AN5977

Application note

How to build a Bluetooth® LE application with STM32WB0 MCUs

Introduction

This application note guides designers through the steps required to build specific Bluetooth® LE applications based on STM32WB0 series microcontrollers, and using the STM32CubeWB0 MCU software package.

It groups together the most important information and lists the aspects to be addressed.

To fully benefit from the information in this document, and to develop an application, the user must be familiar with STM32 microcontrollers, Bluetooth® LE technology, and master system services, such as low-power management and task sequencing.

For more information, such as deep dive concepts, instructions, and examples, refer to the *Bluetooth® LE stack v4.x programming guidelines* (PM0274) or refer to https://www.st.com.

1 General information

The STM32WB0 series device is a powerful ultra-low power 2.4 GHz RF transceiver with an Arm® Cortex®-M0+ microcontroller that can operate up to 64 MHz.

It is suitable to implement an application compliant with the Bluetooth® LE SIG specification. It also provides the capability to get access to the 2.4 GHz radio through a specific low-level driver, which can send and receive packets without using the Bluetooth link layer.

It is also composed of a multitude of smart and high-performance peripherals, a large set of advanced and low-power analog features, and several peripherals tuned for low-power modes.

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

1.1 Reference documents

Table 1. Reference documents

Reference	Document name
[0]	STM32WB09xE datasheet (DS14210)
[1]	STM32WB09xE reference manual (RM0505)
[2]	STM32WB07xC/STM32WB06xC datasheet (DS14676)
[3]	STM32WB07xC/STM32WB06xC reference manual (RM0530)
[4]	STM32WB05xZ datasheet (DS14591)
[5]	STM32WB05xZ reference manual (RM0529)
[6]	Bluetooth® LE stack v4.x Programming Guidelines (PM0274)

1.2 List of acronyms and abbreviations

Table 2. Acronyms and abbreviations

Acronyms	Definitions
ACI	Application command interface
ATT	Attribute protocol
BSP	Board support package
DLE	Data length extension
EXTI	Extended interrupts and event controller
FM	Flash manager
FUOTA	Firmware update over the air
FW	Firmware
GAP	Generic access profile
GATT	Generic attribute profile
HAL	Hardware abstraction layer
HCI	Host controller interface
HSE	High-speed external oscillator
HW	Hardware
IFS	Interframe space
IP	Semiconductor intellectual property core

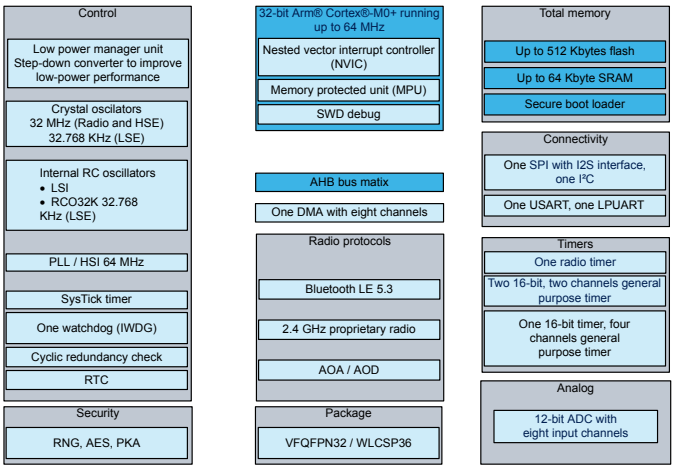
Acronyms	Definitions
ISR	Interrupt service routine
L2CAP	Logical link control and adaptation protocol
LL	Link layer
LSE	Low-speed external oscillator
NVIC	Nested vectored interrupt controller
NVM	Nonvolatile memory
OTA	Over-the-air
OS	Operating system
PDU	Protocol data unit
PHY	Physical layer
PLL	Phase-locked loops
QOS	Quality of service
RTC	Real-time clock
RTOS	Real time operating system
SoC	System on a chip
SW	Software
WFI	Wait for interrupt (Arm assembly code)

2 STM32WB0 overview

2.1 STM32WB0 series key features

This section lists the key features of the STM32WB09xE devices.

Figure 1. STM32WB09xE key features



Refer to STM32WB07xC/STM32WB06xC and STM32WB05xZ datasheets for detailed information about the supported features.

For more information, such as GPIOs, NVIC groups, or memory mapping, refer to the available STM32WB0 series devices datasheets and reference manuals detailed on Section 1.1: Reference documents.

2.2 Software architecture

The software architecture is explained from two points of view:

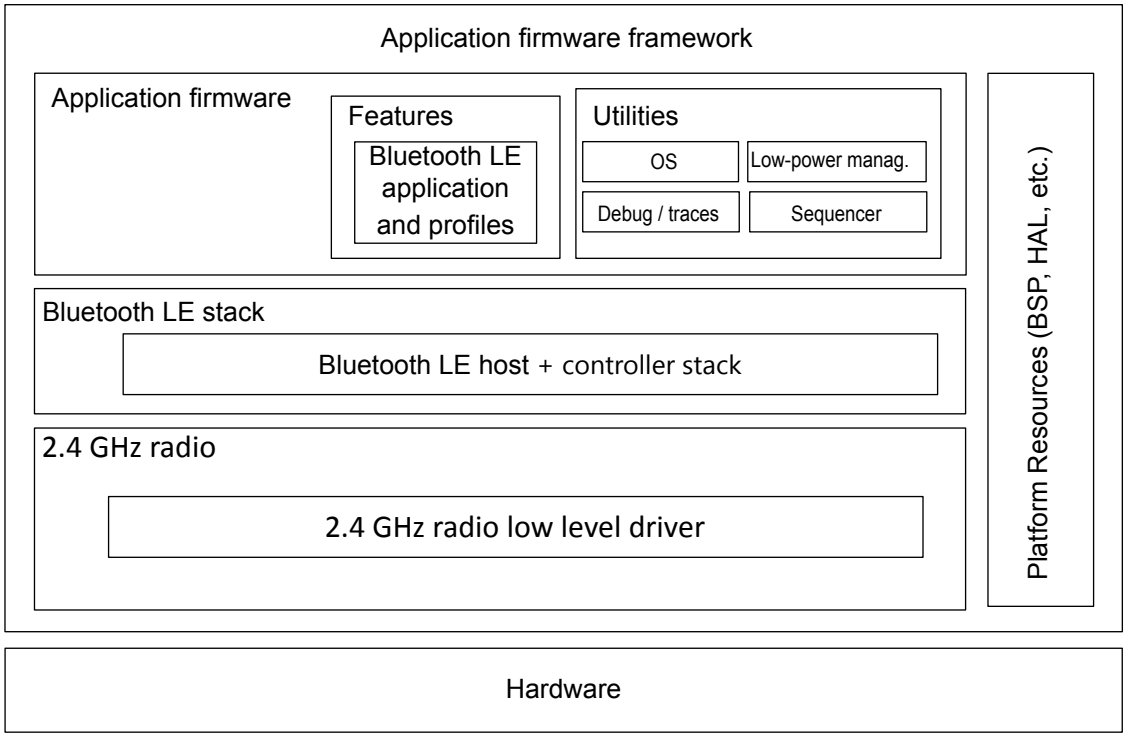
- The application
- The project

2.3 Application view

From the application point of view, the user can rely on multiple modules and software blocks to create an application within the connectivity domain. The framework is referred to as the application firmware framework. The architecture is designed over the following organization:

- Application firmware
- Bluetooth® LE stack or 2.4 GHz radio low-level driver
- Platform resources

Figure 2. **Application firmware framework**



2.4 Application firmware

The user application is based on:

- IPs from STMicroelectronics.
- Features developed by STMicroelectronics for short-range protocols, such as Bluetooth® LE applications and profiles, and 2.4 GHz radio proprietary applications.
- Utilities required by the application and the protocol stacks.
- Generic utilities that provide easy access to basic features such as low-power management or sequencer.

2.5 Bluetooth® LE stack

The Bluetooth® LE stack is the main interface between the application and the hardware for Bluetooth® LE purposes.

The Bluetooth® LE stack manages all networks, and transport protocols that allow the application to communicate with other devices. It also includes the physical layer, the link layer, and the host controller interface.

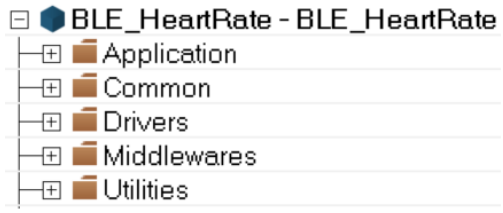
2.6 Platform resources

The platform resources include all the HALs, BSPs, and drivers that ease the platform hardware use.

2.7 Project view

The applications projects are divided into five parts as shown below.

Figure 3. **Application project view organization**



DT76206V1

2.7.1 Application

Application is the core part of the project. It regroups the main information and actions that define the user application.

This section is divided into four subdirectories:

- Core: This directory is the main entry point of the application. It contains all the setup and entry codes such as hardware initialization routines, IRQ setup, scheduler configuration, and tasks registration.
  Examples of files contained in this directory: `main.c`, `app_entry.c`
- Startup: Includes the startup file of the project.
- STM32_BLE: This part is composed of two subparts:
  - App: Main applicative source files, for the example of HR, this repository contains the main application file `app_ble.c` and the Bluetooth® LE services implementation.
  - Target: Dedicated to interface and integration of the libraries and modules present in middleware.
- System: Utilities (USART).

2.7.2 Common

This section contains the centralized system-related modules, interfaces, and utilities (such as PKA, RNG, advanced memory manager) . This files are used by all the STM32_BLE Bluetooth LE applications.

2.7.3 Drivers

The drivers are divided into four component sets:

- Hardware abstraction layer (HAL):
  This layer provides the hardware abstraction drivers and the hardware-interfacing methods to interact with the upper layers (application, libraries, and stacks). The HAL is composed of:
  - HAL drivers:
    A set of portable abstraction APIs based on high level services built around standalone processes. The HAL drivers are functionality oriented, for example to the timer peripheral, for which it is possible to split APIs into several categories following the functions offered by the IPs (such as basic timer, capture, or PWM).
  - Low-layer drivers:
    A set of basic drivers with direct hardware access with no standalone processes. This layer is called either by the applications or by the HAL drivers.
  - HAL core drivers:
    A set of internal drivers providing low level operations to the “complex” peripherals if applicable. They all come with a dedicated source and header files. Some drivers may have an additional file containing extended features, identified by the file extension "`_ex`".
- BSP drivers:
  This layer contains the high-level board services for the different functionalities offered by the hardware (such as LED, audio, and pushbuttons) and the drivers for the external components mounted on the used boards (such as audio codec, and I/O expander).
- CMSIS drivers:
  Cortex® microcontroller software interface standard (CMSIS) drivers that provide a single standard across all Cortex-Mx processor series vendors. It enables code reuse and code sharing across software projects.
- Basic peripheral usage examples:
  This layer encloses the examples built over the STM32 peripheral using the HAL APIs and the low-layer drivers.

2.7.4 Middleware

Libraries and protocol-based components (Bluetooth® LE stack, for example). This directory contains the STM32_BLE middleware, which provides the Bluetooth® LE services management, and system commands.

Horizontal interactions between the components of this layer are directly performed by calling the features APIs while the vertical interaction with the low-level drivers are performed through specific callbacks and static macros implemented in the library interface.

2.7.5 Utilities

Miscellaneous software utilities that provide additional system and media resources services like sequencer tools box, time server, low-power manager, along with several trace utilities and standard library services like, memory, string, timer, and math services.

2.8 Software concepts and features

This chapter covers the main software components available on the STM32WB0 series. Each module concept is described briefly to enhance user appreciation.

For further information, refer to the available documents described in the Section 1.1: Reference documents.

2.8.1 Sequencer

The integration cost of a real time operating system can be disproportional in the case of a simple application as a result of:

- RTOS knowledge required
- Increase in the complexity of the application
- Impact on sizing of RAM/ROM

This is the reason why the sequencer module has been developed, as an alternative to a real time OS.

2.8.1.1 Concepts

The sequencer utility is designed as a simple alternative for simple application cases. However, it does not cover all the services provided by an operating system. For instance, this software solution does not provide preemption mechanisms, so this must be considered in the application design. Instead of tasks designed with a risk of freezing the system, it is recommended to use reentrant functions based on state machines.

Important concepts to consider:

- Task creation: initialize the task and render it callable by the internal scheduler of the sequencer.
- Task enable: enable the task, through a task or an interrupt, so the task can be executed by the scheduler.
- Task pause/resume: pause/resume the task execution from the scheduler point of view, independent whether the task is enabled or not.
- Idle task: if the scheduler has no task to execute, it calls an optional hook function to manage entry in idle mode.
- Task execution: calls the function associated to the task, and the scheduler is locked until the function returns.
- Sequencer: embed a task scheduler that sequences the tasks execution and allows the task to stop until an event reception.

Additionally, the sequencer provides the following features:

- Up to 32 tasks registered
- Request a task to be executed
- Task pause and resume
- Wait for a specific event (not blocking)
- Priority on tasks
- Allow management of an IDLE task

2.8.2 Low-power modes

The STM32WB0 series devices provide the following hardware power save modes to achieve the best compromise between low-power consumption, short start-up time and available wake-up sources:

Deepstop mode

- The system and the bus clocks are stopped.
- Only the essential digital power domain is ON and supplied at 1.0 V.
- The bank RAM0 is kept in retention.
- The other banks of RAM can be in retention or not, depending on the software configuration.
- The low speed clock can run or stop, depending on the software configuration:
  - ON or OFF.
  - Sourced by LSE or by LSI.
- The RTC and the IWDG are active if enabled and the low-speed clock is ON.
- LPUART is active if enabled and the low speed clock is ON (it is valid only on STM32WB05xZ and STM32WB09xE devices).
- The radio wake-up block including its timer is active if enabled and the low speed clock is ON.
- If the low-speed clock is OFF, it is possible to wake up from GPIOs (PA0 to PA15 and PB0 to PB11 on STM32WB07xC/STM32WB06xC devices, and all GPIOs on STM32WB05xZ and STM32WB09xE devices). If it is ON, RTC, IWDG, LPUART (only on STM32WB05xZ and STM32WB09xE devices), radio, and radio timer can also wake up.
- When the wake-up is triggered by previous listed sources, the system reverts to the run mode with all the peripherals on. When exiting from the deepstop mode, the application needs to wait until the high speed oscillator is stable.

Shutdown mode

- The shutdown mode is the least power consuming mode. In shutdown mode, the device is in ultra-low power consumption: all voltage regulators, clocks, and the RF interface are not powered.
- The STM32WB0 series devices can enter shutdown mode by internal software sequence. The only way to exit shutdown mode is by asserting and deasserting the RESET pin or a configurable pulse on PB0 pin (only on STM32WB09xE devices).

Refer to the available STM32WB0 series devices datasheets and reference manuals detailed on Section 1.1: Reference documents.

2.8.2.1 *Low-power management*

The STM32CubeWB0 MCU package provides a software framework to support all the STM32WB0 series devices hardware power save modes.

The power save software combines the low-power requests coming from the application with the radio operating mode, choosing the best power save mode applicable in the current scenario.

The power modes software framework enables the negotiation between the radio module and the application requests and prevents data loss.

When the STM32WB0 device exit from any power save mode, a reset occurs: all the peripheral configurations and the application contexts are lost.

The power save software implements a mechanism to save and restore all the peripheral configurations and the application contexts when a power save procedure is called. So, from the application point of view, the exit from a low-power procedure is fully transparent: when a wake-up from power save occurs, the CPU executes the next instruction after the power save function call.

The power save software framework implements the following power save modes:

- The shutdown level which turns off the device. The only wake-up source is a low pulse on the RSTN pad or a configurable pulse on PB0 (only on STM32WB09xE devices). No smart power management is in place, because there is no memory retention in this power save level configuration.
- The deepstop with low-speed clock enabled which configures the device in deep-sleep and the timer clock sources (LSI or LSE) remain running. A wake-up is possible from all GPIOs enabled for this scope, RTC, IWDG, LPUART (only on STM32WB05xZ and STM32WB09xE devices), radio IP, and the radio timer.
- The deepstop with no low-speed clock enabled which configures the device in in deep-sleep. All the peripherals and clock sources are turned off. A wake-up is possible only from GPIOs enabled for this scope.
- The sleep (WFI) which configures the device in CPU halt. Only the CPU is halted. The rest of the chip continues to run normally. The chip wakes up from any interruption.

2.8.2.2 *Low-power modes examples*

The following section provides some basic examples about how to configure the device on each supported power save mode.

Sleep (WFI)

In this mode only the CPU is stopped. All peripherals continue operating and can wake up the CPU when an interrupt occurs. A typical source code is:

```
/* CPU-HALT (WFI) */
UTIL_LPM_SetStopMode(ACTOR_0, UTIL_LPM_DISABLE);
UTIL_LPM_SetOffMode(ACTOR_0, UTIL_LPM_DISABLE);
UTIL_LPM_EnterLowPower();
```

In this power save mode, the only wake-up sources are the peripheral interrupts.

If the radio operating mode does not allow this low-power request, as it is executing a no stoppable operation, the power save mode is converted automatically inside the power save software in run mode.

Deepstop with low-speed clock enabled

In this low-power mode, the CPU is stopped and all the peripherals are disabled. Only the timer clock sources (LSI or LSE) and the external wake-up source block are running. A wake-up is possible from external wake-up sources, RTC, IWDG, radio, and the radio timer. A typical source code is:

```
/* Configure wakeup source GPIOB0 and RTC */
HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PA8, PWR_WUP_FALLEDG);
HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_RTC, 0);
/* DEEPSTOP with Low Speed Oscillator  ON */
UTIL_LPM_SetStopMode(1 << CFG_LPM_APP, UTIL_LPM_ENABLE);
UTIL_LPM_SetOffMode(1 << CFG_LPM_APP, UTIL_LPM_DISABLE);
UTIL_LPM_EnterLowPower();
```

The wake-up source is the PA8 with detection of wake-up event on falling edge; the application sets the RTC timer to wake up the system when the timeout occurs.

If the radio operating mode is in connection or advertising state, the radio stack accepts the power save mode proposed by the application, but, if necessary, the system can be woken up before the application timeout to follow the connection interval time profile or the advertising interval time profile.

Deepstop with no low-speed clock enabled

In this power save mode, the CPU is stopped and all the peripherals are disabled. Only the external wake-up source block runs. A typical source code is:

```
/* Configure wakeup source PA10 */
HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PA10, PWR_WUP_RISIEDG);
/* DEEPSTOP without Low Speed Oscillator */
UTIL_LPM_SetStopMode(1 << CFG_LPM_APP, UTIL_LPM_ENABLE);
UTIL_LPM_SetOffMode(1 << CFG_LPM_APP, UTIL_LPM_ENABLE);
UTIL_LPM_EnterLowPower();
```

The wake-up source is the PA10 with detection of wake-up event on rising edge.

If the radio module is in connection state, after the negotiation with the radio stack, the power save software changes the power save mode into deepstop with low-speed clock enabled to follow the connection time profile. Otherwise, if the radio module is in an idle state, the radio stack accepts the power save mode deepstop with no low-speed clock requested from the application, and the power save software does not change it.

2.8.2.3 *Low-power current consumption figures*

This section describes the power consumption demonstration application allowing typical current consumption measurements to be shown during Bluetooth® LE advertising and connection phases.

To take these measurements the following configuration has been used:

- Hardware: NUCLEO-WB09KE kit (STM32WB09KE)
- Firmware: BLE_Power_Consumption test application released in the STM32CubeWB0 MCU package
- Power supply: 3.3 V

- Tx Power: 0 dBm
- SMPS ON
- Retention: full RAM retention 64 kbytes
- Crystal Startup time: 780 µs
- System clock: Direct HSE configuration
- Bluetooth® LE clock: 16 MHz
- Advertising interval: 100 ms and 1000 ms, with 25 bytes packet length
- Connection interval: 100 ms and 1000 ms, with empty packets

The BLE_Power_Consumption application configures the STM32WB09KE as Bluetooth® LE peripheral device.

This application is included in the STM32CubeWB0 software package, in Applications, BLE folder. The Bluetooth® LE central role can be covered by another STM32WB09KE device loaded with another application acting as Bluetooth® LE central device and connecting with a connection interval, respectively of 100 or 1000 ms, to the BLE_Power_Consumption peripheral application.

The current consumption values could be measured connecting a DC power analyzer to the MB1801D probe on Nucleo kit.

Alternatively, the Bluetooth® LE central role could be covered by a smarpthone application which connects with the BLE_Power_Consumption with connection interval, respectively, of 100 or 1000 ms.

The following table provides some typical current consumption values in Bluetooth® LE advertising and connections scenarios.

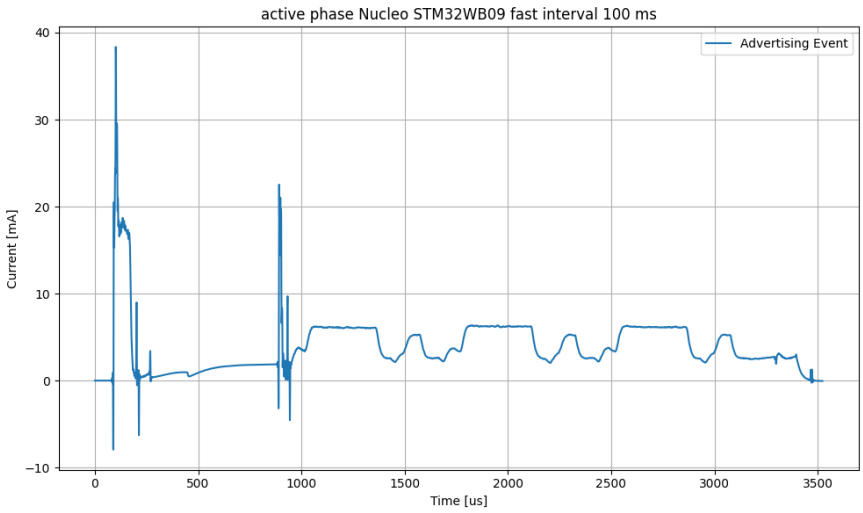
Table 3. Current consumption values

Bluetooth® LE scenario	Real current consumption (µA)(1)
Advertising with an interval of 100 ms	137.69
Advertising with an interval of 1000 ms	15.13
Connection with an interval of 100 ms	55.04
Connection with an interval of 1000 ms	7.22

1. *Values measured connecting a DC power analyzer to the NUCLEO-WB09KE kit running with a real application. Refer to the BLE_PowerConsumption application available on the STM32CubeWB0 package.*

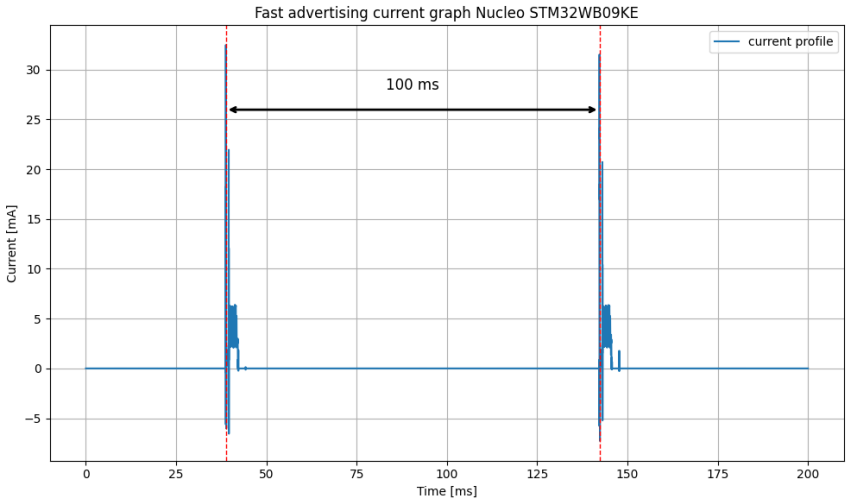
The following images show the advertising procedure snapshots for both intervals (the packet length is 25 bytes).

Figure 4. STM32WB09KE advertising procedure: active phase



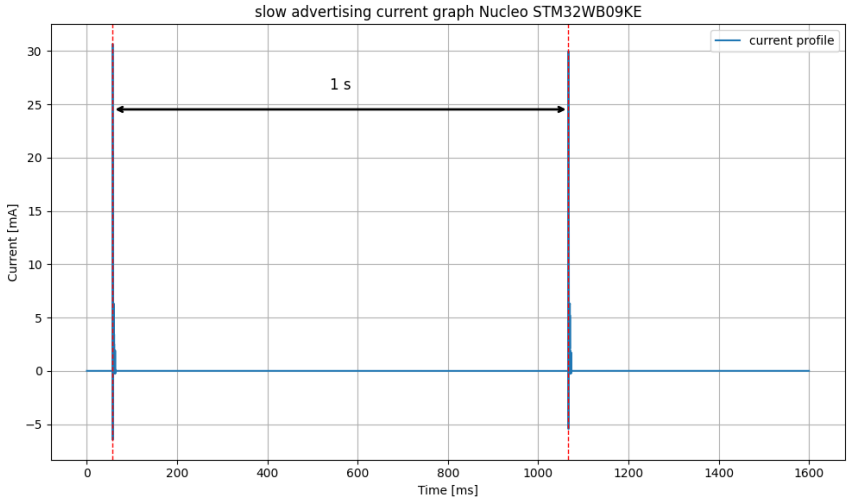
DT73286V2

Figure 5. STM32WB09KE advertising procedure: interval 100 ms



DT73287V2

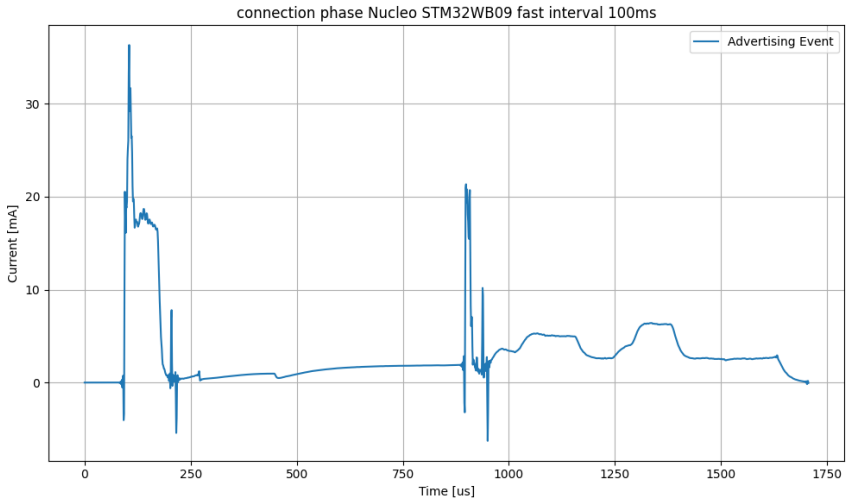
Figure 6. STM32WB09KE advertising procedure: interval 1000 ms



DT73288V2

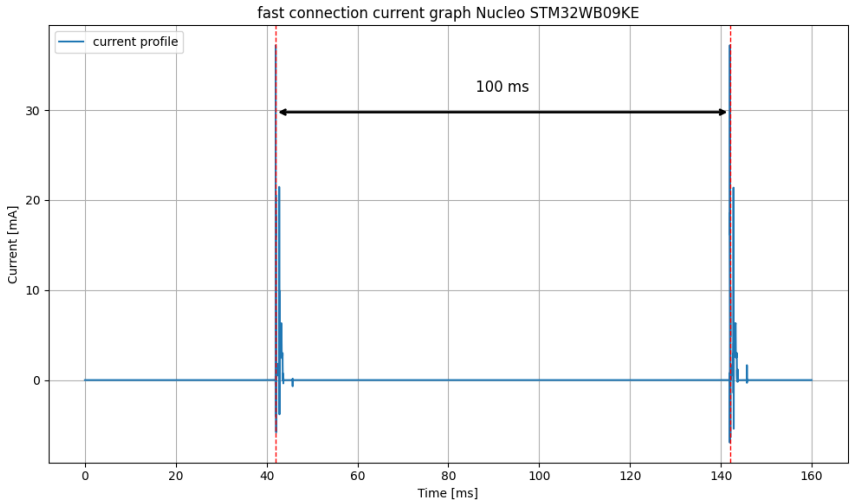
The following images show the connection procedure snapshots for both intervals (with empty packet).

Figure 7. STM32WB09KE connection procedure: active phase



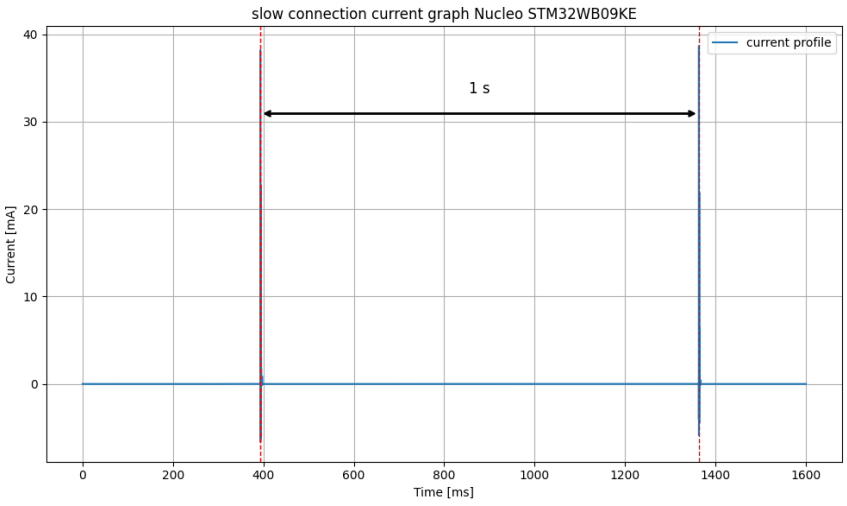
DT73289V2

Figure 8. STM32WB09KE connection procedure: interval 100 ms



DT73291V2

Figure 9. STM32WB09KE connection procedure: interval 1000 ms



DT73290V2

2.8.3 Flash memory management

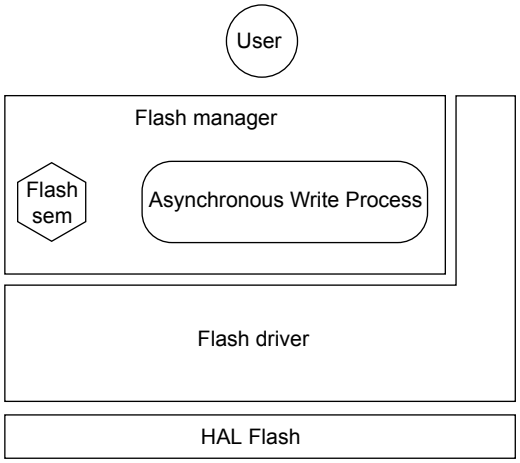
The flash memory management proposes a simple interface to the upper layers to execute operations in flash memory. It manages synchronization between flash memory operations and the Bluetooth® LE LL activity. Thus, users do not have to spend additional effort to synchronize RF timing and flash memory operations.

2.8.3.1 Concepts

The flash memory management relies on different concepts:

1. Asynchronous flash memory operations
   The flash memory operations can be requested via the flash memory manager interface, and executed afterward by the module. The requester is later on notified on the status of the flash memory operation. However, in case of a write operation, the user buffer is held as long as the flash memory operation is not over. If the buffer is updated during the write operation, the user must restart a brand new flash memory operation.
2. Three layer organization
   Flash memory management is based upon a three layers distribution composed by:
   - Flash memory manager: Main user interface for flash memory operation (such as flash memory write, or flash erase).
   - Flash memory driver: low-level driver abstraction layer. Controlled by the flash memory control statuses. It denies flash memory operations close to Bluetooth® LE radio activities.
   - HAL flash memory: low-level driver that interacts directly with the flash memory hardware.

Figure 10. Three-layer organization



DT73245V1

3. Flash memory access
   The flash memory access is protected at two different levels:
   - Flash memory semaphore:
     A semaphore is required to share the flash memory interface between several software modules. The owner of the semaphore is the only one that can request flash memory operations. The semaphore is attributed to the first requester and released once its operation is over.
   - Flash memory control statuses
     Independently from the flash memory semaphore, the flash memory driver provides flags, the flash memory control status, to prevent flash memory operation depending on the system activity. These flags/statuses are checked before any flash memory operation by the flash memory driver.

2.8.4 Trace management: log module

An application often needs to manage traces for debugging purposes or to communicate with another system (such a console and monitoring).

The log module provides a logging mechanism for embedded systems. It allows developers to print logs with different verbose levels and regions, making it easier to debug and troubleshoot issues in the project.

The module is highly customizable and can be configured to meet specific user needs.

2.8.4.1 *Concepts*

The advanced trace is split into two layers:

- Advanced trace services:
  API that regroups all the services available for traces.
- Trace interface:
  Interface between the hardware and the advance trace services layer.

Advanced trace services

The advanced trace offers interesting features to users for trace debugging or logging:

- Trace FIFO management:
  The advanced trace module manages a circular FIFO (the size is customizable) to store all the application data before pushing them on the media. The module guarantees the data integrity and parallel access to the trace services.
- Unchunk mode:
  The unchunk mode is an optional mode that can be enabled inside the utilities' configuration header. The goal of this mode is to guarantee that a frame is not split inside two transfers.
- Timestamp management:
  The timestamp management allows the user to add a timestamp buffer ahead of the frame to output. However, the following steps are necessary:
  - Create a callback function that returns a timestamp buffer and its buffer size.
  - Use the adequate functions that allow timestamp management. This mode can be activated within the utilities' configuration header.
- Verbose level and region management:
  The verbose level and the region are two different levels used to determine whether a frame can be sent or not. If one of these two conditions fails, the frame is discarded.
  Verbose level:
  - The user can define the current verbose level. The default setup is 0, meaning only value frames with a verbose level equal to zero are sent through the advanced trace. If the application verbose level is lower or equal to the current verbose level, the frame is sent or otherwise discarded.

  Region:
  - The user can define the region mask. The default setup is 0, meaning only the application frames with a region equal to zero are sent through the advanced trace. If the application region value is equal to zero or if the value is aligned with the mask region, the frame is sent or otherwise discarded.
- Overrun management:
  This feature aims to indicate when a frame has been discarded due to a full FIFO. The switch enabled mode is not enough to make it functional, the user must provide a callback function that returns an overrun frame and the size of this frame.
  This feature can be activated within the utilities configuration header.

Trace interface

The trace interface handles the hardware interface. The only requirements for the hardware are to provide characteristics such as:

- Ability to send data (8-bit data)
- Ability able to receive data

This hardware layer has been built to match with the UART, but there are other hardware IPs compatible with these requirements: for example USB, I²C, or SPI. The choice is managed by the application designer according to the available resources and the application requirements.

It is necessary to enable the trace module through the application configuration file to be able to use it. Moreover, the hardware interface built on UART makes use of interrupts, so they must be activated to make the module work properly. If enabled, DMA can be used for transmitting data.

2.8.5 Real time software debug

Debug on GPIO allows the user to get real time debugging traces of the application. Debug on GPIO is present at any relevant places. It concerns:

- Start/end of interrupt.

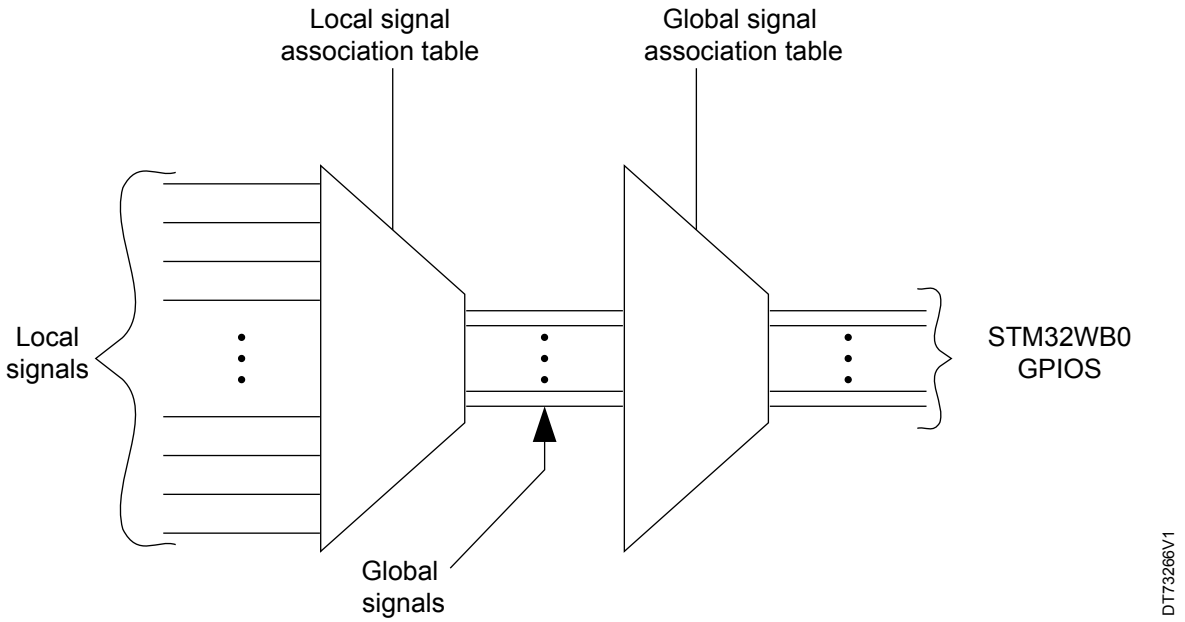
- Start/end of each background process.
- Identification of specific procedures and machine states.

2.8.5.1 *Concepts*

The real time software debug has a two stages in the pipeline architecture:

- Local layer: The first stage determines which local signal is linked to which global signal. The local signal association table is responsible for matching the local and global signals.
- Global layer: The second stage determines which global signal is linked to which GPIO pin. The global signal association table is responsible for the match between the global signals and the GPIO pin.

Figure 11. Real time software debug



Signal selection

Debug signals are divided into different categories such as system signals and application signals. The desired signal is selected in a general configuration file.

Note: The local signals in different modules can have the same ID/number. Therefore, it is necessary to associate the local signals (possibly all used) to the global ones (effectively used).

GPIO configuration

The GPIO associated to the debug signal should be entirely configurable.

For optimization purposes, there is no dedicated callback for getting the associated GPIO at runtime (and no registering). The goal is to reduce code size and processing time (not modify real time on complicated use cases).

Note: It is necessary to associate the software signal to the desired GPIO at compilation time.

2.8.6 FUOTA

The firmware update over the air (FUOTA) allows the user to update the application during runtime without any wire connection.

2.8.6.1 *Concepts*

The FUOTA architecture is based on two components:

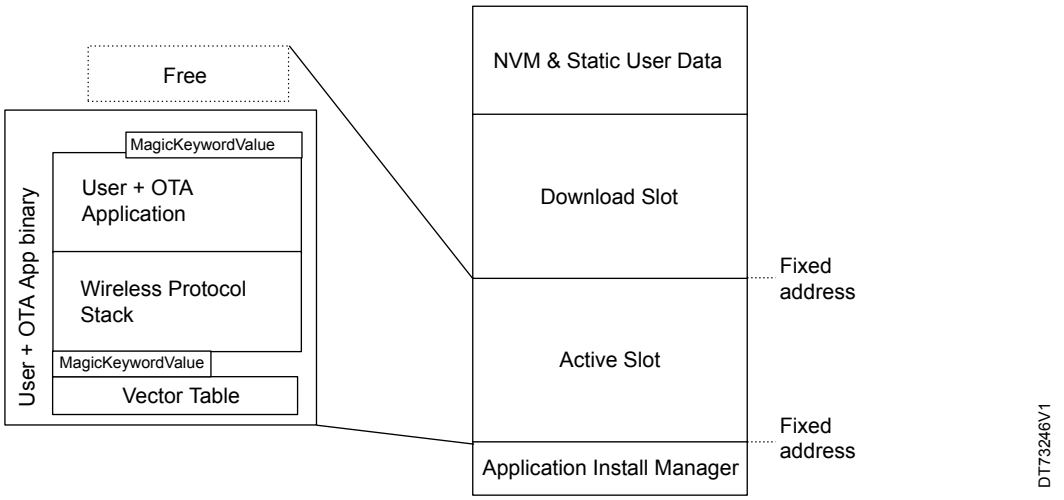
- OTA application: This component is responsible for downloading the raw data of the new application version. It can be based on any short-range wireless protocol handled by the STM32WB0 series devices.
- Application install manager application: This component is responsible for updating the firmware. It substitutes the older version with the new one at the restart. When there is no firmware update to perform, the boot manager application simply acts as a normal boot manager. This application part cannot be updated by OTA.

Memory mapping

The FUOTA memory mapping is organized along four fixed and presized regions:

- Application install manager region: This region starts at the boot address of the CPU. It contains the application install manager binary with its own vector table.
- NVM/User data region: This region handles the NVM and user data. These parts can be split into two if necessary. They can be positioned anywhere in the memory mapping.
- Active slot region: The active slot region is composed of the user app binaries, for example, user application, OTA application, and wireless protocol stack (including its vector table). Its size is fixed and is the same as the download slot. The active slot region address is fixed, but can be placed anywhere.
- Download slot region: This region has the same amount of space as the active slot region. It is designed to host the raw data of the new application version. The download slot region address is fixed, but can be placed anywhere.

Figure 12. FUOTA overview



Active slot

As previously stated, the user application part is composed of different components:

- User + OTA application: This is a single binary implementing both the user and OTA applications. The OTA application is responsible for loading the updated firmware in the download slot. This binary (including both the user and OTA application) can be updated with OTA.
- Wireless protocol stack: This is the protocol stack of the application.
- OTA tag: The OTA tag is a milestone that indicates the end of the active slot binaries. It ensures that the firmware has been fully loaded. The value of the keyword user is a number, but can be a CRC. It is up to the user to implement a postscript for the integrity computation. The keyword user address is placed inside a variable located right after the active slot vector table end. This variable has a fixed address.

2.8.6.2 *FUOTA service and characteristics*

The FUOTA service is a generic attribute profile (GATT) based on the Bluetooth LE profile defined by STMicroelectronics. It has the proprietary UUIDs 0000FE20-cc7a-482a-984a-7f2ed5b3e58f (128 bits) which includes three characteristics:

Table 4. FUOTA service characteristics

Characteristic	Mode	UID	Size
Base address	Write without response	0000FE22-8e22-4541-9d4c-21edae82ed19	5
Confirmation	Indicate	0000FE23-8e22-4541-9d4c-21edae82ed19	1
Raw data	Write without response	0000FE24-8e22-4541-9d4c-21edae82ed19	240

The base address characteristic is used from the FUOTA central device to:

- Indicate the action to perform.
- Provide the address where the new application is stored (offset from flash memory base).
- Inform about the sectors to erase.

The following actions are supported:

- 0x00: Stop all upload
- 0x01: Start user data upload
- 0x02: Start application file upload
- 0x06: End of file transfer
- 0x07: File upload finish
- 0x08: Cancel upload

The confirmation characteristic is used from the FUOTA client device to inform about:

- Ready to receive the file: ready to receive the new binary application.
- Reboot: the new file is fully received.
- Error not free: not ready to receive a new binary application.

The raw data characteristic is used from the FUOTA client device to transfer the file.

At start-up, application with FUOTA starts with fast advertising (80 ms/100 ms).

The advertised data are composed as follows:

Table 5. FUOTA advertised data

Description	Length	AD type	Value
Flags	2	0x01	0x06 (GeneralDiscoverable, BrEdrNotSupported)
Device name	8	0x09	p2pS_XX (XX: last byte of BD address)
Manufacturer data	15	0xFF	[0x30 0x00 (STMicro) 0x02 (ST protocol v2) 0x8D (STM32WB0 series Nucleo kits) 0x83 (p2pserver) 0x00 0x00 0x01 (FUOTA) XX XX XX XX XX XX (BD address)]

The over-the-air upgrade protocol main steps are as follows:

- The server device advertises the specific FUOTA advertising data.
- A client device discovers and connects to the server device.
- The client device writes the base address characteristic with information related to the base address, the sectors to be erased, and start user data upload action.
- The server device sends a confirmation indication to indicate it is ready to receive file.

- The client device starts a sequence of writing raw data characteristic for sending the image data to be uploaded.
- When all image data have been sent, the client device writes the base address characteristic with the end of file transfer action.
- The server device sends a confirmation indication to indicate that the new file is fully received and reboot.

2.8.6.3 *FUOTA application framework*

The application level of the FUOTA service, characteristics, and associated events is done through the ota_app.c file. It handles the following phases:

- Initialization of the FUOTA services (OTA_Init() function).
- Initialization of the context of the application.
- Erases the download slot.
- Receives the notification from the FUOTA service (OTA_Notification() function).
- Initializes base address characteristic.
- Initializes confirmation characteristic.
- Initializes raw data characteristic.
- Manages the GATT event from Bluetooth® LE Stack for the FUOTA operations:
  - Reception of a write command: raw data characteristic value
  - Sent of write response with an OK or KO status.
  - Reception of confirmation characteristics description value: enable or disable indication
  - Notifies the application of the confirmation indication
  - Reception of base address characteristics value
  - Notifies application of the base address modification
  - Reception of raw data characteristics value
  - Notifies application of the raw data modification

The FUOTA service is used in BLE_HeartRate_ota and BLE_p2pServer_ota applications, which are available on STM32CubeWB0 MCU software package.

The application install manager is also provided within the same MCU software package.

Application install manager

Once the firmware update is fully received, the application install manager installs it from the download slot to the active slot.

When there is no firmware update to be installed, it jumps on the reset vector of the application. It cannot be updated with FUOTA.

User application

This is a single binary implementing both the user and the FUOTA application.

The FUOTA application is responsible for downloading the firmware update in the download slot.

This binary (including both the user and the FUOTA application) can be updated with FUOTA.

3 System initialization

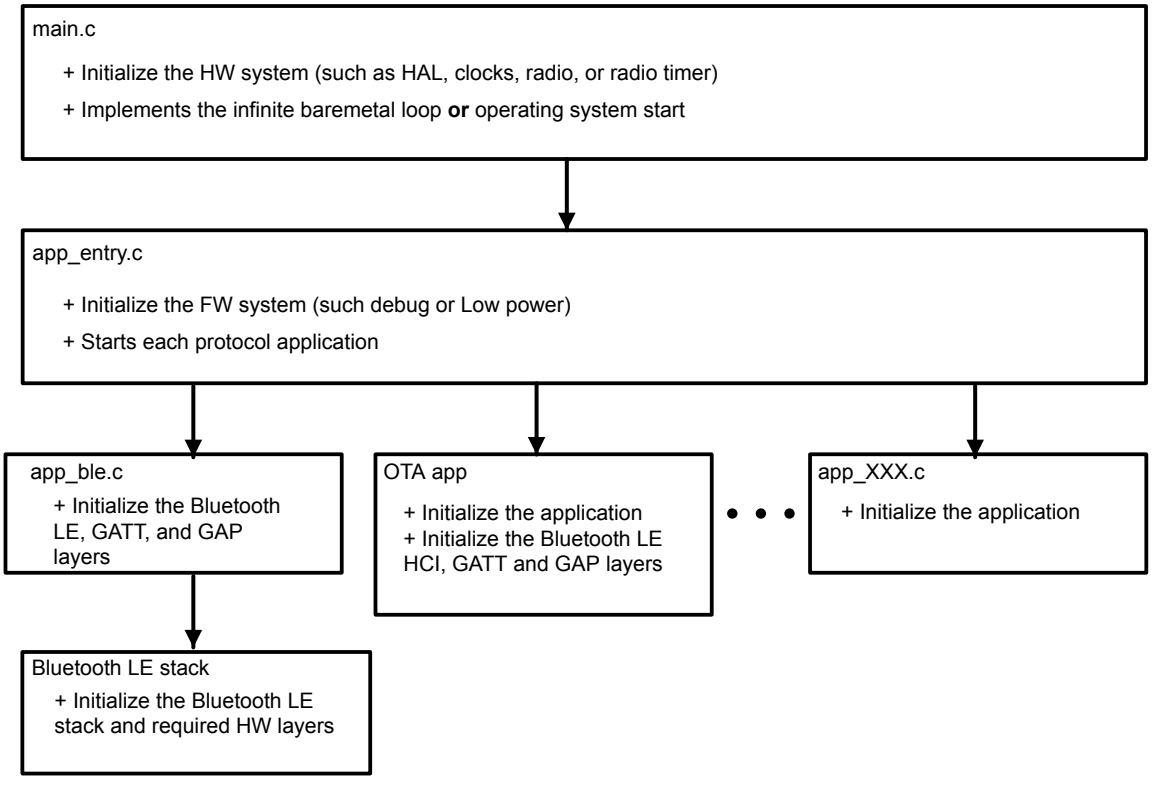
3.1 General concepts

All applications referencing the Bluetooth® LE stack have a four layer architecture:

- `main.c` first entry point, all hardware configuration that is common to any application
- `app_entry.c` all software configuration and implementation that is common to any application
- `app_ble.c/app_xxx.c` application dedicated files
- Initialization of Bluetooth® LE, GATT, and GAP layers

The system is initialized using the steps below.

Figure 13. General architecture



4 Design of a short-range wireless application

4.1 Bluetooth® LE

This chapter aims to highlight a generic Bluetooth® LE application based on STM32_BLE middleware on a STM32WB0 series device.

4.1.1 Overview

A Bluetooth® LE application is composed of different components and modules, but it mostly depends on Bluetooth® LE profiles. These profiles define the capacities and the purpose of the Bluetooth® LE application.

A Bluetooth® LE profile is a collection of one or more Bluetooth® LE services. It also defines the behavior of its services mainly in the context of GAP management.

A Bluetooth® LE profile is composed of three components:

- A GATT management module.
- An application service for each selected Bluetooth® LE service.
- A GAP management module.

A Bluetooth® LE profile is built by linking the Bluetooth® LE stack binary library to the user application.

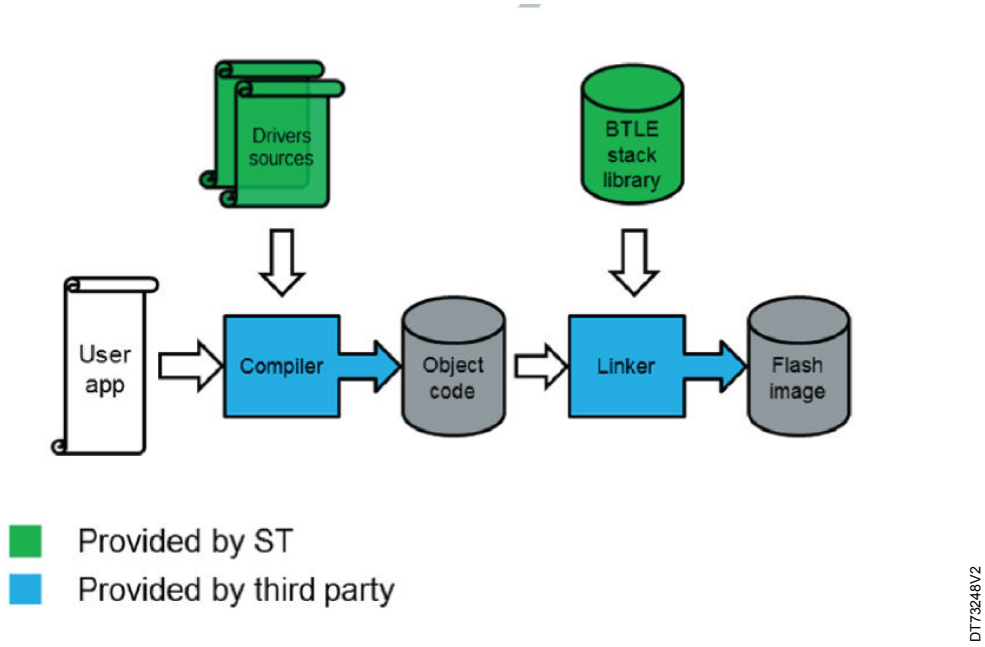
Bluetooth® LE stack v4.x is a standard C library, in binary format, which provides a high-level interface to control STMicroelectronics devices Bluetooth® LE functionalities. The Bluetooth® LE binary library provides the following functionalities:

- Stack APIs for:
  - Bluetooth® LE stack initialization.
  - Bluetooth® LE stack application command interface (HCI command prefixed with `hci_`, and vendor-specific command prefixed with `aci_`).
  - Bluetooth® LE stack state machine handling.
- Stack event dispatcher module:
  - Inform user application about Bluetooth® LE stack events.

- Interrupt handler for radio IP.

  In order to get access to the Bluetooth® LE stack functionalities, a user application is just requested to:
  - Call the related stack APIs.
  - Handle the expected events through the provided stack event dispatcher module.
  - Link the Bluetooth® LE stack binary library to the user application, as described in the figure below:

Figure 14. Bluetooth® LE stack reference application



Note: API is a C function defined by the Bluetooth® LE stack library and called by the user application.

4.1.1.1 *ACI interface*

The ACI interface is a specific STMicroelectronics interface provided to access all the available features in the Bluetooth® LE host stack. The Bluetooth® LE stack library framework allows commands to be sent to Bluetooth® LE stack and it also provides definitions of Bluetooth® LE event packet structures.

The Bluetooth® LE stack APIs use and extend the standard HCI data format defined within the Bluetooth® specifications.

The provided set of APIs supports the following commands:

- Standard HCI commands for controller as defined by Bluetooth® specifications.
- Vendor specific (VS) HCI commands for controller.
- Vendor specific (VS) ACI commands for host (L2CAP, ATT, SM, GATT, GAP).

4.1.1.2 *GATT interface*

The GATT interface is also a specific STMicroelectronics interface provided for use with the ACI interface. Its purpose is to ease management of the Bluetooth® LE services.

The GATT server is in charge of storing the attribute, which composes a profile on a GATT database.

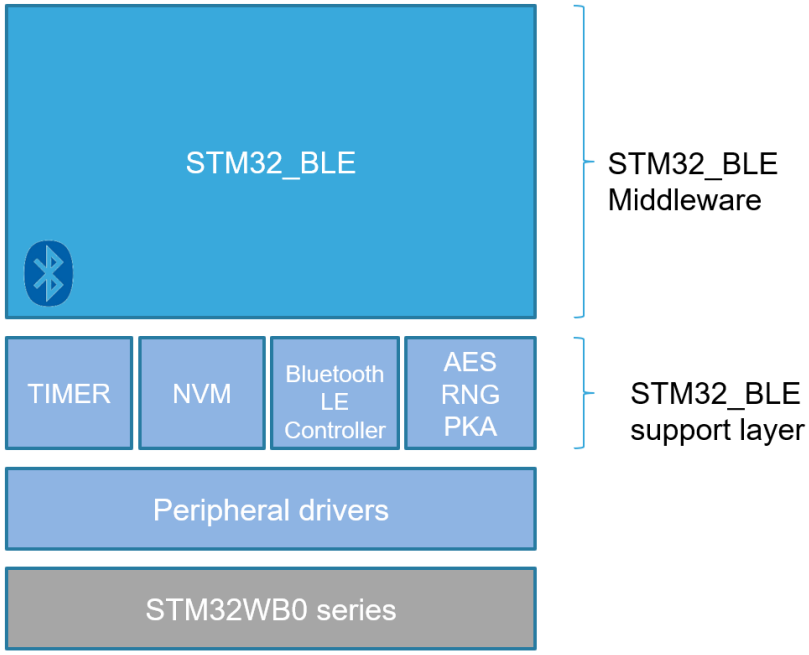
The GATT profile is designed to be used by an application or another profile and defines how to use the contained attributes to obtain information.

For more information about the Bluetooth® LE stack v4.x GATT management, Bluetooth® LE services and characteristics and how to define and handle them, refer to the Bluetooth® LE stack v4.x programming guidelines (PM0274).

4.1.2 Bluetooth® LE stack v4.x architecture

The Bluetooth® LE stack v4.x provides a modular architecture as follows:

Figure 15. Bluetooth® LE stack v4.x architecture



This architecture provides the following new features with related benefits:

1. Code is more modular and testable in isolation:
   - Test coverage is increased
2. Hardware-dependent parts are provided in source form:
   - Radio timer module external to Bluetooth® LE stack (Init API and tick API to be called on user application).
   - NVM module external to Bluetooth® LE stack (Init API and tick API to be called on user application).

Note: Customizing or fixing bugs is made easier.

   - AES, PKA and RNG modules external to Bluetooth® LE stack (Init APIs to be called on user application)
3. Certification targets the protocol part only:
   - It reduces the number of stack versions, since hardware-related problems are mostly isolated in other modules.
   - It reduces the number of certifications.
4. It implements a more flexible and robust radio activity scheduler:
   - It is more robust against late interrupt routines. For example, flash memory writes and/or interrupt disabled by the user.
5. It reduces real-time constraint (less code in interrupt handler):
   - The system gives more time to applications

4.1.2.1 *Bluetooth® LE host stack*

The Bluetooth® LE stack v4.x is a single standard C library called stm32wb0x_ble_stack.a, in binary format, which can be configured in different ways according to user application scenario.

The Bluetooth® LE stack v4.x provides the capability to enable/disable, at compile time, the following Bluetooth® LE stack features based on a user-specific application scenario:

1. Enable/disable controller privacy.
2. Enable/disable LE secure connections.
3. Enable/disable scan capability.
4. Enable/disable data length extension (valid only for the device supporting the data length extension feature).
5. Enable/disable LE 2M and LE Coded PHYs features.
6. Enable/disable extended advertising and scanning features.
7. Enable/disable L2CAP, connection-oriented data service feature (L2CAP-COS).
8. Enable/disable the periodic advertising
9. Enable/disable the periodic advertising with responses
10. Enable/disable constant tone extension (where applicable).
11. Enable/disable LE Power Control and Path Loss Monitoring.
12. Enable/disable the connection capability. It configures support to connections:
    a. If connection option is disabled, connections are not supported; the device is a broadcaster only if scan capability option is disabled, or a broadcaster and observer only if scan capability option is enabled.
    b. If connection option is enabled, connections are supported; device can only act as broadcaster or peripheral if the scan capability option is disabled, or any role (broadcaster, observer, peripheral, and central) if scan capability option is enabled.
13. Enable/disable the connection subrating.
14. Enable/disable the channel classification.
15. Enable/disable the broadcast isochronous streams.
16. Enable/disable the connected isochronous streams.

Dedicated preprocessor options are available in order to select if a specific option must be enabled (1U) or disabled (0U):

```
#define CFG_BLE_CONTROLLER_POWER_CONTROL_ENABLED                    (0U)
#define CFG_BLE_CONTROLLER_SCAN_ENABLED                             (0U)
#define CFG_BLE_CONTROLLER_PRIVACY_ENABLED                          (0U)
#define CFG_BLE_SECURE_CONNECTIONS_ENABLED                          (0U)
#define CFG_BLE_CONTROLLER_DATA_LENGTH_EXTENSION_ENABLED            (0U)
#define CFG_BLE_CONTROLLER_2M_CODED_PHY_ENABLED                     (0U)
#define CFG_BLE_CONTROLLER_EXT_ADV_SCAN_ENABLED                     (0U)
#define CFG_BLE_L2CAP_COS_ENABLED                                   (0U)
#define CFG_BLE_CONTROLLER_PERIODIC_ADV_WR_ENABLED                  (0U)
#define CFG_BLE_CONTROLLER_CTE_ENABLED                              (0U)
#define CFG_BLE_CONTROLLER_POWER_CONTROL_ENABLED                    (0U)
#define CFG_BLE_CONNECTION_ENABLED                                  (0U)
#define CFG_BLE_CONTROLLER_CHAN_CLASS_ENABLED                       (0U)
#define CFG_BLE_CONTROLLER_BIS_ENABLED                              (0U)
#define CFG_BLE_CONNECTION_SUBRATING_ENABLED                        (0U)
#define CFG_BLE_CONTROLLER_CIS_ENABLED                              (0U)
```

The modular configuration options allow the user to exclude some features from the available Bluetooth® LE stack binary library and decrease the overall flash memory.

Some typical Bluetooth® LE configurations could be addressed by using the modular configuration options as follow:

1. Broadcaster only (all modular options set to 0): Send advertising PDU and scan response PDU; receive and process scan request PDU.
2. Broadcaster + observer only (all modular options set to 0, except with `CFG_BLE_CONTROLLER_SCAN_ENABLED`): Send advertising PDU, scan request PDU and scan response PDU; receive and process advertising PDU, scan request PDU and scan response PDU.
3. Basic: All modular options set to 0 except with the capability to connect as a peripheral device (`CFG_BLE_CONNECTION_ENABLED` set to 1).

4. Basic + DLE: All modular options set to 0 except with the connection and the data length extension capabilities (`CFG_BLE_CONNECTION_ENABLED` and `CFG_BLE_CONTROLLER_DATA_LENGTH_EXTENSION_ENABLED` set to 1).
5. Full: All the modular options set to 1.

4.1.2.2 *Footprints*

The library footprints are listed in the following tables.

Table 6. Library footprints

Bluetooth® LE application	Modular configurations	Flash memory footprint (KB)			RAM memory footprint (KB)		
		Total size	Application	Bluetooth® LE stack	Total size	Application	Bluetooth® LE stack
BLE_Heartrate(1)	Only `CFG_BLE_SECURE_CONNECTIONS_ENABLED`, , `CFG_BLE_CONTROLLER_2M_CODED_PHY_ENABLED` and `CFG_BLE_CONNECTION_ENABLED` options are enabled.	124.25	39.82	84.43	15.98	7.03	8.95
BLE_Beacon	Broadcaster only (all modular configuration options are disabled).	34.20	16.11	18.09	10.69	5.51	5.18
BLE_p2pServer(1)	Only `CFG_BLE_SECURE_CONNECTIONS_ENABLED`,`CFG_BLE_CONTROLLER_2M_CODED_PHY_ENABLED` and `CFG_BLE_CONNECTION_ENABLED` options are enabled.	120.95	36.38	82.57	15.62	6.67	8.84
BLE_ TransparentMode	All modular options are enabled.	279.57	67.47	212.10	47.07	20.19	26.88

1. *Further optimization on footprint numbers could be achieved if the LE 2M PHY feature is not required.*

Note: The information in the table above is provided as an example and may not represent the current state of the libraries' footprints. Always refer to the latest software release for the most up to date information.

Note: These numbers are related to the STM32WB09xE applications.

5 How to design an STM32_BLE application using the STM32CubeMX tool

This section describes the key steps to follow to design an STM32_BLE application using the STM32CubeMX tool.

These steps should be followed to define and generate the basic software infrastructure of the STM32_BLE Bluetooth® LE applications provided within the STM32WB0 MCU software package.

The following IPs must be enabled for enabling the STM32_BLE middleware:

- RCC
- RNG
- PKA
- RADIO_TIMER
- RADIO

5.1 STM32CubeMX clock configurations

The following tables provide the available options to configure the overall clock source about RCC (HSE and LSE) on the clock configuration window and the associated implications on the RADIO_TIMER initialization parameters, and on the STM32_BLE CFG_BLE_SLEEP_CLOCK_ACCURACY value.

Note: When selecting LSE or LSI on RCC tab, the user should make sure that the RADIO TIMER IP is initialized inline with the configurations parameters reported on the following tables (refer to the enableInitialCalibration and periodicCalibrationInterval parameters).

Table 7. STM32WB0 STM32CubeMX clock configuration: HSE (RC64MPLL), LSE and RF

STM32CubeMX pinout and configuration		
STM32CubeMX category IPs	**HSE mode and configuration**	**LSI mode and configuration**
RCC	Crystal/Ceramic resonator(1)	Crystal/Ceramic resonator(1)
RADIO_TIMER	-	RADIO_TIMER_InitStruct.XTAL_StartupTime = 320; RADIO_TIMER_InitStruct.enableInitialCalibration = FALSE(1); RADIO_TIMER_InitStruct.periodicCalibrationInterval = 0(1);
STM32_BLE	-	CFG_BLE_SLEEP_CLOCK_ACCURACY to be set according to accuracy of low-speed crystal (ppm)
STM32CubeMX clock configuration		
Refer to the figure below.	Refer to the figure below.	Refer to the figure below.

1. *This value must not be modified.*

Figure 16. STM32CubeMX clock configuration: HSE (RC64MPLL), LSE

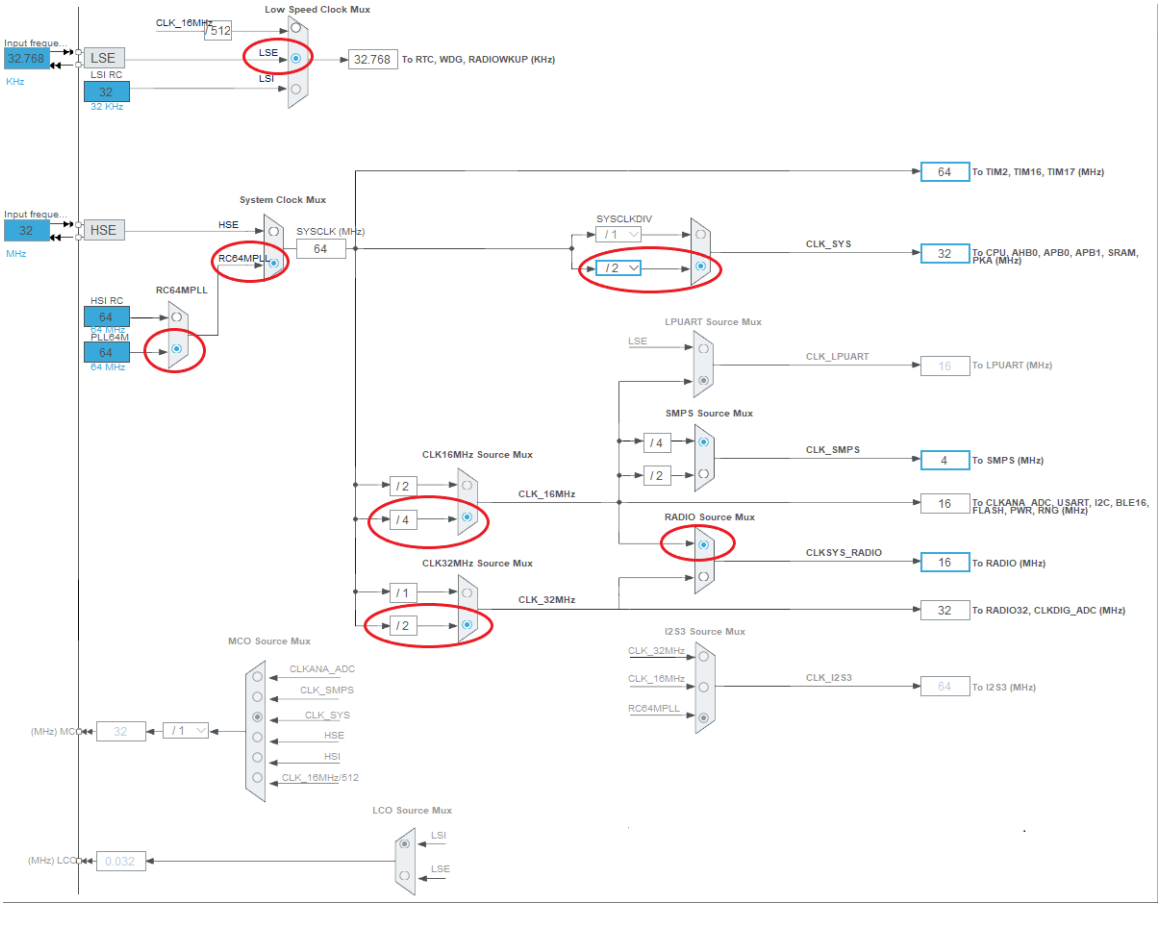


Table 8. STM32WB0 STM32CubeMX clock configuration: HSE (RC64MPLL), LSI and RF

STM32CubeMX pinout and configuration		
STM32CubeMX category IPs	**HSE mode and configuration**	**LSI mode and configuration**
RCC	Crystal/Ceramic resonator(1)	LSE disable(1)
RADIO_TIMER	-	RADIO_TIMER_InitStruct.XTAL_StartupTime=320 RADIO_TIMER_InitStruct.enableInitialCalibration = TRUE(1) RADIO_TIMER_InitStruct.periodicCalibrationInterval = 10000;
STM32_BLE	-	CFG_BLE_SLEEP_CLOCK_ACCURACY=500(1)
STM32CubeMX clock configuration		
-	Refer to the figure below.	Refer to the figure below.

1. *This value must not be modified.*

Figure 17. STM32CubeMX clock configuration: HSE (RC64MPLL), LSI

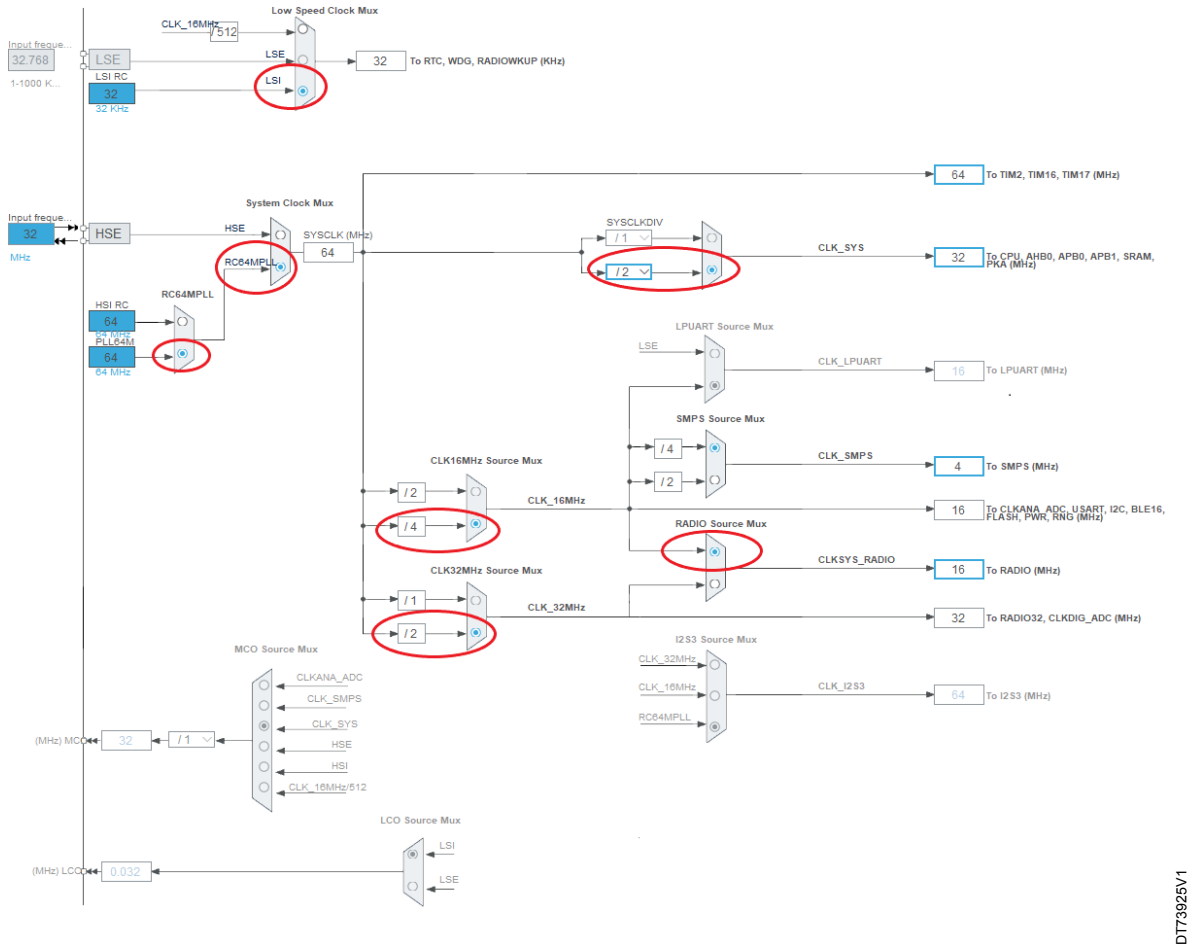


Table 9. STM32WB0 STM32CubeMX clock configuration: HSE (DIRECT HSE), LSE and RF

STM32CubeMX pinout and configuration		
STM32CubeMX category IPs	**HSE mode and configuration**	**LSE mode and configuration**
RCC	Crystal/Ceramic resonator(1)	Crystal/Ceramic resonator(1)
RADIO_TIMER	-	RADIO_TIMER_InitStruct.XTAL_StartupTime=320 RADIO_TIMER_InitStruct.enableInitialCalibration = FALSE(1) RADIO_TIMER_InitStruct.periodicCalibrationInterval = 0;(1)
STM32_BLE	-	CFG_BLE_SLEEP_CLOCK_ACCURACY to be set according to accuracy of low-speed crystal (ppm)
STM32CubeMX clock configuration		
-	Refer to the figure below.	Refer to the figure below.

1. *This value must not be modified.*

Figure 18. STM32CubeMX clock configuration: HSE (DIRECT HSE), LSE

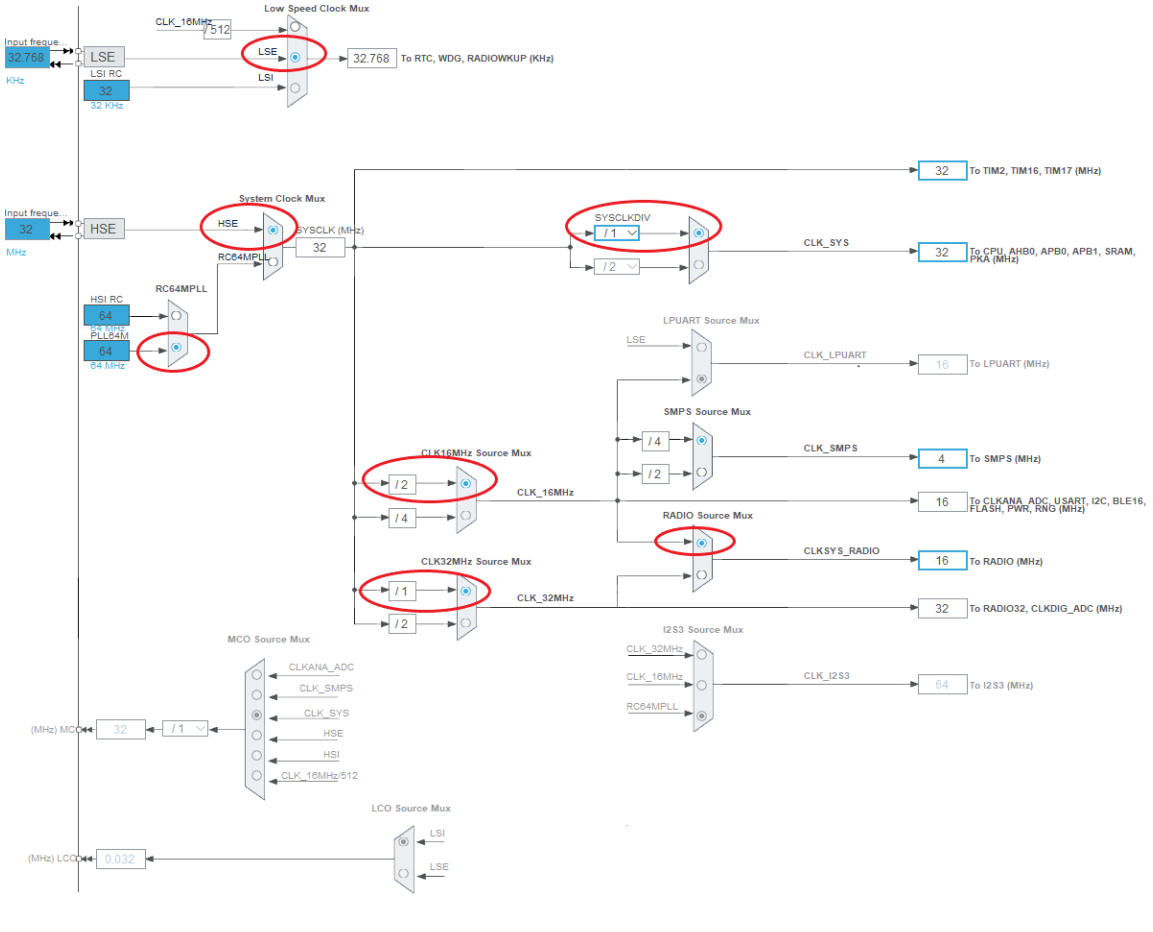


Table 10. STM32WB0 STM32CubeMX clock configuration: HSE (DIRECT HSE), LSI and RF

STM32CubeMX pinout and configuration		
STM32CubeMX category IPs	**HSE mode and configuration**	**LSI mode and configuration**
RCC	Crystal/Ceramic resonator(1)	LSE disable(1)
RADIO_TIMER	-	RADIO_TIMER_InitStruct.XTAL_StartupTime=320 RADIO_TIMER_InitStruct.enableInitialCalibration = TRUE(1) RADIO_TIMER_InitStruct.periodicCalibrationInterval = 10000;(1)
STM32_BLE	-	CFG_BLE_SLEEP_CLOCK_ACCURACY =500(1)
STM32CubeMX clock configuration		
-	Refer to the figure below.	Refer to the figure below.

1. *This value must not be modified.*

Figure 19. STM32CubeMX clock configuration: HSE (DIRECT HSE), LSI

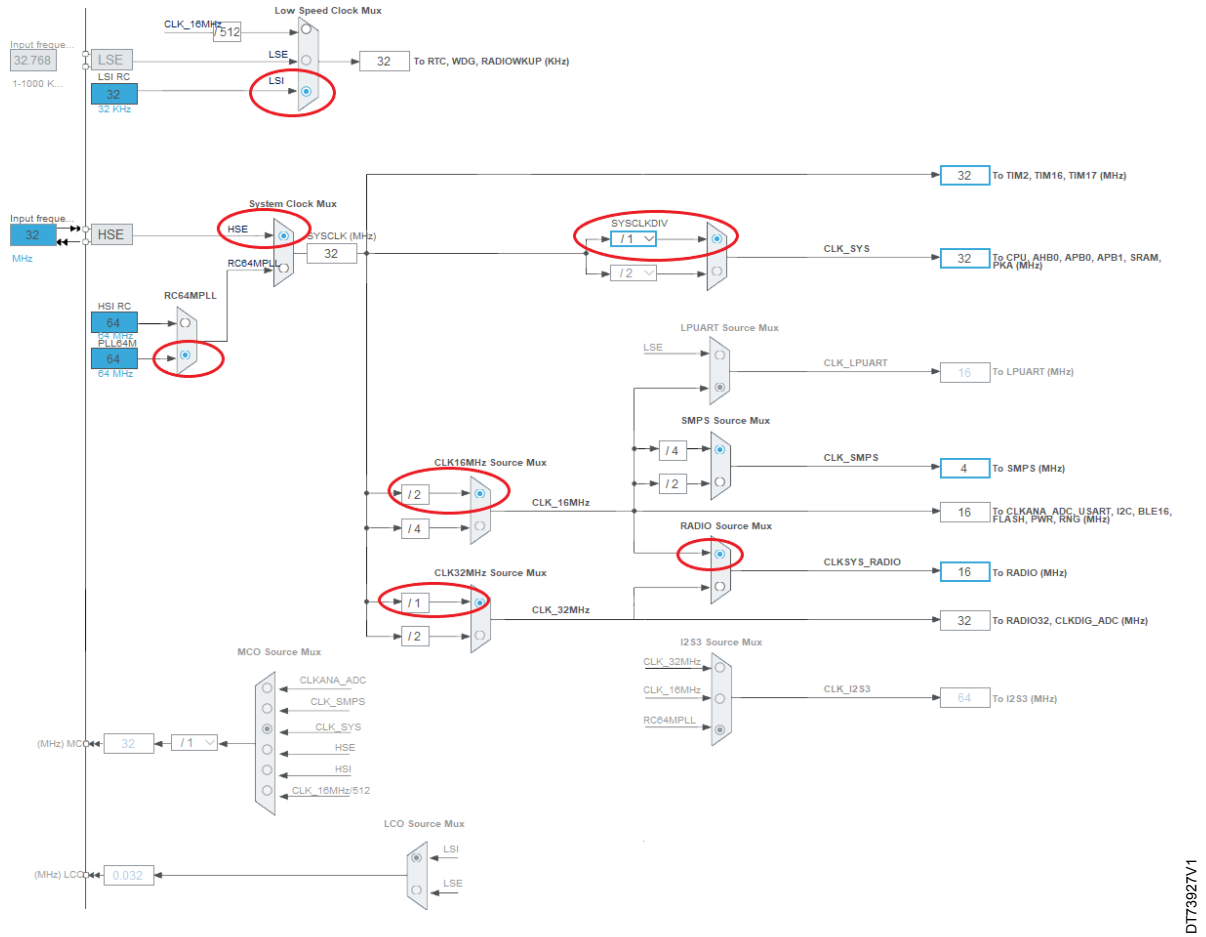


Table 11. STM32WB0 STM32CubeMX clock configuration: HSE (HSI RC), LSE and RF

STM32CubeMX pinout and configuration		
STM32CubeMX category IPs	**HSE mode and configuration**	**LSI mode and configuration**
RCC	HSE crystal/ceramic resonator(1)	Disabled(1)
RADIO_TIMER	-	RADIO_TIMER_InitStruct.XTAL_StartupTime=320 RADIO_TIMER_InitStruct.enableInitialCalibration = FALSE(1) RADIO_TIMER_InitStruct.periodicCalibrationInterval = 0;(1)
STM32_BLE	-	CFG_BLE_SLEEP_CLOCK_ACCURACY to be set according to accuracy of low-speed crystal (ppm)(1)
STM32CubeMX clock configuration		
-	Refer to the figure below.	Refer to the figure below.

1. *This value must not be modified.*

Note: The HSI configuration cannot be used when enabling the RADIO IP. The source code for the HSI initialization with clock divider 1 should be:

```
/* Configure the SYSCLKSource and SYSCLKDivider  */
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
  RCC_ClkInitStruct.SYSCLKDivider = RCC_RC64MPLL_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_WAIT_STATES_1) != HAL_OK)
  {
    Error_Handler();
  }
```

Figure 20. STM32CubeMX clock configuration: HSE (HSI RC), LSE

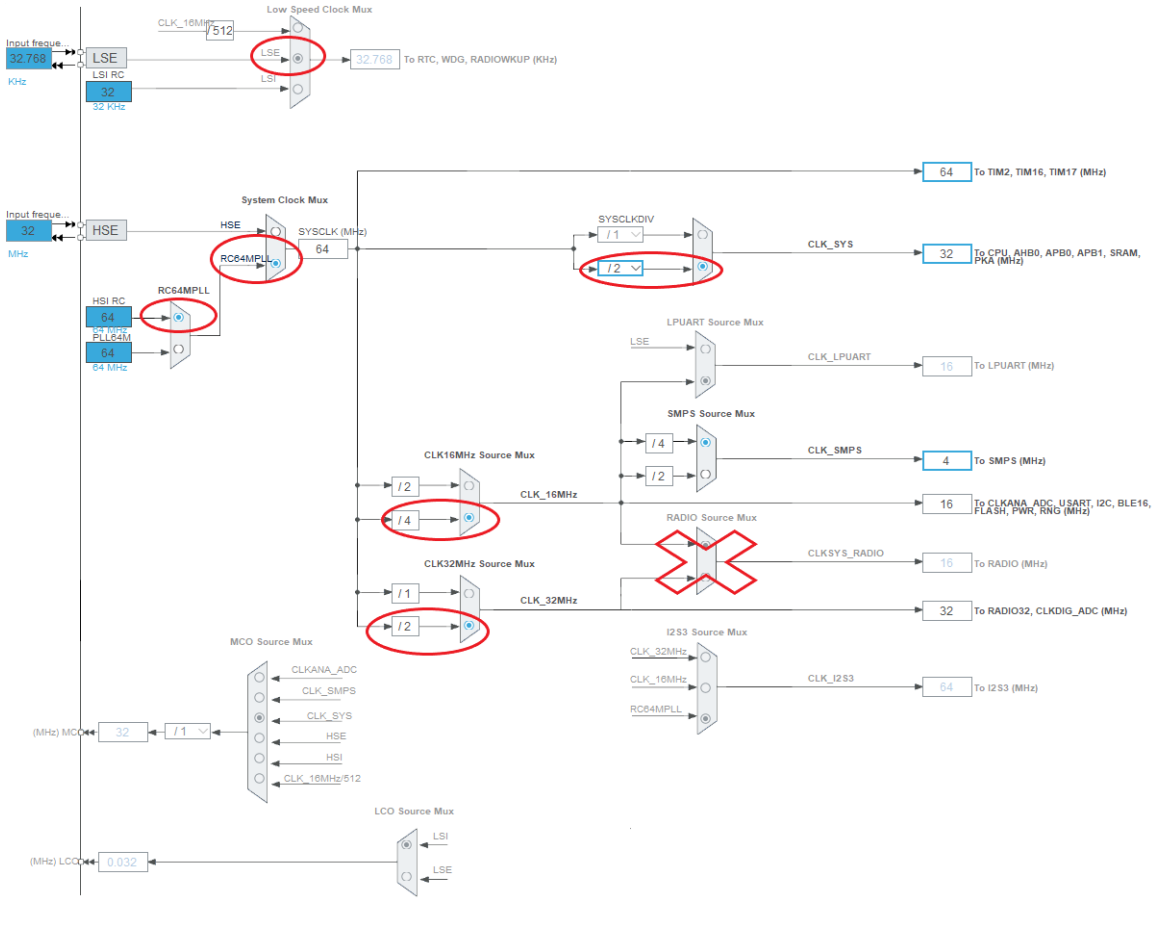


Table 12. STM32WB0 STM32CubeMX clock configuration: HSE (HSI RC), LSI and RF

STM32CubeMX pinout and configuration		
STM32CubeMX category IPs	**HSE mode and configuration**	**LSI mode and configuration**
RCC	HSE crystal/ceramic resonator(1)	Disabled(1)
RADIO_TIMER	-	RADIO_TIMER_InitStruct.XTAL_StartupTime=320 RADIO_TIMER_InitStruct.enableInitialCalibration =TRUE(1) RADIO_TIMER_InitStruct.periodicCalibrationInterval = 10000;(1)

STM32CubeMX pinout and configuration		
STM32_BLE	-	CFG_BLE_SLEEP_CLOCK_ACCURACY = 500(1)
STM32CubeMX clock configuration		
-	Refer to the figure below.	Refer to the figure below.

1. *This value must not be modified.*

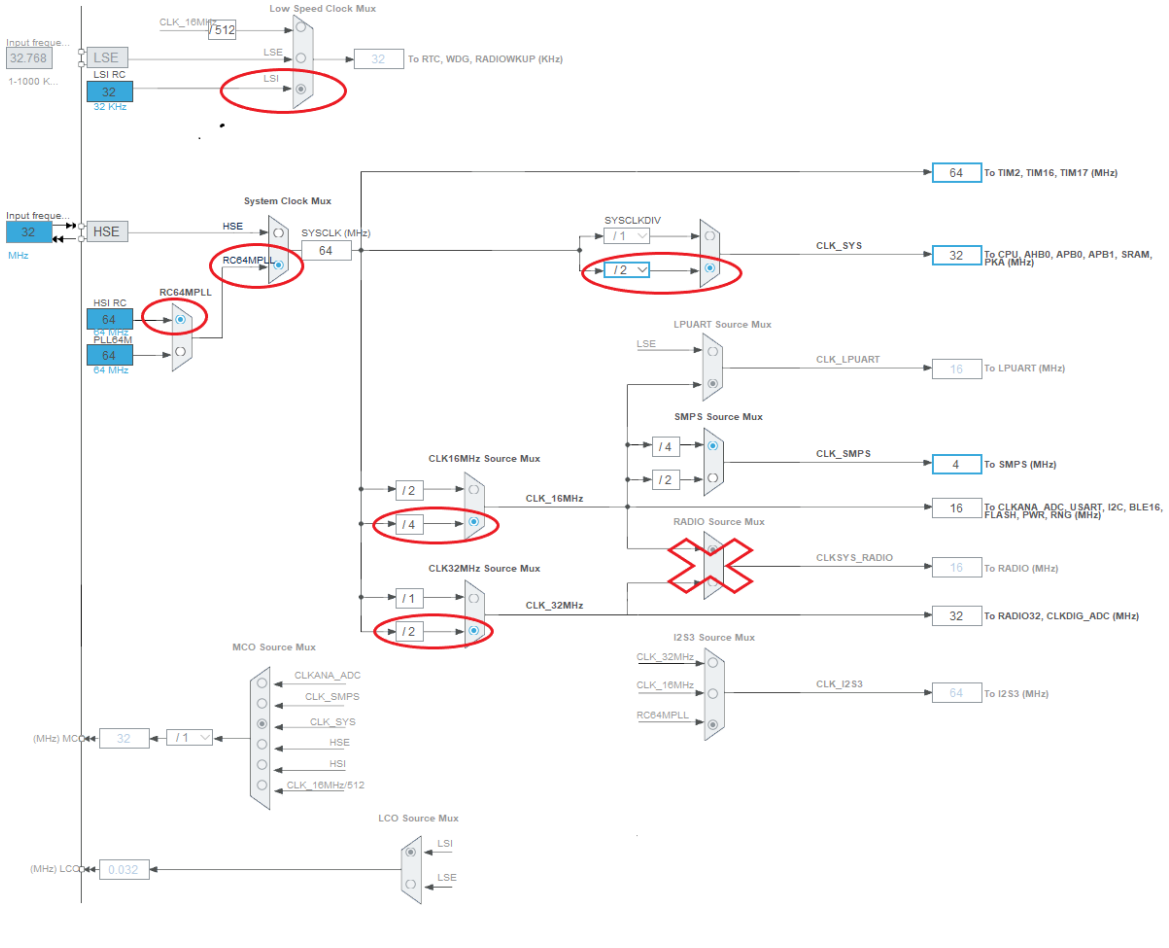
Note: The HSI configuration cannot be used when enabling the RADIO IP.

The source code for the HSI initialization with clock divider 1 should be:

```
/* Configure the SYSCLKSource and SYSCLKDivider  */
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
  RCC_ClkInitStruct.SYSCLKDivider = RCC_RC64MPLL_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_WAIT_STATES_1) != HAL_OK)
  {
    Error_Handler();
  }
```

Figure 21. STM32CubeMX clock configuration: HSE (HSI RC), LSI



DT73929V1

5.2 STM32CubeMX IP configurations when addressing STM32_BLE applications

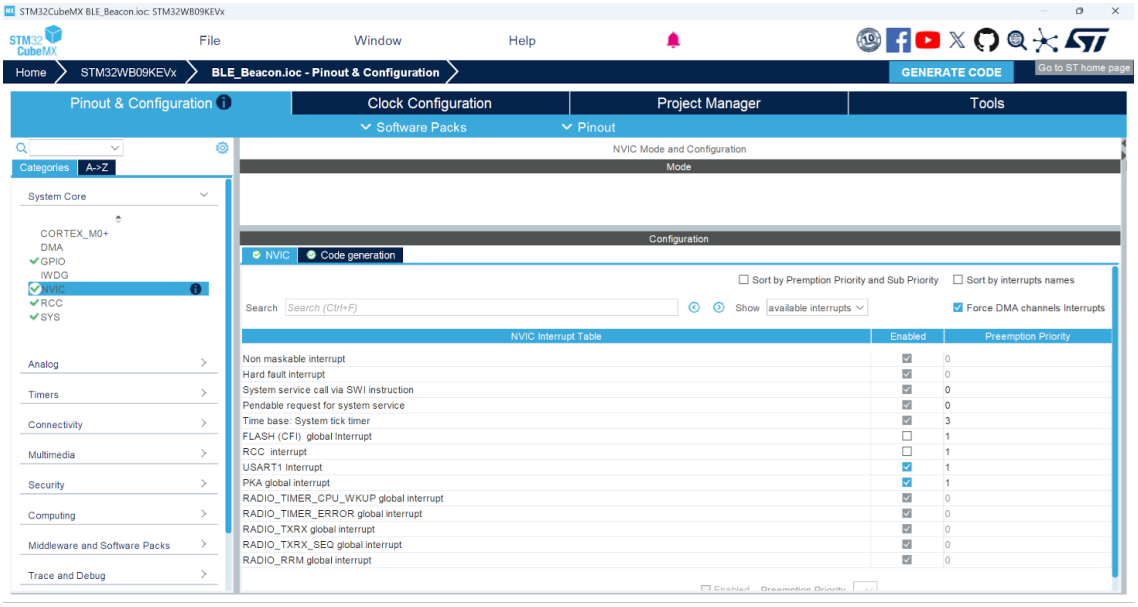
STM32CubeWB0 MCU software package provides a full set of STM32_BLE applications. In order to enable the STM32_BLE Middleware, some IPs must be enabled with specific configurations through STM32CubeMX tool as follow:

1. NVIC IRQs handlers priorities
2. RCC IP as defined on Section 5.1: STM32CubeMX clock configurations
3. RNG IP with the option to do not generate the init API on main.c file (refer to Figure 25. STM32CubeMX IP advanced settings)
4. PKA IP
5. RADIO_TIMER IP as defined on Section 5.1: STM32CubeMX clock configurations, with all IRQ handlers and priority set to 0
6. RADIO IP with all IRQ handlers and priority set to 0
7. USART IP with the option do not generate the init API on main.c file (refer to Figure 25. STM32CubeMX IP advanced settings): this enables to use the BSP COM functionalities and get optimized USART I/O framework.

When targeting STM32_BLE applications or RADIO examples, demonstrations all the IPs IRQs priorities except for RADIO and RADIO_TIMER should be set to a value greater than 0.

As consequence, the following NVIC IRQ priorities should be set through STM32CubeMX tool for targeting the STM32_BLE applications:

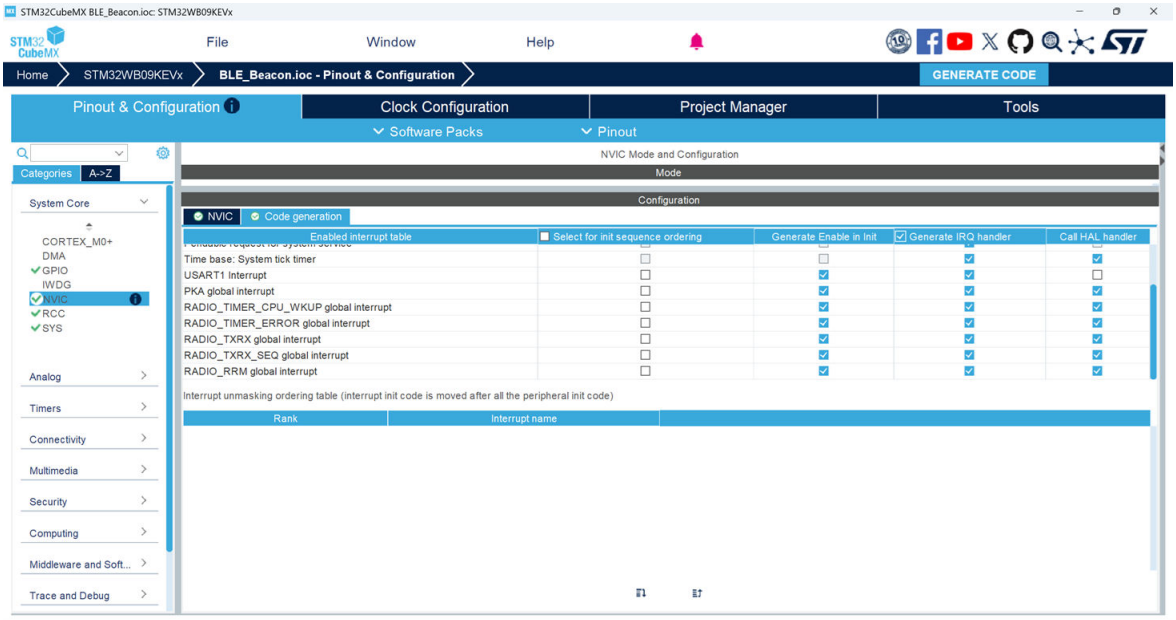
Figure 22. STM32CubeMX NVIC IRQ handle priorities



Note: RADIO_RRM global interrupt is used on Bluetooth LE stack v4.1 or later.

Regarding the generation of IRQ handlers, the following configurations should be set through STM32CubeMX tool for targeting the STM32_BLE applications:

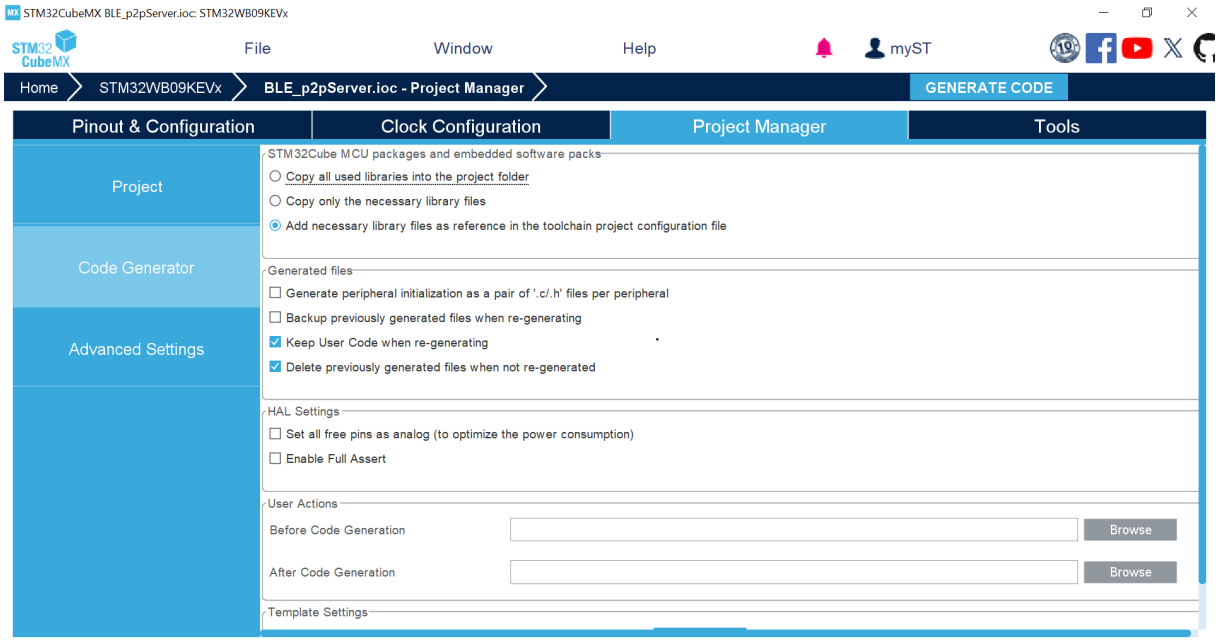
Figure 23. STM32CubeMX NVIC IRQ handler generation



Note: RADIO_RRM global interrupt is used on Bluetooth LE stack v4.1 or later.

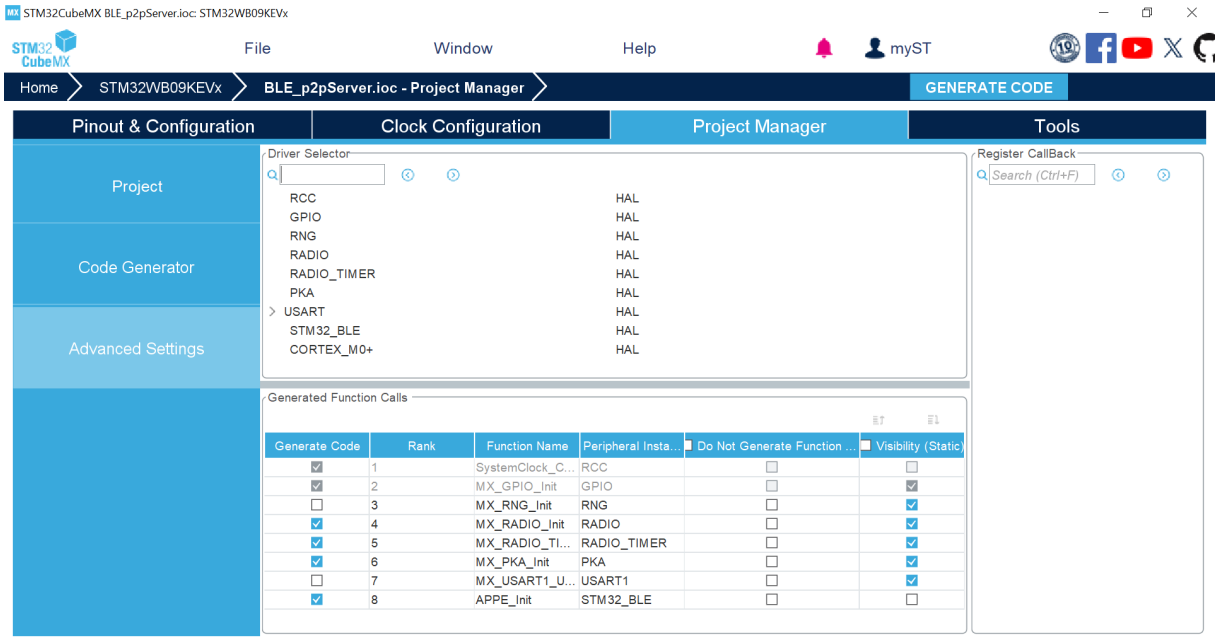
The following setting on STM32CubeMX code generator should be selected in order to properly generate an STM32_BLE application project:

Figure 24. STM32CubeMX IP code generator



The specific IPs configurations used for targeting and STM32_BLE applications can be defined through the STM32CubeMX, Project Manager, Advanced Setting tab as follow:

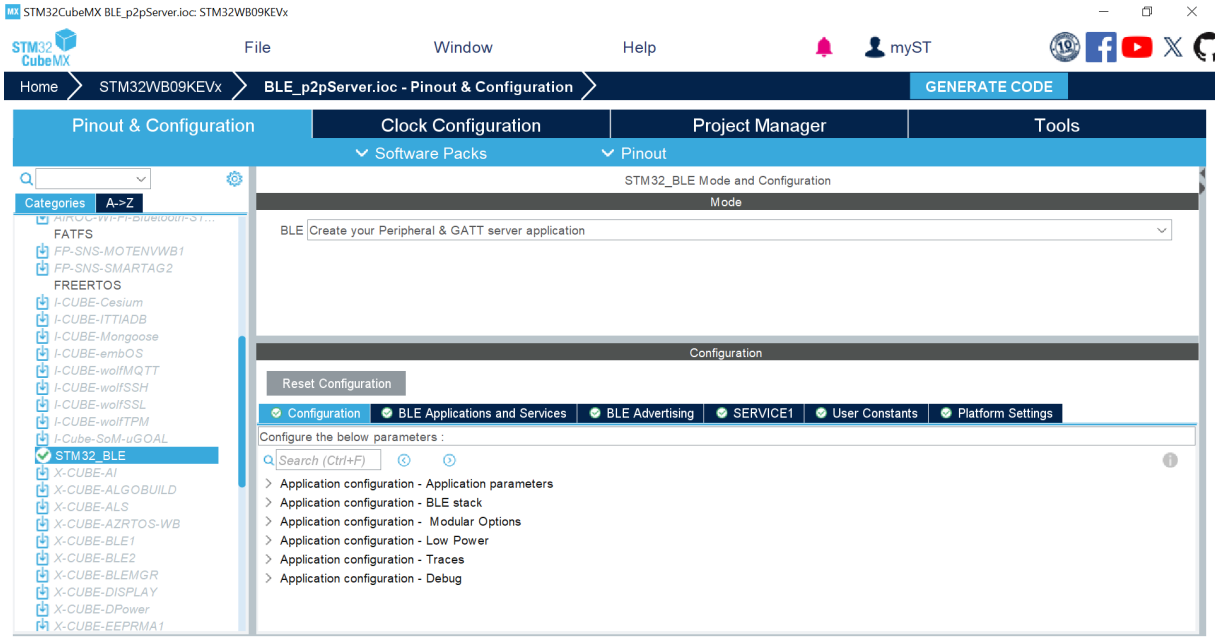
Figure 25. STM32CubeMX IP advanced settings



5.3 How to build STM32_BLE applications with STM32CubeMX

Once the STM32_BLE middleware has been enabled on STM32CubeMX, user can proceed with building a STM32_BLE application based on his Bluetooth® LE application scenario, by using the STM32_BLE Configuration tab.

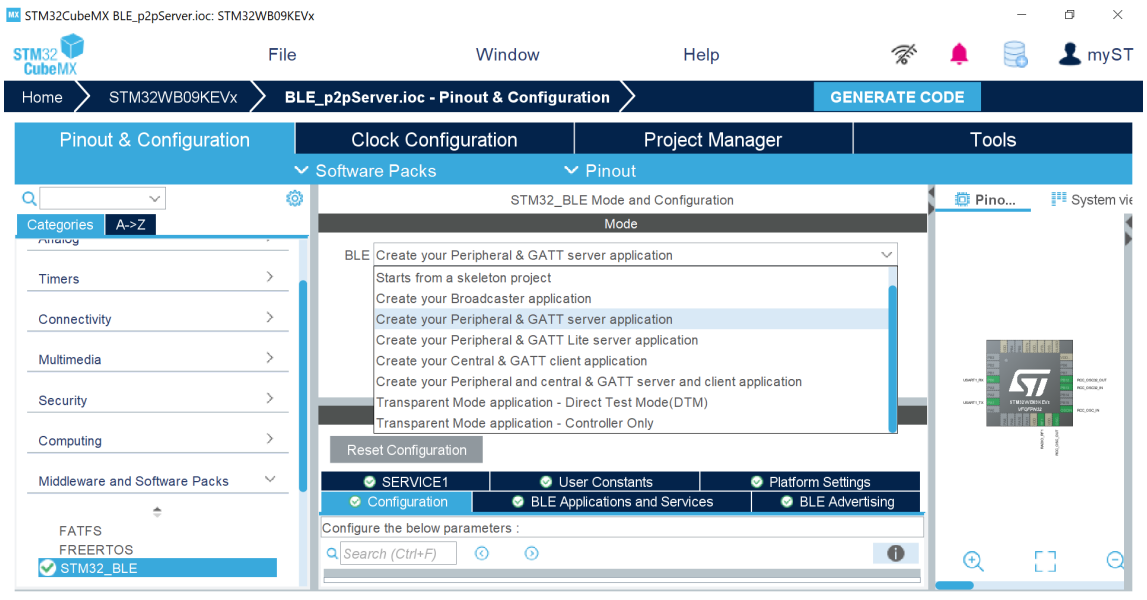
Figure 26. STM32_BLE STM32CubeMX middleware configuration tab



User should perform the following main steps:

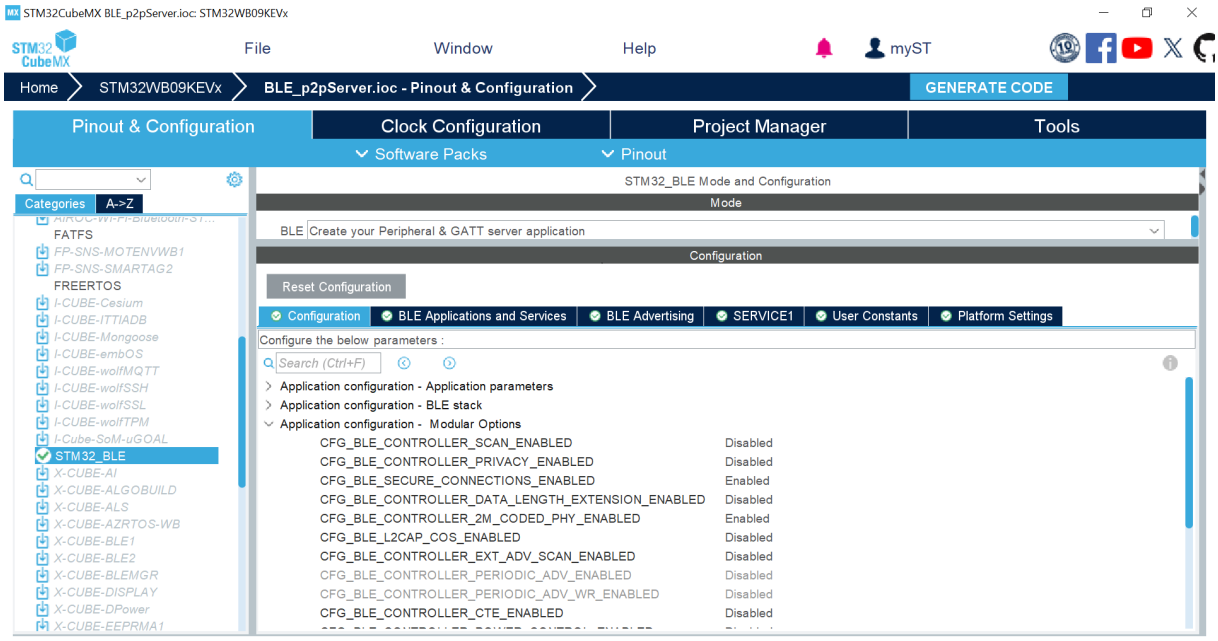
1. Select the Bluetooth® LE application type through the STM32_BLE Mode and Configuration tab. The following options are available:
   a. Start from a Bluetooth® LE skeleton project
   b. Create a broadcaster only application
   c. Create a Peripheral & GATT server application
   d. Create a Central & GATT client application
   e. Create a Peripheral, Central & GATT server, client application
   f. Create a Bluetooth® LE Transparent Mode – Direct test Mode (DTM) application
   g. Create a Bluetooth® LE Transparent Mode HCI controller only – Direct test Mode (DTM) application

Figure 27. STM32_BLE mode and configuration options



Once selected the Bluetooth® LE application scenario, the user defines the specific set of Bluetooth® LE features requested on the user application. This can be done through the STM32_BLE Application Configuration - Modular Options tab which allows selecting which Bluetooth® LE options must be enabled or disabled.

Figure 28. STM32_BLE modular options

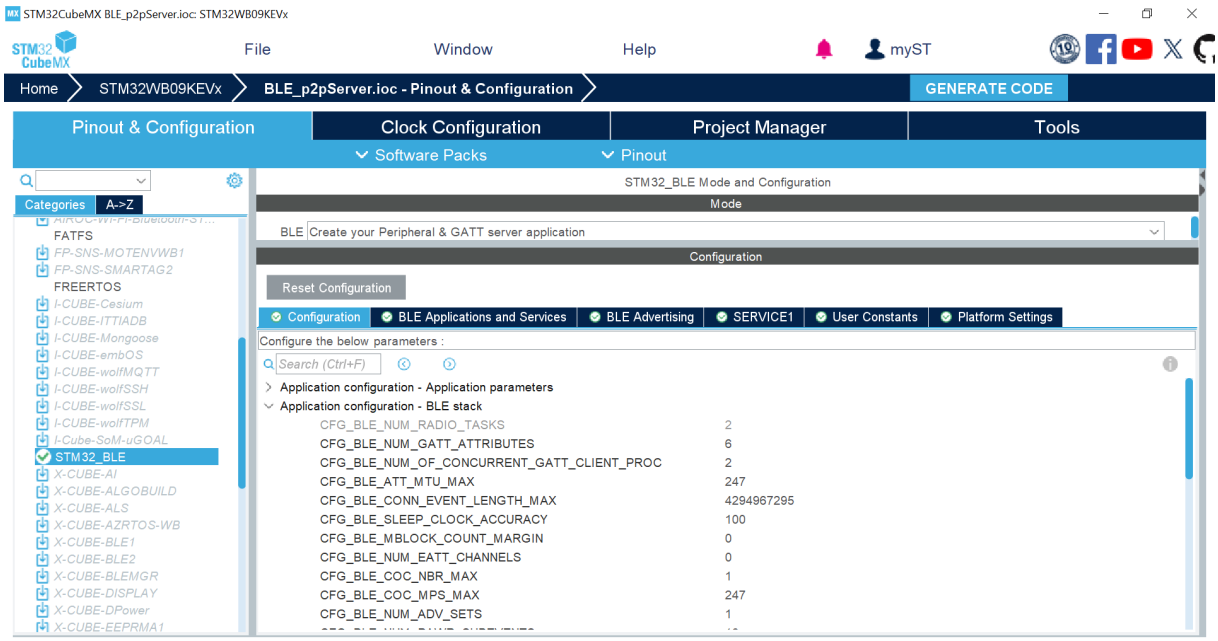


Note:

1. *Each specific STM32_BLE application mode has a different set of default modular configuration options. Target is to activate the minimal requested modular options for providing the Bluetooth LE features needed by the specific application, in order to optimize the memory footprint.*
2. *Refer to the Bluetooth® LE stack v4.x programming guidelines (PM0274) for more detailed information about the Bluetooth LE stack modular configuration options.*

At this stage, user should then define the proper Bluetooth LE stack initialization parameters through the STM32_BLE Application Configuration – BLE stack tab.

Figure 29. STM32_BLE Ble stack initialization parameters

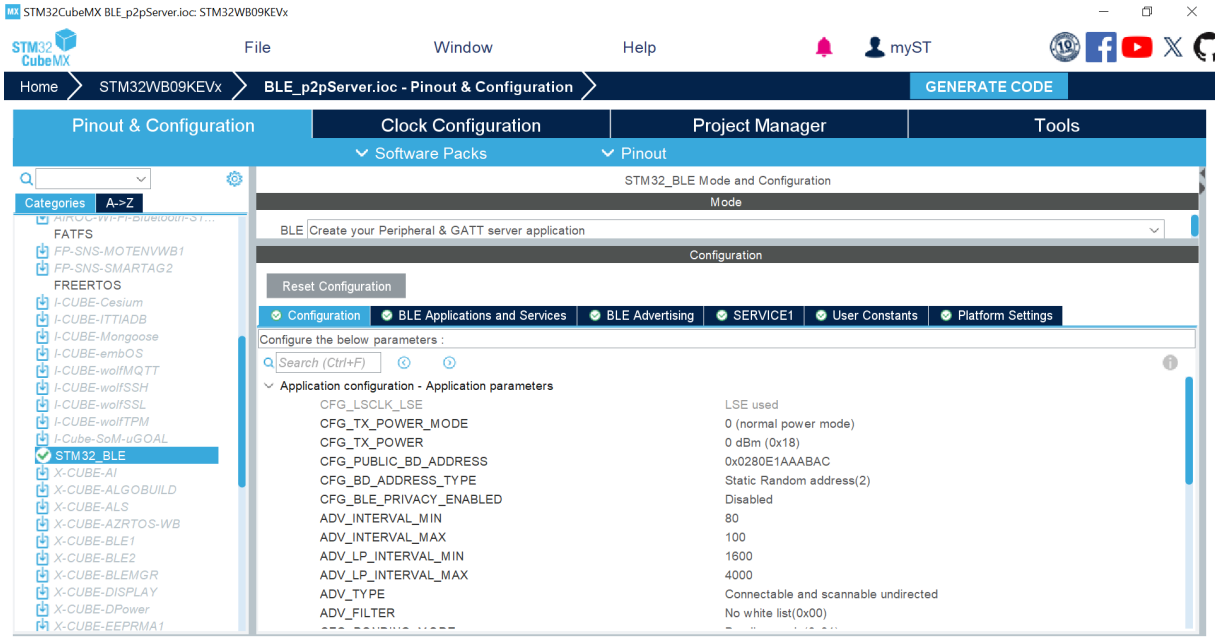


Note: *Refer to the Bluetooth® LE stack v4.x programming guidelines (PM0274) for more detailed information about the Bluetooth® LE stack initialization parameters description and values.*

Once the Bluetooth® LE stack modular configuration options and stack initialization parameters have been defined, the user could set some values related to Bluetooth® LE specific features through the STM32_BLE Application parameters tab:

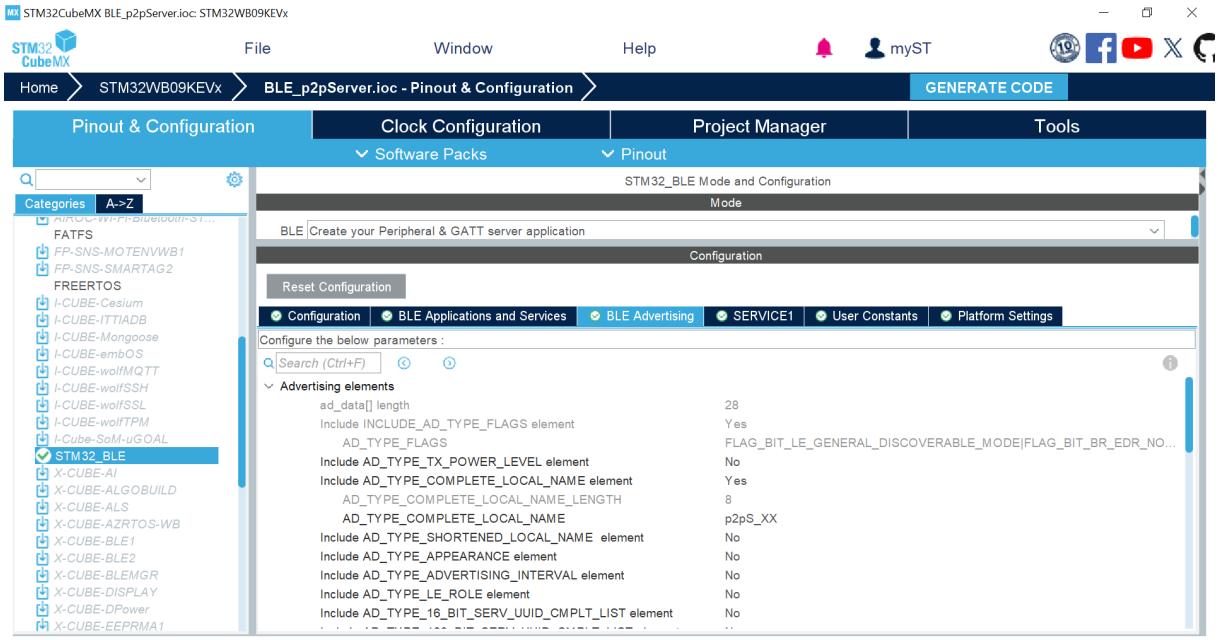
1. TX power
2. Public address
3. Address type
4. Advertising interval
5. Security parameters
6. …

Figure 30. STM32_BLE application parameters



Note: The Bluetooth® LE advertising data and types should be also defined by using the STM32_BLE, BLE Advertising tab.

Figure 31. **10 STM32_BLE BLE advertising tab**



5.4 How to configure the Bluetooth® LE services and characteristics through STM32_BLE STM32CubeMX

When user selects a STM32_BLE application with GATT server capability, user is allowed to define the Bluetooth® LE services and related characteristics by defining the related parameters as described in the images below: names, UUID type and value, Type for services and names, characteristic UUID type and value, characteristic properties, and value length.

Figure 32. **STM32_BLE service definition**

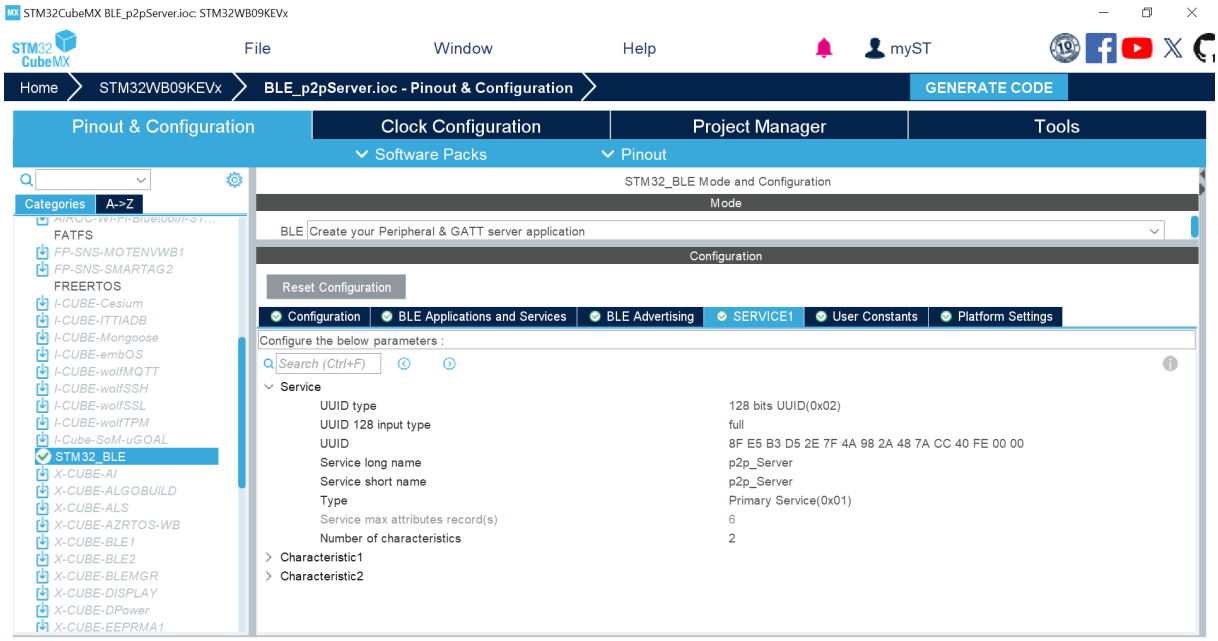
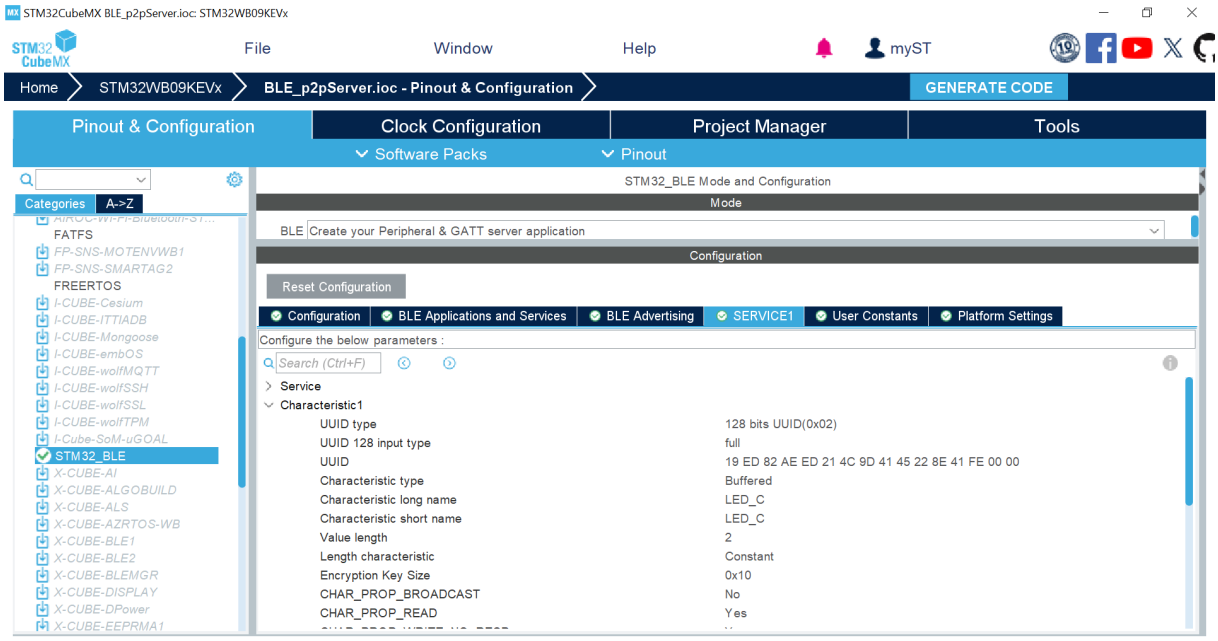


Figure 33. STM32_BLE characteristic definition



The Bluetooth® LE stack v4.x allows to define the following characteristics:

1. Non buffered characteristics. Buffer is not requested on characteristic definition. This option is typically used for handling characteristic linked to real-time data (i.e. sensor data).
2. Buffered characteristics. Buffer is requested on characteristic definition.

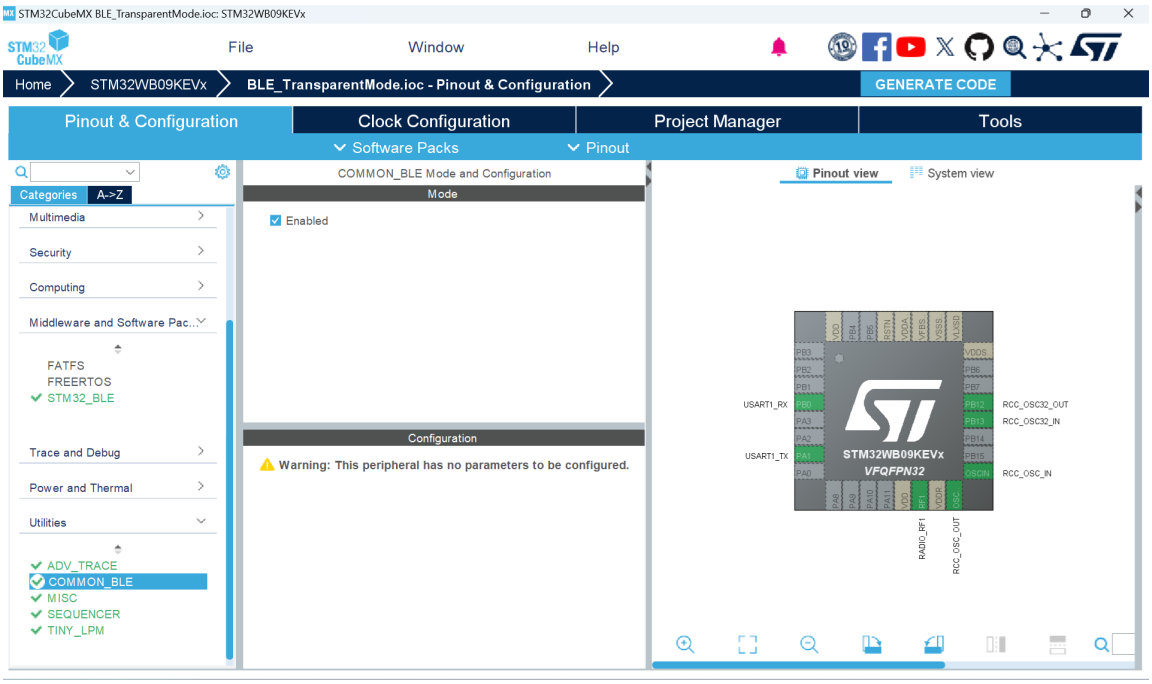
Note: Refer to the Bluetooth® LE stack v4.x programming guidelines (PM0274) for more detailed information about the Bluetooth® LE stack v4.x service and characteristics definition.

5.5 STM32_BLE & Utilities

When STM32_BLE is enabled, the related applications are generated with the STM32 sequencer (stm32_seq.c) and the low-power manager (stm32_lpm.c) utilities software frameworks.

Furthermore, the centralized system files available in the `Projects\Common\BLE` folder are used by default. A dedicated utility COMMON_BLE, enabled by default, selects if the generated application refers to the system files in the new centralized `Common\BLE` folders or not. If not, the system files are generated under the specific application system folder.

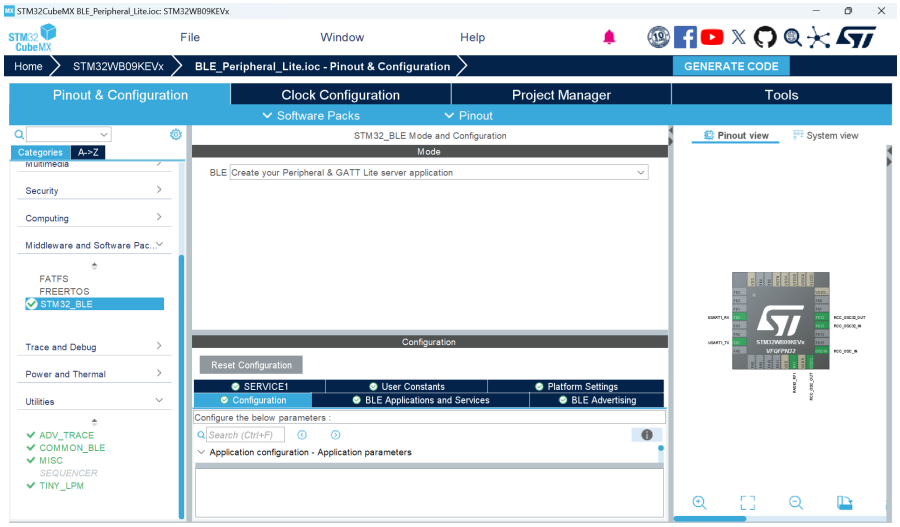
Figure 34. Utilities, COMMON_BLE



DT76209V1

On STM32CubeMX, STM32_BLE middleware, STM32_BLE Mode and configuration options, the user can select the "Create your peripheral and GATT Lite server application". With this option, it's possible to define an application SW framework without a sequencer and low-power manager utilities. The user can target a simple bare metal application framework (no tasks and while (1) with state machines), and a simplified power manager framework, which can be enabled or disabled.

Figure 35. STM32_BLE Peripheral Lite application



DT76208V1

Revision history

Table 13. **Document revision history**

Date	Version	Changes
14-June-2024	1	Initial release.
11-Jul-2024	2	Added reference to STM32CubeWB0 in Section Introduction. Updated Section 4.1.2.2: Footprints..
4-Nov-2024	3	Updated: • Section 2.3: Application view • Figure 3. Application project view organization • Figure 27. STM32_BLE mode and configuration options Added: • Section 2.7.2: Common • Section 5.5: STM32_BLE & Utilities • Section 2.8.2: Low-power modes
05-May-2025	4	Updated: • Figure 1. STM32WB09xE key features • Section 5.1: STM32CubeMX clock configurations
05-Nov-2025	5	Updated: • Section 2.8.2.3: Low-power current consumption figures • Section 4.1.2.2: Footprints • Section 5.2: STM32CubeMX IP configurations when addressing STM32_BLE applications

Contents

List of tables

List of figures