Getting started with STM32CubeWL3 software package for STM32WL3x microcontrollers

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 STM32CubeWL3 for the STM32WL3x product line), 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™ kernel, FatFS, and Sigfox™
  - 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 STM32CubeWL3 MCU Package.

Section 2 describes the main features of STM32CubeWL3 and Section 3 provides an overview of its architecture and of the MCU Package structure.

1 General information

STM32CubeWL3 runs sub-GHz demonstration applications, including Sigfox™ binaries, on STM32WL3x product line microcontrollers based on the Arm® Cortex®-M0+ processor.

The STM32WL3x microcontrollers embed STMicroelectronics's state-of-the-art sub-GHz compliant RF radio peripheral, optimized for ultra-low-power consumption and excellent radio performance, for unparalleled battery lifetime.

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

2 STM32CubeWL3 main features

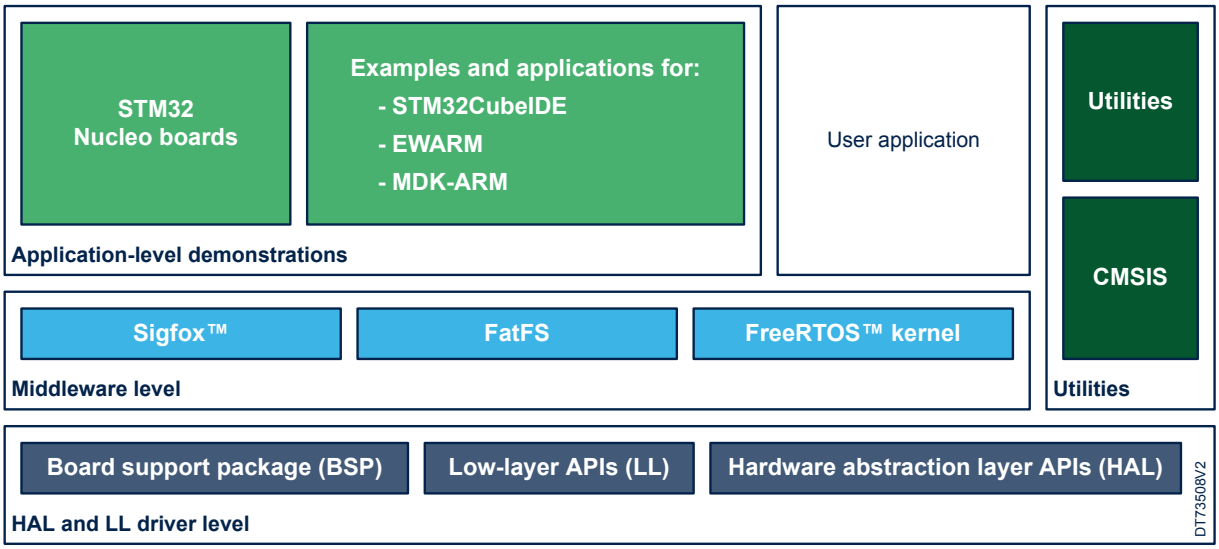
The STM32CubeWL3 MCU Package runs on STM32 32-bit microcontrollers based on the Arm® Cortex®-M0+ processor. It gathers, in a single package, all the generic embedded software components required to develop an application for the STM32WL3x product line microcontrollers.

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 Sigfox™, FatFS, and FreeRTOS™ kernel middleware components.

The STM32CubeWL3 MCU Package also provides several applications and demonstrations implementing all its middleware components.

The STM32CubeWL3 MCU Package component layout is illustrated in Figure 1.

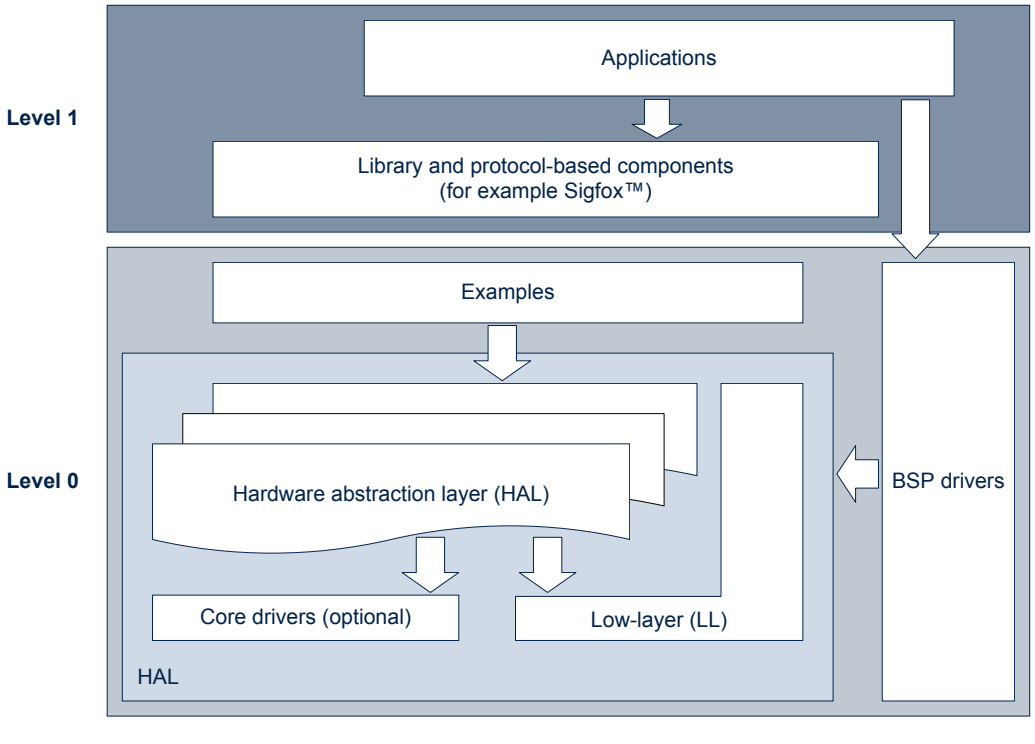
Figure 1. STM32CubeWL3 MCU Package components



3 STM32CubeWL3 architecture overview

The STM32CubeWL3 MCU Package solution is built around three independent levels that easily interact as described in Figure 2.

Figure 2. STM32CubeWL3 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, and 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 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 STM32CubeWL3 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 the user application implementation by providing ready-to-use processes. For example, for the communication peripherals (I^2C, 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 the 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 to show more complex example scenarios with specific peripherals, such as the MRSUBG and LPAWUR.

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 the FreeRTOS™ kernel, FatFS, and Sigfox™ protocol library.

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:

- FreeRTOS™ kernel: implements a real-time operating system (RTOS), designed for embedded systems.
- Sigfox™: implements the Sigfox™ protocol library compliant with the Sigfox™ protocol network and includes the RF test protocol library to test against RF Sigfox™ tools.
- FatFS: implements the generic FAT file system module.

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 STM32CubeWL3 firmware package overview

4.1 Supported STM32WL3x 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 easy portability to other devices.

In addition, with its layered architecture, STM32CubeWL3 offers full support for all the STM32WL3x product line. The user must only define the right macro in `stm32wl3x.h`.

Table 1 shows the macro to define depending on the STM32WL3x product line device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32WL3x product line

Macro defined in `stm32wl3x.h`	STM32WL3x product line devices
stm32wl33	STM32WL33xx microcontrollers

STM32CubeWL3 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 STM32WL3x product line

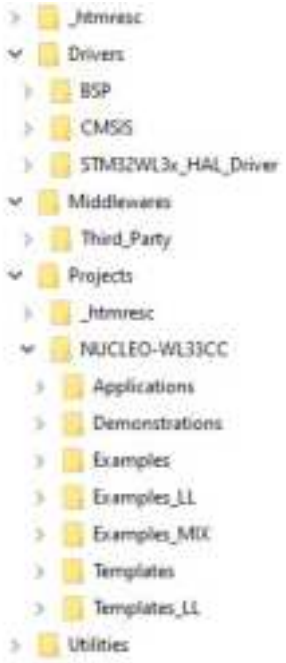
Board	STM32WL3x board supported devices
NUCLEO-WL33CC1	STM32WL33CC
NUCLEO-WL33CC2	STM32WL33CC

The STM32CubeWL3 MCU Package can run on any compatible hardware. The users simply update the BSP drivers to port the provided examples on their boards, if these have the same hardware features (such as LEDs or buttons).

4.2 Firmware package overview

The STM32CubeWL3 MCU Package solution is provided in one single zip package having the structure shown in Figure 3.

Figure 3. STM32CubeWL3 firmware package structure



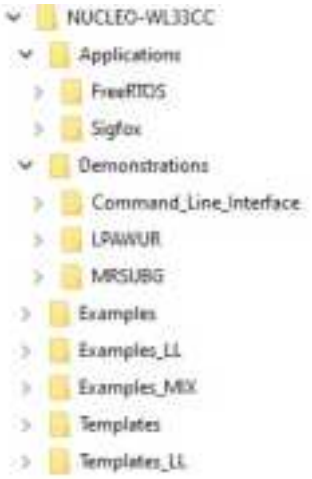
DT73511V2

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

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-WL33CCx boards.

Figure 4. STM32CubeWL3 examples overview



DT73512V2

The examples are classified depending on the STM32CubeWL3 level that they apply to. 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 firmware application for a given board can be quickly built using the template projects available in the `Templates` and `Templates_LL` directories.

Examples, *Examples_LL*, and *Examples_MIX* have the same structure:

- `\Inc` folder containing all the 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.

5 Getting started with STM32CubeWL3

5.1 Running a first example

This section explains how simple it is to run a first example within STM32CubeWL3. It uses as an illustration the generation of a simple LED toggle running on the NUCLEO-WL33CC1 board:

1. Download the STM32CubeWL3 MCU Package.
2. Unzip it, or run the installer if provided, into a directory of your choice.
3. Make sure not to modify the package structure shown in Figure 3. STM32CubeWL3 firmware package structure. Note that it is also recommended to copy the package at a location close to the root volume (meaning `C:\ST` or `G:\Tests`), as some IDEs encounter problems when the path is too long.

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-WL33CC\Examples`.
2. Open the `\GPIO`, then `\GPIO_EXTI` 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 all files and load the image into the target memory.
5. Run the example. For more details, refer to the example readme file.

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

- EWARM:
  1. Under the `Examples` folder, open the `\EWARM` subfolder.
  2. Launch the *Project.eww* workspace (the workspace name might change from one example to another).
  3. Rebuild all files: [**Project**]>[**Rebuild all**].
  4. Load the project image: [**Project**]>[**Debug**].
  5. Run the program: [**Debug**]>[**Go (F5)**].
- MDK-ARM:
  1. Under the `Examples` folder, open the `\MDK-ARM` subfolder.
  2. Open the *Project.uvproj* workspace (the workspace name might 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 the program: [**Debug**]>[**Run (F5)**].
- STM32CubeIDE:
  1. Open the STM32CubeIDE toolchain.
  2. Click on [**File**]>[**Switch Workspace**]>[**Other**] and browse to the STM32CubeIDE workspace directory.
  3. Click on [**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 on 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 STM32Cube 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`". If the peripheral selection and settings are modified, STM32CubeMX updates the initialization part of the code while preserving the main application source code.

To develop a custom project with STM32CubeMX, follow the step-by-step process:

1. 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).
2. 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).

5.2.2 Driver applications

5.2.2.1 HAL application

This section describes the steps required to create a custom HAL application using STM32CubeWL3:

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). The `Template` project provides an empty main loop function. However, it is a good starting point to understand the STM32CubeWL3 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 STM32WL3x product line 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 the 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. **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 must be copied to the project folder (usually the configuration file is named `xxx_conf_template.h`, the fragment *_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 the `HAL_Init()` API to initialize the HAL library, which carries out the following tasks:
   a. Configuration of the flash memory prefetch and SysTick interrupt priority (through macros defined in `stm32wl3x_hal_conf.h`).
   b. Configuration of the SysTick to generate an interrupt every millisecond at the SysTick interrupt priority `TICK_INT_PRIO` defined in `stm32wl3x_hal_conf.h`.
   c. Setting of NVIC group priority to 0.
   d. Call of `HAL_MspInit()` callback function defined in the `stm32wl3x_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 the 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 the DMA interrupt (if needed).
      ◦ Enable the 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 meant to be used.
   d. In the user `main.c` file, initialize the peripheral handle structure then call the peripheral initialization function 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 STM32CubeWL3 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 STM32CubeWL3.

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-WL33CC1).
   The template project provides an empty `main` loop function, which is a good starting point to understand the project settings for STM32CubeWL3. The 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 the code on a given board.
   – It contains the included paths for all the required firmware components.
   – It selects the supported STM32WL3x product line 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 the LL example:
   - Copy/paste the `Templates_LL` folder - to keep the initial source - or directly update an existing `Templates_LL` project.
   - Then, the 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 `stm32wl3x_it.h` file.
     - Replace the `stm32wl3x_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 file `main.h`.

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

5.3 RF applications, demonstrations, and examples

Different types of RF applications, demonstrations, and examples are available in the STM32CubeWL3 package. They are listed in the two sections below.

Sub-GHz examples and demonstrations

These examples demonstrate the main features of the MRSUBG and LPAWUR radio peripherals. These examples are available under:

- `Projects\NUCLEO-WL33CC\Examples\MRSUBG`
- `Projects\NUCLEO-WL33CC\Examples\LPAWUR`
- `Projects\NUCLEO-WL33CC\Demonstrations\MRSUBG`
- `Projects\NUCLEO-WL33CC\Demonstrations\LPAWUR`

Each example or demonstration generally consists of two programs called Tx and Rx acting as transmitter and receiver, respectively:

- Examples/MRSUBG
  - MRSUBG_802_15_4: an implementation of the physical layer defined by the standard 802.15.4. It shows how to configure the radio to transmit or receive 802.15.4 packets.
  - MRSUBG_BasicGeneric: An exchange of STM32WL3x MR_SUBG basic packets.
  - MRSUBG_Chat: A simple application that shows how to use Tx and Rx on the same device.
  - MRSUBG_DatabufferHandler: An example that shows how to swap from Databuffer 0 and 1.
  - MRSUBG_Sequencer AutoAck: An example that transmits and receives packet acknowledgments (ACKs) automatically.
  - MRSUBG_WMBusSTD: An exchange of WM-Bus messages.
  - WakeupRadio: An example to test the LPAWUR radio peripheral.
- Demonstrations/MRSUBG
  - MRSUBG_RTC_Button_TX: This example shows how to set the SoC in deep-stop mode and configure the MRSUBG to wake up the SoC by pressing PB2 to send a frame or after the RTC timer expiration.
  - MRSUBG_Sequencer_Sniff: This example shows how to set the MRSUBG sequencer to operate in sniff mode. This example demonstrates the receiver side and requires another device as a transmitter.
  - MRSUBG_Timer: The application schedules several instances of MRSUBG timer (with autoreload) with different time intervals.
  - MRSUBG_WakeupRadio_Tx: This example explains how to set the SoC in deep stop mode and configure the MRSUBG to wake up the SoC by pressing PB2 to send a frame. This example demonstrates the transmitter side and requires another device as an LPAWUR receiver. The receiver example is located under the `NUCLEO-WL33CC\Demonstrations\LPAWUR\LPAWUR_WakeupRadio_Rx` folder.
- Demonstrations/LPAWUR
  - LPAWUR_WakeupRadio_Rx: This example explains how to set the SoC in deep-stop mode and configure the LPAWUR to wake up the SoC when a frame arrives and is correctly received. This example demonstrates the receiver side and requires another device as a transmitter. The transmitter example is located under the `NUCLEO-WL33CC\Demonstrations\MRSUBG\MRSUBG_WakeupRadio_Tx` folder.

Sigfox™ application

These applications show how to implement a Sigfox™ scenario and use the available Sigfox™ APIs. They are available in the project path `Projects\NUCLEO-WL33CC\Applications\Sigfox\`:

- Sigfox_CLI: This application shows how to use a command-line interface (CLI) to send commands that use the Sigfox™ protocol to send messages and perform precertification tests.
- Sigfox_PushButton: This application allows the evaluation of the STM32WL33xx Sigfox™ device radio capabilities. Pressing PB1 transmits a test Sigfox™ frame.

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 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 peripheral 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* examples.

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 MRSUBG/LPAWUR peripheral examples?

To create a new MRSUBG or LPAWUR example project, either start from the skeleton project provided under `\Projects\NUCLEO-WL33CC\Examples\MRSUBG` or `\Projects\NUCLEO-WL33CC\Examples\LPAWUR`, or from any available project under these same directories.

6.5 How can STM32CubeMX generate code based on embedded software?

STM32CubeMX has a built-in knowledge of STM32 microcontrollers, including their peripherals and software, which allows it to provide a graphical representation to the user and generate `*.h` or `*.c` files based on the user's configuration.

Revision history

Table 3. **Document revision history**

Date	Revision	Changes
29-Mar-2024	1	Initial release.
30-Oct-2024	2	Full integration of STM32CubeWL3 in STM32Cube. Updated: • Introduction • Section 2: STM32CubeWL3 main features • Section 3.2.1: Middleware components • Section 4: STM32CubeWL3 firmware package overview • Section 5.1: Running a first example • Section 5.3: RF applications, demonstrations, and examples Added: • Section 5.1.1: How to run a HAL example • Section 5.2.1: Using STM32CubeMX to develop or update an application • Section 6.4: Is there any template project for MRSUBG/LPAWUR peripheral examples? • Section 6.5: How can STM32CubeMX generate code based on embedded software? Removed: • *PC tools*, including *Navigator*, *STM32WL3 GUI*, and *MR-SUBG Sequencer GUI* • *How can WiSE-Studio IOMapper generate code based on embedded software?* • *Does Navigator allow access to software package resources?*

Contents

List of tables

List of figures