Getting started with STM32CubeWBA for STM32WBA 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 STM32CubeWBA for the STM32WBA series), which include:
  - STM32Cube hardware abstraction layer (HAL), ensuring maximized portability across the STM32 portfolio
  - STM32Cube low-layer APIs, ensuring the best performance and footprints with a high degree of user control over hardware
  - A consistent set of middleware components such as ThreadX, FileX, LevelX, NetX Duo, USBX, touch library, mbed-crypto, TFM, MCUboot, OpenBL, and STM32_WPAN (including Bluetooth® Low Energy profiles and services, Mesh, Zigbee®, OpenThread, Matter, and 802.15.4 MAC layer)
  - 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 STM32CubeWBA MCU Package.

Section 2: STM32CubeWBA main features describes the main features of the STM32CubeWBA MCU Package.
Section 3: STM32CubeWBA architecture overview provides an overview of the STM32CubeWBA architecture and the MCU Package structure.

1 General information

The STM32CubeWBA 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 STM32CubeWBA main features

The STM32CubeWBA MCU Package runs on STM32 32-bit microcontrollers based on the Arm® Cortex®-M33 processor with TrustZone® and FPU.

The STM32CubeWBA gathers, in a single package, all the generic embedded software components required to develop an application for the STM32WBA series microcontrollers. In line with the STM32Cube initiative, this set of components is highly portable, not only within the STM32WBA series microcontrollers but also to other STM32 series.

The STM32CubeWBA is fully compatible with the STM32CubeMX code generator, to generate initialization code. The package includes low-layer (LL) and hardware abstraction layer (HAL) APIs that cover the microcontroller hardware, together with an extensive set of examples running on STMicroelectronics boards. The HAL and LL APIs are available in an open-source BSD license for user convenience.

The STM32CubeWBA MCU Package also contains a comprehensive middleware component constructed around Microsoft® Azure® RTOS middleware, and other in-house and open-source stacks, with the corresponding examples.

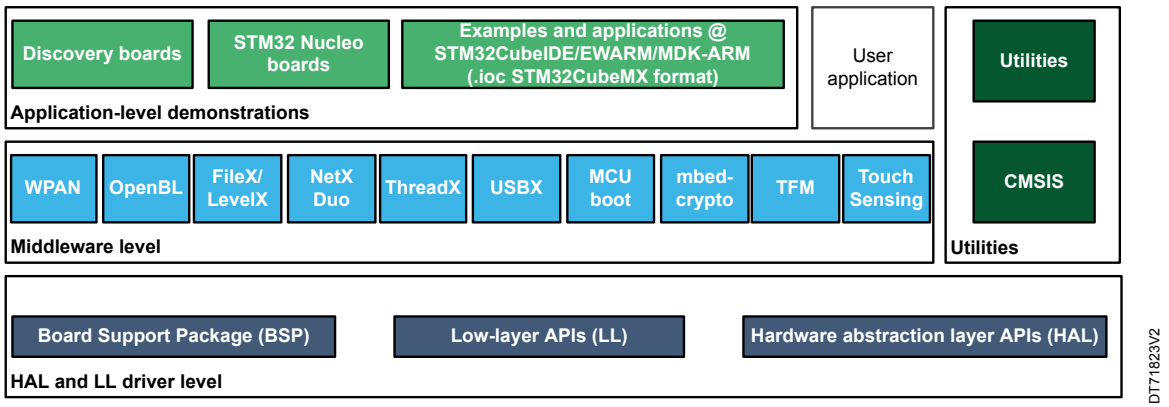
They come with free, user-friendly license terms:

- Integrated and full-featured Azure® RTOS: Azure® RTOS ThreadX
- CMSIS-RTOS implementation with Azure® RTOS ThreadX
- USB Host and Device stacks coming with many classes: Azure® RTOS USBX
- Advanced file system and flash translation layer: FileX, LevelX
- Industrial grade networking stack: optimized for performance coming with many IoT protocols: NetX Duo
- OpenBootloader
- Arm® Trusted Firmware-M (TF-M) integration solution
- mbed-crypto libraries
- ST Netwok Library
- STMTouch touch sensing library solution

Several applications and demonstrations implementing all these middleware components are also provided in the STM32CubeWBA MCU Package.

The STM32CubeWBA MCU Package component layout is illustrated in Figure 1.

Figure 1. STM32CubeWBA MCU Package components



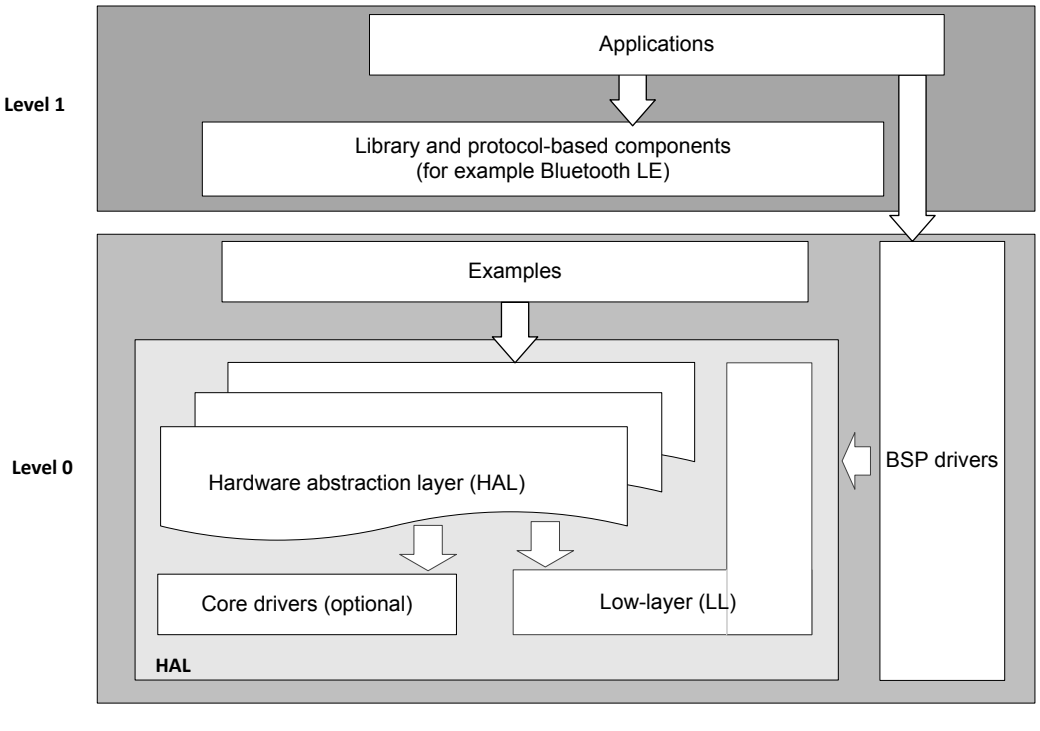
FreeRTOS software expansion

X-CUBE-FREERTOS (FreeRTOS™ software expansion for STM32Cube) provides a full integration of the FreeRTOS™ kernel in the STM32Cube environment for the STM32WBA series microcontrollers. For more information, refer to the *FreeRTOS™ software expansion for STM32Cube* data brief (DB4956), available on st.com.

3 STM32CubeWBA architecture overview

The STM32CubeWBA MCU package solution is built around three independent levels that easily interact as described in Figure 2. STM32CubeWBA MCU package architecture.

Figure 2. STM32CubeWBA 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 (such as LCD, Audio, microSD™, and MEMS drivers). It is composed of two parts:

- Component driver:
  This driver is related to the external device on the board, and not to the STM32 device. The component driver provides specific APIs to theBSP driver external components and could be portable on any other board.
- BSP driver:
  The BSP driver 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()`

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

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

The STM32CubeWBA HAL and LL are complementary and cover a wide range of application 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, which simplify user application implementation by providing ready-to-use processes. For example, for the communication peripherals (I^2S, UART, and others), it provides APIs allowing initializing and configuring the peripheral, managing data transfer based on polling, interrupting, or DMA process, and handling communication errors that may arise during communication. The HAL driver APIs are split into two categories:
  1. Generic APIs, which provide common and generic functions to all the STM32 series microcontrollers.
  2. 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 the 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 lightweight 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 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 to fill initialization data structures with the reset values corresponding to each field.
  - Function for peripheral deinitialization (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 covering Bluetooth® Low Energy (Linklayer, HCI, Stack), Thread, Zigbee®, Matter, OpenBootloader, Microsoft® Azure® RTOS, TF-M, MCUboot, and mbed-crypto.

Horizontal interaction between the components of this layer is done by calling the featured APIs.

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:

- Microsoft® Azure® RTOS
  - Azure® RTOS 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 user mode that allows loading and unloading of prelinked ThreadX modules on the fly through a module manager.
  - NetX Duo
  - FileX
  - USBX
- Bluetooth® Low Energy (BLE): Implements the Bluetooth® Low Energy protocol for the Link and Stack layers.

- MCUboot (open-source software)
- Zigbee® protocols for the stack and related clusters.
- Thread protocol stack and link layer.
- Arm® trusted firmware-M, TF-M (open-source software):
  Reference implementation of the Arm® platform security architecture (PSA) for TrustZone® with the associated secure services.
- mbed-crypto (open-source software):
  The mbed-crypto middleware provides a PSA cryptography API implementation.
- 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 (also called applications) showing how to use it. Integration examples that use several middleware components are provided as well.

4 STM32CubeWBA firmware package overview

4.1 Supported STM32WBA 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 reusability and ensures an easy portability to other devices.

In addition, thanks to its layered architecture, the STM32CubeWBA offers full support of all STM32WBA series. The user has only to define the right macro in `stm32wbaxx.h`.

Table 1 shows the macro to define depending on the STM32WBA series device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32WBA series

Macro defined in stm32wbaxx.h	STM32WBA series devices
stm32wba50xx	STM32WBA50KGU6
stm32wba52xx	STM32WBA52CGU6, STM32WBA52KGU6, STM32WBA52CEU6, STM32WBA52KEU6
stm32wba54xx	STM32WBA54CGU6, STM32WBA54CGU7, STM32WBA54KGU6, STM32WBA54KGU7, STM32WBA54CEU6, STM32WBA54CEU7, STM32WBA54KEU6, STM32WBA54KEU7
stm32wba55xx	STM32WBA55CGU6, STM32WBA55CGU6U, STM32WBA55CGU7, STM32WBA55CEU6, STM32WBA55CEU7
stm32wba62xx	STM32WBA62CIU6, STM32WBA62MIF6, STM32WBA62PII6, STM32WBA62CGU6, STM32WBA62MGF6, STM32WBA62PGI6
stm32wba63xx	STM32WBA63CIU6, STM32WBA63CIU7, STM32WBA63CGU6, STM32WBA63CGU7
stm32wba64xx	STM32WBA64CIU6, STM32WBA64CIU7, STM32WBA64CGU6, STM32WBA64CGU7
stm32wba65xx	STM32WBA65CIU6, STM32WBA65CIU7, STM32WBA65RIV6, STM32WBA65RIV7, STM32WBA65MIF6, STM32WBA65MIF7, STM32WBA65PII6, STM32WBA65PII7, STM32WBA65CGU6, STM32WBA65CGU7, STM32WBA65RGV6, STM32WBA65RGV7, STM32WBA65MGF6, STM32WBA65MGF7, STM32WBA65PGI6, STM32WBA65PGI7

STM32CubeWBA features a rich set of examples and applications at all levels making it easy to understand and use any HAL driver or middleware components. These examples run on the STMicroelectronics boards listed in Table 2.

Table 2. Boards for STM32WBA series

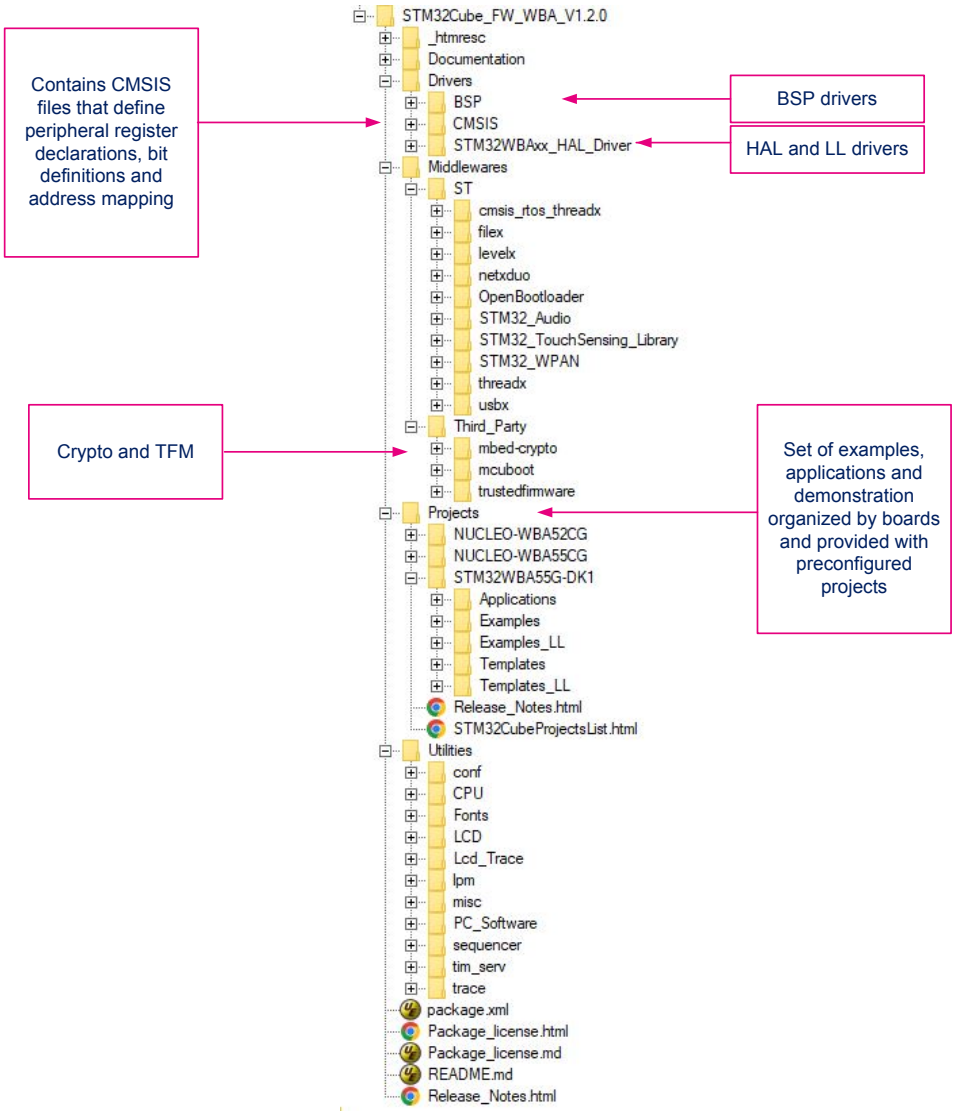
Board	Board STM32WBA supported devices
NUCLEO-WBA55CG	STM32WBA55CGU6
STM32WBA55-DK1	STM32WBA55CGU7
B-WBA5M-WPAN	STM32WBA5MMG
NUCLEO-WBA65RI	STM32WBA65RIV7
STM32WBA65I-DK1	STM32WBA65RIV7

The STM32CubeWBA MCU package can run on any compatible hardware. The user simply updates the BSP drivers to port the provided examples on the board, if the latter has the same hardware features (such as LED, LCD display, and buttons).

4.2 Firmware package overview

The STM32CubeWBA package solution is provided in one single zip package having the structure shown in Figure 3. STM32CubeWBA firmware package structure.

Figure 3. STM32CubeWBA firmware package structure

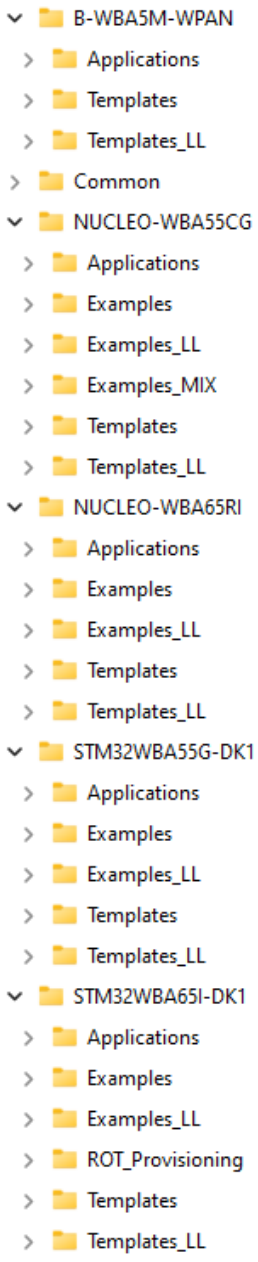


Caution: *The user must not modify the component files. The user can only edit the* `\Projects` *sources.*

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

Figure 4. STM32CubeWBA examples overview shows the project structure for the B-WBA5M-WPAN, NUCLEO-WBA55CG, NUCLEO-WBA65RI, STM32WBA55G-DK1, and STM32WBA65I-DK1 boards.

Figure 4. STM32CubeWBA examples overview



DT71825V3

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

- 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 built thanks to template projects available in the `Templates` and `Templates_LL` directories.

4.2.1 TrustZone® enabled projects

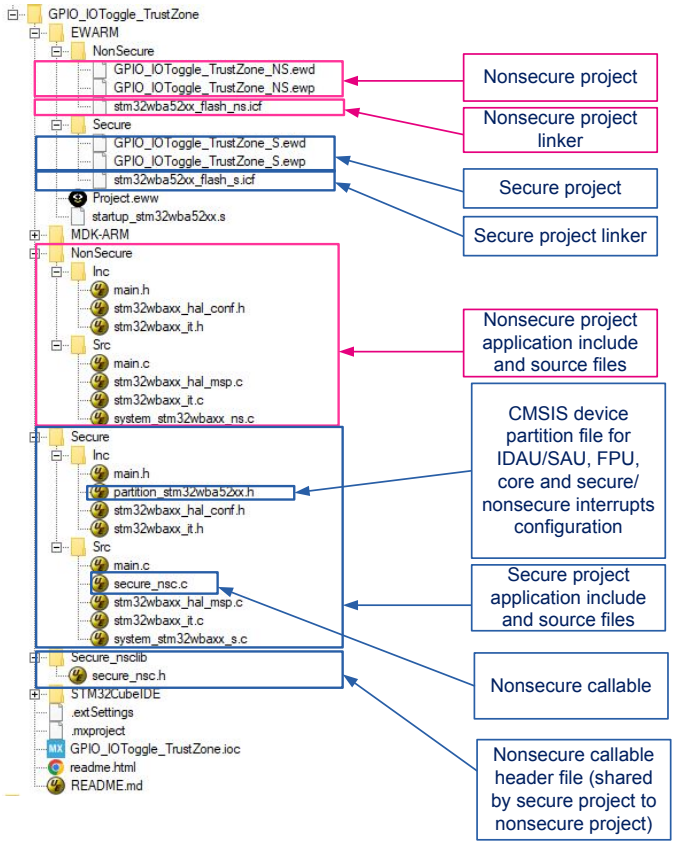
TrustZone® enabled `Examples` names contain the `_TrustZone` prefix. The rule is applied also for `Applications` (except for TFM and SBSFU, which are natively for TrustZone®).

TrustZone®-enabled `Examples` and `Applications` are provided with a multiproject structure composed of secure and nonsecure subprojects as presented in Figure 5. Multiproject secure and nonsecure project structure.

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

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

Figure 5. Multiproject secure and nonsecure project structure



DT71826

The STM32CubeWBA package firmware package provides default memory partitioning in the partition `_<device>.h` files available under: `\Drivers\CMSIS\Device\ST\STM32WBAxx\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 *Secure/nonsecure partitioning using TrustZone® technology* in the RM0495 reference manual.

If the user enables the SAU, a default SAU regions configuration is predefined in partition files as follows:

- SAU region 0: 0x0808 0000 - 0x080F FFFF (nonsecure secure half of flash memory (512 Kbytes)) for STM32WBA5xx
- SAU region 0: 0x0810 0000 - 0x081F FFFF (nonsecure secure half of flash memory (1 Mbyte)) for STM32WBA6xx
- SAU region 1: 0x0BF8 8000 - 0x0BF9 7FFF (nonsecure system memory) for STM32WBA5xx
- SAU region 1: 0x0BF9 0000 - 0x0BFB 7FFF (nonsecure system memory) for STM32WBA6xx
- SAU region 2: 0x0C07 E000 - 0x0C07 FFFF (secure, nonsecure callable) for STM32WBA5xx
- SAU region 2: 0x0C0F E000 - 0x0C0F FFFF (secure, nonsecure callable) for STM32WBA6xx
- SAU region 3: 0x2001 0000 - 0x2001 FFFF (nonsecure SRAM2 (64 Kbytes)) for STM32WBA5xx
- SAU region 3: 0x2004 0000 - 0x2007 FFFF (nonsecure SRAM2 (64 Kbytes)) for STM32WBA6xx
- SAU region 4: 0x4000 0000 - 0x4FFF FFFF (nonsecure peripheral mapped memory)

To match the default partitioning, the STM32WBAxx devices must have the following user option bytes set:

- TZEN = 1 (TrustZone®-enabled device)
- SECWM1_PSTRT = 0x0 SECWM1_PEND = 0x3F (64 out of 128 pages of internal flash memory set as secure) for STM32WBA5xx
- SECWM1_PSTRT = 0x0 SECWM1_PEND = 0x7F (first bank of internal flash memory set as secure) for STM32WBA6xx
- SECWM2_PSTRT = 0x7F SECWM1_PEND = 0x0 (second bank of internal flash memory set as nonsecure) for STM32WBA6xx

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

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` describing the example behavior and needed environment to make it work.
- `*.ioc` file that allows users to open most of the firmware examples within STM32CubeMX.

5 Getting started with STM32CubeWBA

5.1 Running a first HAL example

This section explains how simple it is to run the first example within STM32CubeWBA. It uses as an illustration the generation of a simple LED toggle running on the NUCLEO-WBA55CG board:

1. Download the STM32CubeWBA MCU package.
2. Unzip it into a directory of your choice.
3. Make sure not to modify the package structure shown in Figure 1. It is also recommended to copy the package at a location close to your root volume (meaning `C:\ST` or `G:\Tests`), as 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, which ensures that the security is enabled as described in Section 4.2.1: TrustZone® enabled projects (TZEN=1 (user option byte)).

1. Browse to `\Projects\NUCLEO-WBA55CG\Examples`.
2. Open `\GPIO`, then `\GPIO_IOToggle_TrustZone` folders.
3. Open the project with your preferred toolchain. A quick overview on how to open, build, and run an example with the supported toolchains is given below.
4. Rebuild in sequence all secure and nonsecure project files and load the secure and nonsecure images into the target memory.
5. Run the example: regularly, the secure application toggles LD2 every second, and the nonsecure application toggles LD3 twice as fast. For more details, refer to the readme file of the example.

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

- EWARM:
  1. Under the example folder, open `\EWARM` subfolder.
  2. Launch the Project.eww workspace
  3. Rebuild the xxxxx_S secure project files: [**Project**]>[**Rebuild all**].
  4. Set the xxxxx_NS nonsecure project as Active application (right click on xxxxx_NS project [**Set as Active**])
  5. Rebuild the xxxxx_NS nonsecure project files: [**Project**]>[**Rebuild all**].
  6. Flash the nonsecure binary with [**Project**]>[**Download**]>[**Download active application**] .
  7. Set the xxxxx_S as Active application (right click on xxxxx_S project [**Set as Active**].
  8. Flash the secure binary with the [**Download and Debug**] (Ctrl+D).
  9. Run the program: [**Debug**]>[**Go(F5)**]
- MDK-ARM:
  1. Open the `\MDK-ARM` toolchain.
  2. Open the Multiprojects workspace file `Project.uvmpw`.
  3. Select the xxxxx_s project as Active application ([**Set as Active Project**]).
  4. Build the xxxxx_s project.
  5. Select the xxxxx_ns project as Active project ([**Set as Active Project**]).
  6. Build the xxxxx_ns project.
  7. Load the nonsecure binary ([**F8**]). This downloads `\MDK-ARM\xxxxx_ns\Exe\xxxxx_ns.axf` to flash memory)
  8. Select the Project_s project as Active project ([**Set as Active Project**]).
  9. Load the secure binary ([**F8**]). This downloads `\MDK-ARM\xxxxx_s\Exe\xxxxx_s.axf` to flash memory).
  10. Run the example.
- STM32CubeIDE:
  1. Open the STM32CubeIDE toolchain.
  2. Open the Multiprojects workspace file `.project`.
  3. Rebuild the xxxxx_Secure project.
  4. Rebuild the xxxxx_NonSecure project.
  5. Launch the [**Debug as STM32 Cortex-M C/C++**] application for the secure project.
  6. In the[**Edit configuration**] window, select the [**Startup**] panel, and add load the image and symbols of the nonsecure project.

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

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

5.1.2 Running a first TrustZone® disabled example

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

1. Browse to `\Projects\NUCLEO-WBA55CG\Examples`.
2. Open `\GPIO`, then `\GPIO_EXTI` folders.
3. Open the project with your preferred toolchain. A quick overview on how to open, build, and run an example with the supported toolchains is given below.
4. Rebuild all files and load your image into the target memory.
5. Run the example: Each time the [**USER**] push-button is pressed, the LD1 LED toggles. For more details, refer to the readme file of the example.

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

- EWARM:
  1. Under the example folder, open `\EWARM` subfolder.
  2. Launch the Project.eww workspace (the workspace name may change from one example to another).
  3. Rebuild all files: [**Project**]>[**Rebuild all**].
  4. Load the project image: [**Project**]>[**Debug**].
  5. Run program: [**Debug**]>[**Go (F5)**].
- MDK-ARM:
  1. Under the example folder, open the `\MDK-ARM` subfolder.
  2. Launch the Project.uvproj workspace (the workspace name may change from one example to another).
  3. Rebuild all files:[**Project**]>[**Rebuild all target files**].
  4. Load the project image: [**Debug**]>[**Start/Stop Debug Session**].
  5. Run program: [**Debug**]>[**Run (F5)**].
- STM32CubeIDE:
  1. Open the STM32CubeIDE toolchain.
  2. Click [**File**]>[**Switch Workspace**]>[**Other**] and browse to the STM32CubeIDE workspace directory.
  3. Click [**File**]>[**Import**] , select [**General**]>[**Existing Projects into Workspace**], and then click [**Next**].
  4. Browse to the STM32CubeIDE workspace directory and select the project.
  5. Rebuild all project files: Select the project in the [**Project Explorer**] window then click the [**Project**]>[**Build project**] menu.
  6. Run the program: [**Run**]>[**Debug (F11)**]

5.2 Developing a custom application

Note: Software must enable the instruction cache (ICACHE) to get a 0 wait-state execution from flash memory, and reach the maximum performance and a better power consumption.

5.2.1 Using STM32CubeMX to develop or update an application

In the STM32CubeWBA MCU package, nearly all project examples are generated with the STM32CubeMX tool to initialize the system, peripherals, and middleware.

The direct use of an existing project example from the STM32CubeMX tool requires STM32CubeMX 6.10.0 or higher:

- After the installation of STM32CubeMX, open and if necessary update a proposed project. The simplest way to open an existing project is to double-click on the `*.ioc` file so that STM32CubeMX automatically opens the project and its source files.
- STM32CubeMX generates the initialization source code of such projects. The main application source code is contained by the comments “USER CODE BEGIN” and “USER CODE END”. In case the IP selection and setting are modified, STM32CubeMX updates the initialization part of the code but preserves the main application source code.

For developing a custom project in the STM32CubeMX, follow the step-by-step process:

1. Select the STM32 microcontroller that matches the required set of peripherals.
2. Configure all the required embedded software using a pinout-conflict solver, a clock-tree setting helper, a power consumption calculator, and the utility performing MCU peripheral configuration (such as GPIO or USART) and middleware stacks (such as USB).
3. Generate the initialization C code based on the selected configuration. This code is ready to use within several development environments. The user code is kept at the next code generation.

For more information about STM32CubeMX, refer to the user manual *STM32CubeMX for STM32 configuration and initialization C code generation* (UM1718) and the STM32CubeMX wiki page.

For a list of the available project examples for STM32CubeWBA, refer to the application note *STM32Cube firmware examples for STM32WBA series* (AN5929).

5.2.2 Driver applications

5.2.2.1 HAL application

This section describes the steps required to create a custom HAL application using STM32CubeWBA:

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 STM32CubeWBA).
   The `Template` project provides an empty main loop function. However, it is a good starting point to understand the STM32CubeWBA project settings. The template has the following characteristics:
   – It contains the HAL source code, CMSIS, and BSP drivers, which are the minimum set of components required to develop a code on a given board.
   – It contains the included paths for all the firmware components.
   – It defines the supported STM32WBA series devices, allowing the CMSIS and HAL drivers to be configured correctly.
   – It provides ready-to-use user files preconfigured as shown below:
   HAL initialized with the default time base with Arm® core SysTick.
   SysTick ISR implemented for `HAL_Delay()` purpose.

Note: When copying an existing project to another location, make sure all the included paths are updated.

2. **Add the necessary middleware to the user project (optional)**
   To identify the source files to be added to the project file list, refer to the documentation provided for each middleware. Refer to the applications under `\Projects\STM32xxx_yyy\Applications\<MW_Stack>` (where *<MW_Stack>* refers to the middleware stack, such as ThreadX) to know which source files and include paths must be added.
3. **Configure the firmware components**
   The HAL and middleware components offer a set of build-time configuration options using macros `#define` declared in a header file. A template configuration file is provided within each component, which has to be copied to the project folder (usually the configuration file is named `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 `stm32wbaxx_hal_conf.h`).
   b. Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority `TICK_INT_PRIO` defined in `stm32wbaxx_hal_conf.h`.
   c. Setting of NVIC group priority to 0.
   d. Call of `HAL_MspInit()` callback function defined in `stm32wbaxx_hal_msp.c` user file to perform global low-level hardware initializations.
5. **Configure the system clock**
   The system clock configuration is done by calling the two APIs described below:
   – `HAL_RCC_OscConfig()`: this API configures the internal and external oscillators. The user chooses to configure one or all oscillators.
   – `HAL_RCC_ClockConfig()`: this API configures the system clock source, the flash memory latency, and AHB and APB prescalers.

6. **Initialize the peripheral**
   a. First write the peripheral `HAL_PPP_MspInit` function. Proceed as follows:
      - Enable the peripheral clock.
      - Configure the peripheral GPIOs.
      - Configure the DMA channel and enable DMA interrupt (if needed).
      - Enable peripheral interrupt (if needed).

   b. Edit the `stm32xxx_it.c` to call the required interrupt handlers (peripheral and DMA), if needed.
   c. Write process complete callback functions, if a peripheral interrupt or DMA is planned to be used.
   d. In user `main.c` file, initialize the peripheral handle structure then call the function `HAL_PPP_Init()` to initialize the peripheral.
7. **Develop an application**
   At this stage, the system is ready and the user application code development can start.
   The HAL provides intuitive and ready-to-use APIs to configure the peripheral. It supports polling, interrupts, and a DMA programming model, to accommodate any application requirements. For more details on how to use each peripheral, refer to the rich example set provided in the STM32CubeWBA MCU package.

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

5.2.2.2 *LL application*

This section describes the steps needed to create a custom LL application using STM32CubeWBA.

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-WBA55CG).
   The template project provides an empty main loop function, which is a good starting point to understand the project settings for STM32CubeWBA. Template main characteristics are the following:
   – It contains the source codes of the LL and CMSIS drivers, which are the minimum set of components needed to develop code on a given board.
   – It contains the included paths for all the required firmware components.
   – It selects the supported STM32WBA series device and allows the correct configuration of the CMSIS and LL drivers.
   – It provides ready-to-use user files that are preconfigured as follows:
      - `main.h`: LED and USER_BUTTON definition abstraction layer.
      - `main.c`: System clock configuration for maximum frequency.
2. **Port an existing project to another board**
   To support an existing project on another target board, start from the Templates_LL project provided for each board and available under `\Projects\<STM32xxx_yyy>\Templates_LL`.
   – Select an LL example: To find the board on which LL examples are deployed, refer to the list of LL examples `STM32CubeProjectsList.html`.

3. Port the LL example:
   - Copy/paste the `Templates_LL` folder - to keep the initial source - or directly update the existing `Templates_LL` project.
   - Then porting consists principally in replacing `Templates_LL` files by the `Examples_LL` targeted project.
   - Keep all board specific parts. For reasons of clarity, board specific parts are flagged with specific tags:

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

     Thus, the main porting steps are the following:
     - Replace the `stm32wbaxx_it.h` file
     - Replace the `stm32wbaxx_it.c` file
     - Replace the `main.h` file and update it: Keep the LED and user button definition of the LL template under `BOARD SPECIFIC CONFIGURATION` tags.
     - Replace the `main.c` file and update it:
       Keep the clock configuration of the `SystemClock_Config()` LL template function under `BOARD SPECIFIC CONFIGURATION` tags.
       Depending on the LED definition, replace each LDx occurrence with another LDy available in the `main.h` file.

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

5.2.3 Security applications

This package is delivered with security applications.

5.2.3.1 *OEMxRoT applications*

OEMxRoT provides a Root of Trust solution, including Secure Boot and Secure Firmware Update functionalities (based on MCUboot).

- OEMiRoT stands for OEM immutable (unchangeable) Root of Trust and acts as a first boot stage.
- OEMuRoT (OEM updatable Root of Trust) acts as an optional second boot stage after OEMiRoT.

Both OEMiRoT and OEMuRoT provide the same two services: Secure Boot and Secure Firmware Update.

The solution is used before executing the application and provides an example of a secure service (GPIO toggle), that is isolated from the nonsecure application. The nonsecure application can still use this solution at runtime.

5.2.3.2 *TF-M applications*

The Trusted Firmware-M (TF-M) is an open-source framework that implements the Secure Processing Environment (SPE) for Armv8-M and Armv8.1-M architectures (such as the Cortex®-M33, Cortex®-M55, and Cortex®-M85 processors) or dual-core platforms. It is the platform security architecture reference implementation aligning with PSA Certified™ guidelines, enabling chips, real-time operating systems, and devices to become PSA certified.

STMicroelectronics implemented TF-M integration at its origin in cooperation with the open-source community. The decision to deliver it with the STM32Cube MCU Package and certify it was done to help developers adopt it in a phase of TF-M development and maturation.

TF-M is now stable and mature and shall be available and distributed directly by the TrustedFirmware.org community. STMicroelectronics maintains the delivery of the main HAL and CMSIS device drivers to the TF-M GitHub to make sure the products are compatible with TF-M.

The benefits for the user include:

- Choosing the TF-M version.
- Configuring TF-M according to the application requirements.
- Maintenance and security updates from the community.

For more information on using TF-M on STM32 devices, refer to the following link: https://tf-m-user-guide.trustedfirmware.org/platform/stm/

5.2.3.3 *RF applications*

The RF application is described in this application note: *Building wireless applications with STM32WBA series microcontrollers* (AN5928).

5.3 Getting STM32CubeWBA release updates

The latest STM32CubeWBA MCU package releases and patches are available from STM32WBA Series. They may be retrieved from the `CHECK FOR UPDATE` button in STM32CubeMX. For more details, refer to *Section 3* of the user manual *STM32CubeMX for STM32 configuration and initialization C code generation* (UM1718).

6 FAQ

6.1 When should I use HAL instead of LL drivers?

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

LL drivers offer low-layer register level APIs, with better optimization but less portable. They require in-depth knowledge of product or IP specifications.

6.2 Can I use HAL and LL drivers together? If I can, 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 operate directly on peripheral registers. The *Examples_MIX* example illustrates how to mix HAL and LL.

6.3 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.

6.4 How can STM32CubeMX generate code based on embedded software?

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

Revision history

Table 3. **Document revision history**

Date	Revision	Changes
01-Feb-2023	1	Initial release.
16-Oct-2023	2	Updated the following sections: • Section Introduction • Section 4.1: Supported STM32WBA series devices and hardware • Section 3.2.1: Middleware components • Section 4: STM32CubeWBA firmware package overview • Section 5.2.1: Using STM32CubeMX to develop or update an application • Section 4.2.1: TrustZone® enabled projects Updated the whole document with minor terminology changes.
21-Nov-2024	3	Updated: • Figure 1. STM32CubeWBA MCU Package components • Table 1. Macros for STM32WBA series • Table 2. Boards for STM32WBA series Added FreeRTOS software expansion.
12-Feb-2025	4	Updated: • Section 4.1: Supported STM32WBA series devices and hardware • Section 4.2: Firmware package overview • Section 4.2.1: TrustZone® enabled projects • Section 5.2.1: Using STM32CubeMX to develop or update an application • Section 5.2.3.1: OEMxRoT applications • Section 5.2.3.2: TF-M applications

Contents

List of tables

List of figures