Getting started with STM32CubeWB0 for STM32WB0 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 STM32CubeWB0 for the STM32WB0 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 FreeRTOS™, FAT file system, and STM32_BLE (Bluetooth® Low Energy)
  - 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 STM32CubeWB0 MCU Package.

Section 2 describes the main features of the STM32CubeWB0 MCU Package.

Section 3 provides an overview of the STM32CubeWB0 MCU Package structure and architecture.

1 General information

The STM32CubeWB0 MCU Package runs on ultra-low-power programmable STM32WB0 series microcontrollers based on the Arm® Cortex®-M0+ processor, and supporting the Bluetooth® Low Energy stack.

STM32WB0 series microcontrollers embed STMicroelectronics' state-of-the-art 2.4 GHz RF radio peripheral compliant with the Bluetooth® Low Energy specification, optimized for ultra-low-power consumption and excellent radio performance, for unparalleled battery lifetime.

For information on Bluetooth®, refer to www.bluetooth.com.

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

2 STM32CubeWB0 main features

The STM32CubeWB0 MCU Package runs on STM32WB0 series devices, based on the Arm® Cortex®-M0+ processor with Bluetooth® Low Energy radio.

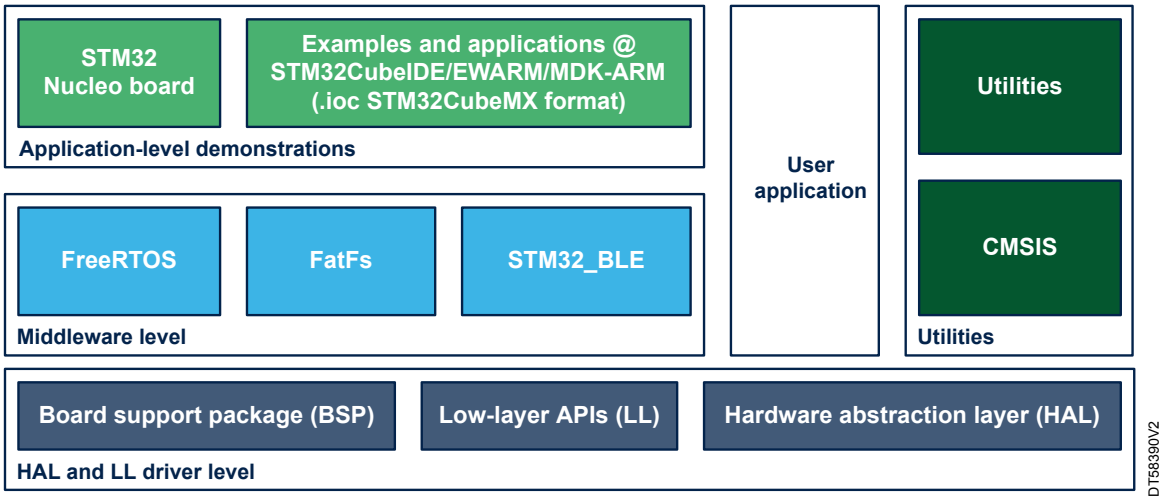
STM32CubeWB0 gathers in a single package all the generic embedded software components required to develop an application for STM32WB0 series microcontrollers.

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. It also includes the STM32_BLE (Bluetooth® Low Energy), FatFs, and FreeRTOS™ middleware components.

Several applications implementing all these middleware components are also provided in the STM32CubeWB0 MCU Package.

The STM32CubeWB0 MCU Package component layout is illustrated in Figure 1. STM32CubeWB0 MCU Package components.

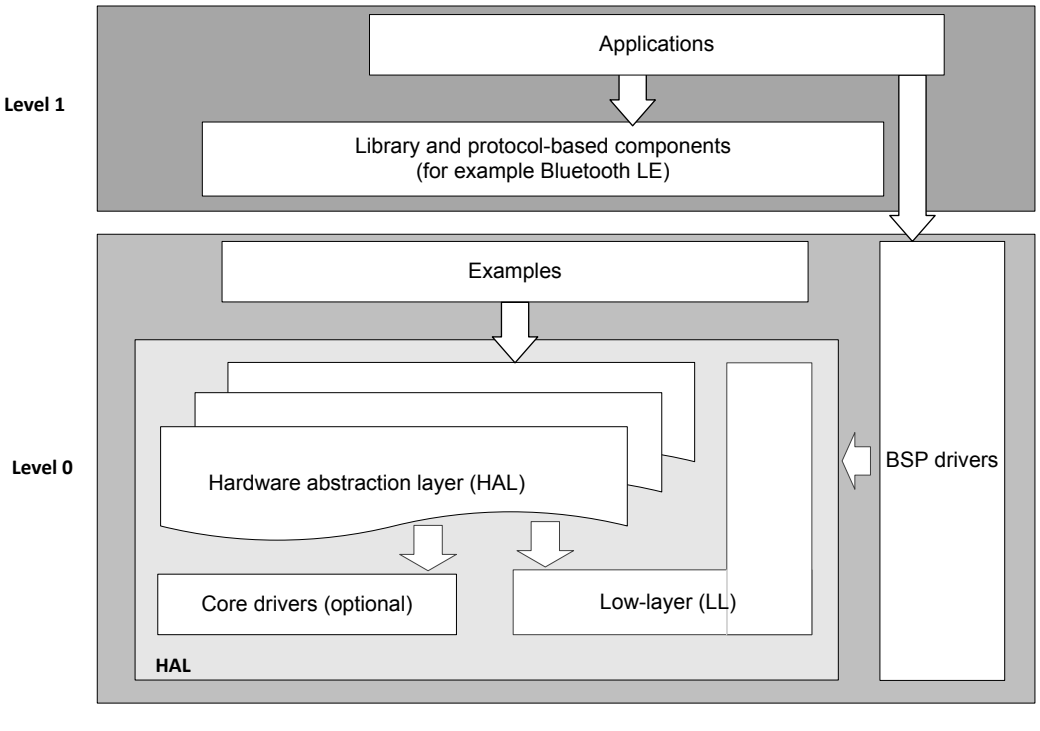
Figure 1. STM32CubeWB0 MCU Package components



3 STM32CubeWB0 architecture overview

The STM32CubeWB0 MCU Package solution is built around three independent levels that easily interact, as described in Figure 2.

Figure 2. STM32CubeWB0 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 LEDs, , buttons, COM 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 of 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 STM32CubeWB0 MCU Package 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 from 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 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.

Note: Demonstration examples are also available in order to show more complex example scenarios with specific IPs, such as the 2.4 GHz radio.

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, FreeRTOS™, and FatFS.

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:

- STM32_BLE: implements the Bluetooth® Low Energy protocol layers (controller and host).
- FreeRTOS™: implements a real-time operating system (RTOS), designed for embedded systems.
- FatFS implements the generic FAT filesystem module.

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.

4 STM32CubeWB0 firmware package overview

4.1 Supported STM32WB0 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 guarantees easy portability to other devices.

In addition, thanks to its layered architecture, STM32CubeWB0 offers full support of the STM32WB0 series. The user only has to define the right macro in `stm32wb0xx.h`.

Table 1 shows the macro to define depending on the STM32WB0 series device available and used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32WB0 series

Macro defined in *stm32wb0x.h*	STM32WB0 series devices
STM32WB05	STM32WB05KZV6, STM32WB05TZF6, STM32WB05KZV7, STM32WB05TZF7
STM32WB07/STM32WB06	STM32WB07CCV6, STM32WB06CCV6, STM32WB07KCV6, STM32WB06KCV6, STM32WB07CCF6, STM32WB06CCF6, STM32WB07CCV7, STM32WB06CCV7, STM32WB07KCV7, STM32WB06KCV7, STM32WB07CCF7, STM32WB06CCF7
STM32WB09	STM32WB09KEV6, STM32WB09TEF6, STM32WB09KEV7, STM32WB09TEF7

STM32CubeWB0 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 below.

Table 2. Boards for STM32WB0 series

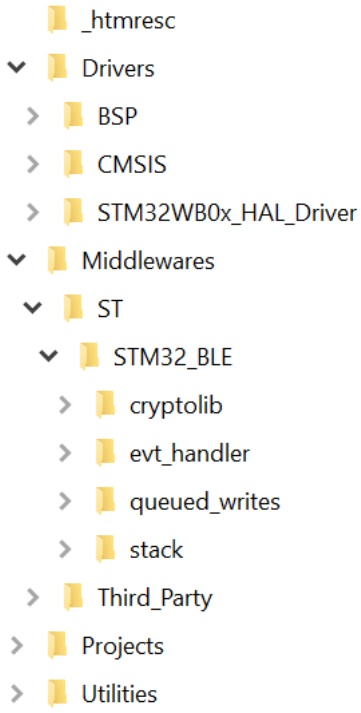
Board	STM32WB0 board supported devices
NUCLEO-WB05KZ	STM32WB05KZV6
NUCLEO-WB07CC	STM32WB07CCV6
NUCLEO-WB09KE	STM32WB09KEV6

The STM32CubeWB0 MCU Package can run on any compatible hardware. The user simply updates the BSP drivers to port the provided examples on his board, if the latter has the same hardware features (such as LED, and buttons).

4.2 Firmware package overview

The STM32CubeWB0 MCU Package is provided in one single zip package, having the structure shown in Figure 3.

Figure 3. STM32CubeWB0 MCU Package structure



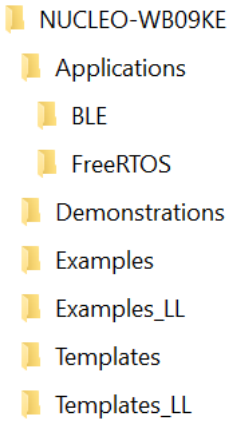
DT58391V3

Caution: The user must not modify the components files. Only the `\Projects` sources can be edited by the user.

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

Figure 4 shows the project structure for the NUCLEO-WB09KE board.

Figure 4. STM32CubeWB0 examples overview



DT58392V1

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. Demonstration examples are also available.
- Level 1 examples are called Applications. They provide typical use cases of each middleware component.

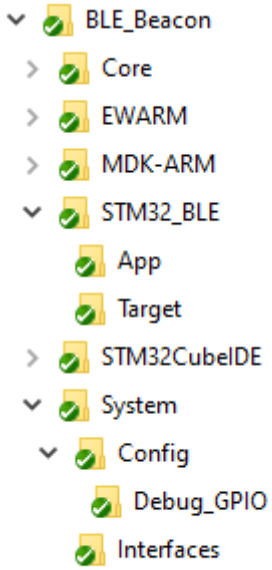
Any LL and HAL firmware examples for a given board can be built quickly with the template projects available in the `Templates` and `Templates_LL` directories.

All *Examples*, *Examples_LL*, and *Examples_MIX* 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.

The STM32_BLE applications have a more structured architecture to address the required components from the Bluetooth® Low Energy stack

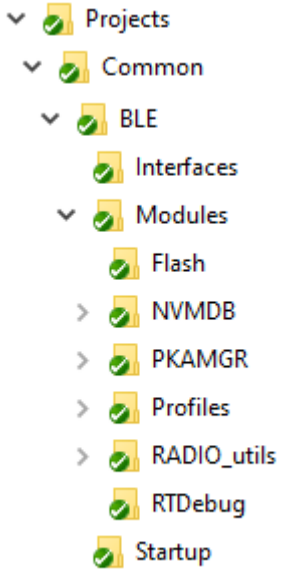
Figure 5. STM32_BLE stack application structure



DT58393V2

Note: Starting from STM32CubeWB0 v1.1.0, some functionalities required for the STM32_BLE applications have been centralized in the `Projects/Common/BLE` folder, shown in Figure 6.

Figure 6. **STM32_BLE stack common BLE folder**



DT75913V1

5 Getting started with the STM32CubeWB0 MCU Package

5.1 Running a first HAL example

This section explains how simple is to run the first example within STM32CubeWB0 MCU Package. It uses as an illustration the generation of a simple LED toggle running on the NUCLEO-WB09KE board:

1. Download the STM32CubeWB0 MCU Package.
2. Unzip it or run installer (if provided), into a directory of your choice.
3. Make sure not to modify the package structure shown in Figure 2. STM32CubeWB0 MCU Package architecture.

5.1.1 How to run a HAL example

Before loading and running an example, it is strongly recommended to read the example readme file for any specific configuration.

1. Browse to `\Projects\NUCLEO-WB09KE\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 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 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 and go to the [**Project**]>[**Build project**] menu.
  6. Run the program: [**Run**]>[**Debug (F11)**].

5.2 Developing a custom application

5.2.1 Using STM32CubeMX to develop or update an application

In the STM32CubeWB0 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 6.12.0 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 with 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).
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).

For a list of the available project examples for STM32CubeWB0, refer to the application note *STM32Cube MCU Package examples for STM32WB0 series* (AN5976).

5.2.2 Driver applications

5.2.2.1 HAL application

This section describes the steps required to create a custom HAL application using STM32CubeWB0:

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` (where `<STM32xxx_yyy>` refers to the board name).
   The `Template` project provides an empty main loop function. However, it is a good starting point to understand the STM32CubeWB0 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 STM32WB0 series devices, allowing the CMSIS and HAL drivers to be configured correctly.
   – It provides ready-to-use user files pre-configured as shown below: HAL initialized with default time base with Arm core SysTick. SysTickISR implemented for `HAL_Delay()` purpose.

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

2. **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.

3. **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 SysTick interrupt priority (through macros defined in `stm32wb0x_hal_conf.h`.
   b. Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority defined in `stm32wb0x_hal_conf.h`.
   c. Setting of NVIC group priority.
   d. Call of `HAL_MspInit()` callback function defined in `stm32wb0x_hal_msp.c` user file to perform global low-level hardware initializations.
4. **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.
5. **Initialize the peripheral**
   a. First write the peripheral initialization 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.
   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 STM32CubeWB0 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 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 STM32CubeWB0.

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-WB09KE).
   The template project provides an empty main loop function, which is a good starting point to understand the project settings for STM32CubeWB0. 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 STM32WB0 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 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 `stm32wb0x_it.h` file
   ◦ Replace the `stm32wb0x_it.c` file
   ◦ Replace the `main.h` file and update it: Keep the LED and user button definition of the LL template under `BOARD SPECIFIC CONFIGURATION` tags.
   ◦ Replace the `main.c` file and update it:
   Keep the clock configuration of the `SystemClock_Config()` LL template function under `BOARD SPECIFIC CONFIGURATION` tags.
   Depending on the LED definition, replace each LDx occurrence with another LDy available in the `main.h` file.

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

5.3 RF applications

5.3.1 Running a first STM32_BLE application

This section explains how to run a first STM32_BLE application with the STM32CubeWB0 MCU Package. It uses as an illustration the generation of a simple beacon advertising running on the NUCLEO-WB09KE board.

1. Download the STM32CubeWB0 MCU Package.
2. Unzip it or run an installer (if provided), into a directory of your choice.
3. Make sure not to modify the package structure shown in Figure 1. STM32CubeWB0 MCU Package components.

5.3.2 How to run an STM32_BLE application

Before loading and running an example, it is strongly recommended to read the example readme file for any specific configuration.

1. Browse to `\Projects\NUCLEO-WB09KE\Applications`.
2. Open the `\BLE`, then `\BLE_Beacon` folder
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: the beacon advertising message can be discovered using a beacon scanner app on a mobile phone. For more details, refer to the readme file of the example.

To open, build and run an application 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 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 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 and go to the [**Project**]>[**Build project**] menu.
  6. Run the program: [**Run**]>[**Debug (F11)**].

Caution: The BLE_Beacon application also configures the STM32WB0 series device in order to enter Deepstop mode when allowed.

When the STM32WB0 series device is in Deepstop mode, the SWD port is disabled. As a consequence, SWD access is disabled and the chip cannot be accessed. In this context, the user can activate the device UART bootloader by setting the PA10 pin to a high value and perform a reset cycle of the device (keeping PA10 at a high value). For example, when using a NUCLEO-WB09KE board, the user can simply fit jumper JP1 in bootloader position and perform a hardware reset through button B4. This allows the activation of the UART bootloader and programs a new application through the SWD. After programming, the user must remove the JP1 jumper.

An alternative approach is to perform a mass erase of the device using the STM32CubeProgrammer tool in UART mode, and then program a new application through the SWD.

5.3.3 Using STM32CubeMX to develop or update an STM32_BLE application

In the STM32CubeWB0 MCU Package, nearly all STM32_BLE 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 6.12.0 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 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 STM32_BLE).
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).

For a list of the available project examples for STM32CubeWB0, refer to the application note *STM32Cube MCU Package examples for STM32WB0 series* (AN5976).

For further information about the steps required to build specific Bluetooth® Low Energy applications based on STM32WB0 series microcontrollers, refer to the application note *How to build a short-range wireless application with STM32WB0 MCUs* (AN5977).

5.3.4 Developing a custom STM32_BLE application

This section describes the steps required to create a custom STM32_BLE application using the STM32CubeWB0 MCU Package.

To create a new project, either start from the BLE_Skeleton project provided for each board under `\Projects\<STM32xxx_yyy>\Applications\BLE` or from any available project under `\Projects\<STM32xx_yyy>\Applications\BLE` (where `<STM32xxx_yyy>` refers to the board name).

The available STM32CubeWB0 MCU Package projects provide the overall application infrastructure for targeting a Bluetooth® Low Energy scenario:

- It contains the HAL source code and 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 STM32WB0 series devices, allowing the correct configuration of the CMSIS and HAL drivers.
- It provides all the system folder components allowing correct support of the Bluetooth® Low Energy stack.
- It provides the middleware folder components supporting the Bluetooth® Low Energy stack.

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

6 FAQ

6.1 When to 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 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 operate directly on peripheral registers. Mixing HAL and LL is illustrated in the *Examples_MIX* example.

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 Is there any template project for 2.4 GHz IP examples?

To create a new 2.4 GHz example project, either start from the RADIO_Skeleton project provided for each board under `\Projects\<STM32xxx_yyy>\Examples\RADIO` or from any available project under `\Projects\<STM32xxy_y yy>\Examples\RADIO (<STM32xxx_yyy>` refers to the board name, such as NUCLEO-WB09KE).

6.5 Which demonstration examples are available?

2.4 GHz IP demonstrations examples are provided within the `\Projects\<STM32xxx_yyy>\Demonstrations\RADIO` folder with a reference template: `RADIO_Skeleton (<STM32xxx_yyy>` refers to the board name, such as NUCLEO-WB09KE).

The RADIO demonstrations examples target complex and advanced 2.4 GHz scenarios (that is, low power management support, over the air firmware upgrade capability).

RADIO_TIMER demonstrations examples are also provided to show the 2.4 GHz radio timer basic capabilities.

6.6 How can STM32CubeMX generate code based on embedded software?

STM32CubeMX has built-in knowledge of STM32 microcontrollers, including their peripherals and software, which allows it to provide 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
15-May-2024	1	Initial release.
03-Oct-2024	2	Updated Section 4.2: Firmware package overview and Figure 5. STM32_BLE stack application structure. Added Figure 6. STM32_BLE stack common BLE folder.

Contents

List of tables

List of figures