Getting started with STM32CubeU5 for STM32U5 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 STM32CubeU5 for the STM32U5 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, USB-PD, touch library, network library, mbed-crypto, TFM, 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 STM32CubeU5 MCU Package.

STM32CubeU5 main features describes the main features of the STM32CubeU5 MCU package. STM32CubeU5 architecture overview and STM32CubeU5 MCU package overview provide an overview of the STM32CubeU5 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 STM32CubeU5 main features

The STM32CubeU5 MCU package runs on STM32 32-bit microcontrollers based on the Arm® Cortex®-M33 processor with Arm® TrustZone® and FPU.

The STM32CubeU5 gathers, in a single package, all the generic embedded software components required to develop an application for the STM32U5 series microcontrollers. In line with the STM32Cube initiative, this set of components is highly portable, not only within the STM32U5 series microcontrollers but also to other STM32 series.

The STM32CubeU5 is fully compatible with the 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.

The STM32CubeU5 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.

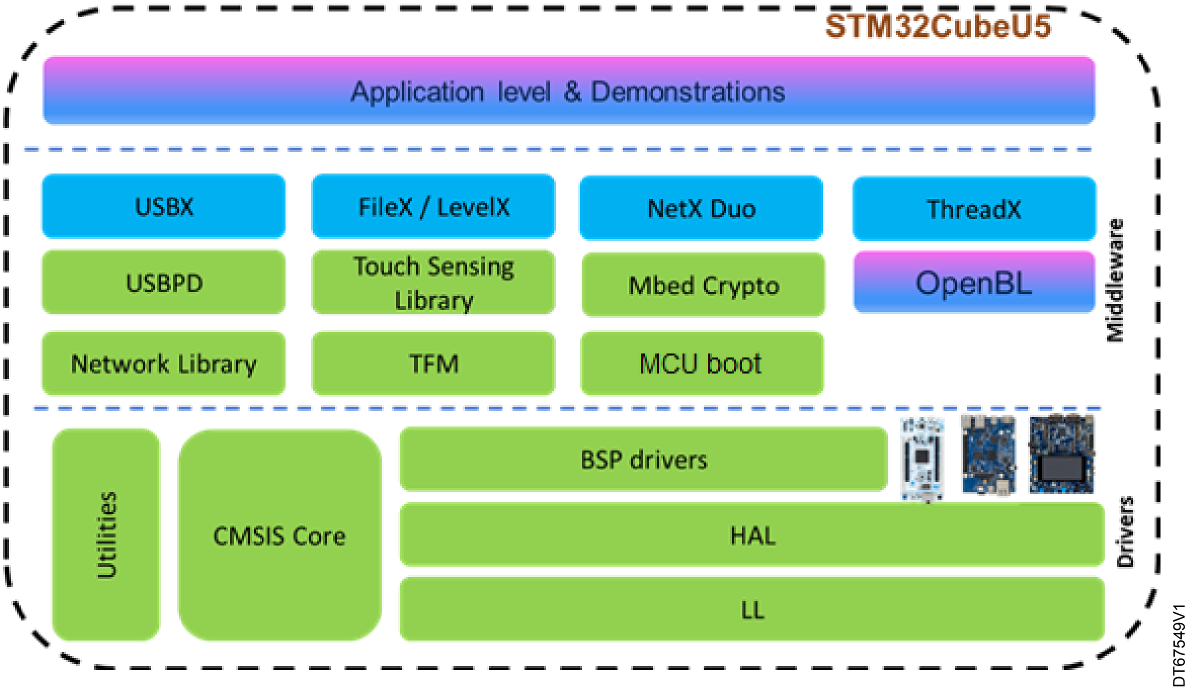
They come with free user-friendly license terms:

- Integrated and full featured RTOS: ThreadX
- CMSIS-RTOS implementation with ThreadX
- USB host and device stacks coming with many classes: USBX
- Advanced file system and flash memory translation layer: FileX / LevelX
- Industrial grade networking stack: optimized for performance coming with many IoT protocols: NetX Duo
- USB PD library
- Open bootloader
- Arm trusted Firmware-M (TF-M) integration solution
- MCU boot
- Mbed Crypto libraries
- STMicroelectronics network library
- STMTouch touch sensing library solution.

Several applications and demonstrations implementing all these middleware components are also provided in the STM32CubeU5 MCU package.

The STM32CubeU5 MCU package component layout is illustrated in Figure 1.

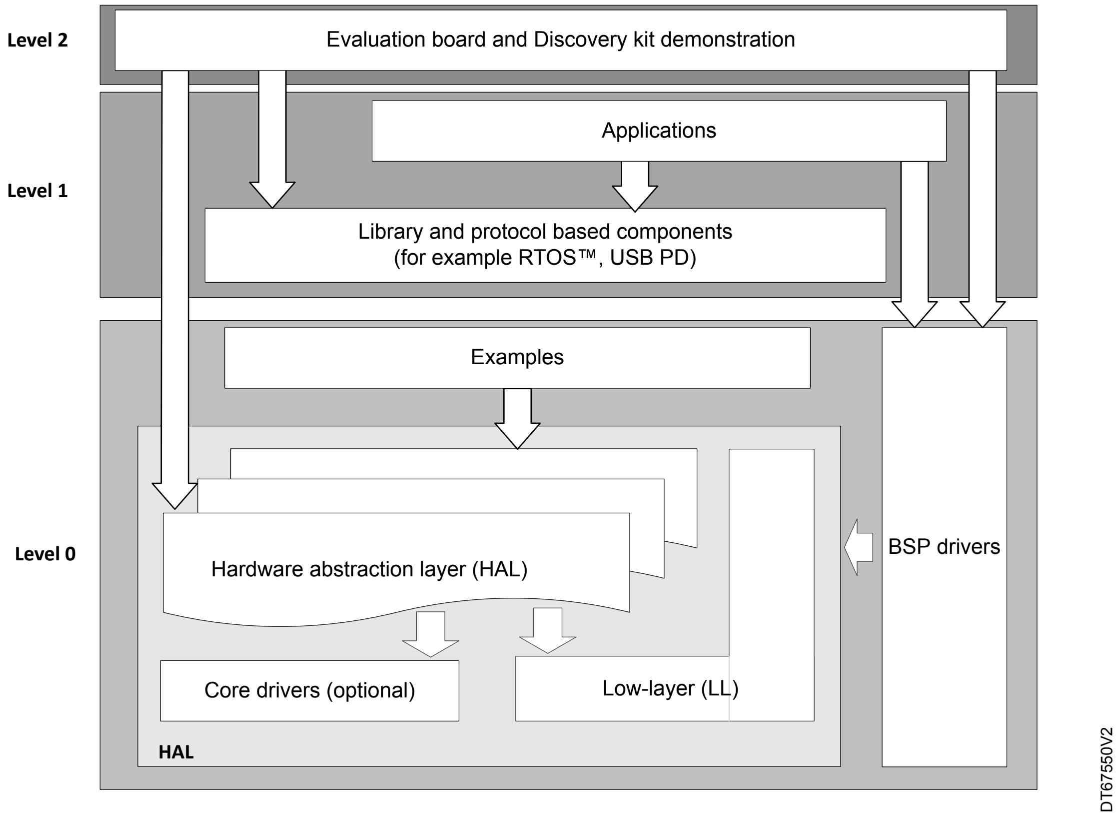
Figure 1. STM32CubeU5 MCU package components



3 STM32CubeU5 architecture overview

The STM32CubeU5 MCU package solution is built around three independent levels that easily interact as described in Figure 2.

Figure 2. STM32CubeU5 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 STM32CubeU5 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, OpenBL) and open source (TF-M, Mbed™ crypto). 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:

- 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 stacks coming with many classes
- USB PD device and Core libraries
  – New USB Type-C® power delivery service. Implementing a dedicated protocol for the management of power in this evolution of the USB.org specification. Refer to http://www.usb.org/developers/powerdelivery/ for more details.
  – 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 the low-level driver through an abstraction layer using the configuration file. The goal is to avoid any dependency between the library and the low-level drivers
- Open bootloader
  – 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.
- 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

Note: An example of the TF-M application is available in the STM32CubeU5 firmware package under \Projects\B-U585I-IOT02A\Applications\TFM.

- MCU boot
- 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).
- Network Library
  – It provides network services on STM32 devices.
  – It provides a socket API (BSD like style) with support of secure or nonsecure connection. It provides also an API to control the lifecycle of the network adapters.
  – Three classes of network adapters are supported WIFI, Ethernet, and cellular. Different WIFI modules are supported from third party vendors.
- The 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 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.

3.4 Utilities

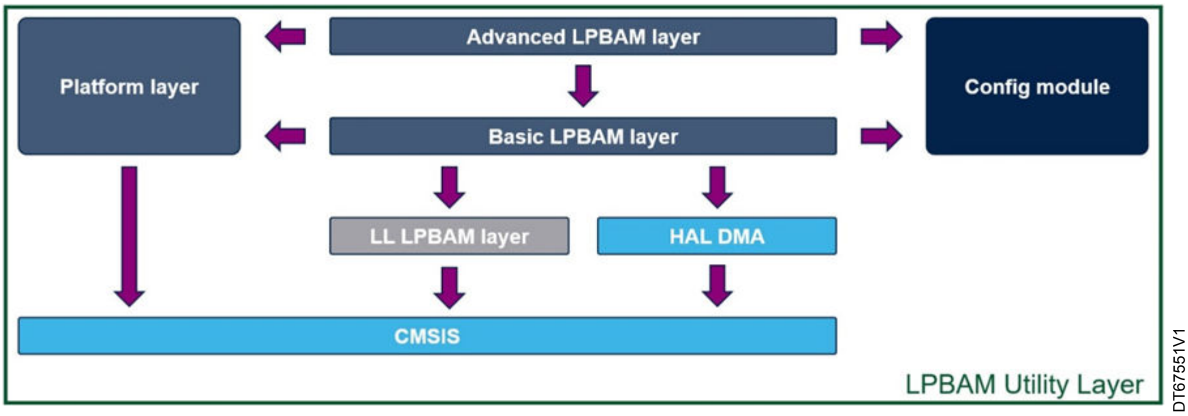
Alike all STM32Cube firmware packages, the STM32CubeU5 provides a set of utilities that offer miscellaneous software and additional system resources services. They can be used either by the application, or the different STM32Cube firmware intrinsic middleware and components.

3.4.1 LPBAM utility

The STM32U5 series embeds the **L**ow **P**ower **B**ackground **A**utonomous **M**ode subsystem, which is s an operating mode that allows peripherals to be functional and autonomous independently from power modes and without any software running. It can chain diverse operations thanks to DMA linked-list transfers.

The LPBAM utility is a set of modular drivers located under the utilities folder in the STM32Cube firmware package. It provides two user levels of abstraction: Basic and advanced levels.

Figure 3. **LPBAM utility layers**



The LPBAM utility is based on layers:

- The first level of abstraction is a hardware agnostic layer named basic LPBAM layer. It gives to the users a full granularity of scenarios creation via transversal APIs.
- The second level of abstraction is a hardware agnostic layer named advanced LPBAM layer. It provides a predefined scenario (set of elementary nodes) that can be customized and concatenated to build an end user application.
- The platform layer contains device specific constants to be used by the applicative side.
- The LL LPBAM layer (low level) is used by the basic and the advanced LPBAM layers. It contains the device specific mechanisms for each supported peripheral.

4 STM32CubeU5 MCU package overview

4.1 Supported STM32U5 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, STM32CubeU5 offers full support of all STM32U5 series. The user only has to define the right macro in *stm32u5xx.h*.

Table 1 shows which macro to define, depending on the STM32U5 series device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32U5 series

Macro defined in stm32u5xx.h	STM32U5 devices
STM32U575xx	STM32U575ZIT6, STM32U575QII6, STM32U575AII6, STM32U575CIU6Q, STM32U575CIT6Q, STM32U575OIY6Q, STM32U575VIT6Q, STM32U575QII6Q, STM32U575ZIT6Q, STM32U575RIT6Q, STM32U575CGU6, STM32U575CGT6, STM32U575RGT6, STM32U575VGT6, STM32U575ZGT6, STM32U575QGI6, STM32U575AGI6, STM32U575CGU6Q, STM32U575CGT6Q, STM32U575OGY6Q, STM32U575VGT6Q, STM32U575QGI6Q, STM32U575ZGT6Q, STM32U575RGT6Q, STM32U575AGI6Q
STM32U585xx	STM32U585CIU6, STM32U585CIT6, STM32U585RIT6, STM32U585VIT6, STM32U585AII6, STM32U585QII6, STM32U585ZIT6, STM32U585OIY6Q, STM32U585VIT6Q, STM32U585QEI6Q, STM32U585RIT6Q, STM32U585AII6Q, STM32U585CIU6Q, STM32U585CIT6Q, STM32U585ZET6Q
STM32U535xx	STM32U535CET6, STM32U535CEU6, STM32U535RET6, STM32U535REI6, STM32U535VET6, STM32U535VEI6, STM32U535CET6Q, STM32U535CEU6Q, STM32U535RET6Q, STM32U535REI6Q, STM32U535VET6Q, STM32U535VEI6Q, STM32U535NEY6Q, STM32U535JEY6Q
STM32U545xx	STM32U545CET6, STM32U545CEU6, STM32U545RET6, STM32U545REI6, STM32U545VET6, STM32U545VEI6, STM32U545CET6Q, STM32U545CEU6Q, STM32U545RET6Q, STM32U545REI6Q, STM32U545VET6Q, STM32U545VEI6Q, STM32U545NEY6Q, STM32U545JEY6Q
STM32U595xx	STM32U595AJH6, STM32U595ZJT6, STM32U595QJI6, STM32U595VJT6, STM32U595RJT6, STM32U595AJH6Q, STM32U595ZJY6QTR, STM32U595ZJT6Q, STM32U595QJI6Q, STM32U595VJT6Q, STM32U595RJT6Q, STM32U595AIH6, STM32U595ZIT6, STM32U595QII6, STM32U595VIT6, STM32U595RIT6, STM32U595AIH6Q, STM32U595ZIY6QTR, STM32U595ZIT6Q, STM32U595QII6Q, STM32U595VIT6Q, STM32U595RIT6Q
STM32U599xx	STM32U599VJT6, STM32U599NJH6Q, STM32U599BJY6QTR, STM32U599ZJY6QTR, STM32U599ZJT6Q, STM32U599VJT6Q, STM32U599NIH6Q, STM32U599ZIY6QTR, STM32U599ZIT6Q, STM32U599VIT6Q
STM32U5A5xx	STM32U5A5AJH6, STM32U5A5ZJT6, STM32U5A5QJI6, STM32U5A5VJT6, STM32U5A5RJT6, STM32U5A5AJH6Q, STM32U5A5ZJY6QTR, STM32U5A5ZJT6Q, STM32U5A5QJI6Q, STM32U5A5VJT6Q, STM32U5A5RJT6Q, STM32U5A5QII3Q
STM32U5A9xx	STM32U5A9NJH6Q, STM32U5A9BJY6QTR, STM32U5A9ZJY6QTR, STM32U5A9ZJT6Q, STM32U5A9VJT6Q
STM32U5F7xx	STM32U5F7VJT6Q, STM32U5F7VJT6, STM32U5F7VIT6Q, STM32U5F7VIT6
STM32U5G7xx	STM32U5G7VJT6Q, STM32U5G7VJT6
STM32U5F9xx	STM32U5F9NJH6Q, STM32U5F9BJY6QTR, STM32U5F9ZJJ6QTR, STM32U5F9ZJT6Q, STM32U5F9VJT6Q, STM32U5F9ZIJ6QTR, STM32U5F9ZIT6Q, STM32U5F9VIT6Q
STM32U5G9xx	STM32U5G9NJH6Q, STM32U5G9BJY6QTR, STM32U5G9ZJJ6QTR, STM32U5G9ZJT6Q, STM32U5G9VJT6Q

STM32CubeU5 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 STM32U5 series

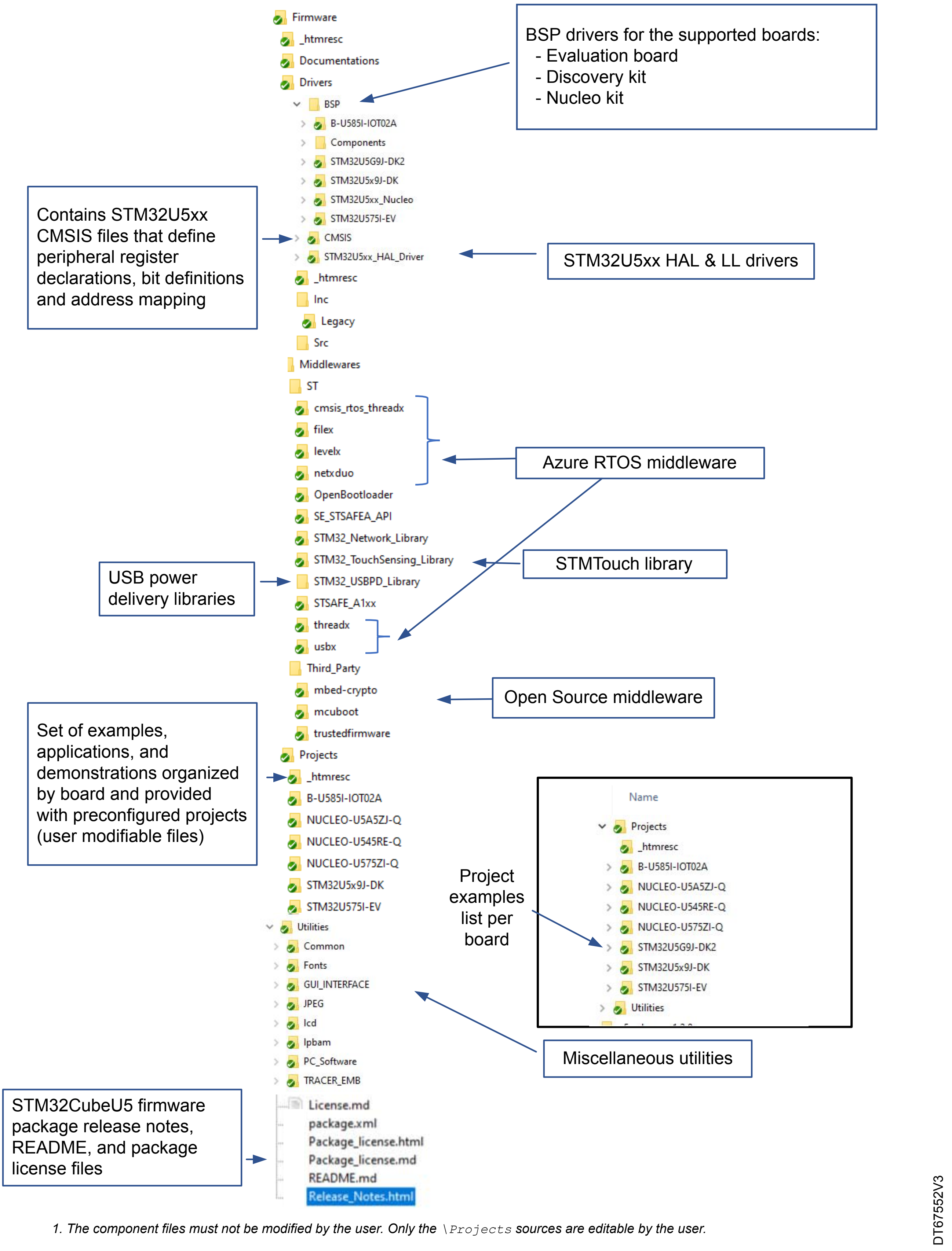
Board	Board STM32U5 supported devices
NUCLEO-U575ZI-Q	STM32U575ZIT6Q
STM32U575I-EV	STM32U575AII6Q
B-U585I-IOT02A	STM32U585AII6Q
NUCLEO-U5A5ZJ-Q	STM32U5A5ZJT6Q
STM32U5A9J-DK	STM32U5A9NJH6Q
NUCLEO-U545RE-Q	STM32U545RET6Q
STM32U5G9J-DK2	STM32U5G9ZJT6Q
STM32U5G9J-DK1	STM32U5G9NJH6Q

The STM32CubeU5 MCU package is able to 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, buttons).

4.2 MCU package overview

The STM32CubeU5 MCU package solution is provided in one single zip package having the structure shown in. Figure 4.

Figure 4. STM32CubeU5 MCU package structure

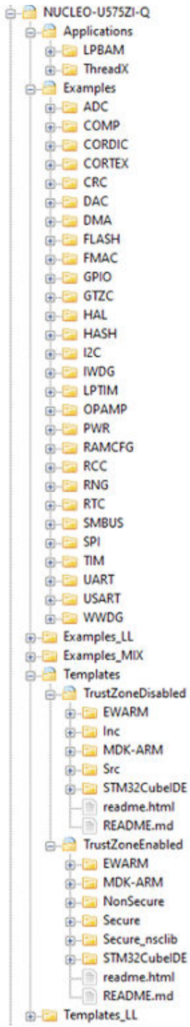


1. *The component files must not be modified by the user. Only the* `\Projects` *sources are editable by the user.*

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

Figure 5 shows the project structure for the NUCLEO-U575ZI-Q board.

Figure 5. STM32CubeU5 examples overview



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* and *Templates_LL* directories.

4.2.1 TrustZone-enabled projects

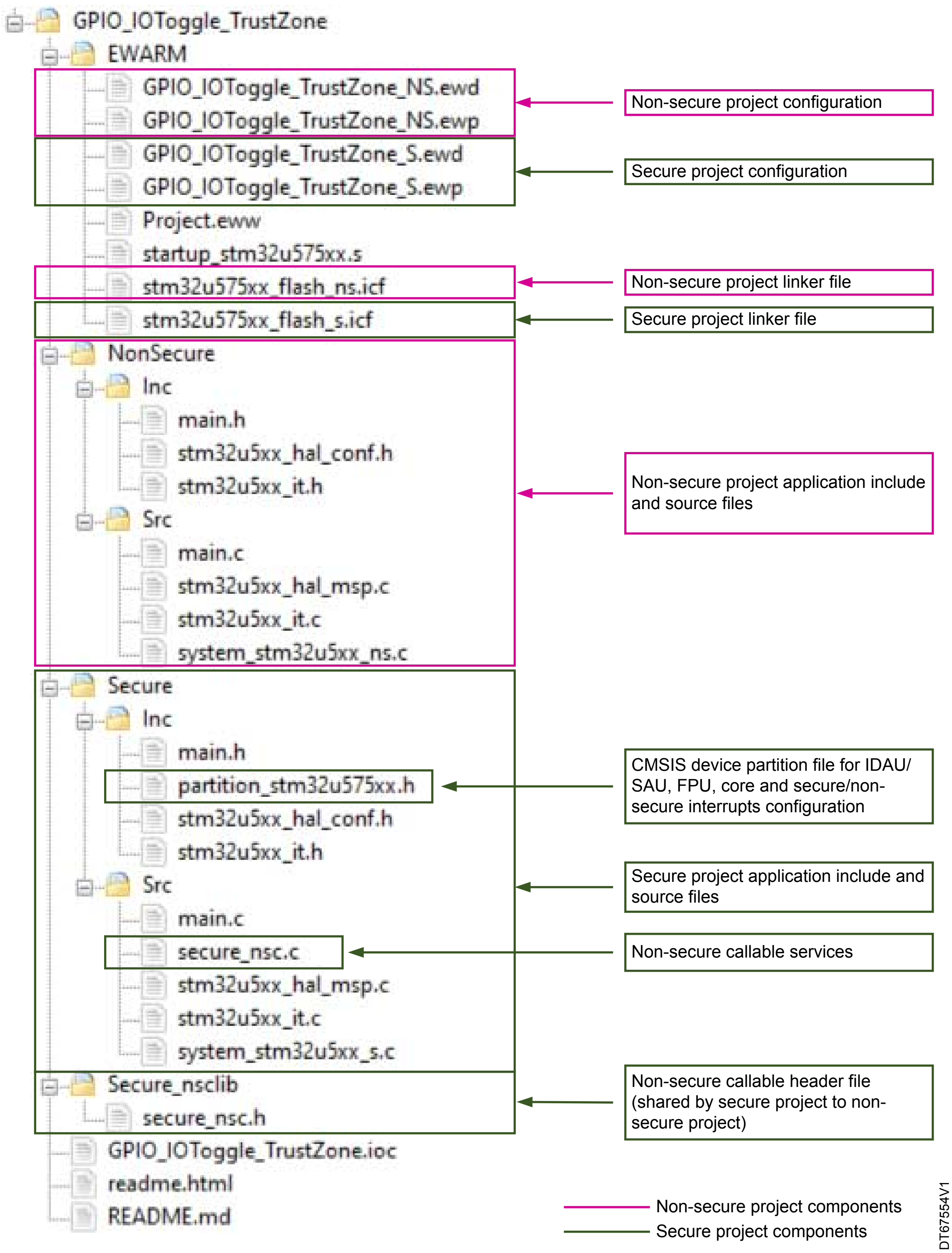
TrustZone-enabled “Examples” names are prefixed with "_TrustZone". This rule is also applied to “Applications” (except TFM, which is native to TrustZone®).

TrustZone-enabled “Examples” and “Applications” are provided with a multiproject structure, composed of secure and nonsecure subprojects, as presented below in Figure 6.

TrustZone-enabled projects are developed according to the CMSIS-5 device template, extended to include the system partitioning header file *partition_<device>.h*. This file is responsible for the principal setup of the secure attribute unit (SAU), the FPU, and the secure/nonsecure interrupts assignment in a secure execution state.

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

Figure 6. Secure and nonsecure multiprojects structure



The STM32CubeU5 firmware package provides default memory partitioning in the `partition_<device>.h` files available in `\Drivers\CMSIS\Device\ST\STM32U5xx\Include\Templates`.

In these partition files, SAU is disabled by default. Consequently, IDAU memory mapping is used for security attribution (refer to figure 3 in the reference manual).

If the user enables SAU, a default SAU region configuration is predefined in the partition files for STM32U575xx and STM32U585xx devices:

- SAU region 0: 0x0C0FE000 - 0x0C0FFFFF (secure, nonsecure callable)
- SAU region 1: 0x08100000 - 0x081FFFFF (nonsecure flash memory Bank2 (1024 Kbytes))
- SAU region 2: 0x20040000 - 0x200BFFFF (nonsecure SRAM3 (512 Kbytes)))
- SAU region 3: 0x40000000-0x4FFFFFFF (nonsecure peripheral mapped memory)
- SAU region 4: 0x60000000-0x9FFFFFFF (nonsecure external memories)
- SAU region 5: 0x0BF90000-0x0BFA8FFF (nonsecure system memory)

To match the default partitioning, for STM32U575xx and STM32U585xx devices, the settings are:

- TZEN=1 (TrustZone-enabled device)
- SECWM1_PSTRT=0x0 SECWM1_PEND=0x7F, meaning all 128 pages of flash memory Bank1 set as secure
- SECWM2_PSTRT=0x1 SECWM2_PEND=0x0, meaning no page of flash memory Bank2 set as secure, hence Bank2 is nonsecure

For STM32U535xx and STM32U545xx devices, the settings are:

- TZEN=1
- SECWM1_PSTRT=0x0 SECWM1_PEND=0x1F, meaning all 32 pages of Bank1 set as secure
- SECWM2_PSTRT=0x1 SECWM2_PEND=0x0, meaning no page of Bank2 set as secure (no page of flash memory Bank2 set as secure, hence Bank2 is nonsecure

For STM32U5FXxx, STM32U5GXxx, STM32U5AXxx and STM32U59Xxx devices, the settings are:

- TZEN=1
- SECWM1_PSTRT=0x0 SECWM1_PEND=0xFF, meaning all pages of Bank1 set as secure
- SECWM2_PSTRT=0xFF SECWM2_PEND=0x0, meaning no page of Bank2 set as secure, hence Bank2 is nonsecure

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.

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

Table 3. Number of examples for each board

Level	STM32U5G9J-DK2	B-U585I-IOT02A	NUCLEO-U545RE-Q	NUCLEO-U575ZI-Q	NUCLEO-U5A5ZI-Q	STM32U575I-EV	STM32U5x9J-DK	Total
Applications	6	21	7	18	6	12	6	76
Demonstrations	0	1	0	0	0	1	0	2
Examples	11	23	15	88	4	27	11	179
Examples_LL	0	0	0	41	0	0	0	41
Examples_MIX	0	0	0	5	0	0	0	5
Templates	2	2	2	2	2	2	2	14
Templates_LL	1	1	1	1	1	1	1	7
Total projects	20	48	25	155	13	43	20	324

5 Getting started with STM32CubeU5

5.1 Running a first example

This section explains how simple it is to run a first example on an STM32U5 series board. The program simply toggles an LED on the NUCLEO-U575ZI-Q board:

Download the STM32CubeU5 MCU package. Unzip it into an appropriate directory. Make sure the package structure shown in Figure 4 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.

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. It insures that the security is enabled as described in section 4.2.1 (TZEN=1 (user option byte)).

1. Browse to *\Projects\NUCLEO-U575ZI-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 nonsecure project files and load the secure and nonsecure images into target memory.
5. Run the example: on a regular basis, the secure application toggles LED1 every second and nonsecure 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
    1. Open *\EWARM* subfolder under the example folder.
    2. Launch the Project.eww workspace.
    3. Set the "xxxxx_S" as an active application (right click on xxxxx_S project **set as active**).
    4. Rebuild the xxxxx_S secure project files: **Project** → **Rebuild all**.
    5. Rebuild the xxxxx_NS nonsecure project files: Right click on xxxxx_NS **project** → **Rebuild all**.
    6. Flash the secure and nonsecure binaries with the **download and debug button (Ctrl+D)**.
    7. Run program: **Debug** → **go(F5)**.

- MDK-ARM
  1. Open the MDK-ARM toolchain.
  2. Open the multiproject workspace file *Project.uvmpw*.
  3. Select the xxxxx_s project as active project (**set as active project**).
  4. Build xxxxx_s project.
  5. Select the xxxxx_ns project as active project (**set as active project**).
  6. Build xxxxx_ns project.
  7. Load the nonsecure binary (**F8**).
     (this downloads the *\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 the *\MDK-ARM\xxxxx_s\Exe\xxxxx_s.axf* to flash memory).
  10. Run the example.

  STM32CubeIDE
  1. Open the STM32CubeIDE toolchain.
  2. Open multiprojects workspace file *.project*.
  3. Rebuild xxxxx_Secure project.
  4. Rebuild xxxxx_NonSecure project.
  5. Launch **debug as STM32 Cortex-M C/C++ application** for the secure project.
  6. Select the **startup** panel in the **edit configuration** window, and load the image and symbols of the nonsecure project.
  7. Load the nonsecure project before loading the secure project.
  8. Click "OK".
  9. 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 or if nothing is mentioned, ensure 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-U575ZI-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
  1. Open *\EWARM* subfolder under the example folder.
  2. Launch the *Project.eww* workspace.

Note: The workspace name may change from one example to another.

  3. Rebuild all files: **Project → Rebuild all**.
  4. Load project image: **Project → Debug**.
  5. Run program: **Debug → go(F5)**.
- MDK-ARM
  1. Open the *\MDK-ARM* subfolder under the example folder.
  2. Launch the *Project.uvprojx* workspace.

Note: The workspace name may change from one example to another.

  3. Rebuild all files: **Project → Rebuild all target files**.
  4. Load project image: **Debug → Start/Stop debug session**.
  5. Run program: **Debug → run (F5)**.

- STM32CubeIDE
  1. Open the STM32CubeIDE toolchain
  2. Click **file** → **switch workspace** → **other** and browse to the STM32CubeIDE workspace directory
  3. Click **file** → **import**, select **General** → **existing projects into workspace** and then click **next**
  4. Browse to the STM32CubeIDE workspace directory and select the project
  5. Rebuild all project files: select the project in the **project explorer** window then click the **project** → **build project** menu
  6. Run program: **Run** → **Debug (F11)**

5.2 Developing a custom application

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 STM32CubeU5 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 6.3.0 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 STM32CubeU5, refer to the STM32Cube firmware examples for the STM32U5 series application note.

5.2.2 HAL application

This section describes the steps required to create a custom HAL application using STM32CubeU5:

1. Create a project
   To create a new project, start either from the template project, provided for each board in \Projects\<STM32xxx_yyy>\Templates, or from any available project in \Projects\<STM32xxy_yyy>\Examples or \Projects\<STM32xx_yyy>\Applications (where <STM32xxx_yyy> refers to the board name, such as STM32CubeU5).
   The template project provides only an empty main loop function, which is a good starting point to understand the STM32CubeU5 project settings. The template has the following characteristics:
   – It contains the HAL source code, CMSIS, and BSP drivers that are the minimum set of components required to develop code on a given board.
   – It contains the include paths for all the firmware components.
   – It defines the supported STM32U5 series devices, allowing the correct configuration of the CMSIS and HAL drivers.
   – It provides ready-to-use user files that are preconfigured 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 updated.

2. **Add the necessary middleware to the project (optional)**
   To identify the source files to be added to the project file list, refer to the documentation provided for each middleware component. Refer to the applications in \Projects\STM32xxx_yyy\Applications\<MW_Stack> (where <MW_Stack> refers to the middleware stack, such as USBX) 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 that has to be copied to the project folder (usually the configuration file is named xxx_conf_template.h, and 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 memory prefetch and SysTick interrupt priority (through macros defined in stm32u5xx_hal_conf.h).
   b. Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority TICK_INT_PRIORITY, defined in stm32u5xx_hal_conf.h, which is clocked by the MSI (at this stage, the clock has not been configured yet and the system is running from the 4 MHz MSI).
   c. Setting the NVIC group priority to 0.
   d. Calling the HAL_MspInit() callback function defined in the stm32u5xx_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/or external oscillators, as well as the PLL source and factors. The user chooses to configure one or all oscillators. They can skip the PLL configuration if there is no need to run the system at high frequency.
   – `HAL_RCC_ClockConfig()`: this API configures the system clock source, the flash memory latency, the AHB prescalers, and the APB prescalers.

   Initialize the peripheral
   a. Write the peripheral `HAL_PPP_MspInit` function. Proceed as follows:
      i. Enable the peripheral clock.
      ii. Configure the peripheral GPIOs.
      iii. Configure the DMA channel and enable DMA interrupt (if needed).
      iv. Enable peripheral interrupt (if needed).
   b. Edit `stm32xxx_it.c` to call the required interrupt handlers (peripheral and DMA), if needed.
   c. Write the process complete callback functions if peripheral interrupt or DMA is going to be used.
   d. In `main.c`, initialize the peripheral handle structure, then call the `HAL_PPP_Init()` function to initialize the peripheral.
6. **Develop an application**
   At this stage, the system is ready and the user application code development can start.
   a. The HAL provides intuitive and ready-to-use APIs to configure the peripheral. It supports polling, interrupts, and a DMA programming model, to accommodate any application requirements. For more details on how to use each peripheral, refer to the rich example set provided in the STM32CubeU5 MCU package.

Caution: In the default HAL implementation, a 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 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 an override possible in case of other implementations in the user file (using a general purpose timer or other 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 STM32CubeU5.

1. **Create a project**
   To create a new project, either start from the Templates_LL project provided for each board in `\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-U575ZI-Q).
   The template project provides an empty main loop function, which is a good starting point to understand the STM32CubeU5 project settings. The template has the following characteristics:
   – 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 include paths for all the required firmware components.
   – It selects the supported STM32U5 device and allows the correct configuration of the CMSIS and LL drivers.
   – It provides ready-to-use user files, which are preconfigured 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**
   a. Start from the Templates_LL project provided for each board, available in `\Projects\<STM32xxx_yyy>\Templates_LL`.
   b. Select an LL example.

Note: 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 an existing `Templates_LL` project.
   – Replace `Templates_LL` files with the `Examples_LL` targeted project.
   – Keep all board-specific parts. For clarity reasons, board-specific parts have been flagged with the following specific tags:

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

   The main porting steps are the following:
   a. Replace the `stm32u5xx_it.h` file.
   b. Replace the `stm32u5xx_it.c` file.
   c. Replace the `main.h` file and update it. Keep the LED and user button definition from the LL template under the `"BOARD SPECIFIC CONFIGURATION"` tags.
   d. Replace the `main.c` file and update it:
      • Keep the clock configuration of the `SystemClock_Config()` LL template function under the `"BOARD SPECIFIC CONFIGURATION"` tags.
      • Depending on LED definition, replace each LEDx occurrence with another LEDy available in the `main.h` file.

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

5.3 Getting STM32CubeU5 release updates

The new STM32CubeU5 MCU package releases and patches are available from www.st.com/stm32u5. 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 STM32CubeU5 MCU package?

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

The middleware stacks made by STMicroelectronics (ex: 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 STM32CubeU5 MCU package?

The STM32CubeU5 MCU package provides BSP drivers and ready-to-use examples for the following STM32U5 series boards:

- NUCLEO-U575ZI-Q
- STM32U575I-EV
- B-U585I-IOT02A
- NUCLEO-U5A5ZJ-Q
- NUCLEO-U545RE-Q
- STM32U5A9J-DK
- STM32U5G9J-DK2
- STM32U5G9J-DK1

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

Yes. STM32CubeU5 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 STM32U5 series devices?

All STM32U5 series 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 configuration of the user option bytes may be done with a STM32CubeProgrammer (STM32CubeProg).

6.5 How to disable TrustZone® on STM32U5 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".

Refer to AN5347 for more details.

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 STM32CubeU5 firmware package provides default memory partitioning in the *partition_<device>.h* files available under:

\Drivers\CMSIS\Device\ST\STM32U5xx\Include\Templates (see Section 4.2.1).

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_stm32u5xx.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 flash memory watermark option bytes (SEC_WMx_PSTRT/SEC_WMx_PEND) to define the secure/non- secure flash memory areas (with 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_<device>.h* files available under:

\Drivers\CMSIS\Device\ST\STM32U5xx\Include\Templates (see Section 4.2.1).

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_<device>.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 STM32CubeU5 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 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 STM32CubeU5 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 *stm32u5xx_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 *stm32u5xx_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 STM32CubeU5 MCU Package releases?

Refer to Getting release updates.

6.20 What is an LPBAM mechanism?

The LPBAM mechanism is a subsystem that operates independently from low power mode and without any software running (CPU intervention).

The low power mode depends on products integration.

6.21 Is the LPBAM based on the DMA "tasks" used to configure the peripherals?

The LPBAM is based on DMA transfers stored on linked-list nodes to configure partially autonomous peripherals.

6.22 How are managed the priorities between LPBAM and others DMA transfer?

The LPBAM is based on DMA transfer. The same DMA channels priority arbitration principle applies.

Revision history

Table 4. **Document revision history**

Date	Revision	Changes
24-Jun-2021	1	Initial release.
24-Feb-2023	2	Updated • Introduction • STM32CubeU5 MCU package architecture • Macros for STM32U5 series • Boards for STM32U5 series • STM32CubeU5 MCU package structure • STM32CubeU5 examples overview • Number of examples for each board • TrustZone-enabled projects • Table 3. Number of examples for each board • What boards are supported by the STM32CubeU5 MCU package?
07-Jul-2023	3	Updated sections: • Table 1. Macros for STM32U5 series • Table 2. Boards for STM32U5 series • Figure 4. STM32CubeU5 MCU package structure • Section 4.2.1 TrustZone-enabled projects • Table 3. Number of examples for each board • Section 6.2 What boards are supported by the STM32CubeU5 MCU package?

Contents

1 General information 2

2 STM32CubeU5 main features 3

3 STM32CubeU5 architecture overview 5

3.1 Level 0 5

3.1.1 Board support package (BSP). 5

3.1.2 Hardware abstraction layer (HAL) and low-layer (LL) 6

3.1.3 Basic peripheral usage examples 6

3.2 Level 1 6

3.2.1 Middleware components 6

3.2.2 Examples based on the middleware components 8

3.3 Level 2 8

3.4 Utilities 8

3.4.1 LPBAM utility. 8

4 STM32CubeU5 MCU package overview 9

4.1 Supported STM32U5 series devices and hardware 9

4.2 MCU package overview 10

4.2.1 TrustZone-enabled projects 12

5 Getting started with STM32CubeU5 16

5.1 Running a first example 16

5.1.1 Running a first TrustZone-enabled example 16

5.1.2 Running a first TrustZone-disabled example 17

5.2 Developing a custom application. 18

5.2.1 Using STM32CubeMX to develop or update an application. 18

5.2.2 HAL application. 18

5.2.3 LL application 21

5.3 Getting STM32CubeU5 release updates 21

6 FAQ 22

6.1 What is the license scheme for the STM32CubeU5 MCU package? 22

6.2 What boards are supported by the STM32CubeU5 MCU package? 22

6.3 Are any examples provided with the ready-to-use toolset projects? 22

6.4 How to enable TrustZone® on STM32U5 series devices? 22

6.5 How to disable TrustZone® on STM32U5 series devices? 22

6.6 How to update the secure/nonsecure memory mapping 23

6.7 How to set up interrupts for secure and nonsecure applications 23

6.8 Why does the system enter in SecureFault_Handler()? 23
6.9 Are there any links with standard peripheral libraries? 23
6.10 Does the HAL layer take advantage of interrupts or DMA? How can this be controlled? . . . 23
6.11 How are the product/peripheral specific features managed? 23
6.12 When should the HAL be used versus LL drivers? 24
6.13 How can LL drivers be included in an existing environment? Is there any LL configuration file as for HAL? 24
6.14 Can HAL and LL drivers be used together? If yes, what are the constraints? 24
6.15 Are there any LL APIs that are not available with HAL? 24
6.16 Why are SysTick interrupts not enabled on LL drivers? 24
6.17 How are LL initialization APIs enabled? 24
6.18 How can STM32CubeMX generate code based on embedded software? 24
6.19 How to get regular updates on the latest STM32CubeU5 MCU Package releases? 24
6.20 What is an LPBAM mechanism? 24
6.21 Is the LPBAM based on the DMA "tasks" used to configure the peripherals? 24
6.22 How are managed the priorities between LPBAM and others DMA transfer? 24

Revision history 25

List of tables 28

List of figures 29

List of tables

Table 1. Macros for STM32U5 series 9
Table 2. Boards for STM32U5 series 10
Table 3. Number of examples for each board 15
Table 4. Document revision history 25

List of figures

Figure 1. STM32CubeU5 MCU package components 4
Figure 2. STM32CubeU5 MCU package architecture 5
Figure 3. LPBAM utility layers 8
Figure 4. STM32CubeU5 MCU package structure 10
Figure 5. STM32CubeU5 examples overview. 12
Figure 6. Secure and nonsecure multiprojects structure. 13