AN5928

Application note

How to build a short range wireless application with STM32WBA MCUs

Introduction

This application note guides designers through the steps required to build specific Bluetooth® Low Energy, Zigbee®, and Thread® applications based on STM32WBA series microcontrollers.

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® Low Energy, Zigbee®, and Thread® technologies, and controller system services, such as low-power management and task sequencing.

For more information, such as deep dive concepts, instructions, and examples, visit the wiki website or refer to www.st.com.

1 List of acronyms and abbreviations

Table 1. Acronyms and abbreviations

Acronyms	Definitions
ACI	Application command interface
ATT	Attribute protocol
BLE	Bluetooth® Low Energy
BSP	Board support package
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
ISR	Interrupt service routine
L2CAP	Logical link control and adaptation protocol
LL	Link layer
LPBAM	Low-power background autonomous mode
LSE	Low-speed external oscillator
NVIC	Nested vectored interrupt controller
NVM	Nonvolatile memory
OS	Operating system
PHY	Physical layer
PLL	Phase-locked loops
QOS	Quality of service
RTC	Real-time clock
RTOS	Real time operating system
SNVMA	Simple NVM arbiter
SoC	System on a chip
SW	software
TZ	TrustZone®
WFI	Wait for interrupt (Arm assembly code)

2 General information

The STM32WBA device is a multiprotocol wireless and ultra-low-power device. It embeds an Arm®CPU Cortex®-M33 core alongside a 2.4 GHz radio compliant with diverse wireless protocols such as Bluetooth® Low Energy (BLE), Zigbee®, or Thread®.

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.

In addition to these, the STM32WBA also provides a dedicated security framework.

For more precise explanation relative to protocols, for example, BLE, etc. or system concepts, for example, low-power, refer to our wiki website.

3 STM32WBA overview

3.1 Software architecture

The software architecture is explained from two points of view:

- The application
- The project

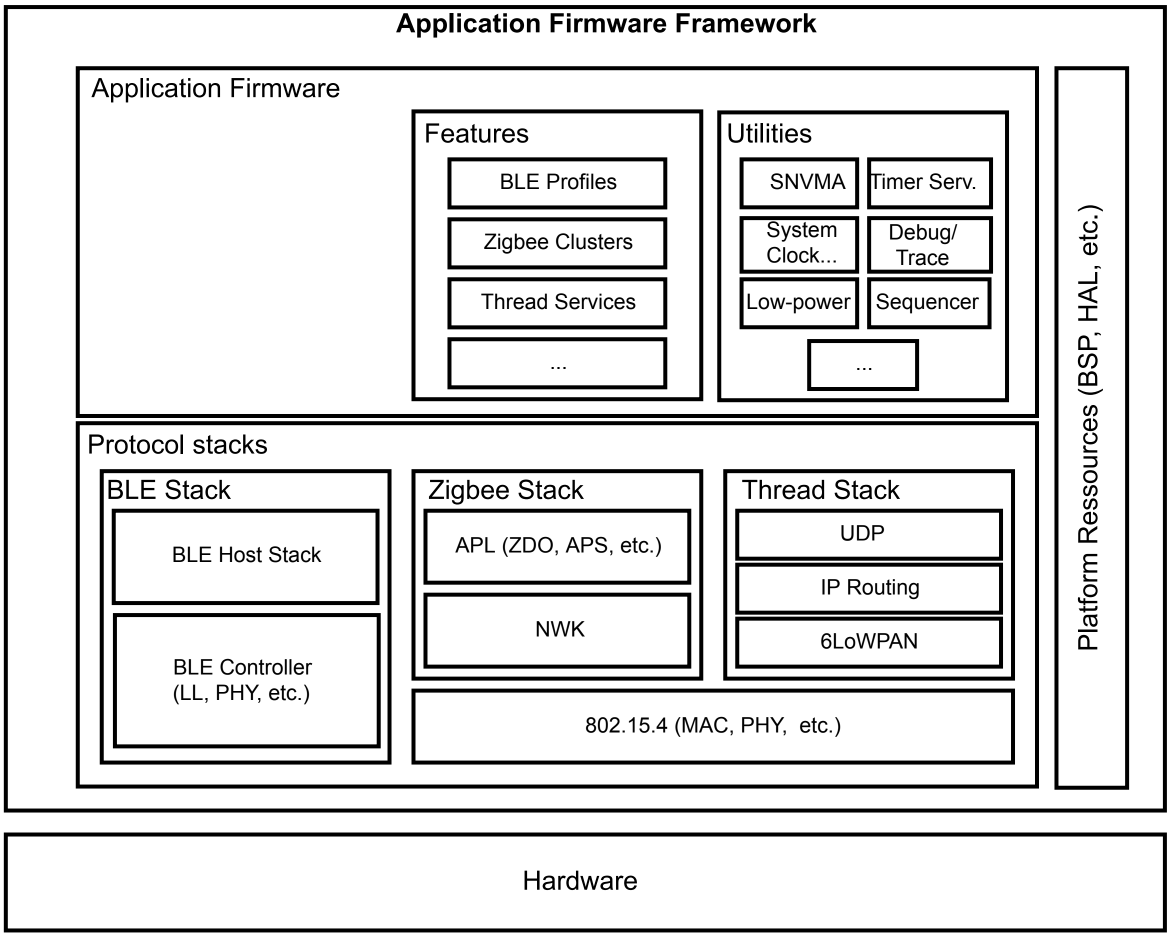
3.1.1 Application view

From the application point of view, the user can rely on multiple modules and SW blocks to create an application. The framework is referred to as the Application Firmware Framework.

The architecture is designed over a 3-level organization:

- Application firmware
- Protocol stack (Bluetooth® Low Energy, Zigbee®, Thread®, etc.
- Platform resources

Figure 1. Application Firmware Framework



3.1.1.1 Application firmware

The user application is based on:

- IPs either from STMicroelectronics or from a third-party supplier.
- Features developed by STMicroelectronics for short-range protocols, such as Bluetooth® Low Energy profiles, etc.

- Utilities required by the application and the protocol stacks.
- Generic utilities that provide easy access to basic features such as low-power management, sequencer, etc.

3.1.1.2 *Protocol stacks*

Multiple protocol stacks can interface with the application and the hardware. For instance, the Bluetooth® Low Energy stack is the main interface for BLE purposes. It is composed of two layers:

- BLE stack: It manages all networks and transport protocols that allow the application to communicate with other devices. Provided by STMicroelectronics, it can also be substituted by the host stack of the user.
- BLE controller: The BLE controller manages the hardware part, the RF state, and guarantees that the RF protocol is correctly followed. It includes the physical layer, the link layer, and the host controller interface.

3.1.1.3 *Platform resources*

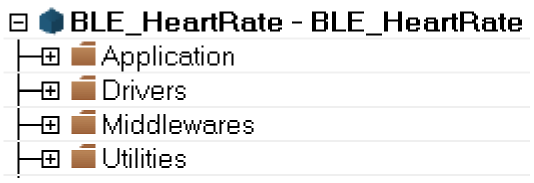
The platform resources include all the HALs, BSPs, and drivers that ease the platform hardware use.

3.1.2 Project view

The project organization differs slightly from the application architecture, as it is repository based. Nevertheless, the above-mentioned concepts are found without difficulties.

This organization is divided into four parts as shown below.

Figure 2. Project view organization



BLE_HeartRate - BLE_HeartRate
Application
Drivers
Middlewares
Utilities

3.1.2.1 *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's setup, scheduler configuration, tasks registration, etc.
  The files contained below this directory are `main.c`, `app_entry.c`, etc.
- Startup: Includes the startup file of the project.
- STM32_WPAN: 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: System-related modules, interfaces, and utilities (PKA, RNG, advanced memory manager, USART, RTDebug, etc.). The user also has access to several configuration files to customize the system experience.

3.1.2.2 *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 (basic timer, capture, PWM, etc.).
  – 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 like the USB and Ethernet. 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 (LED, audio, pushbuttons, etc.) and the drivers for the external components mounted on the used boards (audio codec, IO expander, etc.).
- 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.

3.1.2.3 *Middleware*

Libraries and protocol-based components (BLE stack, for example). This directory contains the BLE services management, system commands, etc.

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.

3.1.2.4 *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.

3.2 SW concepts/features

This chapter covers some of the main software components available on the STM32WBA. Each module concept is briefly described to enhance user appreciation of the platform capabilities.

For further information, visit the online wiki pages. For instance, system modules reference is available: interface, usage, and examples are described for each module on a dedicated page. This information is kept up to date with the latest software release.

3.2.1 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.

3.2.1.1 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

3.2.2 System clock manager

The system clock manager is a feature for system clock management between multiple users. It determines which frequency is best suited to the system needs. This module also handles the clock management during the low-power modes.

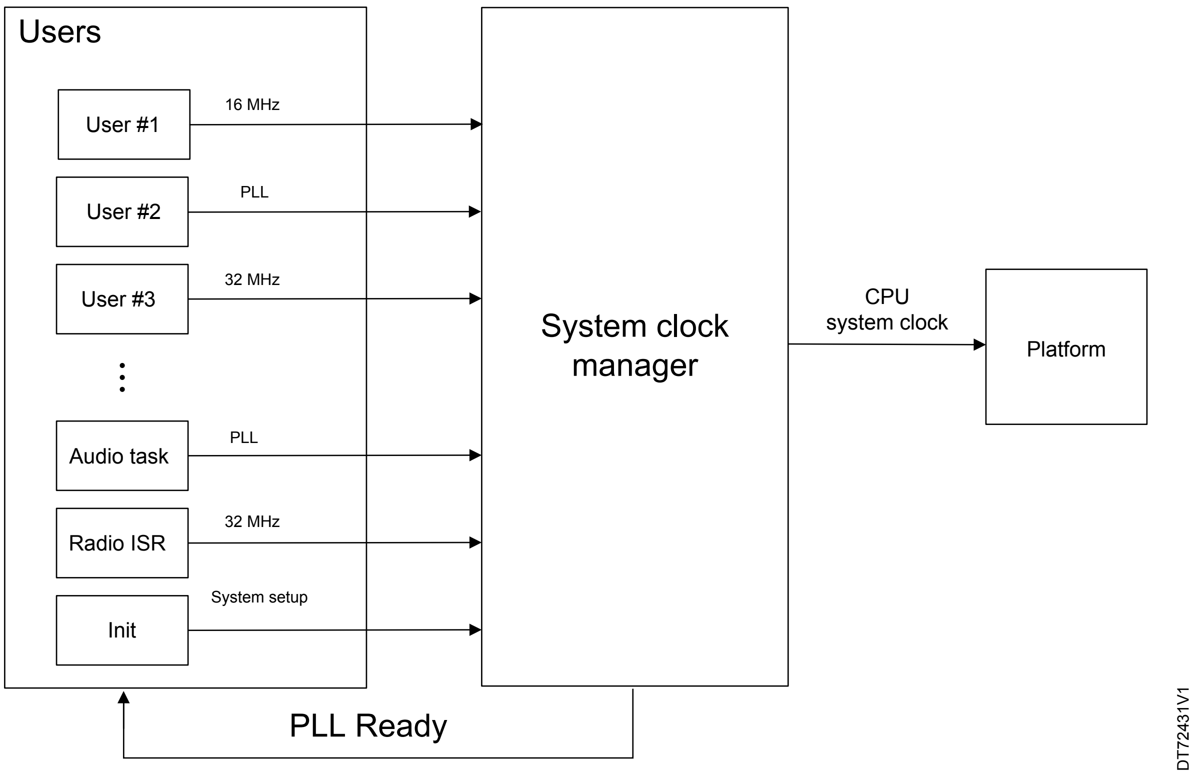
Note: *The clock is cut during several low-power modes.*

3.2.2.1 Concepts

The module is based on a request behavior. Any user, up to 32 users, can request for clock frequency modification. Among the requests, the system clock manager determines the request that fulfills the lower possible configuration.

The best system clock evaluation is realized at each new request.

Figure 3. System clock evaluation



The decrease clock speed request is handled immediately.

However, the increase clock speed request requires more time to setup. This is not blocking as an interrupt mechanism is used to switch to the requested speed. This permits the firmware to keep running at the actual clock speed until the system is ready.

Different clock configurations are possible to request. The supported ones are the following:

- HSE32
- HSE16
- PLL (with a dedicated interface for configuration)

The module also handles the configuration of the flash memory and SRAM latencies, the CPU wait states, the configuration of the VOS and the AHB5 divider configuration, depending on the determined clock speed. For the other impacted modules or systems, the user must adequate and adapt the configurations to the newly set CPU system clock frequency.

3.2.2.2 *Going further*

For more information on the interface, using the interface, and for examples, refer to the wiki system page.

3.2.3 Low-power management

The low-power manager provides a simple interface to receive the input from up to 32 different users and computes the lowest possible power mode the system can use. It also provides hooks to the application before entering or on exiting low-power mode.

3.2.3.1 *Concepts*

From an architecture point of view, the low-power manager has a two section organization. One for the BLE LL and one for the SoC. Both impact each other on their low-power management.

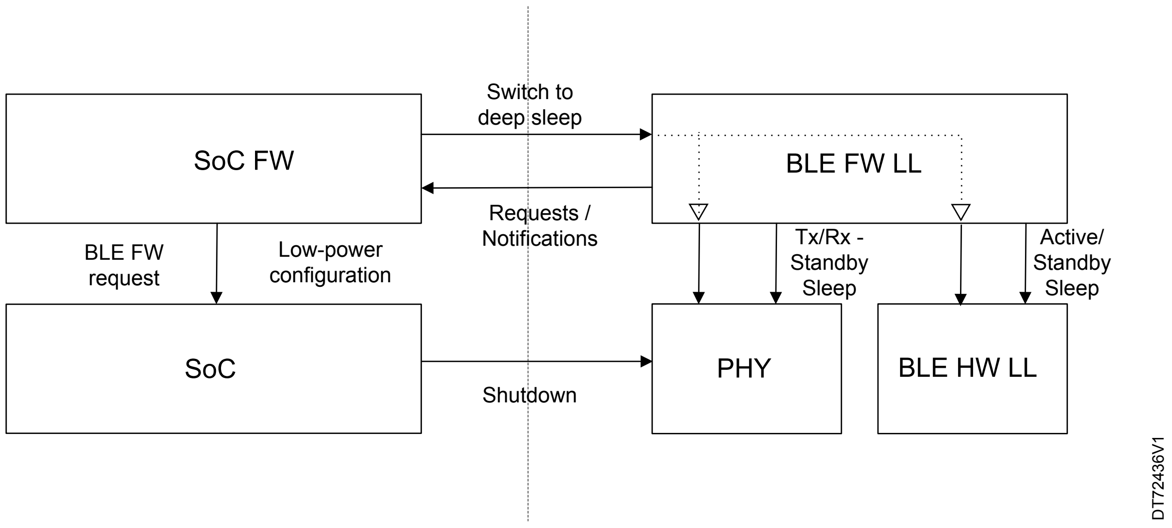


Figure 4. Low-power management

There are numerous different low-power modes available. They all differ according to the section they impact and the intended objective. The different low-power modes of the two sections are as follows:

- BLE IP low-power modes:
  - Hardware LL:
    - Active/Standby: These modes are requested when the radio is operating. Standby is used during IFS.
    - Sleep: This mode is used when there is no radio activity. This is fully handled by the BLE firmware LL. It is possible to access BLE hardware LL registers in this mode. For this, the bus clock needs to be active. To access RXTX SRAM or sequence SRAM, the baseband clock must remain active. This is managed by the LL.
    - Deep-sleep: This mode is used when there is no radio activity and no need to read/write any register in the BLE hardware LL (except the sleep timer and the register to set the low-power mode of the BLE hardware LL). This mode is entered when the BLE hardware LL has remained in idle mode long enough to ensure that the overhead to exit from deep-sleep mode is not significant. This requires profiling to define the threshold value.
  - PHY:
    - Tx/Rx/Standby: These modes are requested when the radio is operating.
    - Sleep: Standby is used during IFS.
    - Shutdown: This mode is applied only when the SoC enters standby (no retention). All radio configurations are lost (calibration, etc.).

- SoC low-power modes:
  - Run mode: The CPU and the system clock are running. It must be used only when it is required to execute code.
  - Sleep mode: Only the CPU is in low-power mode (stops fetching code). The system clock is running. This must be used when a peripheral (that does not have a kernel clock) requires the system clock to operate while the CPU is stopped.
  - Stop0/Stop1 modes: The CPU, the system clock, and the high-speed oscillators are all stopped. LPBAM peripheral can still operate. The 2.4 GHz radio can still be active when in Stop0 Range1. For this, the high-speed system clock and peripheral kernel clocks may be kept running. When a kernel clock is enabled, Stop0 is selected by the SoC hardware. This should be used when standby is not possible.
  - Standby retention: This is the lowest low-power mode targeted in the application. The CPU, the system clock, the high-speed oscillators, and most SoC registers are all lost. It is required to save/restore all contexts and hardware register configurations (that are not retained in the SRAM) to select which content is not lost.
    The time required to save/restore all contexts and hardware registers depends on how many peripherals are used in the application (for example, most of the time GPIO, UART, CPU, DMA, Clocks are used and need to be restored).
    Due to the extra CPU execution time required to exit/enter this mode (mostly due to startup and save/restore operation), it should be used only when the latency to wake-up does not matter and when the “idle” time is long enough so that the power saving in this mode compensates the extra power consumption due to extra CPU processing to enter/exit this mode.
    The radio and BLE hardware LL registers, except for the sleep timer, are lost and need to be restored by the BLE LL firewall. (The BLE LL is in deep-sleep). This should be used, when possible, between radio events to save additional power.
  - Standby mode: SoC registers, radio sleep timer, and BLE hardware LL SRAMs are lost. Only RTC, TAMP, and WKUP are retained. This should be used only when it is decided to restart everything from wake-up. For example, when there is no more radio activity and the application may stop for several days to wake up on a GPIO event or an alarm from the RTC.

Application idle state

The application is either in active state when the CPU executes code or in idle state when the CPU is stopped.

This idle state may be split into three parts:

- Pre-idle task: It holds the last code to be executed before stopping the CPU. It runs outside the critical section so the user may implement a long execution time code without impacting any interrupt latency.
- Idle task: It holds the code to enter low-power mode. A critical section is required to avoid interrupt loss due to race conditions (when WFI is used to stop the CPU). When moving out of low-power mode, the mandatory system setup shall be executed here before any interrupt run. This is required as this is the last code executed before jumping in the interrupt handler.
- Post-idle task: It holds the first code to be executed when moving out of the idle task. It must be executed when the pre-idle task has been executed and only in that case. It runs outside critical sections so the user can implement code with a long execution time without impacting any interrupt latency. At this point, any system setup required must be executed to run the application in Active, but it is not mandatory when followed by an interrupt handler.

3.2.3.2 *Going further*

For more information on the interface, using the interface, and for examples, refer to the wiki system page.

3.2.4 Memory management

An enhanced version of dynamic allocation is available with the STM32WBA series: the advanced memory manager.

It eases dynamic allocations in a multitask/shared environment without adding much complexity.

The new features provided by AMM are the following:

- Support several virtual memories, up to 255
- Minimum size warranty per virtual memory (QOS)
- Callback in case of memory allocation failure

3.2.4.1 *Concepts*

The advanced memory manager comes on top of a basic memory manager, such as Alloc and Free.

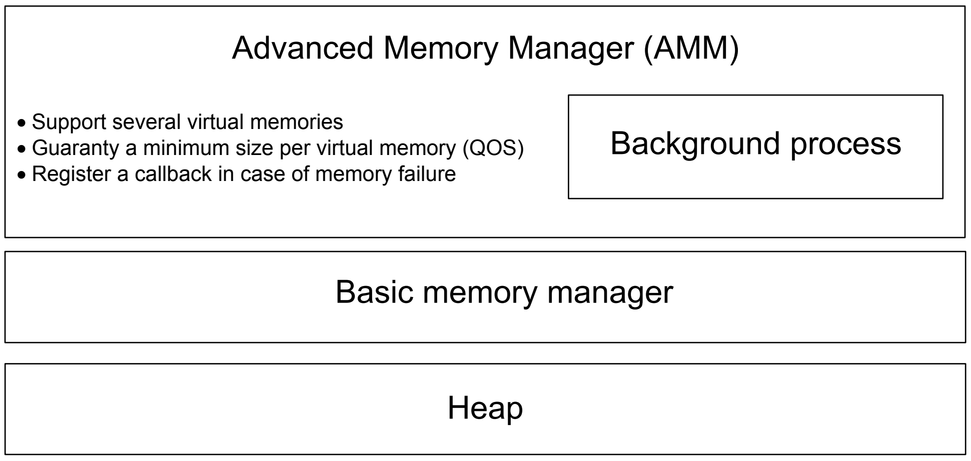

Advanced Memory Manager (AMM)
• Support several virtual memories
• Guaranty a minimum size per virtual memory (QOS)
• Register a callback in case of memory failure
Background process
Basic memory manager
Heap
DT72445V1

Figure 5. Advanced memory manager

Basic memory manager and heap

The basic memory manager can be anything that is capable to allocate/free memory. It must support the capability to merge into one single memory two continuous memories that have been freed (coalescence). It is provided by the user before using the AMM through a registering function.

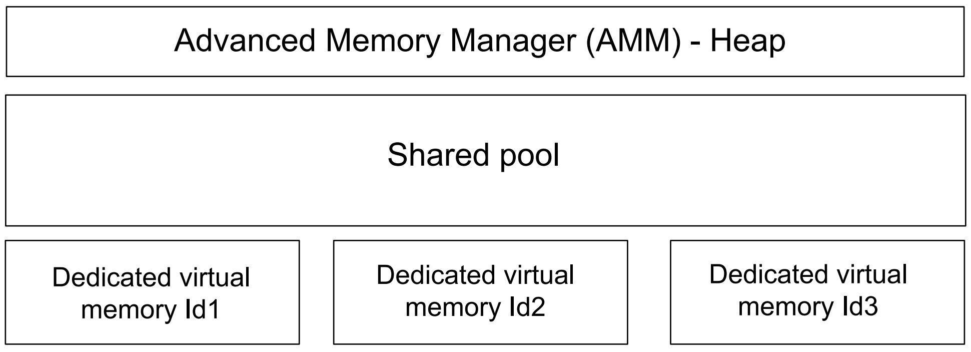
The heap can either be a dedicated memory provided by the application, or the legacy default heap provided by most applications at link time. The size of the heap shall be at least equal to the sum of all the virtual memories combined size.

Virtual memories

In a concurrent execution application, dynamic allocation can encounter some issues with heap sharing. For instance, a task may request a major part of the available heap leading to allocation failure for all other tasks requesting too much memory.

To avoid that kind of issue, the advanced memory manager introduces the concept of virtual memories and shared pool.

Figure 6. Advanced memory manager (AMM)


Advanced Memory Manager (AMM) - Heap
Shared pool
Dedicated virtual memory Id1
Dedicated virtual memory Id2
Dedicated virtual memory Id3
DT72444V1

Virtual memory:

- Dedicated user memory pool, for example, specific VM IDs with up to 255 different IDs
- Ensures that the users have access to the memory amount needed for a minimal execution

Shared pool:

- Mutual memory pool is used by any virtual ID

- Provides memory for an optimal execution
- Size corresponding to the BMM pool size minus the space required for the virtual memories

Retry callback

During program execution, users may encounter some memory allocation failure if there is not enough memory available. Instead of setting up a polling mechanism, in an allocation request, the AMM offers the possibility to register a callback. This callback informs the requester (in an asynchronous way) that some space has been freed, either in the shared pool or in its dedicated virtual pool, and a new allocation request can be submitted.

In some cases, the memory can be freed from a different context than the one it has been allocated from. The callback should not implement any active code and should be used only to set up a new allocation request from the main context.

3.2.4.2 *Going further*

For more information on the interface, using the interface, and for examples, refer to the wiki system page.

3.2.5 Flash memory management

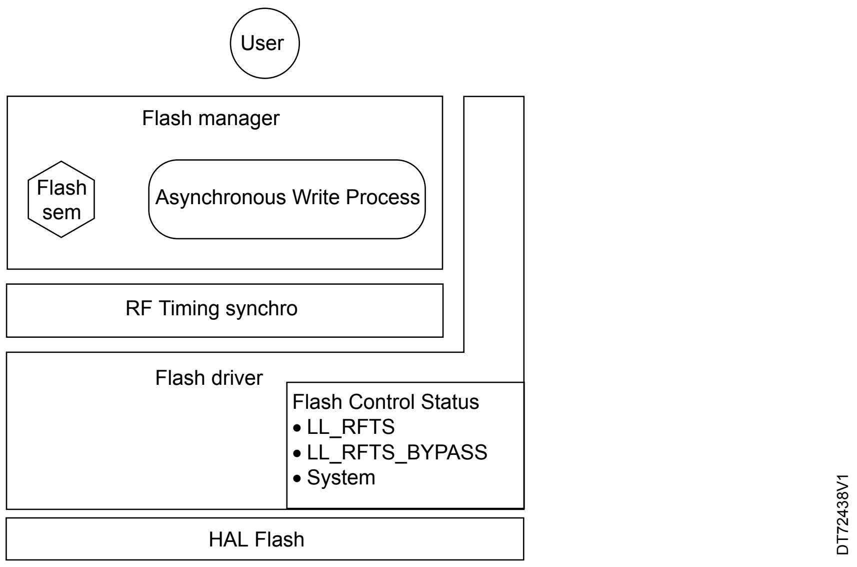
The flash memory management proposes a simple interface to the upper layers to execute operations in flash memory. It manages synchronization between flash memory operations and the BLE LL activity. Thus, users do not have to spend additional effort to synchronize RF timing and flash memory operations.

3.2.5.1 *Concepts*

The flash memory management relies on different concepts:

1. Asynchronous flash memory operations
   All flash memory operations are 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. Four layer organization
   Flash memory management is based upon a four layers distribution composed by:
   – Flash memory manager: Main user interface for flash memory operation (flash memory write, flash erase, etc.).
   – RF timing synchro: Module that realizes synchronization between BLE LL and flash memory operations by activating or deactivating the dedicated flash memory control status.
   – Flash memory driver: Low-level driver abstraction layer. Controlled by the flash memory control statuses.
   – HAL flash memory: Low level driver that interacts directly with the flash memory hardware.

Figure 7. **Four layer organization**

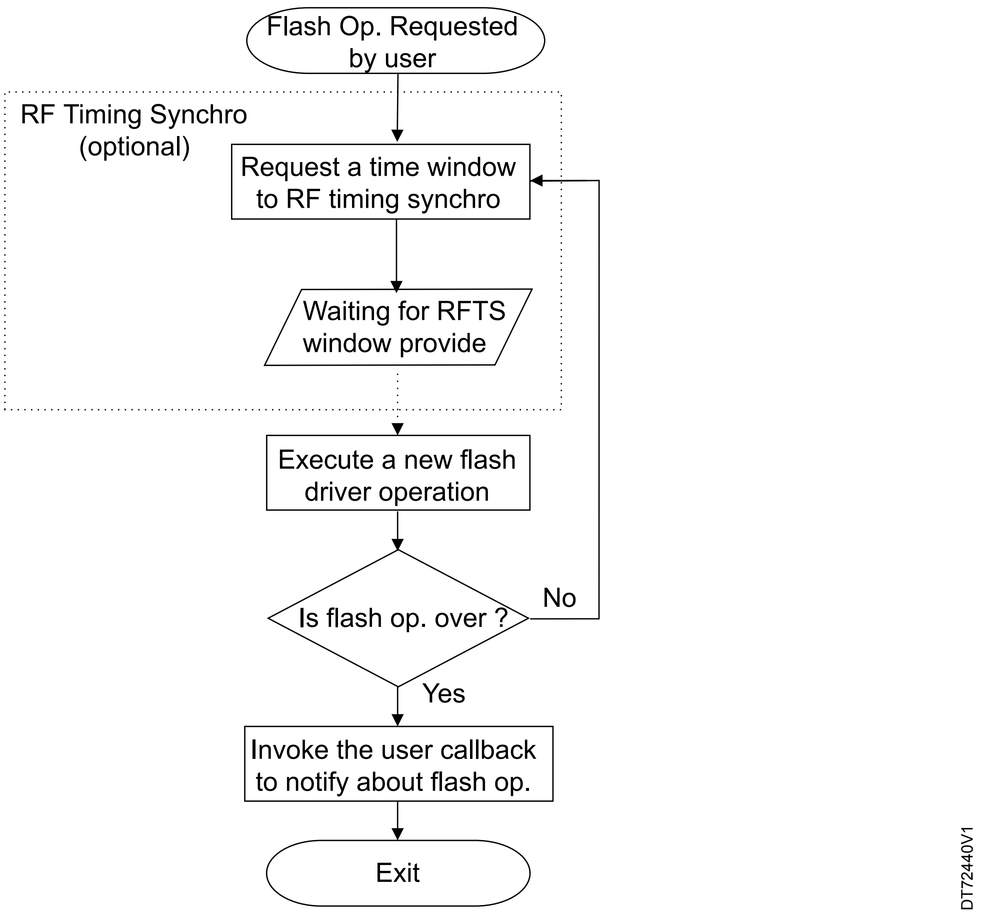


3. Flash memory access
   Flash memory access is protected at two different levels:
   – Flash memory semaphore:
     A semaphore is required to share the flash memory interface between several SW modules. The owner of the semaphore is the only one that can request flash memory operations. The semaphore 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.

Simple routine

The classic flash memory operation routine is represented as follows:

Figure 8. Classic flash memory operation


Flash Op. Requested by user
RF Timing Synchro (optional)
Request a time window to RF timing synchro
Waiting for RFTS window provide
Execute a new flash driver operation
Is flash op. over ?
No
Yes
Invoke the user callback to notify about flash op.
Exit
DT72440V1

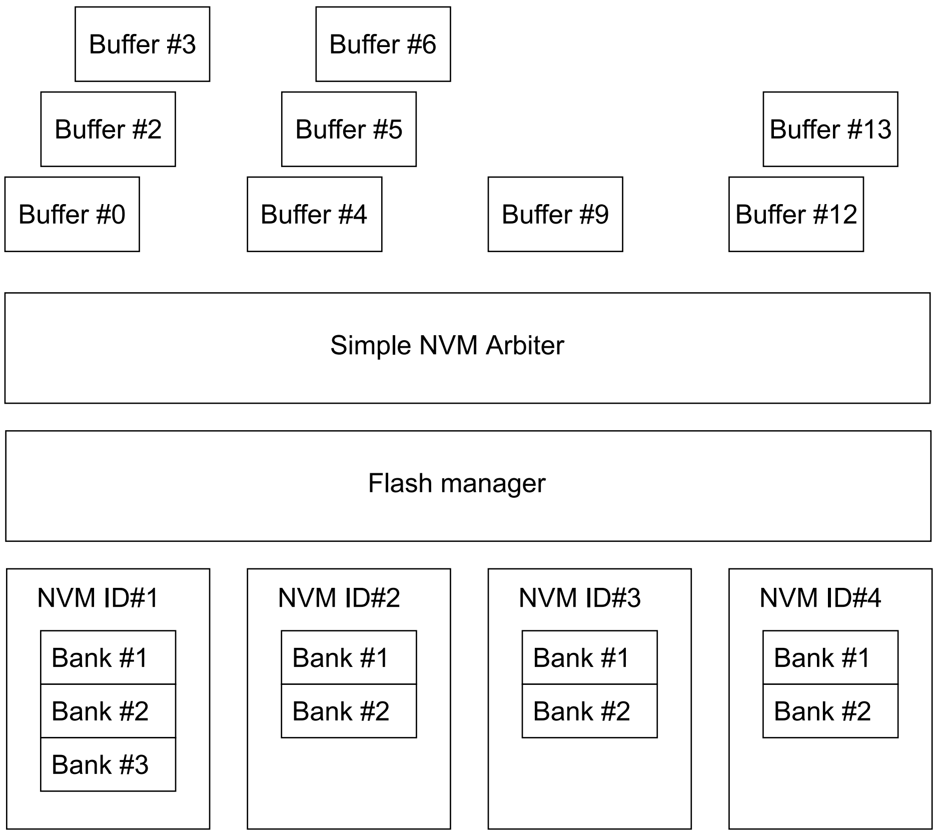
According to the RF state, the flash memory manager adapts its behavior. During RF activity, it requests the help of the RF timing synchro to synchronize flash memory operation and radio activity. Otherwise, the flash memory manager simply executes the flash memory operation until all work is done.

3.2.5.2 *Simple NVM arbiter*

The simple NVM arbiter is a different interface than flash memory operations. It relies on the flash memory manager but adds the possibility to create and manage NVMs, up to 32 NVMs.

NVMs are identified by a unique ID and are composed of multiple banks (at least two, with one for restore and one ready for write). Banks have a sector boundary.

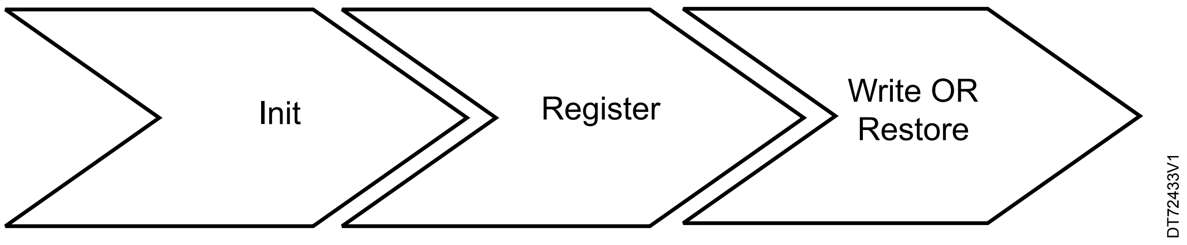
Figure 9. **Simple NVM arbiter**



The user can register up to four buffers in one NVM. During the write operation, all the registered buffers are written in flash memory, that is, a whole bank update. During restore operation, the user restores only one identified buffer.

The classic use of the simple NVM arbiter is shown below:

Figure 10. **Simple NVM arbiter schema**



1. *Initialize the SNVMA.*
2. *Register the buffer(s).*
3. *Execute, as many times as required, the write or restore operation.*

3.2.5.3 *Going further*

For more information on the interface, using the interface, and for examples, refer to the wiki system page.

3.2.6 Trace management: log module

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

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 your project.

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

3.2.6.1 *Concepts*

The log module is organized around three main concepts:

- Configuration
- Timestamp registration
- Printing macros

Configuration

Configuration of the logs regarding different aspects:

- Regions:
  - The user can define multiple regions to organize the application logs clarity.
  - By default, several regions are defined, such as the Bluetooth® Low Energy region (for BLE protocol dedicated logs), or the system region, or the Thread® region.
- Verbose levels:
  - The user can set different levels of verbosity for the application logs, sorting them according to their importance.
  - Depending on the configuration and their level of verbosity, some logs are output or not.
  - There are some already defined verbosity levels that can be reused by the user if needed.
- Colors:
- – Colors are also configurable. The user can set dedicated colors to some regions to ease their readability and highlight them.

Timestamp registration

This allows the user to provide a function to add a timestamp for the log output.

Printing macros

Log printing macros are specific to each region. Printing macros are the interfaces to output logs, they are designed to ease user experience with abstraction and enhance the code clarity.

They manage the verbose level and the region selection to avoid any misutilization. Each region has its own set of dedicated macros.

3.2.6.2 *Going further*

For more information on the interface, using the interface, and for examples, refer to the wiki system page.

3.2.7 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:

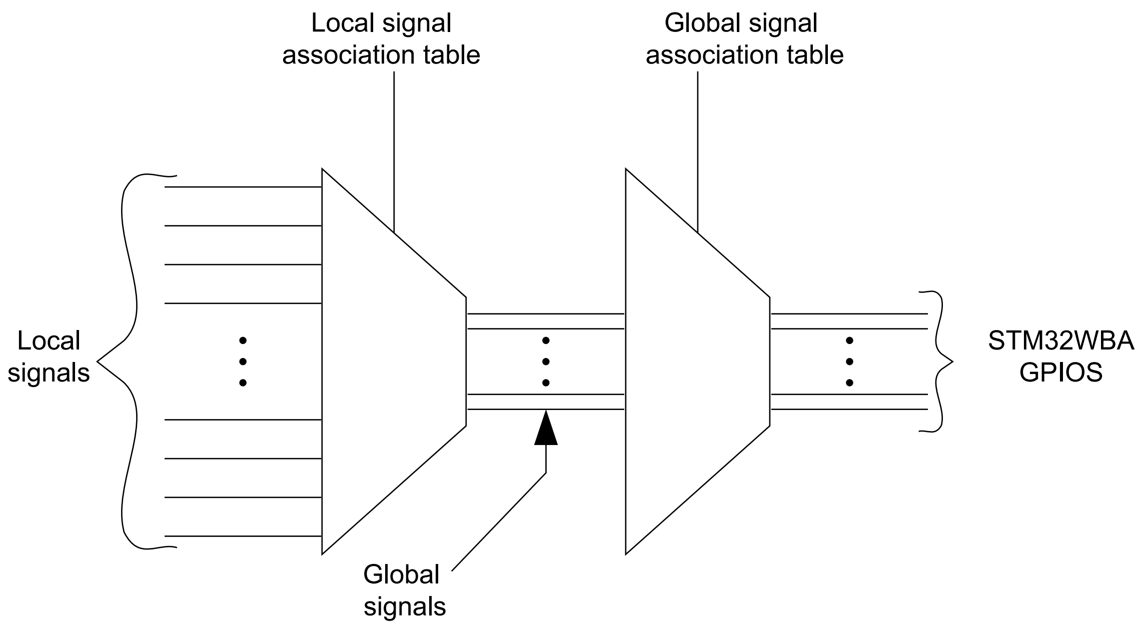
- Start/end of interrupt.
- Start/end of each background process.
- Identification of specific procedures and machine states.

3.2.7.1 *Concepts*

The real time SW 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 SW debug


Local signal association table
Global signal association table
Local signals
STM32WBA GPIOS
Global signals

Signal selection

Debug signals are divided into different categories regarding the concerned layer:

- System SoC signals (interrupts, system services, etc.)
- Link layer signals
- MAC signals
- Host stack signals (BLE, Open Thread, Zigbee, Matter)
- Application signals

As some of these layers are delivered in library format, the debug signal selection cannot be done in the SW component itself.

- The desired signal is selected in a general configuration file.
- In every SW component that supports RT SW debug, all the **local** debugging signals are used. Regarding the **global** signal selection, the **global** signal is used or not.

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

GPIO associated to 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 SW signal to the desired GPIO at compilation time.

3.2.7.2 *Going further*

For more information on the interface, using the interface, and for examples, refer to the wiki system page.

3.2.8 FUOTA

The Firmware Update Over the Air (FUOTA) allows the user to update the application during runtime without any wire connection.

3.2.8.1 *Concepts*

The FUOTA architecture is based on two components:

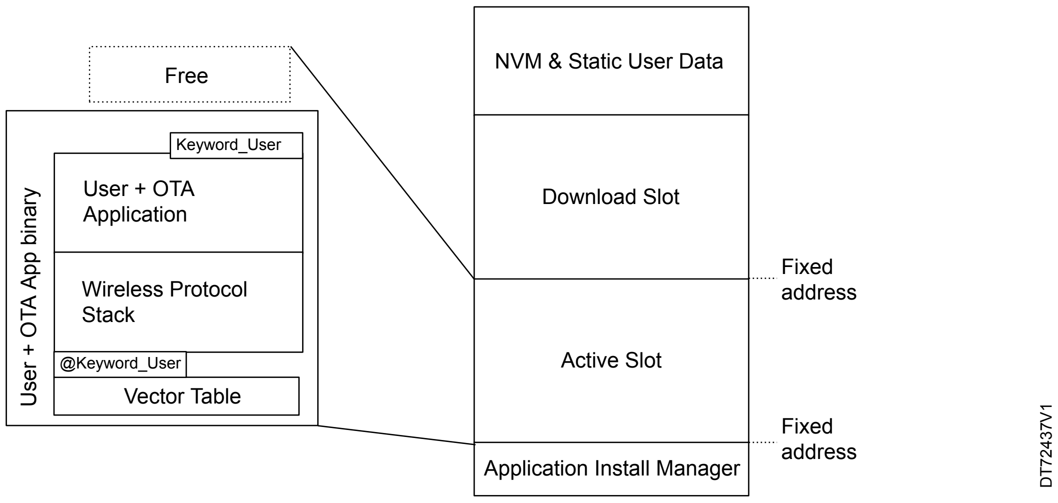
- OTA application: This component is responsible for downloading the raw data of the new application version. It can be based on any short-range wireless protocol handled by the STM32WBA.
- Application install manager application: This component is responsible for updating the firmware. It substitutes 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 update 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.
- Keyword user: The keyword User 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.

3.2.8.2 *Going further*

For more information on the interface, using the interface, and for examples, refer to the FUOTA wiki page

4 System initialization

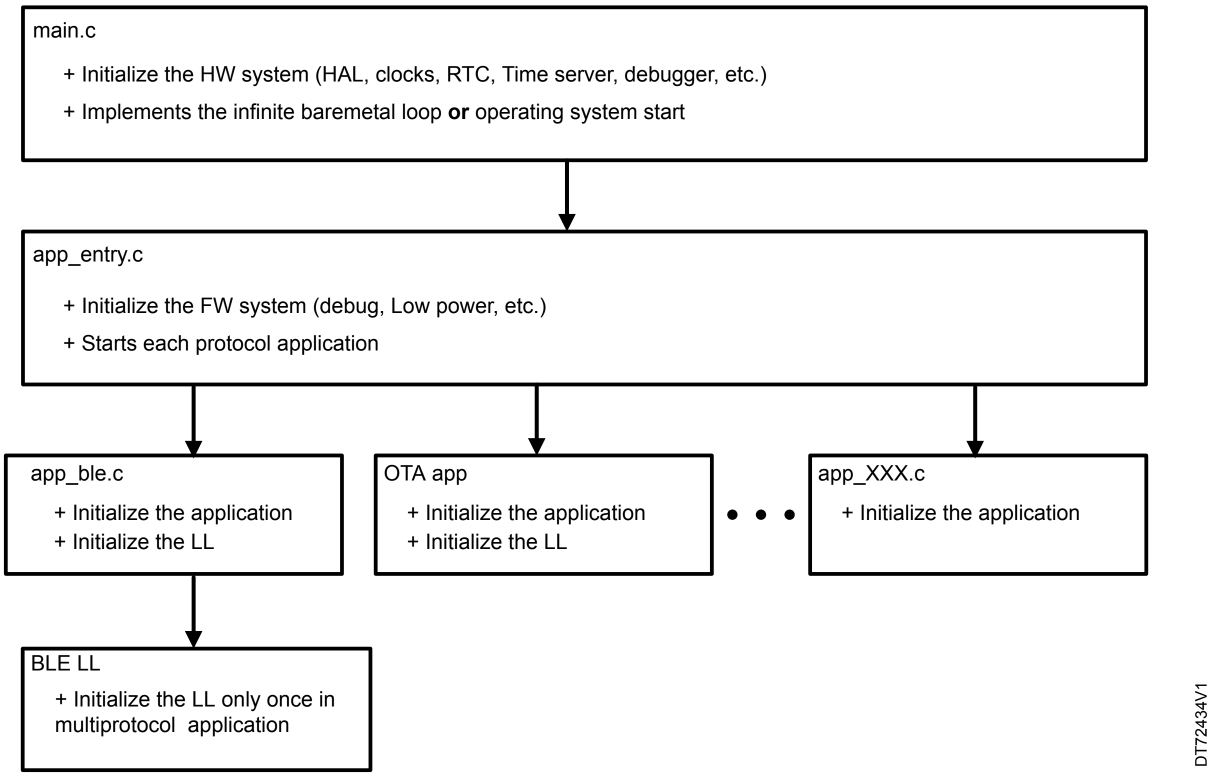
4.1 General concepts

All applications have a four layer architecture:

- `main.c` First entry point, all hardware configuration that is common to any application
- `app_entry.c` All SW configuration and implementation that is common to any application
- `app_ble.c/app_xxx.c` Application dedicated files
- BLE LL/YYY LL: Link layer interface

The system is initialized using the steps below.

Figure 13. General architecture


main.c
+ Initialize the HW system (HAL, clocks, RTC, Time server, debugger, etc.)
+ Implements the infinite baremetal loop **or** operating system start
app_entry.c
+ Initialize the FW system (debug, Low power, etc.)
+ Starts each protocol application
app_ble.c
+ Initialize the application
+ Initialize the LL
OTA app
+ Initialize the application
+ Initialize the LL
app_XXX.c
+ Initialize the application
BLE LL
+ Initialize the LL only once in multiprotocol application
DT72434V1

For detailed information and examples, refer to the wiki page on the system initialization,

5 Design of a short-range wireless application

5.1 Bluetooth® Low Energy

This chapter aims to highlight a generic Bluetooth® Low Energy application architecture on an STM32WBA device.

5.1.1 Overview

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

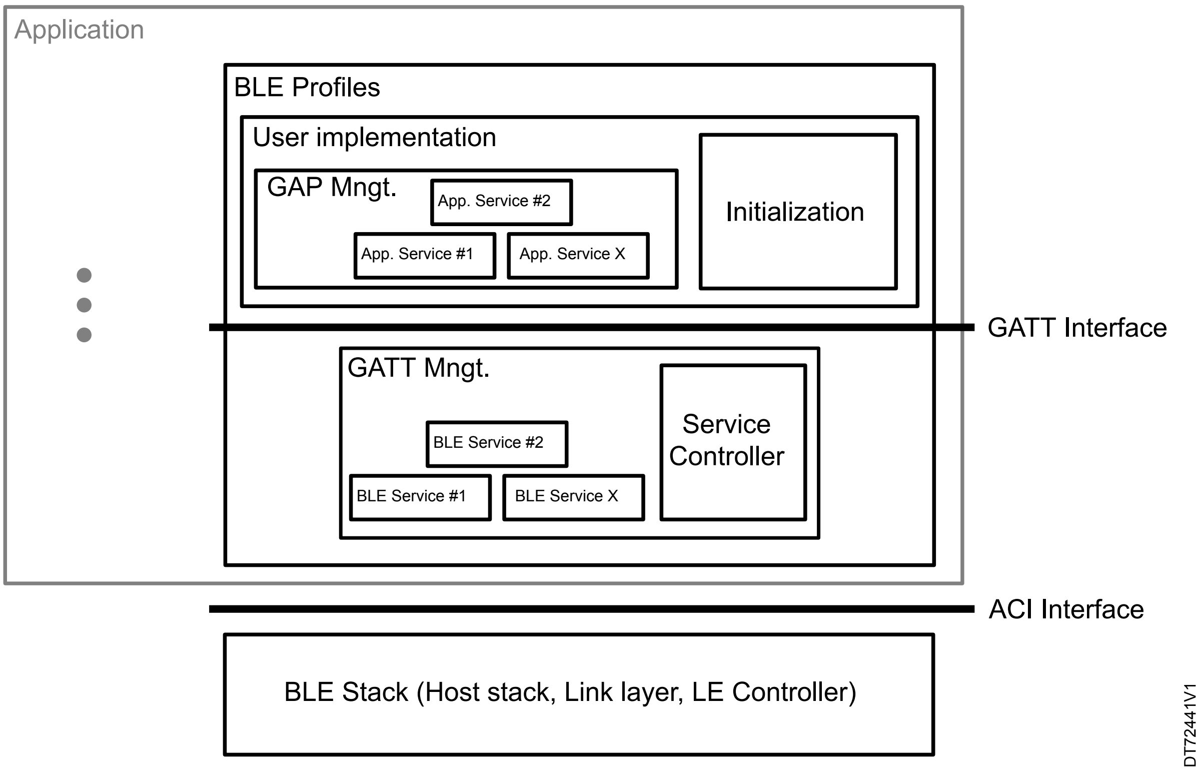
A BLE profile is a collection of one or more BLE services. It also defines the behavior of its services mainly in the context of GAP management.

A BLE profile is composed of three components:

- A GATT management module
- An application service for each selected BLE service
- A GAP management module

The figure below illustrates the design of a generic BLE application:

Figure 14. Generic BLE application design



5.1.1.1 Layer descriptions

The BLE firmware is built on a layering architecture. For BLE configuration, the user can select one of the two available interfaces for the application.

ACI interface

The ACI interface is a specific STMicroelectronics interface provided to access all the available features in the BLE host stack. This interface follows two rules:

- Each command is blocking and does not return until the response is received.
- It reports to the user all asynchronous events received from the BLE host stack.

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 BLE services.

When this interface is used, the ACI GATT interface should no longer be used. There are two exceptions:

- GATT initialization
- Application already implements its BLE service outside the GATT management module

GATT management

The GATT management has two components:

- Service controller:
  - Manages the BLE service initializations, and the events received by the BLE host stack
  - Initializes all enabled BLE services provided in the GATT management module
  - Provides an interface so that the application may initialize its own BLE service implemented outside the GATT management module
  - Manages all events reported by the BLE host stack
  - Forwards the events that are not GATT events to the application
  - Forwards GATT events to each registered BLE service until one of these services reports that the GATT event has been processed

Note: If no registered BLE services is concerned with the GATT event, it is forwarded to the application.

- BLE services:
  - Can either be specified by the BT SIG or custom to STMicroelectronics
  - Each BLE service is implemented in a dedicated file named with the three or four letters, as specified by the BT SIG
  - Each BLE service provides the same features:
    - Creates the service in the BLE stack
    - Implements all mechanisms (that do not require user input) for GATT event reception
    - Provides a simplified interface for value update/receive from the server
    - Adds its characteristics

User implementation

This is the application part of BLE profile creation. The user should implement three components to build a BLE profile:

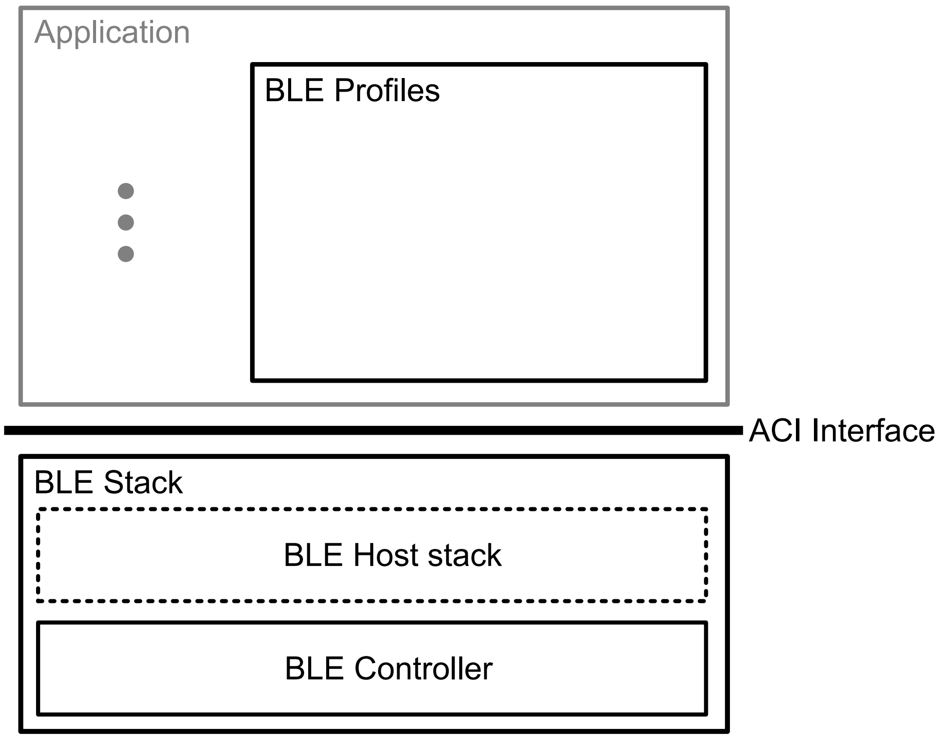
- Initialization:
  Initializes all the software modules of the application, for example, GAP initialization, Service controller initialization, etc.
- GAP management:
  Manages everything that is not related to a GATT event
  Has the BLE connection management role
  Receives notification from the application services to be compliant with a BLE profile

- Application services:
  Application service is the application management of a BLE Service. There is one application service file per BLE Service.
  For an application service, all GATT events (and only those events) are handled in this file.
  Each application service provides at least:
  – An initialization function
  – An implementation of the corresponding BLE service notification function
  – For some profiles, a notification to the GAP management

5.1.2 Configuration

The BLE stack is a two-layer block. One is for the host and the other is for the controller.

Figure 15. **BLE stack**



Features of both parts are provided as two distinct libraries. These two libraries are mandatory for any BLE project, one relying on the other.

5.1.2.1 BLE host stack

The BLE host stack library is configurable in different ways and is provided in four different variants:

- Two variants containing the BLE host stack and the STMicroelectronics LL controller:
  – Full stack (`stm32wba_ble_stack_full.a`)
  – Basic stack (`stm32wba_ble_stack_basic.a`)
- Two variants without the BLE host stack, with the STMicroelectronics LL controller only:
  – Link layer only stack (`stm32wba_ble_stack_llo.a`)
  – Link layer only basic stack (`stm32wba_ble_stack_llobasic.a`)

Depending on the application purpose(s), the user can select one or another configuration. Each variant has its own features and details. The differences between the configurations are listed below.

- Full stack (`stm32wba_ble_stack_full.a`):
  It contains all the legacy stack supported features plus the extended advertising, caching GATT, ACI HCI flow control, isochronous support for audio, L2CAP connection-oriented channels and host enabled with GATT, GAP, and L2CAP features.

Note: This BLE stack variant can be configured to run in controller only mode (the host stack is bypassed).

- Basic stack (`stm32wba_ble_stack_basic.a`):
  It contains only basic supported features or BLE legacy features without extended advertising, neither caching GATT, ACI HCI flow control, isochronous support, nor L2CAP connection-oriented channels. However, the host is here enabled and supports all the basic GATT, GAP, and L2CAP features.
- Link layer only stack (`stm32wba_ble_stack_llo.a`):
  It contains all the features supported by the full stack but does not include the host part (GATT, GAP, and L2CAP features).
- Link layer only basic stack (`stm32wba_ble_stack_llobasic.a`):
  It contains all the features supported by the basic stack but does not include the host part (GATT, GAP, and L2CAP features).

5.1.2.2 *Controller stack*

The controller stack library is only composed of the link layer features. This library is provided in a basic and a full version, for example, LinkLayer_BLE_Full_lib.a and LinkLayer_BLE_Basic_lib.a.

5.1.3 Going further

For more information on the BLE initialization procedure, refer to the wiki page.

5.2 Zigbee®

This chapter aims to highlight a generic Zigbee® application architecture on an STM32WBA device.

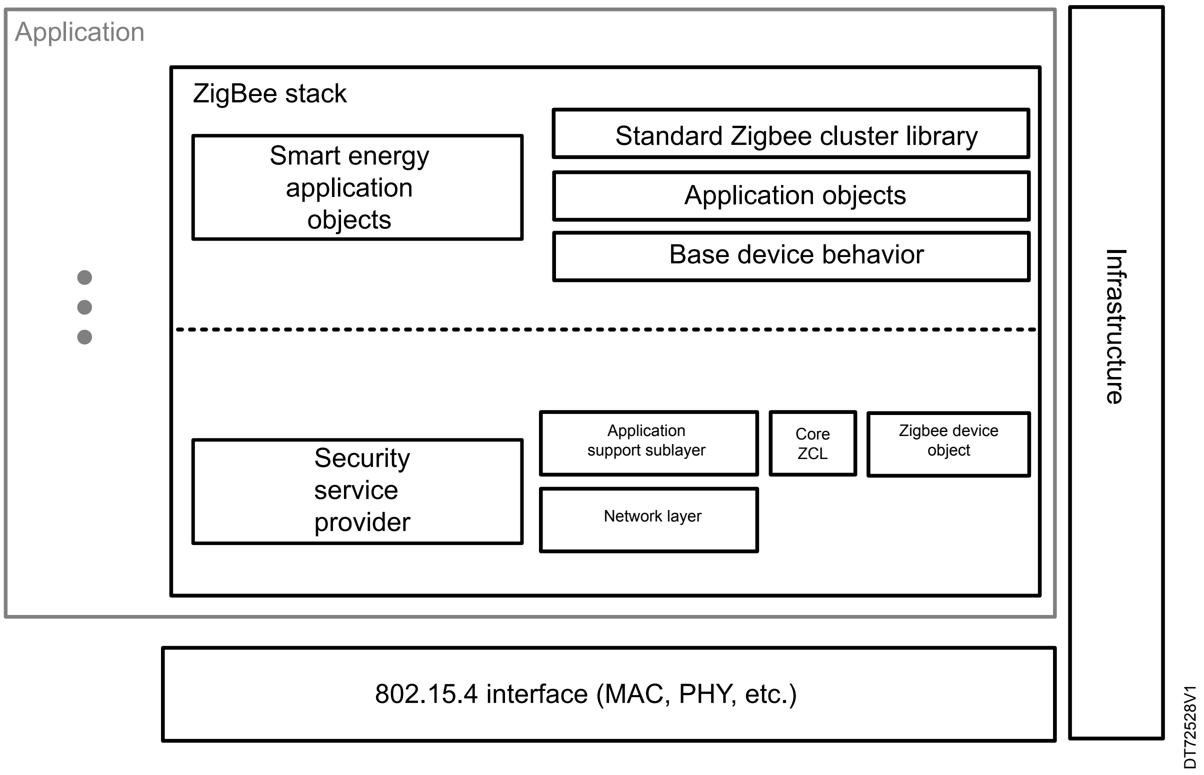
5.2.1 Overview

A Zigbee® application is composed of different components and modules, but it mostly relies on some key points.

- Clusters: Clusters are groups of related commands and attributes that define the functionality of a Zigbee® device. There are two types of clusters: standard clusters and manufacturer-specific clusters.
- Application objects: Application objects are software modules that define the behavior of a Zigbee® device. Each object is associated with one or more clusters and contains code that handles the commands and attributes defined by those clusters.
- Zigbee® profiles: Zigbee® profiles are collections of clusters and application objects that define the behavior of a specific type of Zigbee® device. For example, the Zigbee® Home Automation profile defines the behavior of devices used in home automation applications.

The figure below deals with the design of a generic Zigbee® application:

Figure 16. Generic Zigbee® application design


Application
ZigBee stack
Smart energy application objects
Standard Zigbee cluster library
Application objects
Base device behavior
Security service provider
Application support sublayer
Core ZCL
Zigbee device object
Network layer
802.15.4 interface (MAC, PHY, etc.)
Infrastructure
DT72528V1

5.2.1.1 *Layer descriptions*

Zigbee® stack

The application layer is the highest layer of the ZigBee stack and is responsible for defining the functionality of ZigBee devices. It includes application objects, clusters, and profiles that define the devices' behavior and the data they exchange.

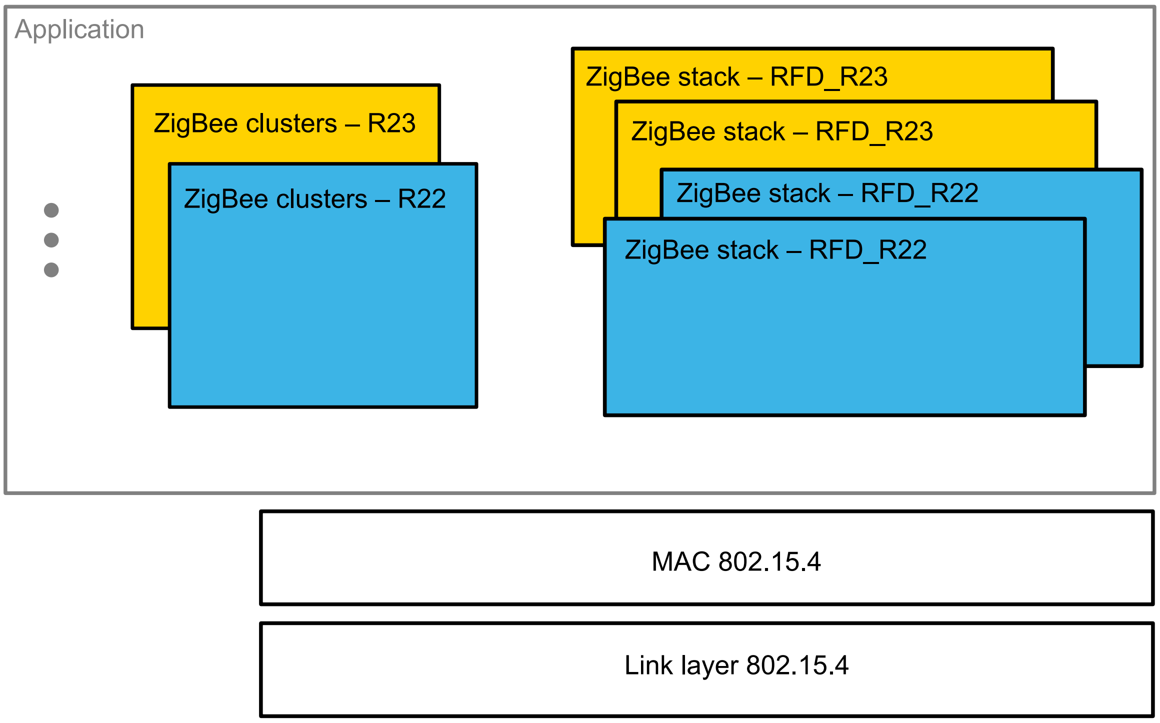
Digging up, here are some of the main components:

- Standard ZigBee Cluster library:
  Cluster can be considered as a group of commands and attributes specific to a dedicated kind of Zigbee application (OnOff, temperature measurement, etc.).
  It defines an interface to specific functionality (commands are actions and attributes are data or state).
  There are two types of clusters: standard clusters and manufacturer-specific clusters.
- Application objects:
  Application objects are software modules that define the behavior of a ZigBee device. Each object is associated with one or more clusters and contains code that handles the commands and attributes defined by those clusters.
-
  - Base device behavior (BDB):
    BDB is a set of rules and procedures that define how ZigBee devices should behave. It is a standard software component that handles fundamental operations such as commissioning, network security, and persistent data management. It is designed to simplify the process of adding new devices to a ZigBee network.
  - Application support sublayer (APS):
    It provides an interface between the network layer (NWK) and the application layer through a general set of services that are used by both ZDO and the manufacturer-defined application object.
    APS is responsible for:
    - Binding management
    - Message forwarding between bound devices
    - Group address definition and management
    - Address mapping from extended address to network address
    - Addresses table
    - Fragmentation and reassembly of packets
    - Reliable data transport
  - ZigBee device object (ZDO):
    The ZDO component handles the device management and communication functions. It includes:
    - Initializing the APS and NWK layer
    - Device discovery
    - Service discovery
    - Network management (including defining operating modes ZC, ZR, ZED)
    - Security management
    - Initiation and/or responding to remote binding requests

5.2.2 Configuration

An STM32WBA Zigbee® application relies on several libraries with configuration possibilities for some of them.

Figure 17. Zigbee® stack



These libraries are provided and are mandatory for any Zigbee® project, one relying on the other.

5.2.2.1 *Zigbee® PRO stack*

The Zigbee® stack is available in three different variants fulfilling different use cases and needs:

- ZigbeeProFFD.a:
  Library corresponding to the Full Feature Device. It is suitable if the application needs to be capable of accepting any role in the network (that is, either router, coordinator or end device).
- ZigbeeProRFD.a:
  Library corresponding to the Reduced Feature Device. This is suitable if the application shall only support an end device role.

Depending on the application purpose(s), the user can select one or another configuration. Each variant has its own features and details.

Note: Both libraries are available in R22 and R23 versions.

5.2.2.2 *Zigbee® clusters*

The Zigbee® Cluster Library (ZCL), ZigBeeCluster.a, is provided and regroups a list of already defined clusters. You can check the full list of available clusters on our wiki website.

Note: Both libraries are available in R22 and R23 versions.

5.2.2.3 *MAC and Link Layer 802.15.4*

These last two libraries handle hardware access and management, providing functionalities to the upper layers.

5.2.3 Going further

For more information on Zigbee®, refer to the wiki page.

5.3 Thread®

This chapter aims to highlight a generic Thread® application architecture on an STM32WBA device.

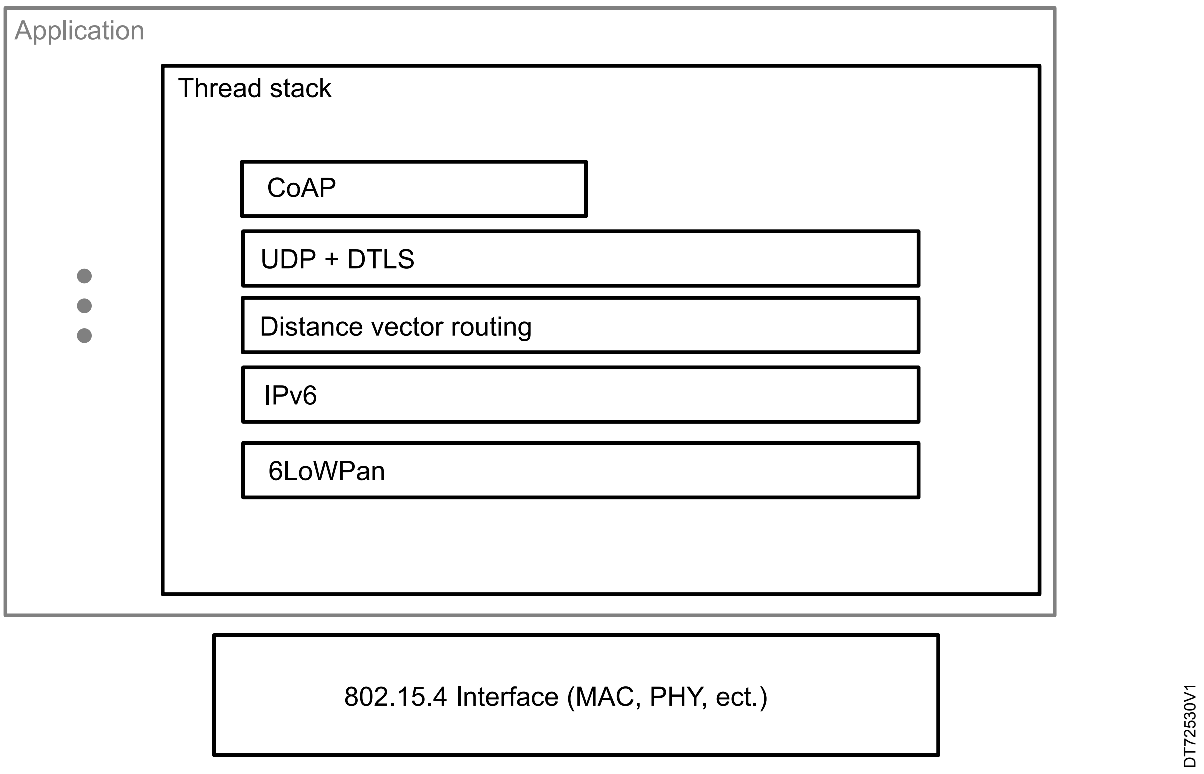
5.3.1 Overview

The Thread® application is also composed of different components and modules, but relies on some key points:

- Thread networks:
  Thread networks are composed of one or more Thread devices that communicate with each other using the Thread protocol. Each Thread device is assigned a unique IPv6 address and can participate in multiple Thread networks
- Thread mesh:
  A Thread mesh is a network topology where each Thread device communicates with other devices in the network, forming a mesh of interconnected devices. This allows for multiple communication paths and provides redundancy in case of device failure.
- Thread border router:
  A Thread border router is a device that connects a Thread network to other networks, such as the internet. It acts as a gateway between the Thread network and other networks, allowing Thread devices to communicate with devices outside the Thread network.
- Thread commissioning:
  Thread commissioning is the process of adding new devices to a Thread network. This involves securely authenticating the new device and assigning it a unique IPv6 address. Commissioning can be done using various methods, such as NFC, QR codes, or manual entry of credentials.

The figure below deals with the design of a generic Thread® application:

Figure 18. Generic Thread® application design



5.3.1.1 *Layer descriptions*

Thread stack

The application layer is the highest layer of the Thread stack and is responsible for defining the functionality of the Thread devices. It includes several protocols, such as the Constrained Application Protocol (CoAP), which is used for machine-to-machine communication, and the Thread Commissioning Protocol, which is used for adding new devices to a Thread network.

- CoAP:
  The CoAP layer is responsible for providing a lightweight, RESTful protocol designed for use in constrained environments, such as low-power wireless networks. It is used to exchange data between devices in a Thread network and is based on the same principles as the Hypertext Transfer Protocol (HTTP) used in the World Wide Web.
- UDP+DTLS:
  The UDP layer is responsible for providing connectionless communication between devices. It is used for sending and receiving CoAP messages. The DTLS Layer is responsible for providing secure communication between devices. It is used to encrypt co-ap messages to prevent unauthorized access and data tampering.
- Distance Vector Routing:
  The distance vector routing protocol is used to ensure that data is routed between devices in the most efficient way possible.
- IPv6:
  The IPv6 layer is responsible for providing end-to-end communication between devices in different Thread networks. It uses the 6LoWPAN protocol to compress IP packets and reduce the size of the data transmitted over the wireless network.
- 6LoWPan:
  The 6LoWPAN layer is responsible for compressing IPv6 packets to reduce their size and enable them to be transmitted over low-power wireless networks. It uses header compression and fragmentation techniques to reduce the size of the packets.

5.3.2 Configuration

An STM32WBA Thread® application relies on several libraries with configuration possibilities for some of them.

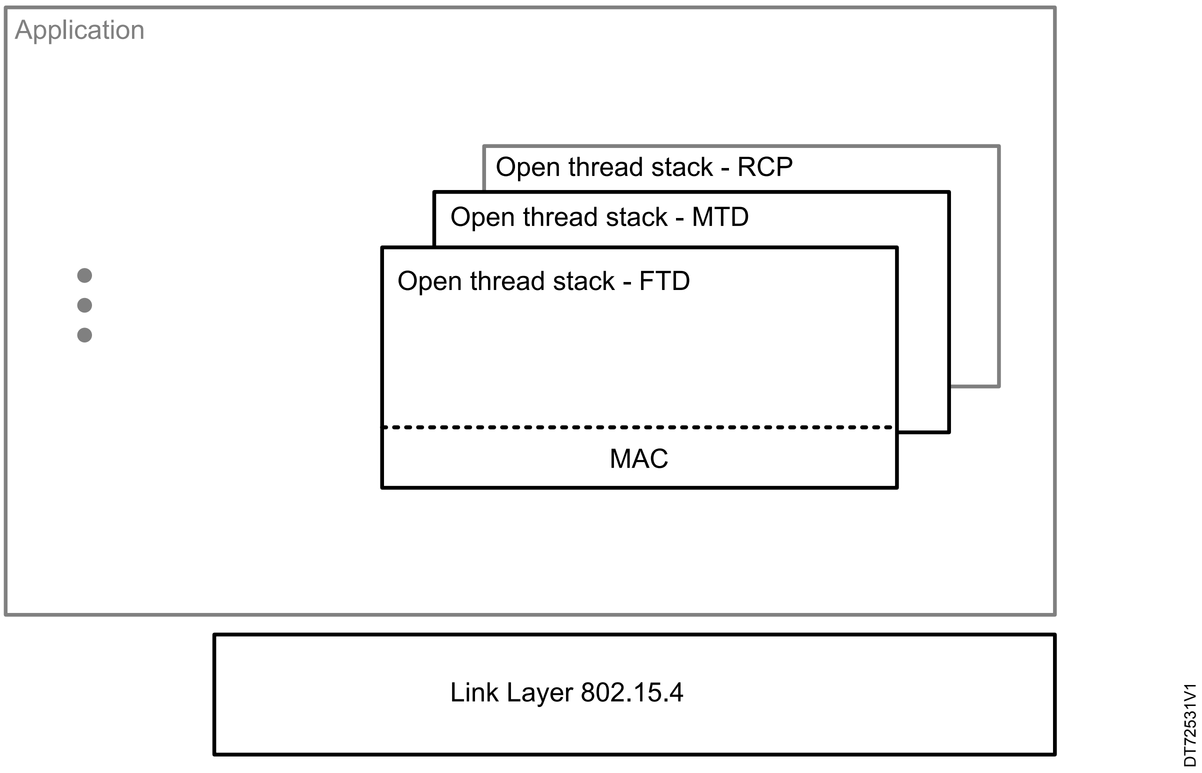


Figure 19. Thread® stack

These libraries are provided and are mandatory for any Thread® project, one relying on the other.

5.3.2.1 *The OpenThread stack is an open-source implementation of the Thread specification, and is implemented in C++ with C-bindings for the OpenThread API.*

The OpenThread stack is available in three different versions fulfilling different use cases and needs:

- Stm32wba_ot_ftd_lib.a:
  Library corresponding to the Full Thread Device. It is suitable if the application needs to be capable of accepting any role in the network (that is, either Router, Router Eligible / End device, or Full End device).
- Stm32wba_ot_mtd_lib.a:
  Library corresponding to the Minimal Thread Device. It is suitable for End device, Sleepy End device or Synchronized End device.
- Stm32wba_ot_rcp_lib.a:
  Library corresponding to the Radio Coprocessor setup. It is suitable for applications that are split between a host and a target, the latter running only radio layers.

Note: The OpenThread stack not only provides the previously defined components but also the MAC layer interface.

Depending on the application purpose(s), the user can select one or another configuration. Each variant has its own features and details.

5.3.2.2 *Link Layer 802.15.4*

The last library handles hardware access and management, providing functionalities to the upper layers.

5.3.3 Going further

For more information on Thread®, refer to the wiki page.

6 Requirements/guidelines

This chapter contains detailed requirements and best practices for the optimal design of a short-range wireless application.

6.1 System

6.1.1 Interrupts

Some restrictions should be followed concerning interruption management. There are some events that cannot be left over, in terms of scheduling. For instance, the LL_High_ISR, must not be delayed for more than 40 µs when in use.

Therefore, the user must organize the interrupt levels as follows:

Table 2. Interrupt levels

Interrupt name	Priority(1)
LL_High_ISR (when the radio is active)	0
RCC (HSERDY and PLL1RDY)	1
User system and peripheral interrupts	2 to 4
LL_High_ISR (when radio is inactive)	5
GPDMA_CHx	Channel 7 (DMA RX: 5) Channel 0 (DMA TX: 6)
User system and peripheral interrupts	7 to 13
SysTick	14
LL_Low_ISR	15

1. *The lower the number, the higher the priority*

Note: *These values might change according to the project protocol(s) and the scheduling method (that is, BareMetal or RTOS). The above values are dedicated to BLE use with sequencer scheduling.*

6.1.2 Performances

There are a few rules to follow for nominal clock configuration:

- Default firmware run speed: 16 MHz - HSE/2 Range2
- Radio event period, run speed: at least 32MHz–HSE Range1
- PLL required run speed: 100 MHz

6.1.3 Processes

The design of each process must give the opportunity for each of them to be executed within a correct time frame window.

Although this is not a strict requirement and some specific processes may not comply with these timings, the recommendations are:

- The sum of each process timing execution should not exceed 300 ms.
- The time execution of each process should not exceed 30 ms.

These timing requirements are especially relevant in a bare metal implementation, as there is no way to stop a running process to execute another one.

When using an operating system, these timing requirements still apply to improve the dynamic between processes that have the same priority.

Revision history

Table 3. Document revision history

Date	Version	Changes
08-Jun-2023	1	Initial release.
09-Apr-2024	2	Updated: • Section 2: General information • Section 3.1.1: Application view • Section 3.1.1.2: Protocol stacks • Section 3.2: SW concepts/features • Section 3.2.6: Trace management: log module • Section 3.2.6.1: Concepts • Section 6.1.1: Interrupts Removed Footprints. Added: • Section 5.2: Zigbee® • Section 5.3: Thread®

Contents

List of figures

List of tables