UM3065

User manual

Getting started with STM32CubeH5 for STM32H5 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 STM32CubeH5 for the STM32H5 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, mbed-crypto, secure manager API, MCUboot, 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 STM32CubeH5 MCU Package.

Section 2 describes the main features of the STM32CubeH5 MCU Package. Section 3 and Section 4 provide an overview of the STM32CubeH5 architecture and MCU Package structure.

1 General information

The STM32CubeH5 MCU Package runs on STM32 32-bit microcontrollers based on the Arm® Cortex®-M33 processor with Arm® TrustZone® and FPU.

Note: Arm and TrustZone are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 STM32CubeH5 main features

The STM32CubeH5 MCU Package runs on STM32 32-bit microcontrollers based on the Arm® Cortex®-M33 processor with TrustZone® and FPU.

The STM32CubeH5 gathers, in a single package, all the generic embedded software components required to develop an application for the STM32H5 series microcontrollers. In line with the STM32Cube initiative, this set of components is highly portable, not only within the STM32H5 series microcontrollers but also to other STM32 series.

The STM32CubeH5 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 STM32CubeH5 MCU Package also contains a comprehensive middleware components constructed around Microsoft® Azure® RTOS middleware and other in-house and open source stacks, with the corresponding examples.

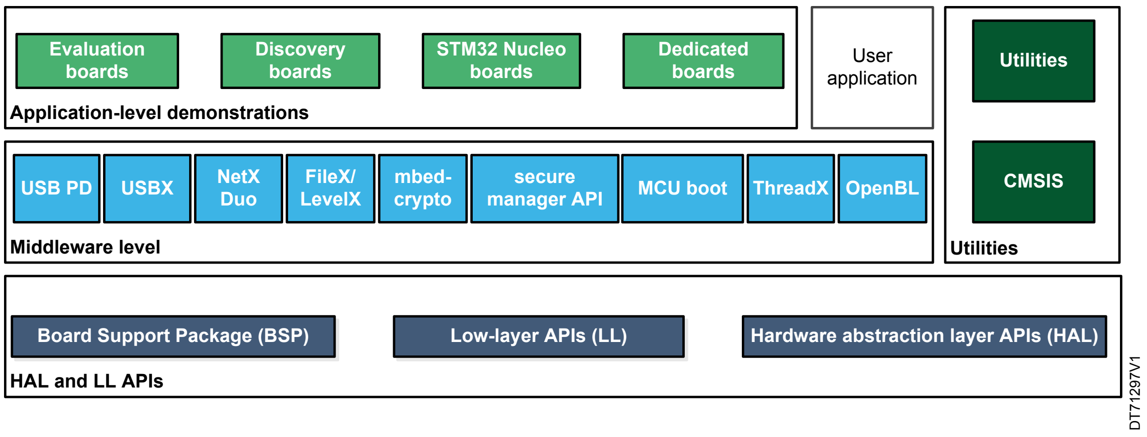
They come with free user-friendly license terms:

- 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 translation layer: FileX / LevelX
- Industrial grade networking stack: optimized for performance coming with many IoT protocols: NetX Duo
- USB PD library
- OpenBootloader
- Secure manager API
- MCU boot
- mbed-crypto libraries
- STM32_Audio library

Several applications and demonstration implementing all these middleware components are also provided in the STM32CubeH5 MCU Package.

The STM32CubeH5 MCU Package component layout is illustrated in the figure below.

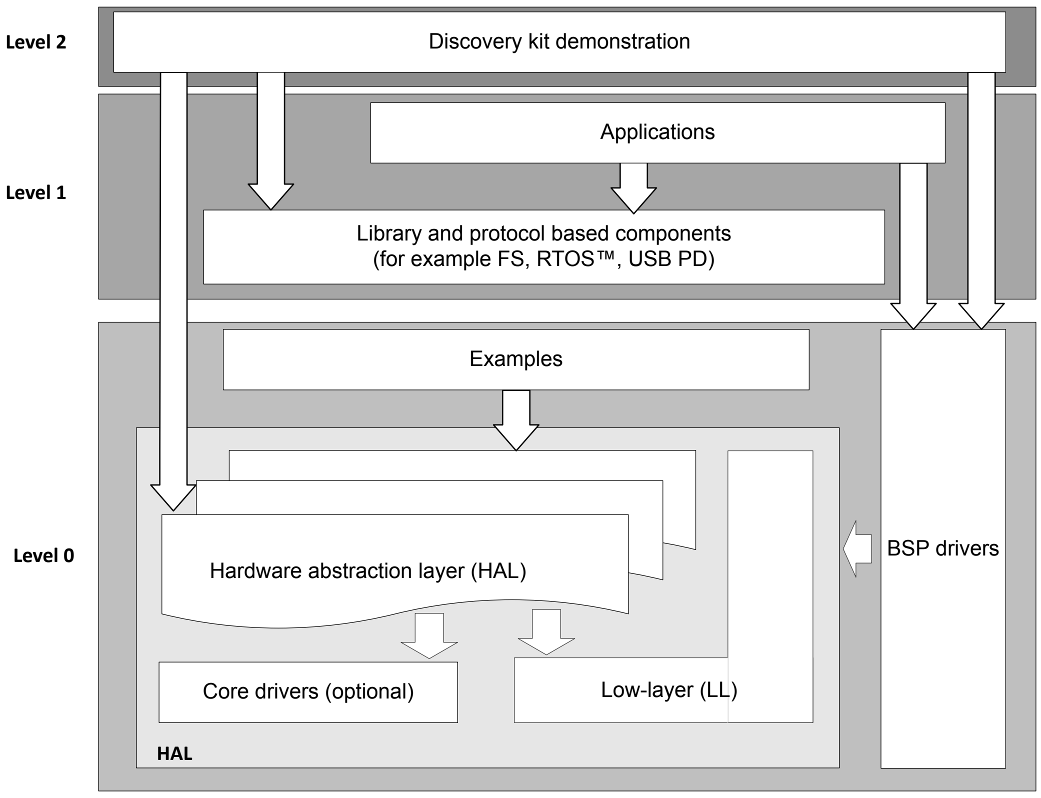
Figure 1. **STM32CubeH5 MCU Package components**



3 STM32CubeH5 architecture overview

The STM32CubeH5 MCU Package solution is built around three independent levels that easily interact as described in the figure below.

Figure 2. **STM32CubeH5 MCU Package architecture**


Level 2
Discovery kit demonstration
Level 1
Applications
Library and protocol based components
(for example FS, RTOS™, USB PD)
Level 0
Examples
Hardware abstraction layer (HAL)
Core drivers (optional)
Low-layer (LL)
HAL
BSP drivers

3.1 Level 0

This level is divided into three sub-layers:

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

3.1.1 Board support package (BSP)

This layer offers a set of APIs relative to the hardware components in the hardware boards (such as LCD, Audio, and microSD™ 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 by just implementing the low-level routines.

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

The STM32CubeH5 HAL and LL are complementary and cover a wide range of applications requirements:

- The HAL drivers offer high-level function-oriented highly-portable APIs. They hide the MCU and peripheral complexity to end user.
  The HAL drivers provide generic multi-instance feature-oriented APIs which simplify user application implementation by providing ready to use processes. As 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 which provides common and generic functions to all the STM32 Series
  – Extension APIs which provide specific and customized functions for a specific family or a specific part number.
- The low-layer APIs provide low-level APIs at register level, with better optimization but less portability. They require a deep knowledge of MCU and peripheral specifications.
  The LL drivers are designed to offer a fast light-weight expert-oriented layer which is closer to the hardware than the HAL. Contrary to the HAL, LL APIs are not provided for peripherals where optimized access is not a key feature, or for those requiring heavy software configuration and/or complex upper-level stack.
  The LL drivers feature:
  – A set of functions to initialize peripheral main features according to the parameters specified in data structures
  – A set of functions used to fill initialization data structures with the reset values corresponding to each field
  – Function for peripheral de-initialization (peripheral registers restored to their default values)
  – A set of inline functions for direct and atomic register access
  – Full independence from HAL and capability to be used in standalone mode (without HAL drivers)
  – Full coverage of the supported peripheral features

3.1.3 Basic peripheral usage examples

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

3.2 Level 1

This level is divided into two sub-layers:

- Middleware components
- Examples based on the middleware components

3.2.1 Middleware components

The middleware is a set of libraries constructed around Microsoft® Azure® RTOS middleware and other in-house (such as OpenBL) and open source (such as mbed-crypto). All are integrated and customized for STM32 MCU devices and enriched with corresponding application examples based on STM32 evaluation boards. Horizontal interactions between the components of this layer are simply done by calling the 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), designed for embedded systems with two functional modes:
  – Common mode: common RTOS functionalities such as thread management and synchronization, memory pool management, messaging, and event handling
  – Module mode: an advanced usage mode that allows 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 file system (FS) / flash 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 management 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 low-level driver through an abstraction layer using the configuration file to avoid any dependency between the library and the low-level drivers.
- OpenBootloader
  This middleware component provides an open source bootloader with exactly the same features as STM32 system bootloader and with the same tools used for system bootloader
- Secure manager API
  This component enables STMicroelectronics installable services that provide callable standard PSA service API for nonsecure application at runtime.
  - PSA standard secure services:
    - Firmware update
    - Internal trusted storage
    - Cryptography: AES, ECC, RSA, SHA, TRNG
    - Initial attestation
  - Software IP protection (PSA isolation level3)
    - Sandbox secure services
- STM32_Audio: Software library to convert PDM data to PCM format
- MCUboot
- 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).

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

3.4 Utilities

Alike all STM32Cube MCU Packages, the STM32CubeH5 provides a set of utilities that offer miscellaneous software and additional system resources services that can be used by either the application or the different STM32Cube Firmware intrinsic middleware and components.

4 STM32CubeH5 MCU package overview

4.1 Supported STM32H5 series devices and hardware

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

In addition, thanks to its layered architecture, the STM32CubeH5 offers full support of all STM32H5 series. The user has only to define the right macro in *stm32h5xx.h*.

Table 1 shows the macro to define depending on the STM32H5 series device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32H5 series

Macro defined in stm32h5xx.h	STM32H5 part numbers
STM32H573xx	STM32H573AII6, STM32H573AII3Q, STM32H573IIT6, STM32H573IIT3Q, STM32H573IIK6, STM32H573IIK3Q, STM32H573MIY3QTR, STM32H573RIT6, STM32H573RIV6, STM32H573VIT6, STM32H573VIT3Q, STM32H573ZIT6, STM32H573ZIT3Q
STM32H563xx	STM32H563AGI6, STM32H563AII6, STM32H563AII3Q, STM32H563IGT6, STM32H563IGK6, STM32H563IIT6, STM32H563IIT3Q, STM32H563IIK6, STM32H563IIK3Q, STM32H563MIY3QTR, STM32H563RGT6,STM32H563RGV6, STM32H563RIT6, STM32H563RIV6, STM32H563VGT6, STM32H563VIT6, STM32H563VIT3Q, STM32H563ZGT6, STM32H563ZIT6, STM32H563ZIT3Q
STM32H562xx	STM32H562AGI6, STM32H562AII6, STM32H562IGT6, STM32H562IGK6, STM32H562IIT6, STM32H562IIK6, STM32H562RGT6, STM32H562RGV6, STM32H562RIT6, STM32H562RIV6, STM32H562VGT6, STM32H562VIT6, STM32H562ZGT6, STM32H562ZIT6
STM32H503xx	STM32H503EBY6TR, STM32H503KBU6, STM32H503CBU6, STM32H503CBT6, STM32H503RBT6 STM32H562AGI6,
STM32H533xx	STM32H533CEU6, STM32H533CET6, STM32H533HEY6TR, STM32H533RET6, STM32H533VET6, STM32H533VEI6, STM32H533ZET6, STM32H533ZEJ6
STM32H523xx	STM32H523CCU6, STM32H523CCT6, STM32H523CEU6, STM32H523CET6, STM32H523HEY6TR, STM32H523RCT6, STM32H523RET6, STM32H523VCT6, STM32H523VCI6, STM32H523VET6, STM32H523VEI6, STM32H523ZCT6, STM32H523ZCJ6, STM32H523ZET6, STM32H523ZEJ6

STM32H5 series 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 STM32H5 series

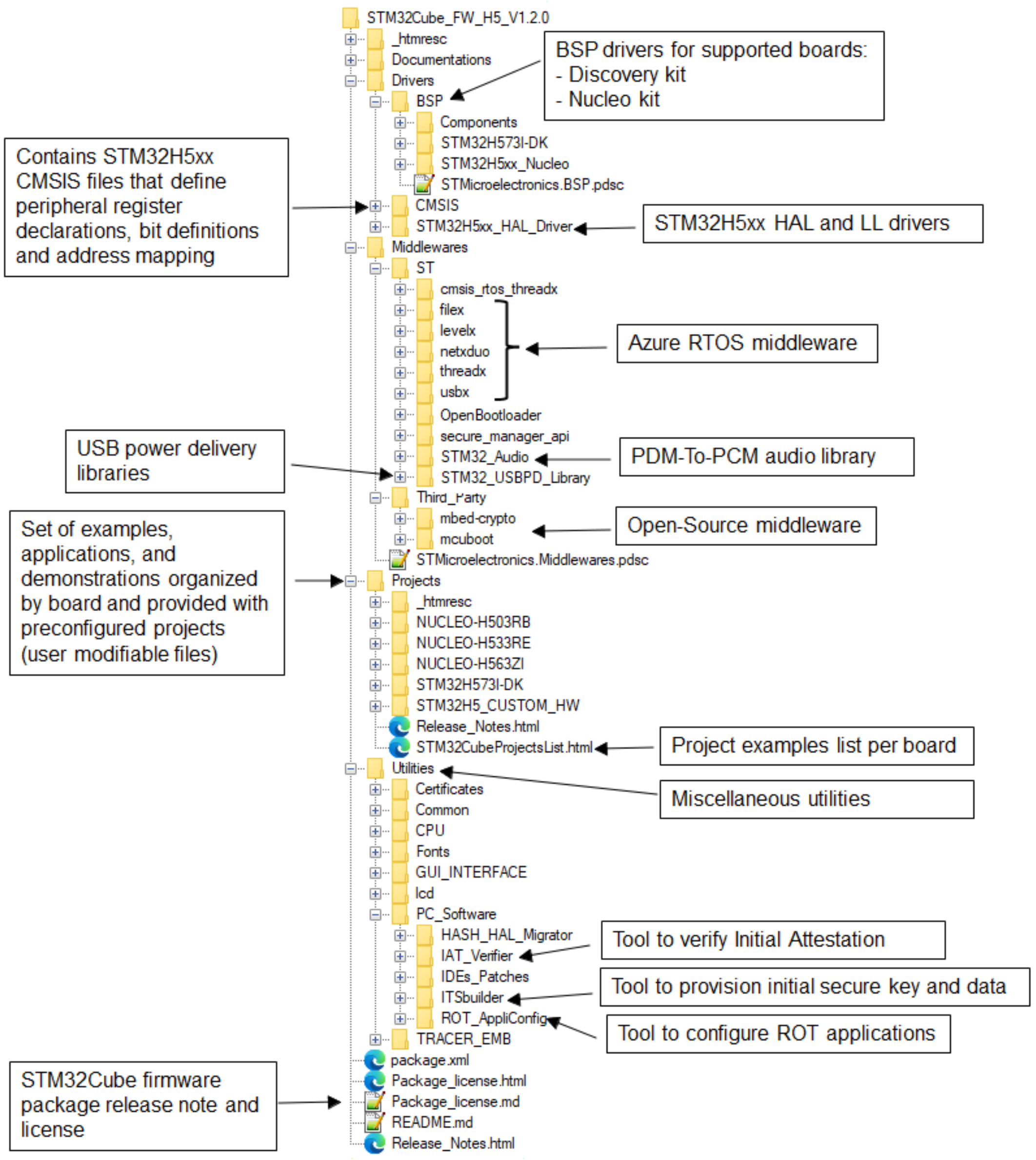
Board	Supported STM32H5 part numbers
NUCLEO-H563ZI	STM32H563ZIT6
NUCLEO-H503RB	STM32H503RBH6
STM32H573I-DK	STM32H573IIK3Q
NUCLEO-H533RE	STM32H533RET6

The STM32CubeH5 MCU Package is able to run on any compatible hardware. The user simply updates the BSP drivers to port the provided examples on his own board, if the latter has the same hardware features (such as LED, LCD display, buttons).

4.2 MCU Package overview

The STM32CubeH5 MCU Package solution is provided in one single zip package having the structure shown in Figure 3.

Figure 3. STM32CubeH5 MCU Package structure

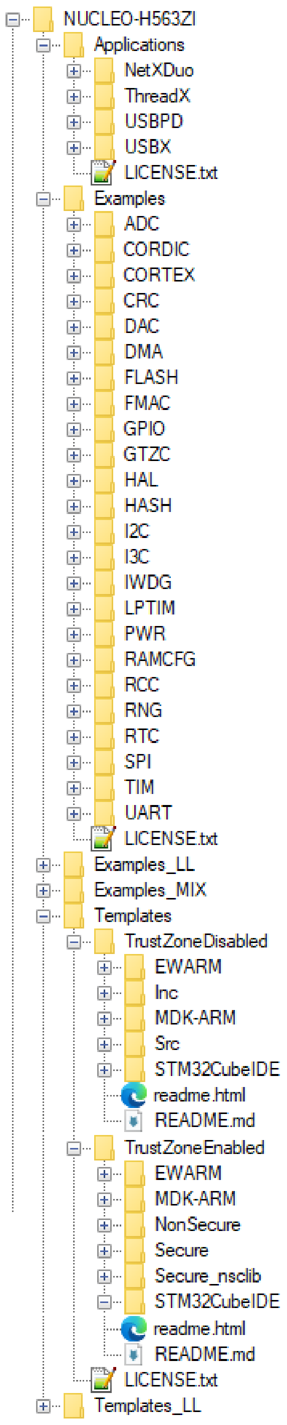


1. *The component files must not be modified by the user. Only the \Projects sources are editable by the user.*

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

Figure 4 shows the project structure for the NUCLEO-H563ZI board.

Figure 4. Overview of STM32CubeH5 examples



```
NUCLEO-H563ZI
    Applications
        NetXDuo
        ThreadX
        USBPD
        USBX
        LICENSE.txt
    Examples
        ADC
        CORDIC
        CORTEX
        CRC
        DAC
        DMA
        FLASH
        FMAC
        GPIO
        GTZC
        HAL
        HASH
        I2C
        I3C
        IWDG
        LPTIM
        PWR
        RAMCFG
        RCC
        RNG
        RTC
        SPI
        TIM
        UART
        LICENSE.txt
    Examples_LL
    Examples_MIX
    Templates
        TrustZoneDisabled
            EWARM
            Inc
            MDK-ARM
            Src
            STM32CubeIDE
            readme.html
            README.md
        TrustZoneEnabled
            EWARM
            MDK-ARM
            NonSecure
            Secure
            Secure_nsclib
            STM32CubeIDE
            readme.html
            README.md
        LICENSE.txt
    Templates_LL
```

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

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

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

4.2.1 TrustZone-enabled projects

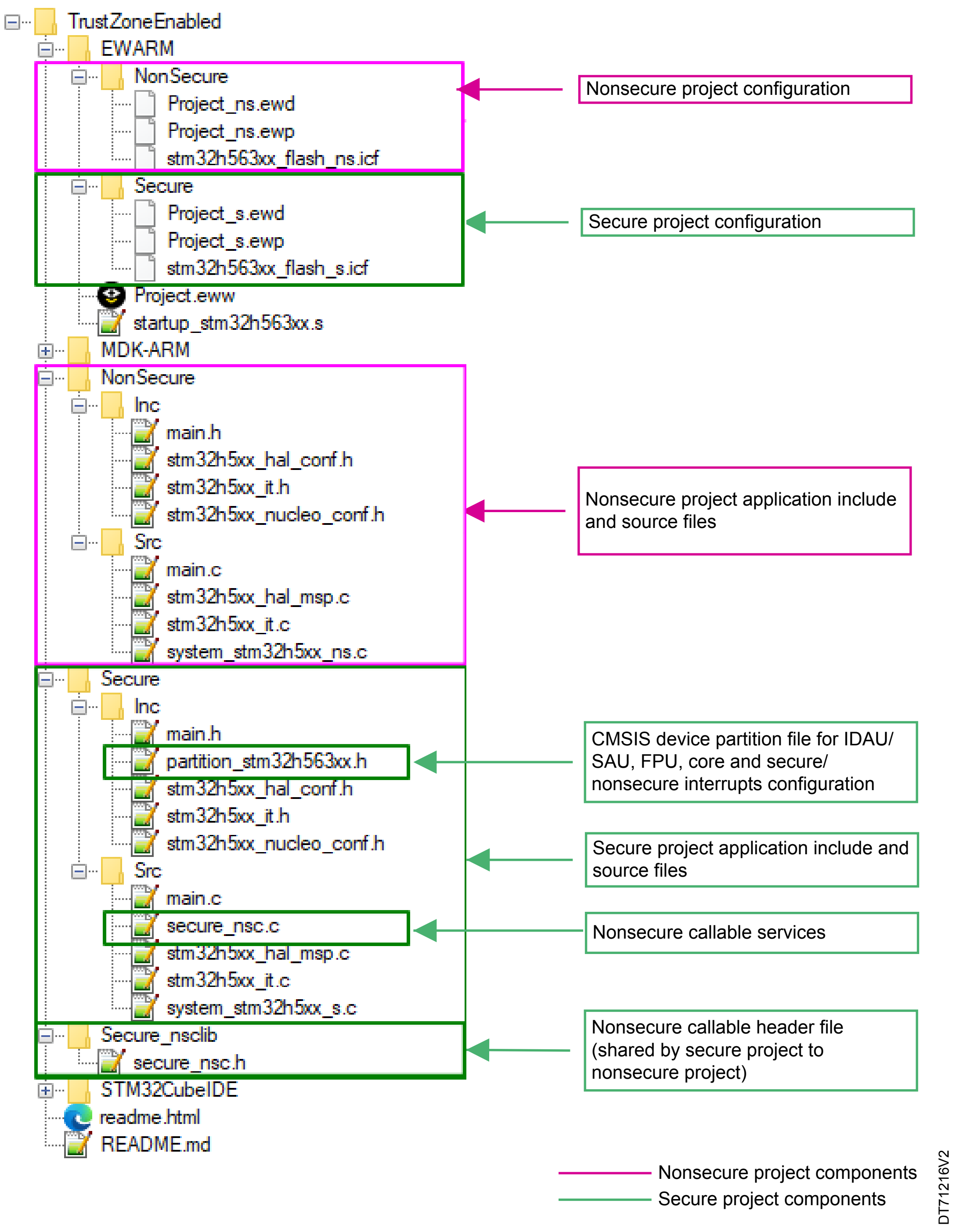
TrustZone-enabled “Examples” names are prefixed with "_TrustZone".

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

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

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

Figure 5. Secure and nonsecure multiprojects structure



The STM32CubeH5 firmware package provides default memory partitioning in the *partition_<device>.h* files available under:

\Drivers\CMSIS\Device\ST\STM32H5xx\Include\Templates.

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

If SAU is enabled by the user, a default SAU region configuration is predefined in partition files as follows:

- SAU region 0: 0x0C0FE000 - 0x0C0FFFFF (secure, nonsecure callable)
- SAU region 1: 0x08100000 - 0x081FFFFF (nonsecure flash memory Bank2 (1024 Kbytes))
- SAU region 2: 0x20050000 - 0x2009FFFF (nonsecure SRAM3 (320 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, the STM32H5 series devices must have the following user option bytes set:
- TZEN = 0xB4 (TrustZone-enabled device)
- SECWM1_STRT=0x0 SECWM1_END = 0x7F (all 128 pages of flash memory Bank1 set as secure)
- SECWM2_STRT=0x1 SECWM2_END = 0x0 (no page of flash memory Bank2 set as secure, hence Bank2 is nonsecure).

Note: The internal flash memory is fully secure by default in TZEN = 0xB4 and user option bytes SECWM1_STRT/SECWM1_END and SECWM2_STRT/SECWM2_END 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 that contains all header files.
- *Src* folder for the sources code.
- *EWARM*, *MDK-ARM*, and *STM32CubeIDE* folders containing the preconfigured project for each toolchain.
- *readme.md* describing the example behavior and needed environment to make it work
- **.ioc* file that allows users to open most of firmware examples within STM32CubeMX

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

Table 3. Number of examples for each board

Level	NUCLEO-H503RB	NUCLEO-H563ZI	STM32H573I-DK	NUCLEO-H533RE	STM32H5_CUSTOM_HW	Total
Templates_LL	1	1	1	1	-	**4**
Templates	2	3	6	5	-	**16**
Templates_Board	-	-	-	1	-	**1**
Examples_MIX	7	9	0	0	-	**16**
Examples_LL	36	32	0	6	-	**74**
Examples	50	78	26	37	2	**193**
Demonstrations	0	0	1	0	-	**1**
Applications	4	16	32	6	-	**58**
Total	**100**	**139**	**66**	**56**	**2**	**363**

5 Getting started with STM32CubeH5

5.1 Running a first example

This section explains how simple it is to run a first example on an STM32H5 series board. The program simply toggles a LED on the NUCLEO-H563ZI board:

Download the STM32CubeH5 MCU Package. Unzip it into an appropriate directory. Make sure the package structure shown in Figure 3. STM32CubeH5 MCU Package structure 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

Prior to loading and running a TrustZone-enabled example, it is mandatory to read the example readme file for any specific configuration that insures that the security is enabled as described in Section 4.2.1: TrustZone-enabled projects (TZEN = 0xB4 (user option byte)).

1. Browse to *\Projects\NUCLEO-H563ZI\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. Under the example folder, open *\EWARM* sub-folder
  2. Launch the Project.eww workspace
  3. Set the "xxxxx_S" as 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 **Download and Debug button (Ctrl+D)**
  7. Run program: **Debug** → **Go(F5)**
- MDK-ARM
  1. Open the MDK-ARM toolchain
  2. Open Multi-projects 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 Multi-projects 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. In the **Edit configuration** window, select the **Startup** panel, and add load image and symbols of the nonsecure project.
  7. Be careful, the nonsecure project has to be loaded before the secure project.
  8. Then click "OK".
  9. Run the example on debug perspective.

5.1.2 Running a first TrustZone-disabled example

Prior to loading and running a TrustZone-disabled example, it is mandatory to read the example readme file for any specific configuration or if nothing is mentioned, ensure that the board device has the security disabled (TZEN = 0xC3 (user option byte)). See FAQ for doing the optional regression to TZEN = 0xC3.

1. Browse to *\Projects\NUCLEO-H563ZI\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. Under the example folder, open *\EWARM* sub-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. Under the example folder, open *\MDK-ARM* sub-folder
  2. Launch the *Project.uvprojx* workspace

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

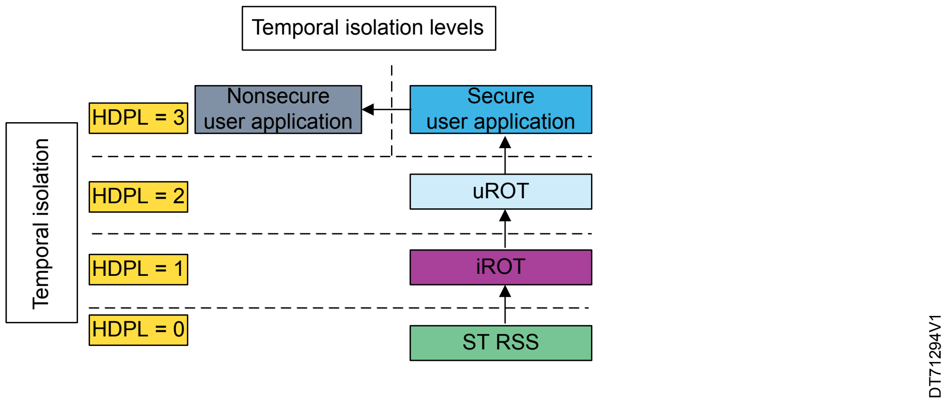
  3. Rebuild all files: **Project** → **Rebuild all target files**
  4. Load project image: **Debug** → **Start/Stop Debug Session**
  5. Run program: **Debug** → **Run (F5)**.
- STM32CubeIDE
  1. Open the STM32CubeIDE toolchain
  2. Click **File** → **Switch Workspace** → **Other** and browse to the STM32CubeIDE 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)**

5.1.3 Running a first Root of Trust (ROT) example

5.1.3.1 Bootpath overview

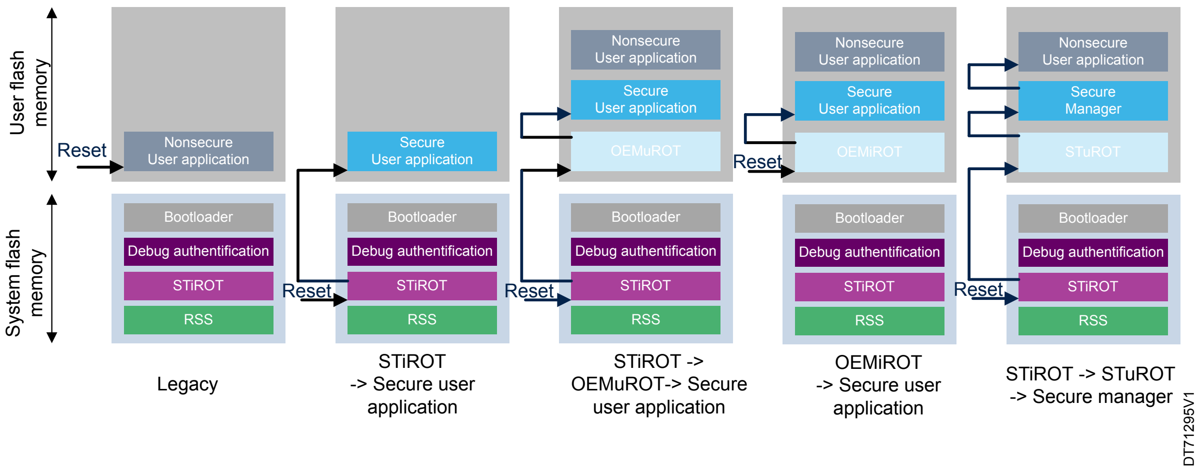
The STM32H5xx devices support temporal isolation through Hide Protection Level (HDPL).

Figure 6. Temporal isolation levels on STM32H5 series



Several bootpaths are demonstrated on STM32H5xx devices. They consist of one or two boot stages provided by STMicroelectronics or implemented by original equipment manufacturers (OEMs).

Figure 7. Security bootpath supported on STM32H5 Series

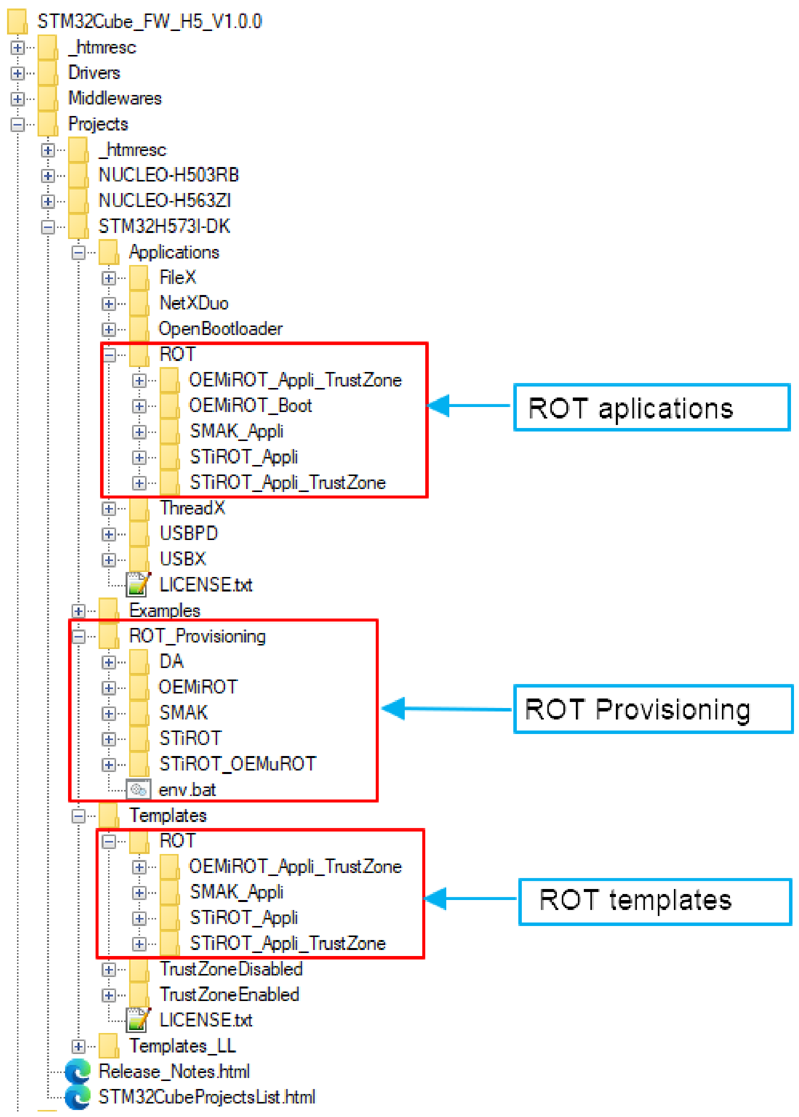

User flash memory
System flash memory
Reset
Nonsecure User application
Secure User application
OEMuROT
OEMiROT
Secure Manager
STuROT
Bootloader
Debug authentification
STiROT
RSS
Legacy
STiROT -> Secure user application
STiROT -> OEMuROT-> Secure user application
OEMiROT -> Secure user application
STiROT -> STuROT -> Secure manager
DT71295V1

5.1.3.2 ROT applications

Prior to loading and running an ROT application, check the application readme file for any specific configuration that ensures that the related bootpath is enabled.

The ROT applications can be found under *\Projects\STM32H573I-DK\Applications\ROT*, *Projects\ NUCLEO-H563ZI\Applications\ROT* and *Projects\ NUCLEO-H503RB\Applications\ROT*. For STM32H573I-DK, they are organized as shown in the figure below.

Figure 8. **ROT application structure**



```
STM32Cube_FW_H5_V1.0.0
  _htmresc
  Drivers
  Middlewares
  Projects
    _htmresc
    NUCLEO-H503RB
    NUCLEO-H563ZI
    STM32H573I-DK
      Applications
        FileX
        NetXDuo
        OpenBootloader
        ROT
          OEMiROT_Appli_TrustZone
          OEMiROT_Boot
          SMAK_Appli
          STiROT_Appli
          STiROT_Appli_TrustZone
        ThreadX
        USBPD
        USBX
        LICENSE.txt
      Examples
      ROT_Provisioning
        DA
        OEMiROT
        SMAK
        STiROT
        STiROT_OEMuROT
        env.bat
      Templates
        ROT
          OEMiROT_Appli_TrustZone
          SMAK_Appli
          STiROT_Appli
          STiROT_Appli_TrustZone
        TrustZoneDisabled
        TrustZoneEnabled
        LICENSE.txt
      Templates_LL
    Release_Notes.html
    STM32CubeProjectsList.html
```

For *Projects\ NUCLEO-H563ZI\Applications\ROT* and *Projects\ NUCLEO-H503RB\Applications\ROT*, the ROT application structure is similar except that STiROT and SMAK are not available.

5.1.3.3 *OEMiROT, STiROT, and STiROT_OEMuROT bootpaths*

To run the OEMiROT, STiROT, or STiROT_OEMuROT bootpath, proceed as follows:

1. First, configure the user environment using the `env` script available under the *ROT_Provisionning* folder.

Caution: Make sure the STM32TrustedPackageCreator option is selected during the STM32CubeProgrammer installation, as it is used by the provisioning script.

2. Select the required bootpath, then launch the provisioning script located under each bootpath folder.
3. Once the provisioning script for the desired bootpath has started, follow the instructions displayed on the terminal. They guide you through the following steps:
   a. Configuration management: option byte key (OBK) generation (configuration and debug authentication.
   b. Image generation: image build (secure boot and application).
   c. Provisioning: image programming and OBK provisioning.

4. Once the steps above are executed, reset the target, and connect the terminal emulator via the ST-LINK virtual communication port to get the application menu.

Caution: Do not change the product state from OPEN to a higher state without having provisioned the debug authentication (certificate and permissions). Otherwise, the MCU becomes unusable. The provisioning script ensures that the provisioning of the debug authentication is performed before modifying the product state, so that the device can be reinitialized.

5.1.3.4 *Secure manager access kit (SMAK) bootpath*

To run the SMAK bootpath, proceed as follows:

1. Before using the SMAK application (SMAK_Appli), download the secure manager package from *www.st.com*, and install it under *STM32H573I-DK\Applications\ROT\SMAK_Appli\Binary*.
2. First configure the user environment using the `env` script available under the *ROT_Provisionning* folder.

Caution: Make sure the STM32TrustedPackageCreator option is selected during the STM32CubeProgrammer installation, as it is used by the provisioning script.

3. Then launch the provisioning script available under *ROT_Provisionning\SMAK*, and follow the instructions displayed on the terminal. They guide you through the following steps:
   a. Configuration:
      ◦ OBK generation (configuration, debug authentication) and OB generation
      ◦ SFI generation
      ◦ SMAK_Appli configuration
   b. Installation: Secure manager programming and OBK provisioning through SFI.
   Once the above steps are complete, the product state is TZ-CLOSED so that the MCU secure area is closed.
4. Open the *NonSecure SMAK_Appli* project using your preferred toolchain, connect the terminal emulator via the ST-LINK virtual port, build the project, then download it.
5. Follow the instructions displayed on the terminal to access the demonstrations on the internal trusted storage, the cryptography, the initial attestation, and firmware update Platform Security Architecture (PSA) services.

5.1.3.5 *Debug authentication (DA) regression*

After having run an ROT application, the device can be erased and reinitialized by erasing the flash memory and by switching back the product to OPEN state. This can be done by running the regression script located in the *ROT_Provisioning/DA* folder.

5.2 Developing a custom application

The instruction cache (ICACHE) must be enabled by software to get a 0 wait-state execution from flash memory and external memories, and reach the maximum performance and a better power consumption.

5.2.1 Using STM32CubeMX to develop or update an application

In the STM32CubeH5 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 6.8.0 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 *STM32CubeMX for STM32 configuration and initialization C code generation* (UM1718).

For a list of the available example projects for the STM32CubeH5, refer to the STM32Cube firmware examples for STM32CubeH5 application note.

5.2.2 HAL application

This section describes the steps required to create a custom HAL application using STM32CubeH5:

1. Create a project
   To create a new project, start either from the *Template* project provided for each board under *\Projects\<STM32xxx_yyy>\Templates* or from any available project under *\Projects\<STM32xxy_yyy>\Examples* or *\Projects\<STM32xx_yyy>\Applications* (where *<STM32xxx_yyy>* refers to the board name, such as STM32CubeH5).
   The Template project provides an empty main loop function, however it is a good starting point to understand the STM32CubeH5 project settings. The template has the following characteristics:
   – It contains the HAL source code, CMSIS and BSP drivers which are the minimum set of components required to develop a code on a given board.
   – It contains the include paths for all the firmware components.
   – It defines the supported STM32H5 series devices, allowing the CMSIS and HAL drivers to be configured correctly.
   – It provides read-to-use user files 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 user project (optional)**
   To identify the source files to be added to the project file list, refer to the documentation provided for each middleware. Refer to the applications under *\Projects\STM32xxx_yyy\Applications\<MW_Stack>* (where *<MW_Stack>* refers to the middleware stack, such as USBX) to know which source files and which include paths must be added.
3. **Configure the firmware components**
   The HAL and middleware components offer a set of build time configuration options using macros `#define` declared in a header file. A template configuration file is provided within each component which has to be copied to the project folder (usually the configuration file is named *xxx_conf_template.h*, the word '*_template*' needs to be removed when copying it to the project folder). The configuration file provides enough information to understand the impact of each configuration option. More detailed information is available in the documentation provided for each component.
4. **Start the HAL Library**
   After jumping to the main program, the application code must call `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 *stm32h5xx_hal_conf.h*).
   b. Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority `TICK_INT_PRIO` defined in *stm32h5xx_hal_conf.h*, which is clocked by the CSI (at this stage, the clock is not yet configured and thus the system is running from the 4 MHz CSI).
   c. Setting of NVIC group priority to 0.
   d. Call of `HAL_MspInit()` callback function defined in *stm32h5xx_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 oscillator or all oscillators. The PLL configuration can be skipped 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 and AHB and APB prescalers.

   Initialize the peripheral
   a. First write the peripheral `HAL_PPP_MspInit` function. Proceed as follows:
      ◦ Enable the peripheral clock.
      ◦ Configure the peripheral GPIOs.
      ◦ Configure the DMA channel and enable DMA interrupt (if needed).
      ◦ Enable peripheral interrupt (if needed).
   b. Edit the *stm32xxx_it.c* to call the required interrupt handlers (peripheral and DMA), if needed.
   c. Write process complete callback functions if peripheral interrupt or DMA is planned to be used.
   d. In user *main.c* file, initialize the peripheral handle structure then call the function `HAL_PPP_Init()` to initialize the peripheral.
6. **Develop an application**
   At this stage, the system is ready and user application code development can start.
   a. The HAL provides intuitive and ready-to-use APIs to configure the peripheral. It supports polling, interrupts and a DMA programming model, to accommodate any application requirements. For more details on how to use each peripheral, refer to the rich example set provided in the STM32CubeH5 MCU Package.

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

5.2.3 LL application

This section describes the steps needed to create a custom LL application using STM32CubeH5.

1. **Create a project**
   To create a new project, either start from the Templates_LL project provided for each board under *\Projects\<STM32xxx_yyy>\Templates_LL* or from any available project under *\Projects\<STM32xxy_yyy>\Examples_LL* (*<STM32xxx_yyy>* refers to the board name, such as NUCLEO-H563ZI).
   The template project provides an empty main loop function, which is a good starting point to understand the project settings for STM32CubeH5. 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 include paths for all the required firmware components.
   – It selects the supported STM32H5 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 & USER_BUTTON definition abstraction layer.
     ◦ main.c: system clock configuration for maximum frequency.
2. **Port an existing project to another board**
   To port an existing project to another target board, start from the Templates_LL project provided for each board and available under *\Projects\<STM32xxx_yyy>\Templates_LL*:
   a. Select a LL example
      To find the board on which LL examples are deployed, refer to the list of LL examples *STM32CubeProjectsList.html*.
3. Port the LL example
   – Copy/paste the *Templates_LL* folder - to keep the initial source - or directly update existing *Templates_LL* project.
   – Then porting consists principally in replacing *Templates_LL* files by the *Examples_LL* targeted project.
   – Keep all board specific parts. For reasons of clarity, board specific parts have been flagged with specific tags:

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

   Thus the main porting steps are the following:
     ◦ Replace the *stm32h5xx_it.h* file
     ◦ Replace the *stm32h5xx_it.c* file
     ◦ Replace the main.h file and update it: keep the LED and user button definition of the LL template under `'BOARD SPECIFIC CONFIGURATION'` tags.
     ◦ Replace the main.c file and update it:
       Keep the clock configuration of the `SystemClock_Config()` LL template function under `'BOARD SPECIFIC CONFIGURATION'` tags.
       Depending on LED definition, replace each LEDx occurrence with another LEDy available in *main.h*.

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

5.3 Getting STM32CubeH5 release updates

The new STM32CubeH5 MCU Package releases and patches are available from www.st.com/stm32h5. 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 STM32CubeH5 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.

6.2 What boards are supported by the STM32CubeH5 MCU Package?

The STM32CubeH5 MCU Package provides BSP drivers and ready-to-use examples for the following STM32H5 series boards:

- NUCLEO-H563ZI
- NUCLEO-H503RB
- STM32H573I-DK
- NUCLEO-H533RE

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

Yes. STM32CubeH5 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 STM32H5 Series devices?

All STM32H5 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 user option bytes configuration may be done with STM32CubeProgrammer (STM32CubeProg).

6.5 How to disable TrustZone® on STM32H5 Series devices?

The TrustZone® security can be disabled through the STM32CubeProgrammer by following the sequence below:

1. Set the TZEN bit to 0xC3.
2. Click *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 STM32CubeH5 MCU Package provides default memory partitioning in the *partition_<device>.h* files available under:

\Drivers\CMSIS\Device\ST\STM32H5xx\Include\Templates (see 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 non-secure memory address aliases):

- If SAU is enabled, nonsecure area update (internal flash and SRAMs) (see *partition_stm32h5xx.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 watermark option bytes (SEC_WMx_STRT/SEC_WMx_END) 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\STM32H5xx\Include\Templates (see 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_<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 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 STM32CubeH5 HAL and LL drivers are the replacement of the standard peripheral library:

- The HAL drivers offer a higher abstraction level compared to the standard peripheral APIs. They focus on the features that are common to the peripherals rather than hardware. A set of user-friendly APIs allows a higher abstraction level which in turn makes them easily portable from one product to another.
- The LL drivers offer low-layer registers level APIs. They are organized in a simpler and clearer way avoiding direct register accesses. LL drivers also include peripheral initialization APIs, which are more optimized compared to what is offered by the SPL, while being functionally similar. Compared to HAL drivers, these LL initialization APIs allows an easier migration from the SPL to the STM32CubeH5 LL drivers, since each SPL API has its equivalent LL API(s).

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 *stm32h5xx_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. Mixing HAL and LL is illustrated in the "Examples_MIX" example.

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

Yes, there are. A few Cortex® APIs have been added in *stm32h5xx_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, that allows to provide 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 STM32CubeH5 MCU Package releases?

Refer to Section 5.3: Getting STM32CubeH5 release updates.

Revision history

Table 4. Document revision history

Date	Revision	Changes
06-Feb-2023	1	Initial release.
05-Feb-2024	2	Updated: • Document title. • Section 3.2.1: Middleware components • Table 1. Macros for STM32H5 series • Table 2. Boards for STM32H5 series • Figure 3. STM32CubeH5 MCU Package structure • Table 3. Number of examples for each board • Section 6.1: What is the license scheme for the STM32CubeH5 MCU Package? • Section 6.2: What boards are supported by the STM32CubeH5 MCU Package?

Contents

1 General information 2

2 STM32CubeH5 main features 3

3 STM32CubeH5 architecture overview 4

3.1 Level 0 4

3.1.1 Board support package (BSP). 4

3.1.2 Hardware abstraction layer (HAL) and low-layer (LL) 5

3.1.3 Basic peripheral usage examples 5

3.2 Level 1 5

3.2.1 Middleware components 5

3.2.2 Examples based on the middleware components 6

3.3 Level 2 6

3.4 Utilities 6

4 STM32CubeH5 MCU package overview. 7

4.1 Supported STM32H5 series devices and hardware 7

4.2 MCU Package overview 8

4.2.1 TrustZone-enabled projects 10

5 Getting started with STM32CubeH5 13

5.1 Running a first example 13

5.1.1 Running a first TrustZone-enabled example 13

5.1.2 Running a first TrustZone-disabled example 14

5.1.3 Running a first Root of Trust (ROT) example 15

5.2 Developing a custom application. 17

5.2.1 Using STM32CubeMX to develop or update an application. 17

5.2.2 HAL application. 18

5.2.3 LL application 20

5.3 Getting STM32CubeH5 release updates 20

6 FAQ 21

6.1 What is the license scheme for the STM32CubeH5 MCU Package? 21

6.2 What boards are supported by the STM32CubeH5 MCU Package? 21

6.3 Are any examples provided with the ready-to-use toolset projects? 21

6.4 How to enable TrustZone® on STM32H5 Series devices? 21

6.5 How to disable TrustZone® on STM32H5 Series devices? 21

6.6 How to update the secure / nonsecure memory mapping 21

6.7 **How to set up interrupts for secure and nonsecure applications** 21

6.8 Why does the system enter in SecureFault_Handler()? 22
6.9 Are there any links with standard peripheral libraries? 22
6.10 **Does the HAL layer take advantage of interrupts or DMA? How can this be controlled?** 22
6.11 How are the product/peripheral specific features managed? 22
6.12 **When should the HAL be used versus LL drivers?** 22
6.13 How can LL drivers be included in an existing environment? Is there any LL configuration file as for HAL? 22
6.14 **Can HAL and LL drivers be used together? If yes, what are the constraints?** 22
6.15 Are there any LL APIs which are not available with HAL? 22
6.16 Why are SysTick interrupts not enabled on LL drivers? 23
6.17 How are LL initialization APIs enabled? 23
6.18 How can STM32CubeMX generate code based on embedded software? 23
6.19 How to get regular updates on the latest STM32CubeH5 MCU Package releases? 23

Revision history 24

List of tables 27

List of figures 28

List of tables

Table 1. Macros for STM32H5 series 7
Table 2. Boards for STM32H5 series 7
Table 3. Number of examples for each board 12
Table 4. Document revision history 24

List of figures

Figure 1. STM32CubeH5 MCU Package components 3
Figure 2. STM32CubeH5 MCU Package architecture 4
Figure 3. STM32CubeH5 MCU Package structure 8
Figure 4. Overview of STM32CubeH5 examples. 9
Figure 5. Secure and nonsecure multiprojects structure. 11
Figure 6. Temporal isolation levels on STM32H5 series 15
Figure 7. Security bootpath supported on STM32H5 Series 15
Figure 8. ROT application structure. 16