols: NetX Duo
- OpenBootloader
- MCUboot
- Mbed TLS libraries
- STMTouch touch sensing library solution

Several applications and demonstrations implementing all these middleware components are also provided in the STM32CubeU3 MCU Package.

The STM32CubeU3 MCU Package component layout is illustrated in Figure 1 below.

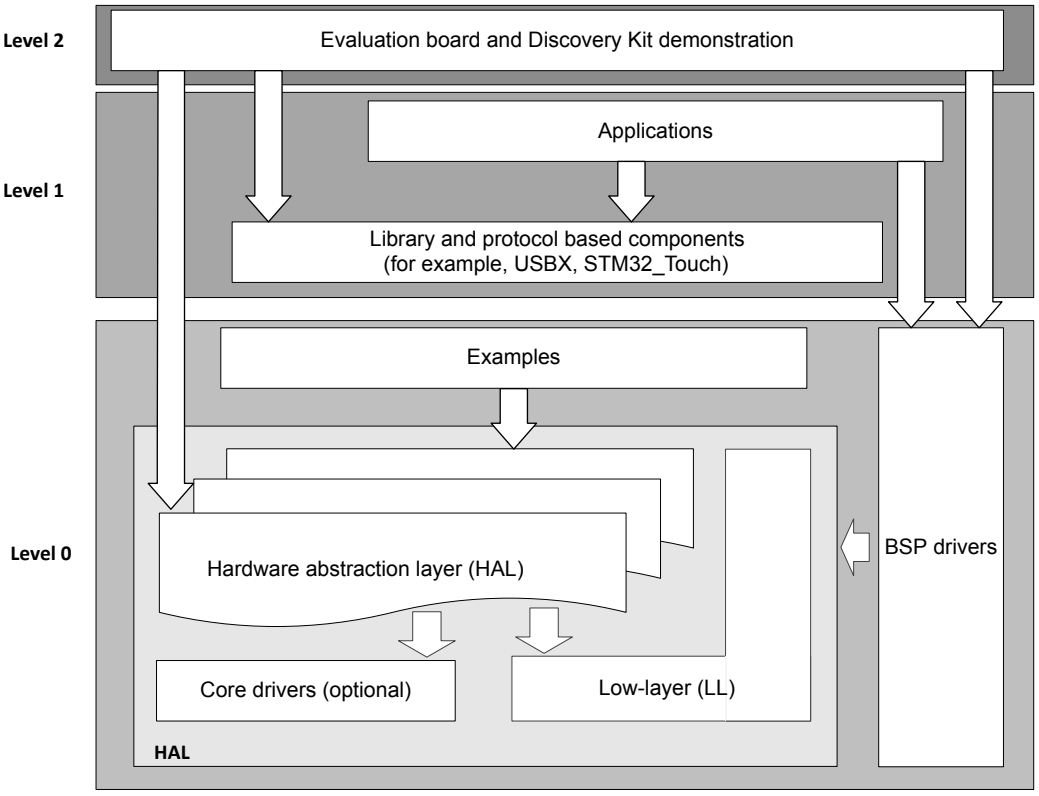
Figure 1. STM32CubeU3 MCU Package components



3 STM32CubeU3 architecture overview

The STM32CubeU3 MCU Package solution is built around three independent levels that easily interact, as described in Figure 2.

Figure 2. STM32CubeU3 MCU Package architecture



3.1 Level 0

This level is divided into three sublayers:

- 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. The hardware boards can be LCD, audio, microSD™, and MEMS drivers. 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 provides specific APIs to the BSP driver external components and could be portable on any other board.
- 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()`

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

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

The STM32CubeU3 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 that simplify user application implementation by providing ready to use processes. As an example, for the communication peripherals (I2S, 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 that provide common and generic functions to all the STM32 series
  - Extension APIs that 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, 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 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 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

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 sublayers:

- Middleware components
- Examples based on the middleware components

3.2.1 Middleware components

The middleware is a set of libraries constructed around the Microsoft®Azure® RTOS middleware and other in-house (STMTouch touch sensing, OpenBootloader) and open-source stacks (Mbed TLS). All are integrated and customized for STM32 MCU devices. They are also enriched with corresponding application examples based on STM32 evaluation boards. Horizontal interactions between the components of this layer are 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.

The main features of each middleware component are as follows:

- Azure® RTOS ThreadX
  - A real-time operating system (RTOS) that is designed for embedded systems with two functional modes.
  - Common mode includes common RTOS functionalities. These functionalities are thread management and synchronization, memory pool management, messaging, and event handling.
  - Module mode is an advanced usage mode. It enables loading and unloading of prelinked ThreadX modules on-the-fly through a module manager.
- NetX Duo
  - Industrial grade networking stack: optimized for performance coming with many IoT protocols.

- FileX/LevelX
  - Advanced flash memory file system (FS)/flash memory translation layer (FTL): fully featured to support NAND/NOR flash memories
- USBX
  - USB Host and Device, coming with many classes
- OpenBootloader
  - It provides an open-source bootloader with exactly the same features as the STM32 system bootloader and with the same tools used for the system bootloader.
- MCUboot
- Mbed TLS
  - 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 on a proven surface charge transfer acquisition principle.

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 Level 2

This level is composed of a single layer that consists of 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.

4 STM32CubeU3 MCU Package overview

4.1 Supported STM32CubeU3 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 what MCU is used. This improves the reusability of the library code and ensures easy portability to other devices.

In addition, owing to its layered architecture, STM32CubeU3 offers full support of all STM32U3 series. The user only has to define the right macro in `stm32u3xx.h`.

Table 1 shows which macro to define, depending on the STM32U3 series device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32U3 series

Macro defined in stm32u3xx.h	STM32U3 devices
STM32U375XX	STM32U375CET6, STM32U375CET6Q, STM32U375CEU6, STM32U375CEU6Q, STM32U375CEY6QTR, STM32U375CGT6, STM32U375CGT6Q, STM32U375CGU6, STM32U375CGU6Q, STM32U375CGY6QTR, STM32U375KEU6, STM32U375KGU6, STM32U375REI6, STM32U375REI6Q, STM32U375RET6, STM32U375RET6Q, STM32U375REY6GTR, STM32U375REY6QTR, STM32U375RGI6, STM32U375RGI6Q, STM32U375RGT6, STM32U375RGT6Q, STM32U375RGY6GTR, STM32U375RGY6QTR, STM32U375VEI6, STM32U375VEI6Q, STM32U375VET6, STM32U375VET6Q, STM32U375VGI6, STM32U375VGI6Q, STM32U375VGT6, STM32U375VGT6Q
STM32U385XX	STM32U385CGT6, STM32U385CGT6Q, STM32U385CGU6, STM32U385CGU6Q, STM32U385CGY6QTR, STM32U385KGU6, STM32U385RGI6, STM32U385RGI6Q, STM32U385RGT6, STM32U385RGT6Q, STM32U385RGY6GTR, STM32U385RGY6QTR, STM32U385VGI6, STM32U385VGI6Q, STM32U385VGT6, STM32U385VGT6Q

STM32CubeU3 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 STM32U3 series

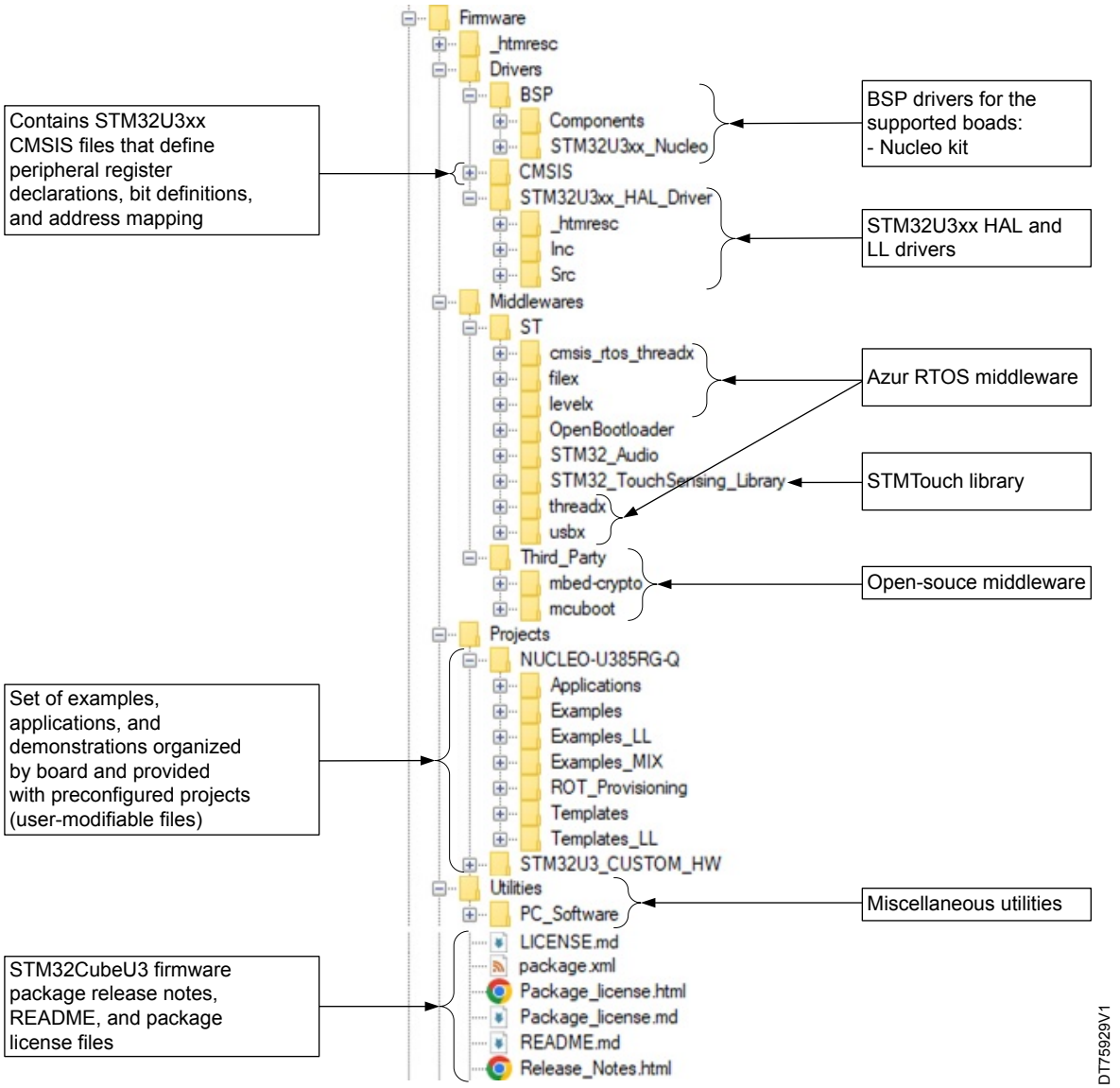
Board	Supported STM32U3 devices
NUCLEO-U385RG-Q	STM32U385RGT6Q

The STM32CubeU3 MCU Package can run on any compatible hardware. The user simply updates the BSP drivers to port the provided examples on their own board, if the latter has the same hardware features (such as LED, LCD display, or buttons).

4.2 MCU Package overview

The STM32CubeU3 MCU Package solution is provided in one single zip package having the structure shown in Figure 3.

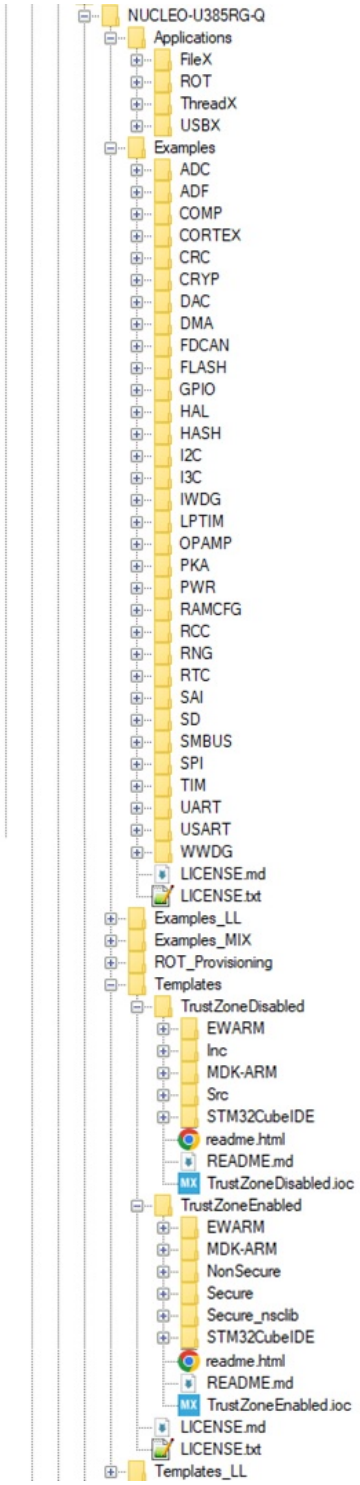
Figure 3. STM32CubeU3 MCU Package structure



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

Figure 4 shows the project structure for the NUCLEO-U385RG-Q board.

Figure 4. **STM32CubeU3 examples overview**



DT75930V1

The examples are classified depending on the STM32Cube level that 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 built quickly using the template projects available in the *Templates*, *Templates_Board*, and *Templates_LL* directories.

4.2.1 TrustZone®-enabled projects

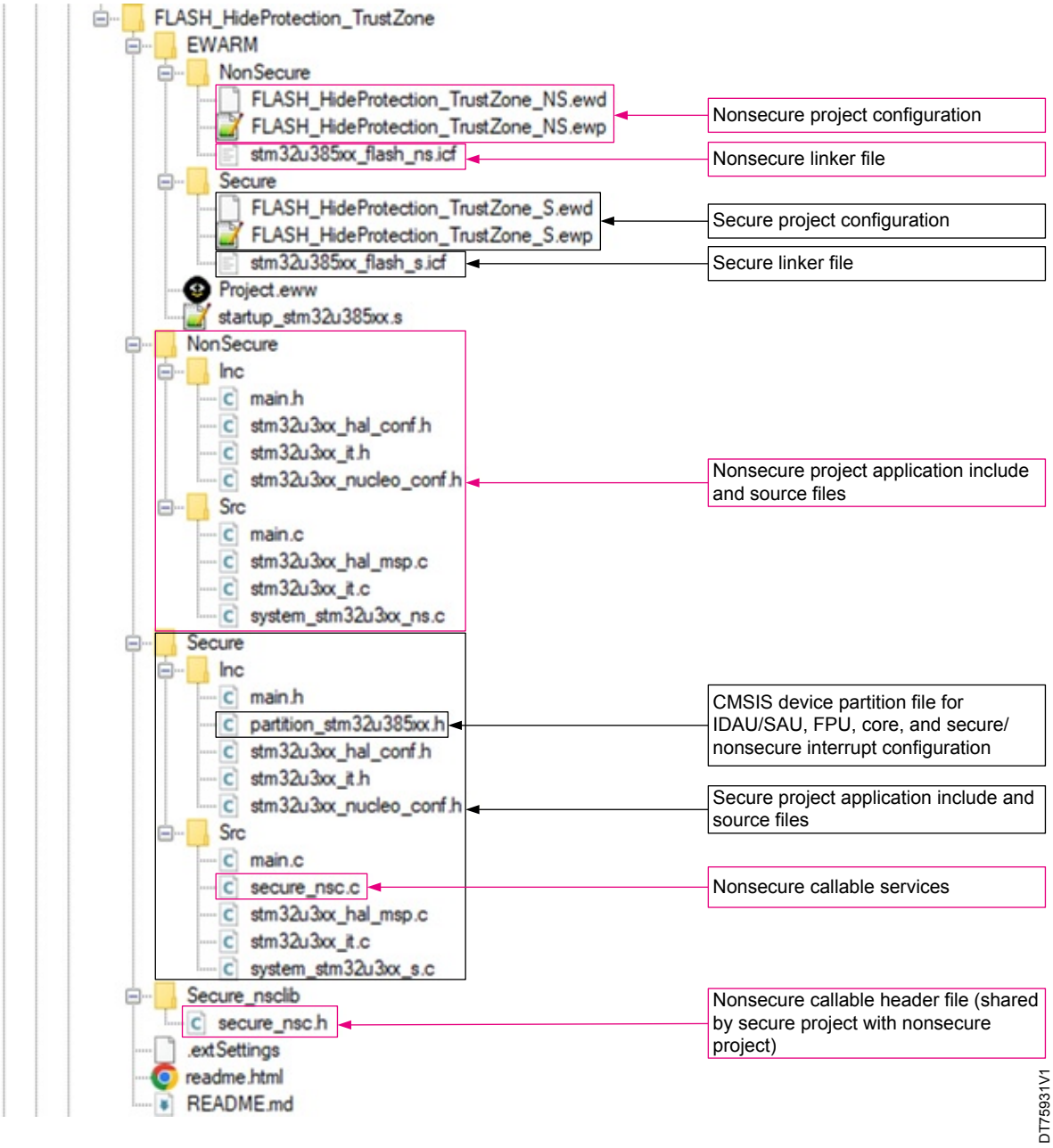
TrustZone® enabled `Examples` names contain the `_TrustZone` prefix. The rule is applied also for `Applications` (except for TFM and SBSFU, which are natively for TrustZone®).

TrustZone®-enabled `Examples` and `Applications` are provided with a multiproject structure composed of secure and nonsecure subprojects as presented in Figure 1.

TrustZone®-enabled projects are developed according to the CMSIS-5 device template, extended to include the system partitioning header file `partition_stm32u3xxxx.h`, who is mainly responsible for the setup of the secure attribute unit (SAU), the FPU, and the secure/nonsecure interrupts assignment in the secure execution state.

This setup is performed in the secure CMSIS `SystemInit()` function, which is called at startup before entering the secure application `main()` function. Refer to Arm® TrustZone®-M documentation of software guidelines.

Figure 5. Secure and nonsecure multiprojects structure



The STM32CubeU3 firmware package provides default memory partitioning in the `partition_stm32u3xxxx.h` files available in `\Drivers\CMSIS\Device\ST\STM32U3xx\Include\Templates`.

In these partition files, SAU is disabled by default. Consequently, IDAU memory mapping is used for security attribution (refer to figure 4 in the reference manual).

If the user enables SAU, a default SAU region configuration is predefined in the partition files for STM32U375xx and STM32U385xx devices:

- Region 0 : 0x0000 0000 - 0x080F FFFF (nonsecure external memories, flash memory)
- Region 1 : 0x0BF8 0000 - 0x0BFA FFFF (nonsecure system memory)
- Region 2 : 0x0C07 E000 - 0x0C07 FFFF (secure nonsecure callable)
- Region 3 : 0x1000 0000 - 0x17FF FFFF (nonsecure external memory)
- Region 4 : 0x2001 8000 - 0x2002 FFFF (nonsecure SRAM1, 96 Kbytes)
- Region 5 : 0x2003 8000 - 0x2003 FFFF (nonsecure SRAM2, 32 Kbytes)
- Region 6 : 0x4000 0000 - 0x4FFF FFFF (nonsecure peripheral mapped memory)
- Region 7 : 0x9000 0000 - 0x9FFF FFFF (nonsecure external memory)

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

All examples have the same structure:

- `\Inc` folder, containing all header files.
- `\Src` folder, containing the source code.
- `\EWARM`, `\MDK-ARM`, and `\STM32CubeIDE` folders, containing the preconfigured project for each toolchain.
- `README.md` and `readme.html` files, describing the example behavior and required environment to make it work.
- `.ioc` file, allowing users to open most of the firmware examples within STM32CubeMX.

5 Getting started with STM32CubeU3

5.1 Running a first example

This section explains how simple it is to run a first example on an STM32U3 series board. The program simply toggles a LED on the NUCLEO-U385RG-Q board:

Download the STM32CubeU3 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 is too long.

5.1.1 Running a first TrustZone®-enabled example

Before loading and running a TrustZone® enabled example, it is mandatory to read the example readme file for any specific configuration, which ensures that the security is enabled as described in Section 4.2.1: TrustZone®-enabled projects (TZEN=1 (user option byte)).

1. Browse to `\Projects\NUCLEO-U385RG-Q\Examples`.
2. Open `\GPIO`, then `\GPIO_IOToggle_TrustZone` folders.
3. 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.
4. Rebuild in sequence all secure and nonsecure project files and load the secure and nonsecure images into the target memory.
5. Run the example: LD2 remains ON for two seconds while in the nonsecure application, before it is handed over to the secure application, which toggles it every second. 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` subfolder.
  2. Launch the Project.eww workspace
  3. Rebuild the xxxxx_S secure project files: [**Project**]>[**Rebuild all**].
  4. Rebuild the xxxxx_NS nonsecure project files: [**Project**]>[**Rebuild all**].
  5. Program the secure and nonsecure binaries using [**Download and Debug**] (Ctrl+D).
  6. Run the program: [**Debug**]>[**Go(F5)**]
- MDK-ARM:
  1. Open the `\MDK-ARM` toolchain.
  2. Open the Multiprojects workspace file `Project.uvmpw`.
  3. Select the xxxxx_s project as Active application ([**Set as Active Project**]).
  4. Build the xxxxx_s project.
  5. Select the xxxxx_ns project as Active project ([**Set as Active Project**]).
  6. Build the xxxxx_ns project.
  7. Load the nonsecure binary ([**F8**]). This downloads `\MDK-ARM\xxxxx_ns\Exe\xxxxx_ns.axf` to flash memory)
  8. Select the Project_s project as Active project ([**Set as Active Project**]).
  9. Load the secure binary ([**F8**]). This downloads `\MDK-ARM\xxxxx_s\Exe\xxxxx_s.axf` to flash memory).
  10. Run the example.
- STM32CubeIDE:
  1. Open the STM32CubeIDE toolchain.
  2. Open the Multiprojects workspace file `.project`.
  3. Rebuild the xxxxx_Secure project.
  4. Rebuild the xxxxx_NonSecure project.
  5. Launch the [**Debug as STM32 Cortex-M C/C++**] application for the secure project.
  6. In the [**Edit configuration**] window, select the [**Startup**] panel, and load the image and symbols of the nonsecure project.

Important: The nonsecure project must be loaded before the secure project.

  7. Click [**Ok**].
  8. Run the example on debug perspective.

5.1.2 Running a first TrustZone®-disabled example

Before loading and running a TrustZone® disabled example, it is mandatory to read the example readme file for any specific configuration. If there are no specific mentions, ensure that the board device has security disabled (TZEN=0 (user option byte)). See FAQ for doing the optional regression to TZEN = 0.

1. Browse to `\Projects\NUCLEO-U385RG-Q\Examples`.
2. Open `\GPIO`, then `\GPIO_EXTI` folders.
3. 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.
4. Rebuild all files and load your image into the target memory.
5. Run the example: Each time the [**USER**] push-button is pressed, the LD1 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` subfolder.
  2. Launch the Project.eww workspace (the workspace name may change from one example to another).
  3. Rebuild all files: [**Project**]>[**Rebuild all**].
  4. Load the project image: [**Project**]>[**Debug**].
  5. Run program: [**Debug**]>[**Go (F5)**].
- MDK-ARM:
  1. Under the example folder, open the `\MDK-ARM` subfolder.
  2. Launch the Project.uvproj workspace (the workspace name may change from one example to another).
  3. Rebuild all files:[**Project**]>[**Rebuild all target files**].
  4. Load the 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 the program: [**Run**]>[**Debug (F11)**]

5.2 Developing a custom application

Note: The instruction cache (ICACHE) must be enabled by software to get a zero wait-state execution from flash memory and external memories. It also reaches the maximum performance and a better power consumption.

Note: The data cache (DCACHE) introduced on an S-AHB system bus of Cortex®-M33 processor to improve the performance of data traffic to/from external memories, must be enabled when using external memories.

5.2.1 Using STM32CubeMX to develop or update an application

In the STM32CubeU3 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 6.13.0 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.
- STM32CubeMX generates the initialization source code of such projects. 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 user manual *STM32CubeMX for STM32 configuration and initialization C code generation* (UM1718, available from st.com).

For a list of the available example projects for STM32CubeU3, refer to the STM32Cube firmware examples for the STM32U3 series application note.

5.2.2 HAL application

This section describes the steps required to create a custom HAL application using STM32CubeU3:

1. **Create a project**
   To create a new project, start either from the `Template` project provided for each board under `\Projects\<STM32U3xx_yyy>\Templates` or from any available project under `\Projects\<STM32U3xx_yyy>\Examples` or `\Projects\<STM32U3xx_yyy>\Applications` (where `<STM32U3xx_yyy>` refers to the board name, such as NUCLEO-U385RG-Q).
   The `Template` project provides an empty main loop function. However, it is a good starting point to understand the STM32CubeU3 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 STM32U3 series devices, allowing the CMSIS and HAL drivers to be configured correctly.
   – It provides ready-to-use user files that are preconfigured as follows:
     ◦ HAL initialized with the 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\STM32U3xx_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 `stm32u3xx_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 `stm32u3xx_hal_conf.h`).
   b. Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority `TICK_INT_PRIO` defined in `stm32u3xx_hal_conf.h`.
   c. Setting of NVIC group priority to 0.
   d. Call of `HAL_MspInit()` callback function defined in `stm32u3xx_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.

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 `stm32u3xx_it.c` to call the required interrupt handlers (peripheral and DMA), if needed.
   c. Write process complete callback functions, if a peripheral interrupt or DMA is planned to be used.
   d. In the user `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 the user 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 example set provided in the STM32CubeU3 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 a 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 STM32CubeU3.

1. **Create a project**
   To create a new project, either start from the Templates_LL project provided for each board under `\Projects\<STM32U3xx_yyy>\Templates_LL`, or from any available project under `\Projects\<STM32U3xx_yyy>\Examples_LL` (*<STM32xxx_yyy>* refers to the board name, such as NUCLEO-U385RG-Q).
   The template project provides an empty main loop function, which is a good starting point to understand the project settings for STM32CubeU3. 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 STM32U3 series device and allows the correct configuration of the CMSIS and LL drivers.
   – It provides ready-to-use user files that are preconfigured 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\<STM32U3xx_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 the 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 `stm32u3xx_it.h` file
     - Replace the `stm32u3xx_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 STM32CubeU3 release updates

The new STM32CubeU3 MCU Package releases and patches are available from www.st.com/stm32u3. 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).

6 FAQ

6.1 What is the license scheme for the STM32CubeU3 MCU Package?

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

The middleware stacks made by STMicroelectronics (for example, the USBPD 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 (TFM) have user-friendly license terms. For more details, refer to the appropriate middleware license agreement.

6.2 What boards are supported by the STM32CubeU3 MCU Package?

The STM32CubeU3 MCU Package provides BSP drivers and ready-to-use examples for the following STM32U3 series boards:

- NUCLEO-U385RG-Q

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

Yes. STM32CubeU3 provides a rich set of examples and applications. They come with the preconfigured projects for IAR Embedded Workbench®, Keil®, and STM32CubeIDE.

6.4 How to enable TrustZone® on STM32U3 series devices?

All STM32U3 series devices support TrustZone®. The factory default state is TrustZone® disabled. The TrustZone® security is activated with the TZEN option bit in the FLASH_OPTR register. The configuration of the user option bytes can be done using STM32CubeProgrammer (STM32CubeProg).

6.5 How to disable TrustZone® on STM32U3 series devices?

The following sequence is needed to disable TrustZone®:

- Boot from user flash memory:
  1. Make sure that secure and nonsecure applications are well loaded and executed (jump done on nonsecure application).
  2. Set RDP to level 1 through STM32CubeProgrammer if not yet done. Then only hotplug connection is possible during nonsecure application execution.
  3. Use a power supply different than STLINK in order to be able to connect to the target.
  4. Uncheck the "TZEN" box and set RDP to level 0 (option byte value 0xAA), then click on "Apply".
- Boot from RSS:
  1. Make sure to apply a high level on the BOOT0 pin (make sure that "nSWBOOT0 option byte" is checked).
  2. Set RDP to level 1 through STM32CubeProgrammer if not yet done. Then only hotplug connection is possible during nonsecure application execution.
  3. Use a power supply different than STLINK in order to be able to connect to the target.
  4. Uncheck the "TZEN" box and set RDP to level 0 (option byte value 0xAA), then click on "Apply".

6.6 How to update the secure/nonsecure memory mapping

In case of memory isolation for secure and nonsecure applications, the secure and nonsecure applications share the same internal flash memory and embedded SRAMs.

The STM32CubeU3 firmware package provides default memory partitioning in the `partition_stm32u3xxxx.h` files available under: `\Drivers\CMSIS\Device\ST\STM32U3xx\Include\Templates` (refer to Section 4.2.1: TrustZone®-enabled projects).

Any memory map partitioning change between secure and nonsecure applications requires the following updates and alignments (without overlap between secure and nonsecure memory space and using secure and nonsecure memory address aliases):

- If SAU is enabled, nonsecure area update (internal flash memory and SRAMs) (see `partition_stm32u3xx.h` file).
- Secure and nonsecure linker files update to correctly locate the secure and nonsecure code and data.
- Update the nonsecure address to jump to (in secure `main.c` and nonsecure reset handler in nonsecure linker file).
- Update the nonsecure boot base address option bytes (NSBOOTADD0) with the nonsecure address to jump to (using STM32CubeProgrammer).
- Update the flash memory watermark option bytes (SEC_WMx_PSTRT/SEC_WMx_PEND) to define the secure/nonsecure flash memory areas (using STM32CubeProgrammer).

6.7 How to set up interrupts for secure and nonsecure applications

At MCU core level, all interrupts are set to secure at system reset. This default state is visible in the `partition_stm32u3xxxx.h` files available under: `\Drivers\CMSIS\Device\ST\STM32U3xx\Include\Templates` (refer to Section 4.2.1: TrustZone®-enabled projects).

One of these template files is intended to be copied in the secure application for the selected device. This is to set the interrupt line targets by modifying the `partition_stm32u3xxxx.h` file, either to target the secure (default) or nonsecure application vector table. This insures that interrupts are set up when the application enters the secure `main()`.

6.8 Why does the system 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 nonsecure application to secure address.

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

6.9 Are there any links with standard peripheral libraries?

The STM32CubeU3 HAL and LL drivers are the replacement of the standard peripheral library (SPL):

- 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 the hardware. a set of user-friendly APIs allows a higher abstraction level that 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 allow an easier migration from the SPL to the STM32CubeU3 LL drivers, since each SPL API has its equivalent LL API.

6.10 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).

6.11 How are the product/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 available on some products/lines only.

6.12 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.

6.13 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 *stm32u3xx_ll_ppp.h* file(s).

6.14 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. The mixing of HAL and LL is illustrated in the "Examples_MIX" example.

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

Yes, there are. A few Cortex® APIs have been added in *stm32u3xx_ll_cortex.h*, for instance for accessing SCB or SysTick registers.

6.16 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.

6.17 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.

6.18 How can STM32CubeMX generate code based on embedded software?

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

6.19 How to get regular updates on the latest STM32CubeU3 MCU Package releases?

Refer to Section 5.3.

Revision history

Table 3. **Document revision history**

Date	Revision	Changes
24-Oct-2024	1	Initial release.

Contents

List of tables

List of figures