
How to build wireless applications with STM32WB MCUs

Introduction

This document guides designers through the steps required to build specific Bluetooth® Low Energy or 802.15.4 applications based on STM32WB 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 technology, 802.15.4 OpenThread protocol, Zigbee® protocol, and 802.15.4 MAC layer. It must then master system services, such as low power management and task sequencing.

Contents

1	References	11
2	List of acronyms and abbreviations	12
3	Software overview	13
3.1	Supported stacks	13
3.2	BLE application	15
3.3	Building a BLE application on top of the HCI layer interface	16
3.4	Thread application	17
3.5	MAC 802_15_4 application	17
3.6	BLE and Thread application in concurrency	17
4	STM32WB software architecture	18
4.1	Main principles	18
4.2	Memory mapping	19
4.3	Shared peripherals	20
4.4	Sequencer	27
4.4.1	Implementation	28
4.4.2	Interface	28
4.4.3	Detailed interface and behavior	28
4.5	Timer server	31
4.5.1	Implementation	31
4.5.2	Interface	32
4.5.3	Detailed interface and behavior	32
4.6	Low power manager	34
4.6.1	Implementation	35
4.6.2	Interface	35
4.7	Flash memory management	35
4.7.1	CPU2 timing protection	36
4.7.2	CPU1 timing protection	38
4.7.3	Conflict between RF activity and flash memory management	38
4.8	Debug information from CPU	39
4.8.1	GPIO	39

4.8.2	SRAM2	40
4.9	FreeRTOS low power	41
4.10	Device information table	42
4.11	ECCD error management	43
5	System initialization	46
5.1	General concepts	46
5.2	CPU2 startup	46
6	PLL management	48
6.1	How to switch the system clock between HSE and PLL	48
6.1.1	Case 1: Before CPU2 is started	48
6.1.2	Case 2: CPU2 is started	49
7	Step by step design of a BLE application	51
7.1	Initialization phase	51
7.2	Advertising phase (GAP peripheral)	51
7.3	Discoverable and connectible phase (GAP central)	52
7.4	Services and characteristic configuration (GATT server)	53
7.5	Service and characteristic discovery (GATT client)	54
7.6	Security (pairing and bonding)	55
7.6.1	Security modes and level	56
7.6.2	Security commands	56
7.6.3	Security information commands	57
7.7	Privacy feature	58
7.8	How to use the 2 Mbps feature	59
7.9	How to update connection parameters	59
7.10	Event and error code description	59
8	BT-SIG and proprietary GATT-based BLE application	61
8.1	Transparent mode - Direct test mode (DTM)	61
8.1.1	Purpose and scope	61
8.1.2	Transparent mode application principle	62
8.1.3	Configuration	62
8.1.4	RF certification - Application implementation	64

8.2	Heart rate sensor application	64
8.2.1	How to use STM32WB heart rate sensor application	65
8.2.2	STM32WB heart rate sensor application - Middleware application	66
8.3	STMicroelectronics proprietary advertising	70
8.4	Proprietary P2P application	73
8.4.1	P2P server specification	73
8.4.2	How to use the P2P server application	75
8.4.3	P2P server application - Middleware application	75
8.4.4	P2P client application - Middleware application	78
8.5	FUOTA application	83
8.5.1	CPU1 user flash memory mapping	83
8.5.2	BLE FUOTA application startup	84
8.5.3	BLE FUOTA services and characteristics specification	85
8.5.4	Flow description example to upload new CPU1 application binary	86
8.5.5	Application example with smart phone	88
8.5.6	How to use the reboot request characteristics	90
8.5.7	Power failure recovery mechanism for CPU1 application	92
8.6	Application tips	92
8.6.1	How to set Bluetooth device address	92
8.6.2	How to set IR (Identity Root) and ER (Encryption Root)	94
8.6.3	How to add a task to the sequencer	95
8.6.4	How to use the timer server	95
8.6.5	How to start the BLE stack - SHCI_C2_BLE_Init()	96
8.6.6	BLE GATT DB and security record in NVM	101
8.6.7	How to calculate the maximum number of bonded devices that can be stored in NVM	101
8.6.8	NVM write access	102
8.6.9	How to maximize data throughput	102
8.6.10	How to add a custom BLE service	102
8.6.11	How to use BLE commands in blocking mode	103
9	Building a BLE application on top of the HCI layer interface	104
10	Thread	105
10.1	Overview	105
10.2	How to start	105
10.3	Thread configuration	106

10.4	Architecture overview	106
10.5	Inter core communication	107
10.6	OpenThread API	108
10.7	Usage of the OpenThread APIs	109
10.7.1	OpenThread instance	109
10.7.2	OpenThread call back management	109
10.8	System commands for Thread applications	110
10.8.1	Non-volatile Thread data	111
10.8.2	Low-power support	112
11	Step by step design of an OpenThread application	113
11.1	Initialization phase	113
11.2	Set-up the Thread network	113
11.3	CoAP request	113
11.3.1	Creating an otCoapResource	114
11.3.2	Sending a CoAP request	114
11.3.3	Receiving a CoAP request	114
11.4	Commissioning	115
11.5	CLI	115
11.6	Traces	116
12	STM32WB OpenThread application	117
12.1	Thread_Cli_Cmd	117
12.2	Thread_Coap_DataTransfer	117
12.3	Thread_Coap_Generic	117
12.4	Thread_Coap_Multiboard	117
12.5	Thread_Commissioning	118
12.6	Thread_FTD_Coap_Multicast	118
12.7	Thread_SED_Coap_Multicast	118
12.8	Thread FUOTA	119
12.8.1	Principle	119
12.8.2	Memory mapping	119
12.8.3	Thread FUOTA protocol	122
12.8.4	FUOTA application startup procedure	123
12.8.5	Applications	124

13	MAC IEEE Std 802.15.4-2011	126
13.1	Overview	126
13.2	Architecture	126
13.3	API	126
13.4	How to start	127
13.4.1	Board configuration	127
13.4.2	MAC radio protocol processor CPU2 firmware	128
13.4.3	MAC application processor firmware	128
13.4.4	Output	129
13.4.5	MAC IEEE Std 802.15.4-2011 system	130
13.4.6	Integration recommendations	130
14	Annexes	133
14.1	Detailed flow of the device initialization	133
14.2	Mailbox interface	135
14.2.1	Interface API	136
14.2.2	Detailed interface behavior	137
14.3	Mailbox interface - Extended	141
14.3.1	Interface API	141
14.3.2	Detailed interface and behavior	142
14.4	ACI interface	147
14.4.1	Detailed interface and behavior	148
14.5	Vendor specific HCI commands for controller	153
14.6	STM32WB system commands and events	155
14.6.1	Commands	155
14.6.2	Events	158
14.7	BLE - Set 2 Mbps link	158
14.8	BLE - Connection update procedure	159
14.9	BLE - Link layer data packet	160
14.10	Thread overview	161
14.10.1	Introduction	161
14.10.2	Main characteristics	162
14.10.3	Layers	162
14.10.4	Mesh topology	164
14.10.5	Thread configuration	165

15	Conclusion	167
16	Revision history	168

List of tables

Table 1.	Stacks supported by STM32WB series microcontrollers	13
Table 2.	Semaphores	21
Table 3.	Interface functions	28
Table 4.	Interface functions	32
Table 5.	Interface functions	35
Table 6.	Advertising phase API description	52
Table 7.	GAP central APIs	52
Table 8.	GATT client APIs	54
Table 9.	Security commands	57
Table 10.	Security information commands	57
Table 11.	2 Mbps feature commands	59
Table 12.	Proprietary connection data	59
Table 13.	Direct test mode functions	62
Table 14.	Heart rate service functionalities	67
Table 15.	HR sensor application control	70
Table 16.	AD structure according to the Bluetooth 5 Core specification Vol. 3 part C	71
Table 17.	STM32WB manufacturer specific data	71
Table 18.	Group B features - Bit mask	71
Table 19.	Device ID Enum	71
Table 20.	P2P service and characteristic UUIDs	74
Table 21.	P2P specification	74
Table 22.	P2P service functionalities	76
Table 23.	FUOTA service and characteristics UUID	85
Table 24.	Base address characteristics specification	86
Table 25.	File upload confirmation reboot request characteristics specification	86
Table 26.	Raw data characteristics specification	86
Table 27.	Reboot request characteristics specification	86
Table 28.	MO firmwares available for Thread	105
Table 29.	Files for Thread configuration	106
Table 30.	Interface APIs	136
Table 31.	Interface APIs	141
Table 32.	BLE transport layer interfaces	147
Table 33.	Vendor specific HCI commands	153
Table 34.	System interface commands	155
Table 35.	User system events	158
Table 36.	Document revision history	168

List of figures

Figure 1.	Protocols supported by STM32WB series microcontrollers	14
Figure 2.	STM32WB series microcontrollers BLE HCI layer model	15
Figure 3.	BLE application and wireless firmware architecture	16
Figure 4.	Memory mapping	19
Figure 5.	Timing for entering/exiting Stop mode on CPU1	22
Figure 6.	Algorithm to enter Stop mode on CPU1	23
Figure 7.	Algorithm to exit Stop mode on CPU1	24
Figure 8.	Algorithm to use RNG on CPU1	25
Figure 9.	Algorithm to use USB on CPU1	26
Figure 10.	Algorithm to write/erase data in the flash memory	37
Figure 11.	CPU1 and flash memory operation versus PESD bit	38
Figure 12.	Format of version and memory information	43
Figure 13.	ECC management in NMI interrupt handler	45
Figure 14.	System initialization	46
Figure 15.	Algorithm to switch the system clock from PLL to HSE	48
Figure 16.	Algorithm to switch the system clock from HSE to PLL	49
Figure 17.	GATT-based BLE application	61
Figure 18.	Transparent mode with P-NUCLEO-WB55 board and ST-LINK VCP	63
Figure 19.	Transparent mode with P-NUCLEO-WB55 board and level shifter	64
Figure 20.	Simple setup with BLE RF tester and P-NUCLEO board	64
Figure 21.	Heart rate profile structure	65
Figure 22.	Simple setup with BLE RF tester and P-NUCLEO board	65
Figure 23.	Smart phone - ST BLE sensor with heart rate application	66
Figure 24.	Heart rate project - Interaction between middleware and user application	70
Figure 25.	P2P server to client demonstration	73
Figure 26.	P2P server to ST BLE sensor smart phone application	73
Figure 27.	P2P server/client communication sequence	74
Figure 28.	P2P server connected to ST BLE sensor smart phone application	75
Figure 29.	P2P server software communication	78
Figure 30.	P2P client software communication	83
Figure 31.	FUOTA memory mapping	84
Figure 32.	FUOTA startup procedure	85
Figure 33.	FUOTA process with heart rate	87
Figure 34.	P2P server - Application firmware selection	88
Figure 35.	P2P server - Application firmware update	89
Figure 36.	Heart rate sensor notification	90
Figure 37.	User option bytes setting	106
Figure 38.	Software architecture	107
Figure 39.	OpenThread functions calls	108
Figure 40.	OpenThread callback	108
Figure 41.	OpenThread stack API directory structure	109
Figure 42.	OpenThread callback management	110
Figure 43.	Storage of non-volatile data	111
Figure 44.	Configurable CLI UART (LPUART or USART)	115
Figure 45.	Traces for Thread applications	116
Figure 46.	Thread FUOTA network topology	119
Figure 47.	OTA server (Thread_Ota_Server) flash memory mapping	120
Figure 48.	FUOTA client flash memory mapping initial state	120

Figure 49.	FUOTA server flash memory mapping after CPU1 binary transfer.	121
Figure 50.	FUOTA server flash memory mapping after CPU2 binary transfer.	121
Figure 51.	Thread FUOTA protocol	122
Figure 52.	FUOTA startup procedure	123
Figure 53.	Update procedure	124
Figure 54.	MAC 802.15.4 software architecture	126
Figure 55.	MAC API dedicated to application core	127
Figure 56.	Option bytes configuration for MAC 802.15.4.	127
Figure 57.	MAC 802.15.4 simple application	128
Figure 58.	MAC 802.15.4 applications - Directory structure	129
Figure 59.	Coordinator start	130
Figure 60.	Node start, requesting association, and data send.	130
Figure 61.	Coordinator receiving association request and data.	130
Figure 62.	MAC 802.15.4 layer abstraction	131
Figure 63.	Traces on MAC 802.15.4 application	132
Figure 64.	System initialization	133
Figure 65.	System ready event notification	134
Figure 66.	BLE initialization	135
Figure 67.	Transport layer initialization	137
Figure 68.	BLE channel initialization	138
Figure 69.	BLE command sent by the mailbox	139
Figure 70.	ACL data sent by the mailbox.	139
Figure 71.	System command sent by the mailbox.	140
Figure 72.	BLE and system user event received by the mailbox.	140
Figure 73.	System transport layer initialization	142
Figure 74.	System command sent by the system transport layer	143
Figure 75.	System user event reception flow.	145
Figure 76.	shci_resume_flow() usage example	146
Figure 77.	BLE transport layer initialization	148
Figure 78.	ACI command flow	149
Figure 79.	BLE user event receive flow	151
Figure 80.	hci_resume_flow() usage example	152
Figure 81.	2 Mbps set-up flow	159
Figure 82.	Master initiates the connection update with HCI command	160
Figure 83.	Slave initiates the connection update with L2CAP command.	160
Figure 84.	Data packet breakdown	161
Figure 85.	Application GATT data format	161
Figure 86.	Thread protocol letters	162
Figure 87.	6LoWPAN packet fragmentation	163
Figure 88.	Thread network topology	165
Figure 89.	Link with the external world	165
Figure 90.	Thread device roles	166

1 References

- [1] UM2550⁽¹⁾ Getting started with STM32CubeWB for STM32WB Series
- [2] RM0434⁽¹⁾ Multiprotocol wireless 32-bit MCU Arm[®]-based Cortex[®]-M4 with FPU, Bluetooth[®] Low-Energy and 802.15.4 radio solution
- [3] AN5270⁽¹⁾ STM32WB Bluetooth[®] Low Energy wireless interface
- [4] UM2442⁽¹⁾ Description of STM32WB HAL and low-layer drivers
- [5] UM2288⁽¹⁾ STM32CubeMonitor-RF software tool for wireless performance measurements
- [6] AN5185⁽¹⁾ ST firmware upgrade services for STM32WB series
- [7] Bluetooth[®] specification Bluetooth Core Specification (v4.0, v4.1, v4.2, v5.0)
- [8] MAC IEEE Std 802.15.4-2011 Specification of the 802_15_4 MAC standard
- [9] Thread specification Thread specification V1.1 (Thread Group)
- [10] AN5506⁽¹⁾ Getting started with Zigbee[®] on STM32WB Series

1. Available on www.st.com.

2 List of acronyms and abbreviations

ACI	Application command interface
ATT	Attribute protocol
BLE	Bluetooth® Low Energy
CLI	Command line interface
CoAP	Constrained application protocol
CPU1	Cortex®-M4 core
CPU2	Cortex®-M0+ core
D2D	Device to device
DUT	Device under test
FUOTA	Firmware update over the air
FUS	Firmware upgrade service
GAP	Generic access profile
GATT	Generic attribute profile
HCI	Host controller interface
L2CAP	Logical link control adaptation layer protocol
LTK	Long-term key
OTA	Over the air
PDU	Protocol data unit
P2P	Peer to peer
RFU	Reserved for future use
SIG	Special interest group
SM	Security manager
UUID	Universally unique identifier

3 Software overview

3.1 Supported stacks

The STM32WB series microcontrollers are based on Arm^{®(a)} cores.

Select CPU2 firmware to load, based on the target application.

The STM32WB series microcontroller ecosystem supports different stacks (see [Table 1](#)), controlled by the application through specific interfaces, as shown in [Figure 1](#).

As shown in [Figure 2](#), CPU2 can provide a BT HCI standard interface, and a different BLE stack can run on CPU1.

Table 1. Stacks supported by STM32WB series microcontrollers

Stacks supported	Associated firmware
BLE	stm32wb5x_BLE_HCI_AdvScan_fw.bin stm32wb5x_BLE_HCILayer_fw.bin stm32wb5x_BLE_LLD_fw.bin stm32wb5x_BLE_Stack_full_fw.bin stm32wb5x_BLE_Stack_light_fw.bin
Thread	stm32wb5x_Thread_FTD_fw.bin stm32wb5x_Thread_MTD_fw.bin
BLE and Thread	stm32wb5x_BLE_Thread_dynamic_fw.bin stm32wb5x_BLE_Thread_static_fw.bin
BLE and MAC 802_15_4	stm32wb5x_BLE_Mac_802_15_4_fw.bin
BLE and Zigbee	stm32wb5x_BLE_Zigbee_FFD_dynamic_fw.bin stm32wb5x_BLE_Zigbee_FFD_static_fw.bin stm32wb5x_BLE_Zigbee_RFD_dynamic_fw.bin stm32wb5x_BLE_Zigbee_RFD_static_fw.bin stm32wb5x_Zigbee_FFD_fw.bin stm32wb5x_Zigbee_RFD_fw.bin
MAC 802_15_4	stm32wb5x_Mac_802_15_4_fw.bin stm32wb5x_Phy_802_15_4_fw.bin

arm

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

Figure 1. Protocols supported by STM32WB series microcontrollers

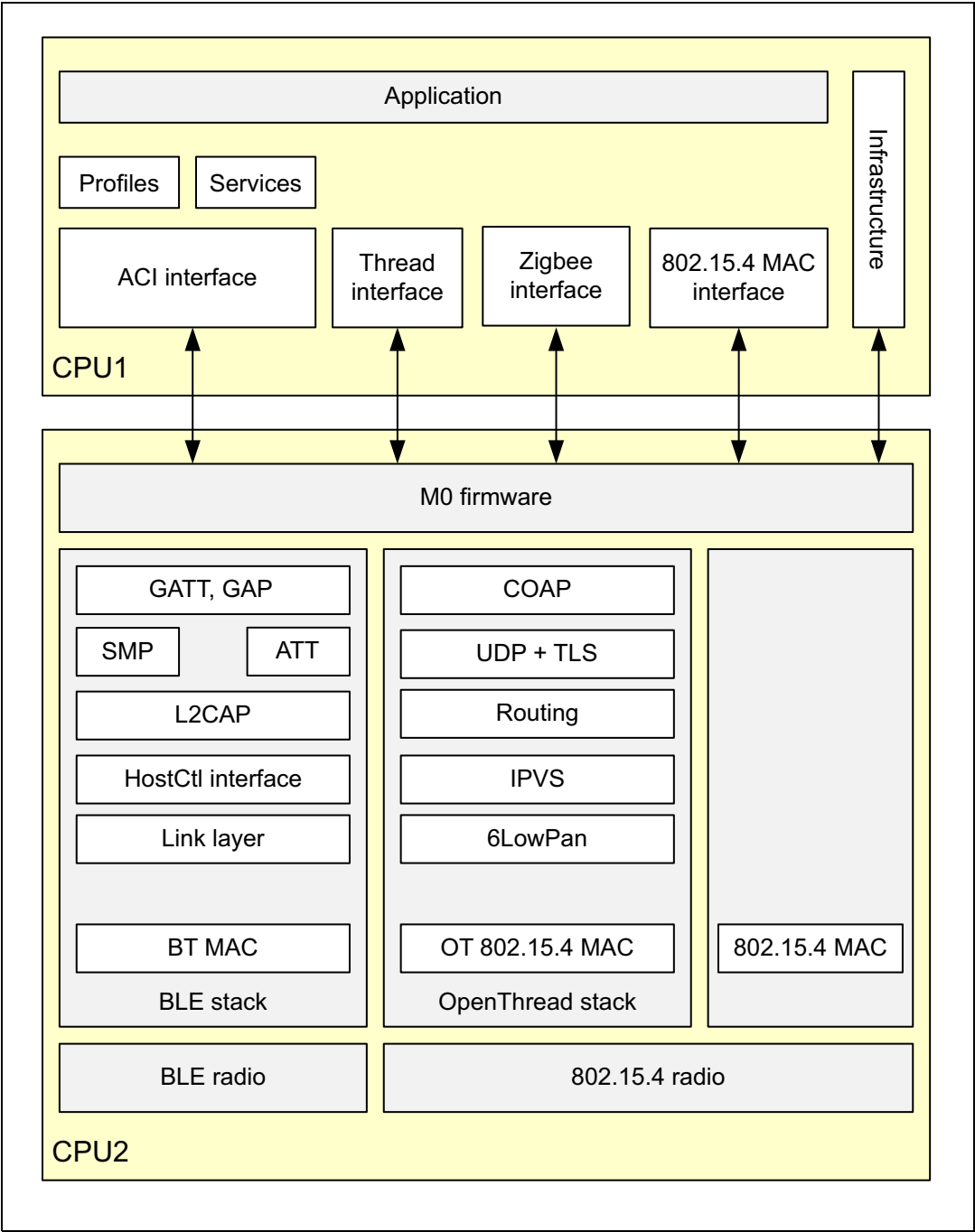
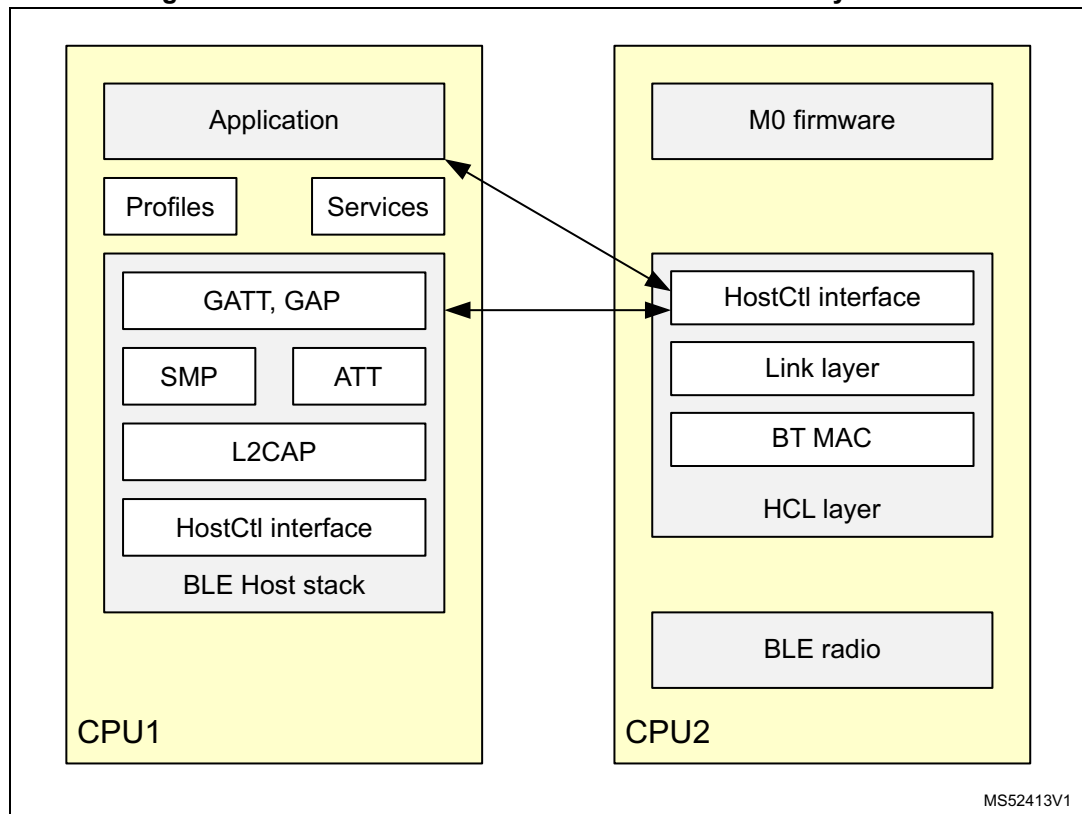


Figure 2. STM32WB series microcontrollers BLE HCI layer model



3.2 BLE application

The STM32WB architecture separates the BLE profiles and applications, running on the application CPU1, from the real-time aspects residing in the BLE peripheral.

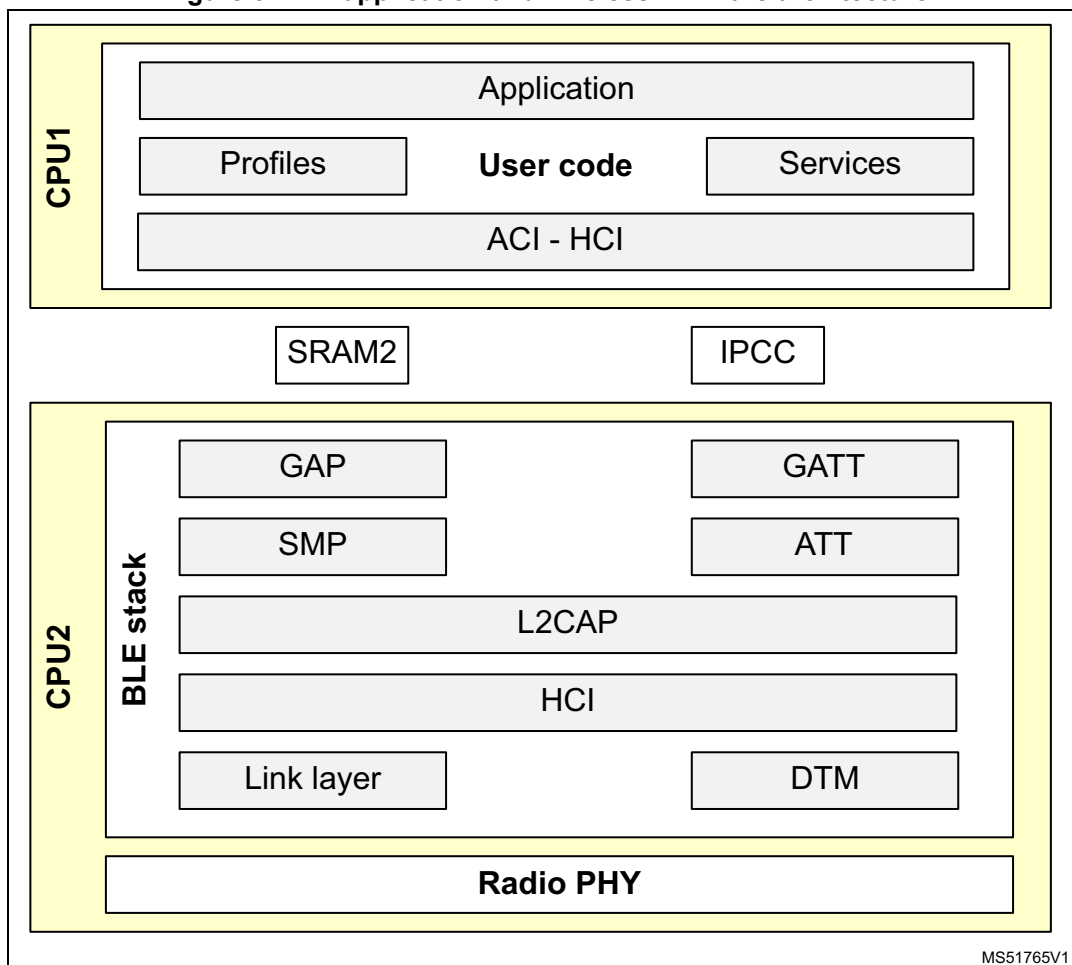
The BLE peripheral incorporates a CPU2 processor containing the stack handling the link layer up to the GAP and the GATT layers. It also incorporates the physical 2.4 GHz radio.

The application CPU1 collects and computes the data to transfer to the BLE.

CPU2 contains the LE controller and the LE host needed to manage all real time link layer and radio PHY interaction, which includes:

- Low power manager to control Low-power mode
- Debug trace to output information about activities
- Mailbox / IPCC to interface the BLE stack (LL, GAP, GATT)

Figure 3. BLE application and wireless firmware architecture



3.3 Building a BLE application on top of the HCI layer interface

The CPU2 can be used as a BLE HCI layer co-processor. In this case, the user either implements its own HCI application, or uses an existing open source BLE host stack.

Most BLE host stacks use a UART interface to communicate with a BLE HCI co-processor. The equivalent physical layer on STM32WB series microcontroller is the mailbox, as described in [Section 14.2: Mailbox interface](#).

The mailbox provides an interface for both the BLE channel and the System channel. The BLE host stack is responsible for building the command buffer to be sent over the BLE channel on the mailbox, and must provide an interface to report events received over the mailbox. In addition to the BLE host stack adaptation over the mailbox, the user must notify the mailbox driver when an asynchronous packet can be released.

The System channel is not handled by a BLE host stack. The user must implement a custom transport layer to build the system command buffer to be sent to the mailbox driver and to manage the event received from the mailbox (including the notification to release an asynchronous buffer to the mailbox driver). It can also use the mailbox extended driver (as described in [Section 14.3: Mailbox interface - Extended](#)), which provides an interface on top

of the transport layer responsible for building the system command buffer and to manage the system asynchronous event.

The BLE_TransparentMode project can be used as an example to build an application on top of a BLE HCI layer co-processor using the mailbox as described in [Section 12.2: Thread_Coap_DataTransfer](#).

3.4 Thread application

The OpenThread stack runs on CPU2 core and exports a set of APIs on CPU1 side in order to build a complete Thread application. Three CPU2 firmwares support the Thread protocol:

- **sm32wb5x_Thread_FTD_fw**: In this case, the device supports all Thread roles except the border router (such as Leader, Router, End device, Sleepy end device).
- **stm32wb5x_Thread_MTD_fw**: In this case, the device can act only as an end or sleepy end device). This configuration saves memory space compared to the FTD one.
- **stm32wb5x_BLE_Thread_fw**: In this case, the device supports both Thread (FTD) and BLE in static concurrent mode (refer to [Section 3.6](#) for more details).

3.5 MAC 802_15_4 application

When downloading the STM32wb5x_Mac_802_15_4_fw CPU2 firmware, CPU1 can access directly the 802_15_4 MAC layer and build its own application on top of it.

3.6 BLE and Thread application in concurrency

The STM32WB series microcontrollers supports a “static concurrent mode” (also named “switched mode”).

Both stacks (BLE and Thread) are embedded in the stm32wb5x_BLE_Thread_fw CPU2 firmware, available from www.st.com. Switch from one protocol to the other is done through a system application command. In this mode, the system disables the operational protocol before activating the other one. The STM32WB device switches from BLE to Thread after completely stopping the BLE stack, and vice versa. These transitions can take several seconds, as the network needs to be reattached each time.

4 STM32WB software architecture

4.1 Main principles

- All the code running on CPU2 is delivered as an encrypted binary
- Black box for customer perspective
- All the code running on CPU1 is delivered as source code
- Communication between CPUs is done via “mailbox”

The standard STM32Cube delivery package includes STM32WB resources such as:

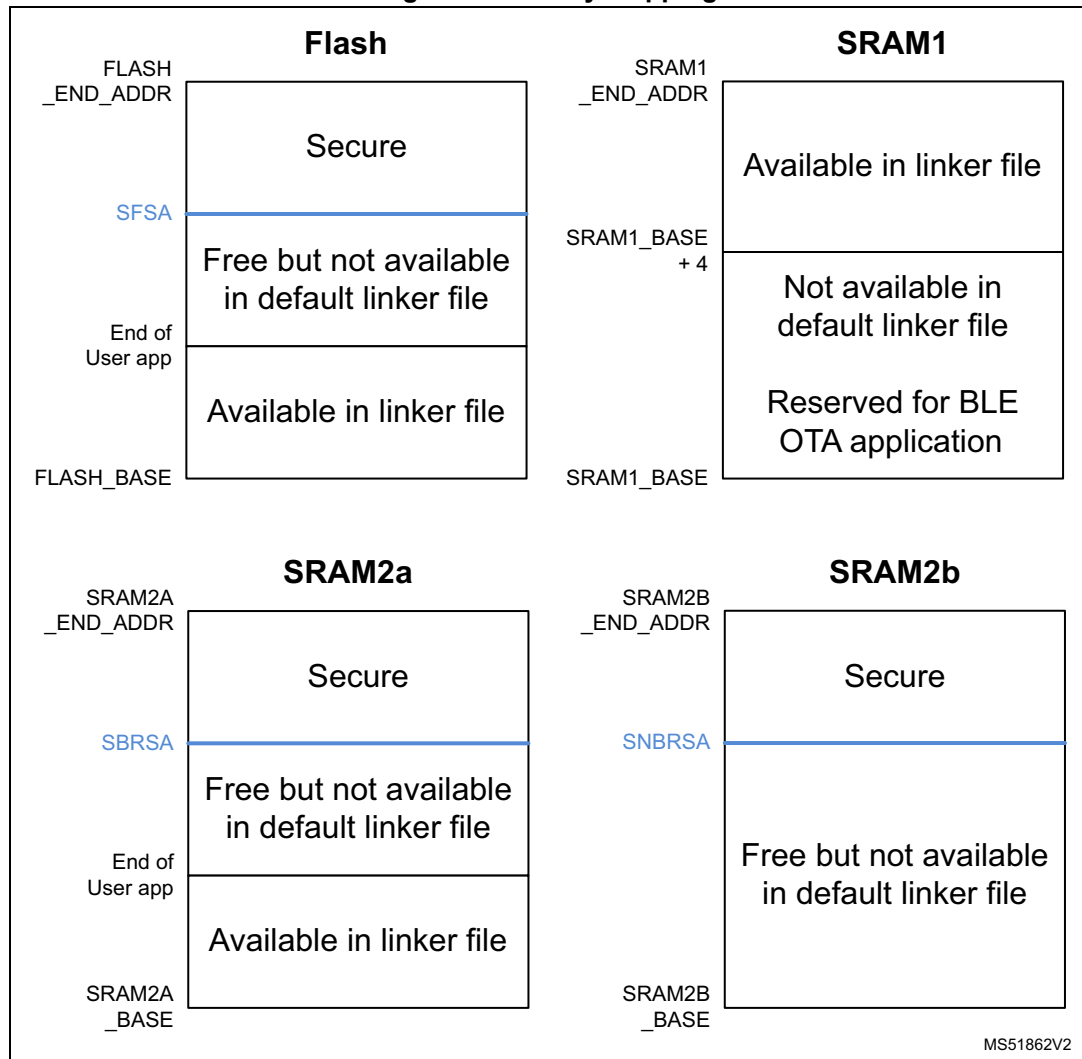
- HAL/LL to access the hardware registers
- BSP
- Middleware (such as FreeRTOS, USB devices).

In addition, the following application provide efficient system integration:

- A sequencer to execute the tasks in background and enter Low-power mode when there is no activity
- A timer server to provide virtual timers running on RTC (in Stop and Standby modes) to the application.

4.2 Memory mapping

Figure 4. Memory mapping



MS51862V2

The flash, SRAM2a and SRAM2b memories contain a secure section, which cannot be read nor written by CPU1. The secure start address for each memory can be read from the option byte, indicated in blue in [Figure 4](#):

- SFSA for the flash memory
- SBRSA for the SRAM2a (retained in Standby)
- SNBRSA for the SRAM2b.

These option bytes are only written by the FUS running on CPU2. This is done on each CPU2 update installed by the FUS.

The user application must take into account that the available memory can vary between different versions of the RF stack. The available space for the user application can be obtained from the release notes for STM32WB coprocessor wireless binaries. The install address for the RF stack is also the boundary address for the user flash memory area.

Ensure that some margin is included in CPU2 domain to support updates during the product lifetime.

The boundary granularity is 4 Kbytes for the flash memory and 1 Kbyte for SRAM2a and SRAM2b.

The linker file is identical to all delivered BLE/Thread applications (except for BLE_Thread_Static, BLE_HeartRate_ota and BLE_p2pServer_ota). The available memories are chosen to fit all provided applications. For applications like BLE, where CPU2 memory requirements are smaller, it is possible to update the linker file to allocate more memory to the application.

To optimize the available memory for a dedicated application, the linker file must be updated inline with the following guidelines:

- Flash memory: the end of available memory address can be moved up to the SFSA address. When a CPU2 update is required, there must be enough free memory just below the secure memory to upload a new encrypted CPU2 FW update. The size of the memory required depends on CPU2 FW to be updated (BLE, Thread or concurrent BLE/Thread), see [1].
- SRAM1: the first unavailable 32 bits in the linker file are only required for the BLE_OTA application. For all other applications, the start address can be moved from SRAM1_BASE + 4 to SRAM1_BASE.
- SRAM2a: the end of available memory address can be moved up to the SBRSA address. When CPU2 update support required, there must be some free sectors just below the secure memory to support new CPU2 FW updates requiring more sectors to be secure.
- SRAM2b: The SRAM2b is not part of the linker file because it is all secure for any FW CPU2 supporting the Thread protocol. For BLE only applications, the linker file can be updated with a new section to map RW data into the SRAM2B from SRAM2B_BASE up to the SNBRSA address. When CPU2 update support required, there must be some free sectors just below the secure memory to support new CPU2 FW updates requiring more sectors to be secure.

STOP2 is the deepest low power mode supported when RF is active. When the user application must enter Standby mode, it must first stop all RF activities, and fully reinitialize CPU2 when coming out of Standby mode. The user application can use the full non secure SRAM2a to store its own content (that needs to be retained in Standby mode).

4.3 Shared peripherals

AES2 is reserved to the CPU2 and must never be used/accessed by the CPU1.

AES1 is reserved to the CPU1 and is never used/accessed by the CPU2. The only case when the CPU2 accesses the AES1 is when the CPU1 requests to write a user key on the customer key storage area. This is described in [6].

All other peripherals concurrently accessible by both CPUs are protected by hardware semaphores. Before accessing these peripherals, the associated semaphore must first be taken, and released afterwards.

Table 2. Semaphores

Semaphore	Purpose
Sem0	RNG - All registers
Sem1	PKA - All registers
Sem2	Used to share between CPU1 and CPU2 the capability to write/erase data in FLASH
Sem3	RCC_CR RCC_EXTCFGR RCC_CFGR RCC_SMPSCR
Sem4	Clock control mechanism for the Stop mode implementation
Sem5	RCC_CRRCR RCC_CCIPR
Sem6	Used by CPU1 to prevent CPU2 from writing/erasing data in flash memory
Sem7	Used by CPU2 to prevent CPU1 from writing/erasing data in flash memory
Sem8	Ensures that CPU2 does not update the BLE persistent data in SRAM2 when CPU1 is reading them
Sem9	Ensures that CPU2 does not update the Thread persistent data in SRAM2 when CPU1 is reading them

If the application needs to use semaphores for inter task control, it is recommended to start using Sem31 downwards to be compatible with future wireless firmware updates on CPU1, where new features requiring additional semaphores can be added.

Sem0 is used to share the RNG between the two CPUs. The semaphore is taken by the CPU2 for an interval depending upon the number to be generated and upon the RNG source clock speed. To relax the latency to get these numbers, it is recommended to generate at startup a pool of numbers and fill the pool in a low priority task when some of them are retrieved by the application to keep it full. The usage of Sem0 is shown in [Figure 8](#).

Sem 0 can be used in the USB use case too. When the USB is not used anymore and needs to be switched off by the application, Sem 0 must be taken before switching off the CLK48 clock. This is required because USB and RNG share the same clock, and CPU2 could use RNG at the same time when CPU1 needs to switch off the USB (see [Figure 9](#)).

Sem1 is used to share the PKA IP between the two CPUs.

Sem2 is used to share between the two CPUs the capability to write/erase data in FLASH.

Sem2 must be taken before starting more than a single write/erase procedure, and released when they are completed. The semaphore must be taken/released to surround a couple of write/erase procedure. The semaphore is taken by the CPU2 for an interval depending upon the number of data to be written in the flash memory and upon the number of sectors to erase. BLE stack writes to flash memory the pairing information (when bonding is enabled) and the GATT attribute cache.

Sem3 is used for the low power management. It must not be locked for more than 500 µs by the CPU1 when there is BLE RF activity. The algorithm is detailed in [Figure 6](#) and [Figure 7](#).

Sem4 is used to handle race condition on the switch of the system clock when a CPU exits low power mode while the other one enter low power mode. The algorithm is detailed in [Figure 6](#) and [Figure 7](#).

Sem3 and Sem4 are used in the examples to enter/exit Stop mode.

The user must ensure that the algorithms shown in [Figure 6](#) and [Figure 7](#) are executed before and after wake-up from Stop mode. These routines (see [Figure 5](#)) are usually implemented inside the IDLE task of sequencer or RTOS. The implementation takes advantage of the fact that when WFI is called from critical section, the MCU wakes up upon interrupt request, but instead of executing ISR it continues to execute the next instruction after WFI. Only after exiting the critical section the ISR is executed.

```
PRIMASK = 1; // Mask all interrupts (enter critical section)
PWR_EnterStopMode()
WFI
PWR_ExitStopMode()
PRIMASK = 0; // Unmask all interrupts (exit critical section)
```

Figure 5. Timing for entering/exiting Stop mode on CPU1

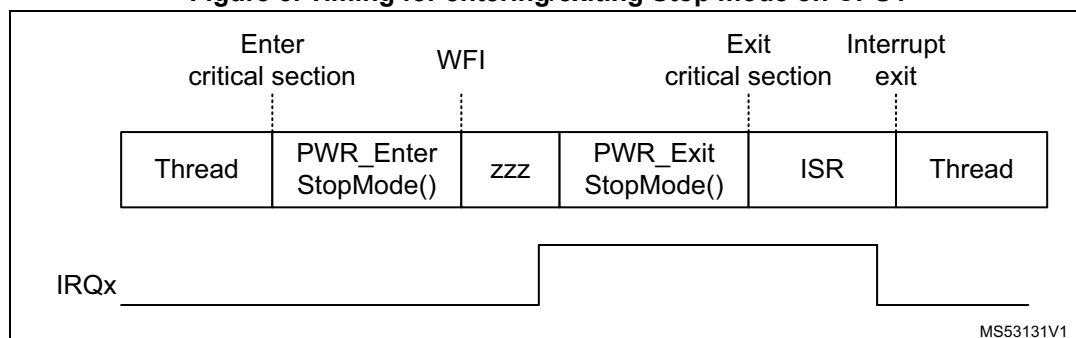


Figure 6. Algorithm to enter Stop mode on CPU1

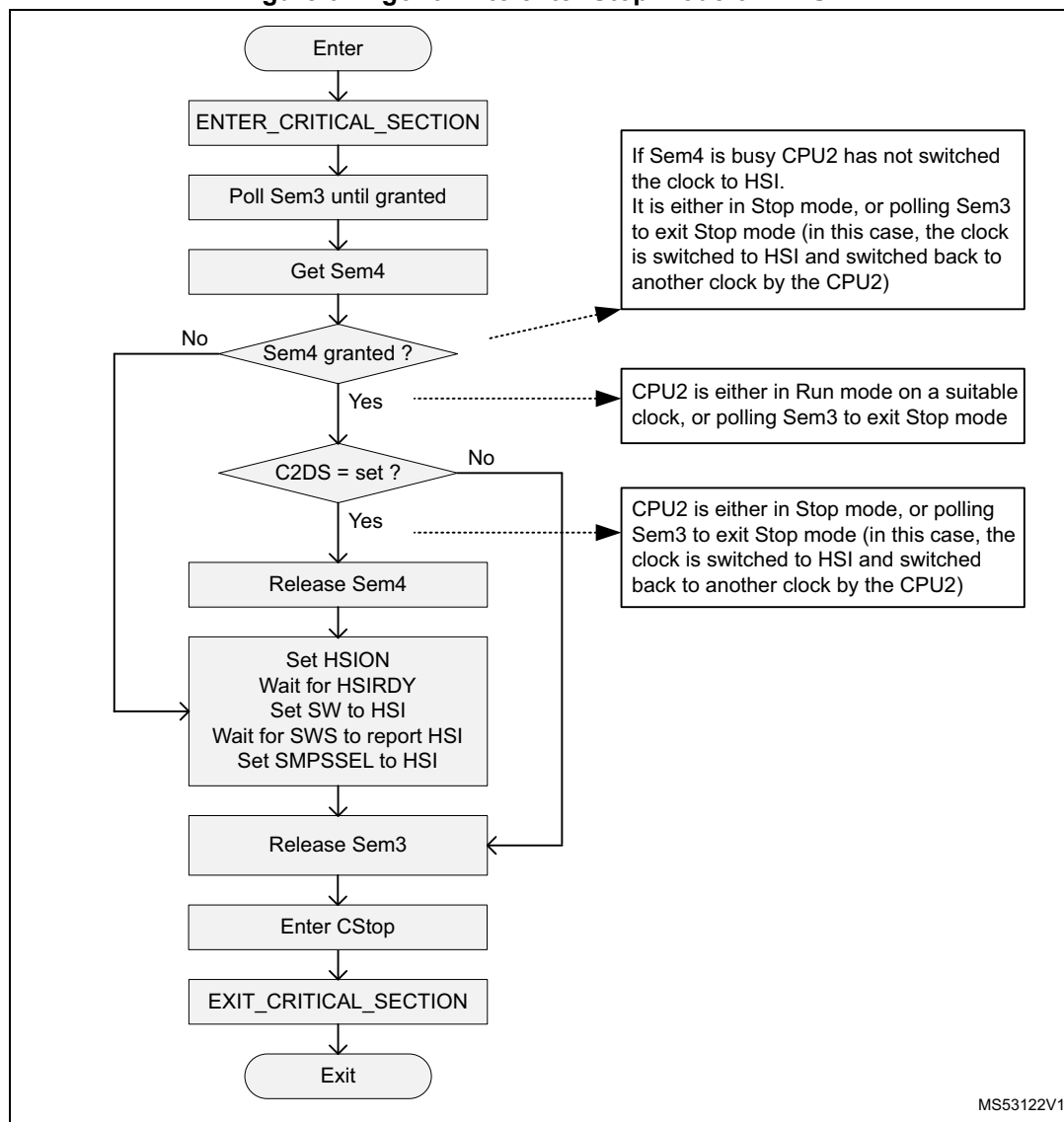
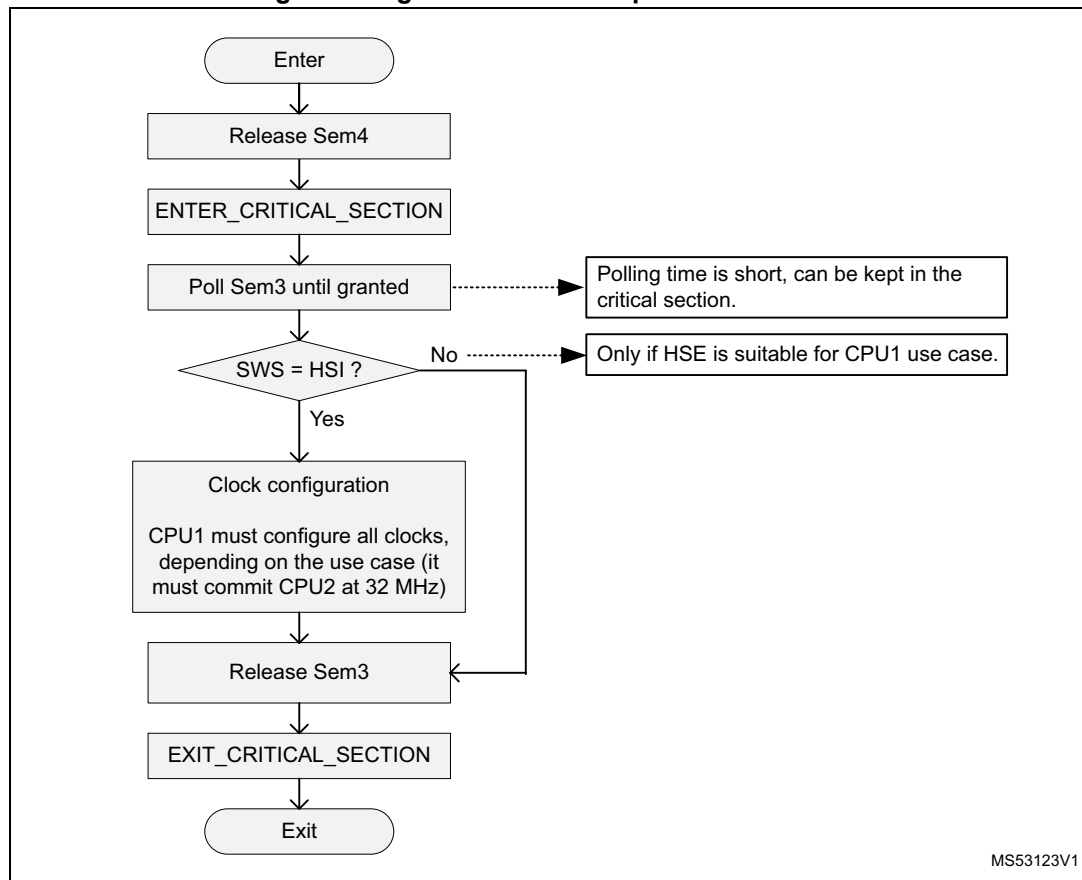


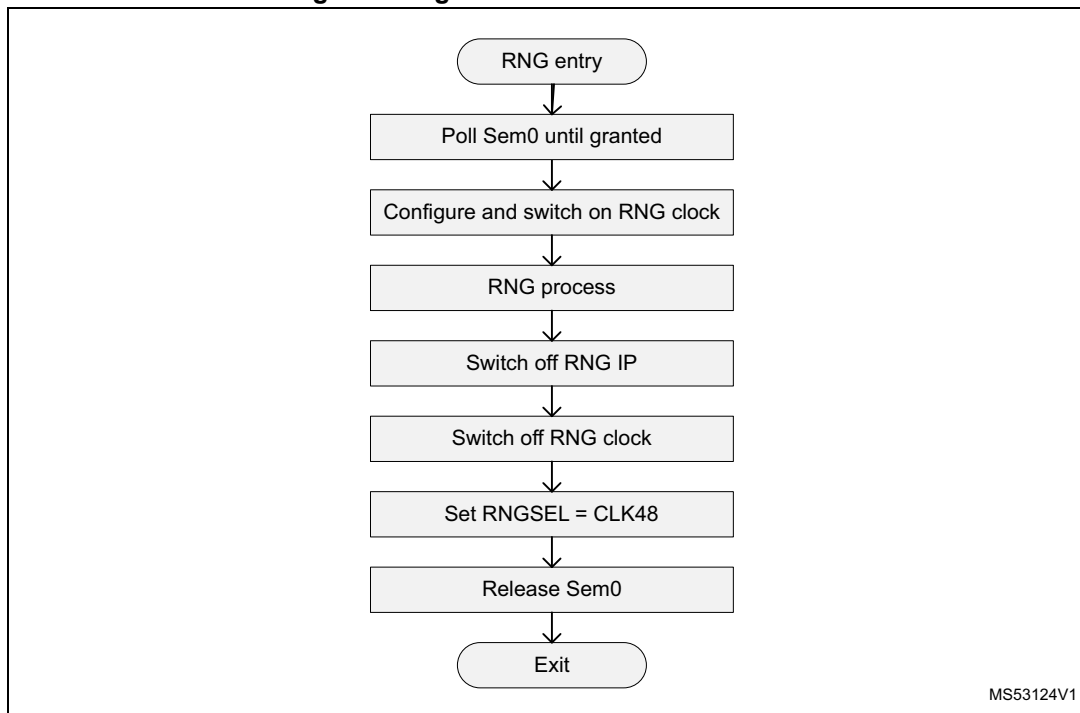
Figure 7. Algorithm to exit Stop mode on CPU1



Sem5 is used to control the RNG/USB CLK48 source clock. The CPU2 updates or switches off the clock only when the RNG IP (Sem0) is used.

To avoid a race condition with the CPU2, when the CPU1 needs to switch off the clock it must always first get Sem0, even if not using the RNG IP. This mechanism is shown in the BLE P-NUCLEO-WB55.USB dongle examples (see [Section 8.1.3: Configuration](#) and [Figure 9](#)). This does not impact the CPU2.

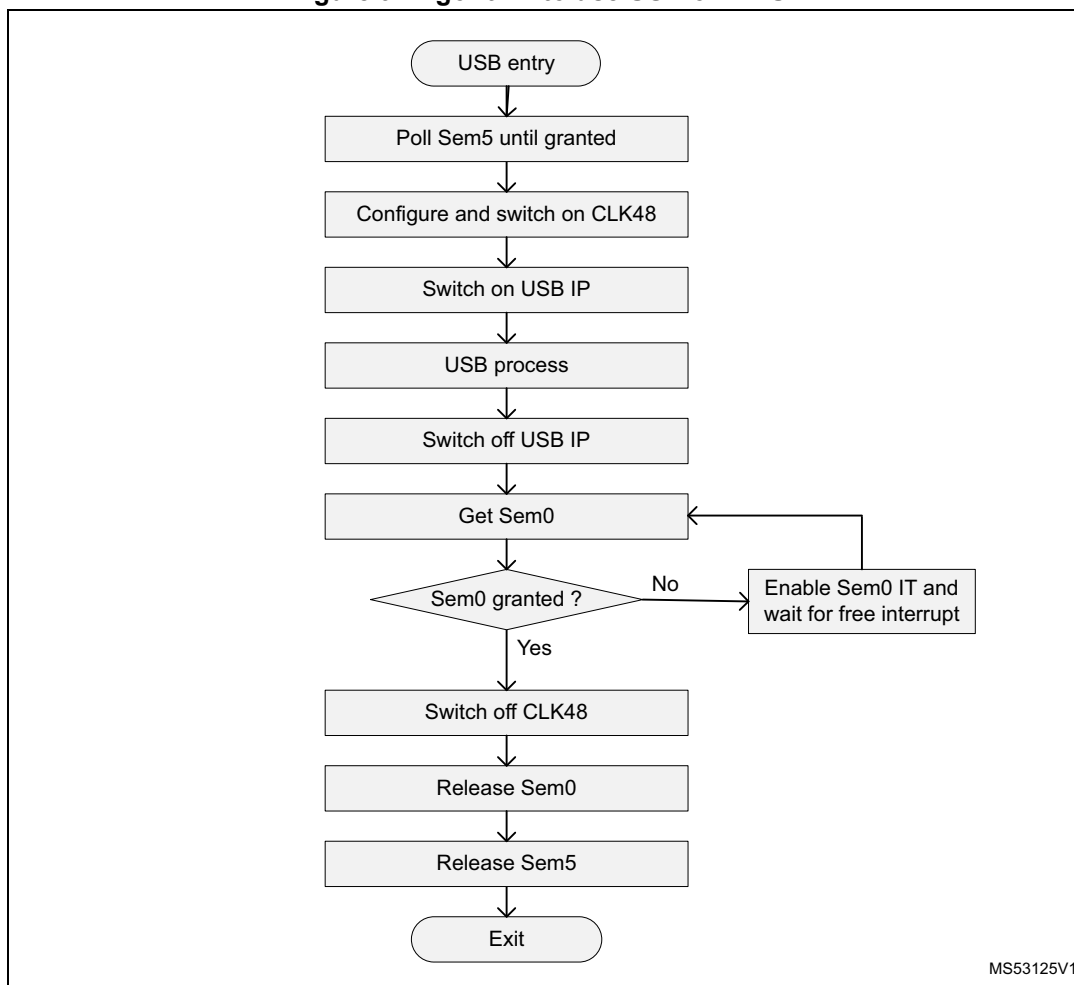
Figure 8. Algorithm to use RNG on CPU1



Note:

Sem5 is not taken because the CPU2 does not take it without taking first Sem0. This algorithm can be updated to take Sem5 before configuring the RNG clock source.

Figure 9. Algorithm to use USB on CPU1



The USB and RNG IPs share the same source clock. Before switching off the clock, the USB driver must first check whether the CPU2 requires the clock or not. To avoid a race condition with the CPU2, the CPU1 must first get Sem0 (RNG semaphore, CPU2 does not use USB) before switching off the clock.

If Sem0 is busy, the CPU1 must wait for Sem0 to be free to switch off the clock. This is required because there can be a race condition when CPU1 releases the USB and CPU2 releases the RNG at the same time, leading to the oscillator to be kept on.

Sem6 is used to protect the CPU1 timing versus write/erase operations requested by the CPU2. The CPU1 shall get Sem6 to prevent the CPU2 or other CPU1 processes to either write or erase data in flash memory. There is no time limit on how long the CPU1 can keep the semaphore, but, as long as the semaphore is taken, the CPU2 is unable to write either the pairing or client descriptor information in the memory.

CPU1 must release Sem6 only if it can afford being stalled for the time required to finish the write or erase operation.

The CPU2 implements the algorithm described in [Figure 10](#), similarly to the CPU1. Before writing or erasing data in flash memory, it tries to get Sem6 and, if successful, writes/erases

data and releases the semaphore. When the CPU1 needs to protect its timing, it polls Sem6 until it gets it.

Sem7 is used to protect the CPU2 timing versus write/erase flash memory operation requested by CPU1. The CPU1 must get Sem7 before writing or erasing. Sem7 must be taken and released for each single write or erase operation, but for not more than 0.5 ms in addition to the write/erase timing. To comply with this requirement the code must be executed in the critical section. The algorithm is described in [Figure 10](#).

Sem8 is used to ensure that CPU2 does not update the BLE persistent data in SRAM2 while CPU1 reads them.

The CPU2 can be configured to store the BLE persistent data either in the internal NVM storage on CPU2 or in the SRAM2 buffer provided by the user application. This can be configured with the system command *SHCI_C2_Config()* when the CPU2 is requested to store persistent data in SRAM2, so it can write data in this buffer when needed. To read consistent data with the CPU1 from the SRAM2 buffer, the flow must be:

1. CPU1 takes Sem8
2. CPU1 reads all persistent data from SRAM2 (most of the time, the goal is to write these data into an NVM managed by CPU1)
3. CPU1 releases Sem8

There is no timing constraint on how long this semaphore can be kept.

Sem9 is used to ensure that CPU2 does not update the Thread persistent data in SRAM2 while CPU1 reads them.

The CPU2 can be configured to store the Thread persistent data either in the internal NVM storage on CPU2 or in the SRAM2 buffer provided by the user application. This can be configured with the system command *SHCI_C2_Config()* when the CPU2 is requested to store persistent data in SRAM2, so it can write data in this buffer when needed. To read consistent data with the CPU1 from the SRAM2 buffer, the flow must be:

1. CPU1 takes Sem9
2. CPU1 reads all persistent data from SRAM2 (most of the time, the goal is to write these data into an NVM managed by CPU1)
3. CPU1 releases Sem9

There is no timing constraint on how long this semaphore can be kept.

4.4 Sequencer

The sequencer executes the registered functions one by one. It has the following features:

- supports up to 32 functions
- requests functions to be executed
- enables / disables the execution of a function
- provides a blocking interface based on the reception of an event.

The sequencer provides a simple background scheduling function. It provides a hook to implement a secure way Low-power mode (no event loss) when the sequencer does not have any pending tasks to be executed. It also provides an efficient mechanism for the application to wait for a specific event before moving forward. When the sequencer is waiting for a specific event, it provides a hook where the application can either enter low-power mode, or execute some other code.

4.4.1 Implementation

To use the sequencer, the application must:

- set the number of maximum of supported functions (this is done by defining a value for UTIL_SEQ_CONF_TASK_NBR)
- register a function to be supported by the sequencer with UTIL_SEQ_RegTask()
- start the sequencer by calling UTIL_SEQ_Run() to run a background while loop
- call UTIL_SEQ_SetTask() when a function needs to be executed.

4.4.2 Interface

Table 3. Interface functions

Function	Description
<code>void UTIL_SEQ_Idle(void);</code>	Called (in critical section - PRIMASK) when there is nothing to execute.
<code>void UTIL_SEQ_Run(UTIL_SEQ_bm_t mask_bm)</code>	Requests the sequencer to execute functions that are pending and enabled in the mask mask_bm.
<code>void UTIL_SEQ_RegTask(UTIL_SEQ_bm_t task_id_bm, uint32_t flags, void (*task)(void))</code>	Registers a function (task) associated with a signal (task_id_bm) in the sequencer. The task_id_bm must have a single bit set.
<code>void UTIL_SEQ_SetTask(UTIL_SEQ_bm_t task_id_bm,</code>	Requests the function associated with the task_id_bm to be executed. The task_prio is evaluated by the sequencer only when a function has finished. If several functions are pending at any one time, the one with the highest priority (0) is executed.
<code>void UTIL_SEQ_PauseTask(UTIL_SEQ_bm_t task_id_bm)</code>	Disables the sequencer to execute the function associated with task_id_bm.
<code>void UTIL_SEQ_ResumeTask(UTIL_SEQ_bm_t task_id_bm)</code>	Enables the sequencer to execute the function associated with task_id_bm.
<code>void UTIL_SEQ_WaitEvt(UTIL_SEQ_bm_t evt_id_bm)</code>	Requests the sequencer to wait for a specific event evt_id_bm and does not return until the event is set with UTIL_SEQ_SetEvt().
<code>void UTIL_SEQ_SetEvt(UTIL_SEQ_bm_t evt_id_bm)</code>	Notifies the sequencer that the event evt_id_bm occurred (the event must have been first requested).
<code>void UTIL_SEQ_EvtIdle(UTIL_SEQ_bm_t task_id_bm, UTIL_SEQ_bm_t evt_waited_bm)</code>	Called while the sequencer is waiting for a specific event.
<code>void UTIL_SEQ_ClrEvt(UTIL_SEQ_bm_t evt_id_bm)</code>	Clears the pending event.
<code>UTIL_SEQ_bm_t UTIL_SEQ_IsEvtPend(void)</code>	Returns the evt_id_bm of the pending event.

4.4.3 Detailed interface and behavior

The sequencer is a packaging of while loops to call functions when requested by the user:

```
while(1)
{
```

```

    if(task_id1)
    {
        task_id1 = 0;
        Fct1();
    }

    if (task_id2)
    {
        task_id2= 0;
        Fct2();
    }

    __disable_irq();
    If (! (task_id1| task_id2))
    {
        UTIL_SEQ_Idle();
    }
    __enable_irq();
}

```

void UTIL_SEQ_Run(UTIL_SEQ_bm_t mask_bm)

Implements the body of the while (1) loop. The mask_bm parameter is the list of functions that the sequencer is allowed to execute. Each function is associated with one bit in that mask_bm. At the end of the startup, this API must be called in a while (1) loop with mask_bm = (~0) to allow the sequencer to execute any pending function.

void UTIL_SEQ_Idle(void)

Called under the critical section (set with the CortexM PRIMASK bit - all interrupts are masked) when the sequencer does not have any function to executed. This is where the application must enter the Low-power mode.

void UTIL_SEQ_RegTask(UTIL_SEQ_bm_t task_id_bm, uint32_t flags, void (*task)(void))

Informs the sequencer to add the function task associated with the flag task_id_bm to its while loop.

void UTIL_SEQ_SetTask(UTIL_SEQ_bm_t task_id_bm , UTIL_SEQ_bm_t task_prio)

Sets the flag task_id_bm for the scheduler to call the associated function.

The task_prio is evaluated by the sequencer when it needs to decide which function to call next. This can be done only when the execution of the current function is finished. When several functions have their flag set, the one with the higher priority is executed (0 is the highest). This API can be called several times before the function is actually executed with a different priority. In that case, the sequencer records the highest priority. Whatever the

number of API calls before the function is executed, the sequencer runs the associated function only once.

```
void UTIL_SEQ_PauseTask( UTIL_SEQ_bm_t task_id_bm ) :
```

Informes the sequencer not to execute the function associated with the flag task_id_bm even if it is set. If the API UTIL_SEQ_SetTask() is called after UTIL_SEQ_PauseTask(), the request is recorded but the function is not executed. The mask associated with UTIL_SEQ_PauseTask() is independent from the mask associated with void UTIL_SEQ_Run(UTIL_SEQ_bm_t mask_bm).

A function can be executed only when its flag is set and enabled in both masks (default case).

```
void UTIL_SEQ_ResumeTask( UTIL_SEQ_bm_t task_id_bm ) :
```

Cancels the request done by UTIL_SEQ_PauseTask(). If this API is called when no UTIL_SEQ_PauseTask() has been requested, it has no effect.

```
void UTIL_SEQ_WaitEvt( UTIL_SEQ_bm_t evt_id_bm )
```

When this API is called, it does not return until the associated evt_id_bm signal is set. Only one bit in the evt_id_bm 32-bit value needs to be set. While the sequencer is waiting for this event, it calls UTIL_SEQ_EvtIdle() in a while loop on the event evt_id_bm. This must be used to replace all code where a polling is made on a flag before moving forward.

```
void UTIL_SEQ_SetEvt( UTIL_SEQ_bm_t evt_id_bm )
```

Must be called only when UTIL_SEQ_WaitEvt() has already been called. It sets the signal evt_id_bm the function UTIL_SEQ_WaitEvt() is waiting for. Calling this API before the UTIL_SEQ_WaitEvt() function makes the call to UTIL_SEQ_WaitEvt() return immediately as the flag is already set.

```
void UTIL_SEQ_EvtIdle(UTIL_SEQ_bm_t task_id_bm, UTIL_SEQ_bm_t  
evt_waited_bm)
```

Called while the API void UTIL_SEQ_WaitEvt() is waiting for the signal to be set with UTIL_SEQ_SetEvt().

This API is weakly implemented in the sequencer to call UTIL_SEQ_Run(0), so, while waiting for this event to occur, the function UTIL_SEQ_Idle() allows the system to enter low-power mode while waiting for the flag.

The application can implement this API to pass parameters different from 0 to the UTIL_SEQ_Run(mask_bm). Each bit set to 1 in the mask_bm requests the sequencer to execute the function associated with this flag when it is set with UTIL_SEQ_SetTask(). This means that when the function UTIL_SEQ_WaitEvt() is called, while it is waiting for the requested event to return, it can execute the unmasked functions when their flag is set, or call UTIL_SEQ_Idle() if no tasks is pending execution by the sequencer.

```
void UTIL_SEQ_ClrEvt( UTIL_SEQ_bm_t evt_id_bm )
```

This API can be called when, in some applications, the API UTIL_SEQ_WaitEvt() needs to be called while Evt is already set. In that case, the Evt must be cleared.

```
UTIL_SEQ_bm_t UTIL_SEQ_IsEvtPend( void ):
```

This API returns the Evt that is currently pending. When several UTIL_SEQ_WaitEvt() are nested, it returns the last one, which means the one that makes the deeper UTIL_SEQ_WaitEvt() to return to its caller.

4.5 Timer server

The timer server has the following features:

- Up to 255 virtual timers depending on available RAM capacity
- Single shot and repeated mode
- Stops a virtual timer and restarts it with a different timeout value
- Deletes a timer
- Timeout from 1 to $2^{32} - 1$ ticks

The timer server provides multiple virtual timers sharing the RTC wake-up timer. Each virtual timer can be defined as either single shot or a repeated timer. When a repeated timer comes to the end of a cycle, the user is notified and the virtual timer is automatically restarted with the same timeout. When a single shot timer ends, the user is notified and the virtual timer is set to the pending state (which means it is kept registered and can be restarted at any time). The user can stop a virtual timer and restart it with a different timeout value. When a virtual timer is no longer needed, the user must delete it to free the slot in the timer server.

The timer server can be used concurrently with the calendar.

4.5.1 Implementation

To use the timer server, the application must:

- Configure the RTC IP. When the calendar is required in the application, the RTC configuration must be compatible with the calendar settings requirement. When the calendar is not used, the RTC can be optimized for a Timer Server usage only.
- Initialize the timer server with HW_TS_Init().
- Implement HW_TS_RTC_Int_AppNot() (optional). When not implemented, the timer callback is called in the RTC interrupt handler context.
- Create a virtual timer with HW_TS_Create().
- Use the virtual timer with HW_TS_Stop(), HW_TS_Start().
- Delete the virtual when not needed using HW_TS_Delete().

4.5.2 Interface

Table 4. Interface functions

Function	Description
<code>void HW_TS_Init(HW_TS_InitMode_t TimerInitMode, RTC_HandleTypeDef *hrtc);</code>	Initializes the timer server.
<code>HW_TS_ReturnStatus_t HW_TS_Create(uint32_t TimerProcessID, uint8_t *pTimerId, HW_TS_Mode_t TimerMode, HW_TS_pTimerCb_t pTimerCallBack)</code>	Creates a virtual timer.
<code>void HW_TS_Stop(uint8_t TimerID)</code>	Stops a virtual timer.
<code>void HW_TS_Start(uint8_t TimerID, uint32_t timeout_ticks)</code>	Starts a virtual timer.
<code>void HW_TS_Delete(uint8_t TimerID)</code>	Deletes a virtual timer.
<code>void HW_TS_RTC_Wakeup_Handler(void);</code>	Timer server handler to be called from the RTC interrupt handler.
<code>uint16_t HW_TS_RTC_ReadLeftTicksToCount(void);</code>	Returns the number of ticks to count before the next interrupt.
<code>void HW_TS_RTC_Int_AppNot(uint32_t TimerProcessID, uint8_t TimerID, HW_TS_pTimerCb_t pTimerCallBack)</code>	Reports to the application that a virtual timer has expired.
<code>void HW_TS_RTC_CountUpdated_AppNot(void)</code>	Reports to the application that the number of ticks before the next interrupt has been updated by the timer server.

4.5.3 Detailed interface and behavior

The timer server provides virtual timers that run while the system is in Low-power mode down right down to Standby mode.

`void HW_TS_Init(HW_TS_InitMode_t TimerInitMode, RTC_HandleTypeDef *hrtc) :`

This command initializes the timer server based on the RTC IP configuration that must be made upfront.

TimerInitMode selects the timer server boot mode. When Standby mode is supported and the device wakes up from standby, set TimerInitMode to `hw_ts_InitMode_Limited` so that the timer server context is not reset. Otherwise, TimerInitMode must be set to `hw_ts_InitMode_Full` to run full initialization.

hrtc is the Cube HAL RTC handle.

`HW_TS_ReturnStatus_t HW_TS_Create(uint32_t TimerProcessID,
uint8_t *pTimerId,
HW_TS_Mode_t TimerMode,
HW_TS_pTimerCb_t pTimerCallBack) :`

pTimerId

This is the id returned by the timer server to the caller that needs to be used to Stop/Start/Delete the created timer.

TimerMode

The timer mode can be either in single shot or repeated mode. When in single shot mode, the timer is stopped when the timeout is reached. In repeated mode, it is restarted with the same previously programmed value at each timeout. This mode is fixed when the timer is created. To change the mode, the timer must be deleted and a new one must be created. Note that in this case the new allocated pTimerId can be different.

pTimerCallback

User callback on timeout.

TimerProcessID

This is defined by the user and is expected to be used in HW_TS_RTC_Int_AppNot(). When the timer is created, only the caller knows the Id that has been allocated. The TimerProcessID is returned in the HW_TS_RTC_Int_AppNot() with the pTimerCallback so that relevant decision can be done when implementing HW_TS_RTC_Int_AppNot().

void HW_TS_Stop(uint8_t TimerID)

Stops the timer TimerID. It has no effect if the timer is not running. The timer TimerID must have been created. When the timer is stopped, the TimerID remains allocated in the timer server so that the same timer (with the same TimerMode and same pTimerCallback) can be restarted with a different value.

void HW_TS_Start(uint8_t TimerID, uint32_t timeout_ticks)

Starts the timer TimerID with the timeout_ticks value. The value of the timeout_ticks depends on the configuration of the RTC IP. If the TimerID is already running, it is first stopped in the timer server and restarted with the new timeout_ticks value.

void HW_TS_Delete(uint8_t TimerID)

Deletes the TimerID from the timer server. The TimerID can be allocated to a new virtual timer. This API can be called on a running TimerID. In that case, it is first stopped and then deleted.

void HW_TS_RTC_Wakeup_Handler(void)

This interrupt handler must be called by the application in the RTC interrupt handler. This handler clears all required status flag in the RTC and EXTI peripherals.

```
uint16_t HW_TS_RTC_ReadLeftTicksToCount(void)
```

This API returns the number of ticks left to be counted before an interrupt is generated by the timer server. It can be used when the system needs to enter Low-power mode and decide which Low-power mode to apply, depending on when the next wake-up is expected.

When the timer is disabled (no timer in the list), it returns 0xFFFF.

```
void HW_TS_RTC_Int_AppNot(uint32_t TimerProcessID,  
uint8_t TimerID,  
HW_TS_pTimerCb_t pTimerCallBack)
```

This API must be implemented by the user application.

It notifies the application when a timer expires. This API is running in the RTC wake-up interrupt context and the application can prefer to call the pTimerCallBack as a background task depending on how much code is executed in the pTimerCallBack. As long as the TimerID is only known to the caller, the TimerProcessID can be used to identify to which module this pTimerCallBack belongs, and the application can assess if it can be called in the RTC wake-up interrupt context or not.

```
void HW_TS_RTC_CountUpdated_AppNot(void):
```

This API must be implemented by the user application.

This API notifies the application that the counter has been updated. This is expected to be used along with the HW_TS_RTC_ReadLeftTicksToCount () API. The counter can have been updated since the last call of HW_TS_RTC_ReadLeftTicksToCount () and before entering Low-power mode. This notification provides the application a way to solve the race condition to reevaluate the counter value before entering Low-power mode

4.6 Low power manager

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 exit of low-power mode.

The Low power manager provides the following features:

- Up to 32 users
- Stop mode and Off mode (standby and shutdown).
- Low-power mode selection
- Low-power mode execution
- callback when entering or exiting low-power mode
- Run mode not supported, when the application must stay in this mode, it must not call UTIL_LPM_EnterModeSelected ().

There is nothing to do to control the CPU2 low-power mode, the wireless firmware sets on its own the best low-power mode configuration of CPU2 as soon as CPU2 is started.

The low-power mode selection of CPU2 must be written to SHUTDOWN before the start of the CPU2, to cover the cases when the application starts but the CPU2 does not. In such cases, the reset value of the low-power mode selection must be overwritten to allow the

device to enter in low-power mode. Otherwise, it is impossible to go lower than Stop 0 mode until CPU2 starts, because of the reset value of the CPU2 low-power mode selection.

4.6.1 Implementation

The low power manager can handle up to 32 users with different low-power mode requests.

To use the low power manager, the application must:

- create a user Id
- call either UTIL_LPM_SetOffMode() or UTIL_LPM_SetStopMode() at any time with the defined user Id to set the requested Low-power mode
- call void UTIL_LPM_EnterLowPower() in background.

4.6.2 Interface

Table 5. Interface functions

Function	Description
UTIL_LPM_ModeSelected_t UTIL_LPM_ReadModeSel(void)	Returns the selected low-power mode to be applied.
UTIL_LPM_SetOffMode(UTIL_LPM_bm_t lpm_id_bm, UTIL_LPM_State_t state)	Enables or disables the Off mode for any user at any time.
void UTIL_LPM_SetStopMode(UTIL_LPM_bm_t lpm_id_bm, UTIL_LPM_State_t state)	Enables or disables the Stop mode for any user at any time.
void UTIL_LPM_EnterLowPower(void)	Enters the selected low-power mode.
void UTIL_LPM_EnterSleepMode(void)	API called before entering Sleep mode.
void UTIL_LPM_ExitSleepMode(void)	API called on exiting Sleep mode.
void UTIL_LPM_EnterStopMode(void)	API called before entering Stop mode.
void UTIL_LPM_ExitStopMode(void);	API called on exiting Stop mode.
void UTIL_LPM_EnterOffMode(void)	API called before entering Off mode.
void UTIL_LPM_ExitOffMode(void);	API called on exiting Off mode. This is called only if the MCU did not enter the mode as expected.

4.7 Flash memory management

The STM32WB share one single bank between CPU1 and CPU2. When the flash memory is either being written or erased, there is no way to fetch instruction from it.

When the CPU executes code from flash memory, it stalls as soon as a write or erase operation is started.

When the CPU executes code from SRAM, the CPU is not stalled while a write or erase operation is ongoing (assuming it does not read data from flash memory).

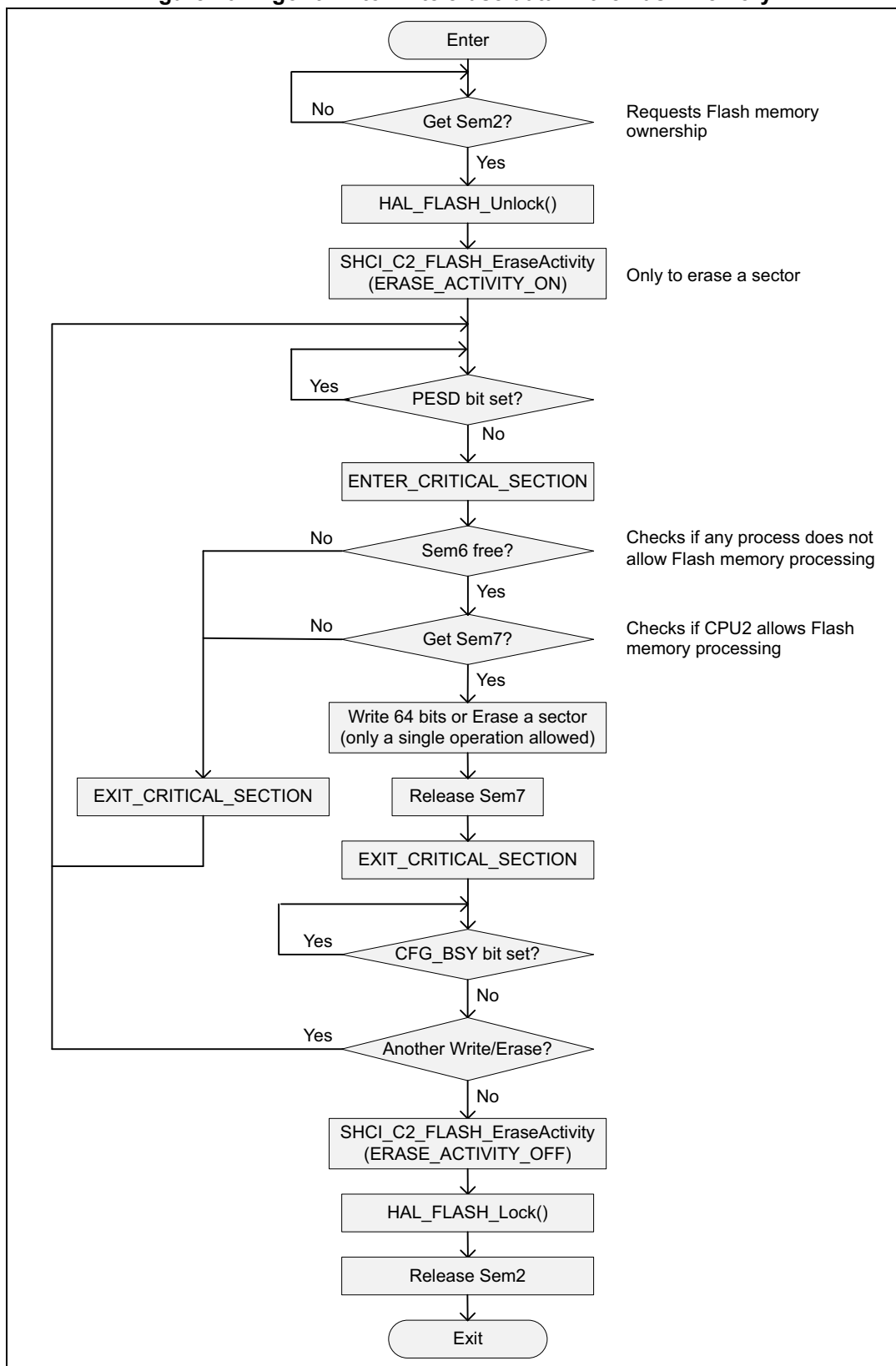
4.7.1 CPU2 timing protection

For security reason, CPU2 is prevented to execute any code from SRAM. To protect the CPU2 timing, it uses Sem7 to enable or disable flash memory operation requests from CPU1.

The application on CPU1 must implement the algorithm shown in [Figure 10](#) to write or erase the flash memory, and also implement in its own driver the following actions (outside the critical section defined in [Figure 10](#)):

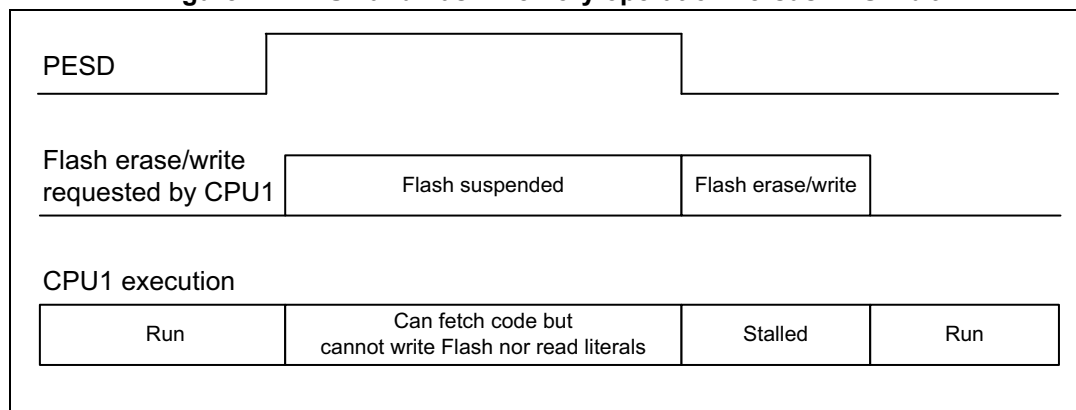
- Take Sem2 before any access to the flash memory and release it when it no longer needs it to access the IP
- When the user driver needs to erase sectors, it must first send the command `SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_ON)`. When all concerned sectors are erased, it must send the command `SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_OFF)`, see [Section 4.7.1](#).
- When the CPU2 timing protections uses the PESD bit mechanism (which is the case by default, see [Section 4.7.1](#)), the FLASH driver must either poll the `CFGBSY` bit from the `FLASH_SR` register or read back the memory until the value is the one to be written.

Figure 10. Algorithm to write/erase data in the flash memory



By default, CPU2 uses the PESD bit mechanism (from FLASH_SR register) to protect its BLE timing and not Sem7. The algorithm is still valid although checking Sem7 is useless. The drawback is that if the PESD bit is set by CPU2 at the same time when CPU1 starts a write or erase operation, CPU1 can fetch code but cannot read literals from the memory, even if the code to be executed requires this action. It is difficult to control whether CPU1 will be stalled or not when the PESD mechanism is used. Additionally, there is no interrupt signal on PESD bit release by CPU2, so asynchronous software flow is not possible.

Figure 11. CPU1 and flash memory operation versus PESD bit



The CPU2 use of PESD or Sem7 mechanism to protect the BLE timing is configurable by CPU1 with the system command `SHCI_C2_SetFlashActivityControl()`. Although it can be sent at any time, it is recommended to send it during the initialization phase.

By default, CPU2 protects its timing versus write operation requested by CPU1. When CPU1 needs to start an erase operation, it must first send the system command `SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_ON)`. When it does not expect to request erase operation anymore, it must send the system command `SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_OFF)`. These commands do not need to be sent for each single erase operation. It is recommended to enable the protection before requesting the first erase operation and to send the disable protection after the last erase operation has been performed.

4.7.2 CPU1 timing protection

When CPU1 needs to make sure it will not be stalled due to flash memory operation (write or erase) requested by CPU2, it must take Sem6. CPU2 does not request any flash memory operation until Sem6 is released.

When Sem6 is taken, it means CPU2 is already in the process to execute a flash memory operation (it is either close to be started or it has just finished). CPU1 must poll Sem6 to get it when it needs to prevent CPU2 to request any flash memory operation.

CPU2 uses the same algorithm described in [Figure 10](#).

4.7.3 Conflict between RF activity and flash memory management

Even if CPU1 does not use Sem6 to prevent CPU2 to start flash memory operations, there are some cases when it is impossible for CPU2 to start the erase.

Write is always possible, but when the NVM is full on CPU2, a write request may need an erase. In this case, data are not written until the erase is executed.

When the CPU2 needs to protect its timing versus erase (either because it has been notified by CPU1 with the command `SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_ON)` or because internally it needs to erase some sectors), any memory operation is forbidden 25 ms before the radio activity until the end of it. To execute the erase operation when BLE is running, the application must ensure that there is a radio idle time longer than 25 ms.

- BLE advertising: the advertising interval must be longer than 25 ms + advertising packet length to be able to execute flash memory erase operation.
- BLE connected: the connection interval must be longer than 25 ms + packet length, to be able to execute flash memory erase operation.
- Data throughput use case: When data streaming packet is sent, the radio is kept active to send as much data as possible between two connection intervals. Therefore, the radio can not be in idle long enough to fit an erase operation. When the device is master, it can reduce the Connection Event Length parameter (with either the `aci_gap_create_connection()` or `aci_gap_start_connection_update()` command) to prevent the device filling completely the interval between two connection intervals. When the device is slave, it must request the master to increase the connection interval so that the data to be sent fit only part of the interval between two connections events.

CPU2 needs to write data in flash memory when:

- after the pairing phase, only if bonding is enabled (first pairing or if the pairing is requested with the `force_rebond` parameter), to store the security information
- after a disconnection to store the GATT database, if the device has been previously bonded
- when `aci_gatt_store_db` command is called to store the GATT database for all active connections, if the device has been previously bonded
- when `aci_gap_clear_security_db` command is called to clear the bonding table (write to invalidate one or several records, security and GATT information)
- when `aci_gap_remove_bonded_device` command is called to remove a specified device from bonding table (write to invalidate the record, security and GATT information related to the specified device)

4.8 Debug information from CPU

4.8.1 GPIO

It is possible to output on GPIOs most of the real time activity of CPU2 such as background tasks, interrupt handlers and BLE IP Core signals. Assignment of a signal to particular GPIO is fully configurable from the CPU1 side except for the BLE IP Core signals as they are driven by HW. Therefore, the BLE IP Core GPIOs must be enabled only if not used by the application. The full configuration is made in the file `app_debug.c` located in `\Core\Src` for each application.

HW signals

The `aRfConfigList[]` table holds the list of GPIO driven by the hardware according to the radio activity. There are four parameters for each signal to monitor:

```
{ GPIOA, LL_GPIO_PIN_9, 0, 0},      /* DTB13 - Tx/Rx Start */
```

The first two parameters define the GPIO used (in this example, PA9 is used to output DTB13). These two parameters cannot be modified. To monitor the signal, the associated GPIO must be available on the board.

The third parameter is used to enable (1) or disable (0) the signal. All signals are set to 0 by default.

The fourth parameter is unused, keep it at 0.

To monitor a signal on the associated GPIO, the third parameter must be set to 1, and the BLE_DTB_CFG compiler switch at the top of the file must be set to 7.

The most useful signal is DTB13, which shapes all radio activity.

SW signals

The aGpioConfigList [] table holds the list of GPIO that are driven by the software. There are four parameters for each signal to monitor:

```
{ GPIOA, LL_GPIO_PIN_0, 0, 0}, /* BLE_ISR - Set on Entry / Reset on Exit */
```

The first two parameters define the GPIO used to output the signal. These are fully configurable. The user can select any GPIO unused in the application.

The third parameter is used to enable (1) or disable (0) the signal. All signals are set to 0 by default.

The fourth parameters is unused and must be kept to 0.

To monitor one signal on the associated GPIO, the third parameter must be set to 1.

4.8.2 SRAM2

Hardfault

When the CPU2 enters the hardfault interrupt handler, it can output different information before running an infinite loop.

It can set a GPIO if enabled in app_debug.c - aGpioConfigList [].

It writes in SRAM2A the following data:

@SRAM2A_BASE: 0x1170FD0F	Keyword that identifies a hardfault issue
@SRAM2A_BASE + 4	Program counter value that generated the hardfault
@SRAM2A_BASE + 8	Link register value when the instruction that generated the hardfault has been executed
@SRAM2A_BASE + 12	Stack pointer value when the instruction that generated the hardfault has been executed

Security attack

When the buffers provided to the CPU2 to exchange data through the mailbox are not in the unsecure SRAM, the CPU2 enters an infinite loop and writes the keyword 0x3DE96F61 at address SRAM2A_BASE.

4.9 FreeRTOS low power

Whatever the stack running on CPU2, the FreeRTOS low power mode shares the same implementation on the CPU1 for all wireless applications.

The HAL tick is mapped to TIM17 so that it does not collide with the systick reserved for FreeRTOS. The file `stm32wbxx_hal_timebase_tim.c` from `\Applications\BLE\BLE_HeartRateFreeRTOS\Core\Src` implements the HAL functions:

- `HAL_InitTick()`
- `HAL_SuspendTick()`
- `HAL_ResumeTick()`

The TIM17 user interrupt handler `HAL_TIM_PeriodElapsedCallback()` is implemented in `main.c` to increment the tick used by the HAL. This implementation can be customized to select another timer.

When FreeRTOS is in idle mode, the systick is switched off and replaced with a low power timer. The file `freertos_port.c` from `\Applications\BLE\BLE_HeartRateFreeRTOS\Core\Src` implements the tickless mode

- `vPortSuppressTicksAndSleep()` is reimplemented to support the tickless mode based on the low power mode available on STM32WB devices
- `vPortSetupTimerInterrupt()` is reimplemented to start a low power timer available on STM32WB devices

The current implementation is using the Timer server running on RTC. The timer selection can be changed by reimplementing the following functions:

- `LpTimerInit()` to initialize the low power timer to use.
- `LpTimerCb()` in case something more than just wake-up is required. In the current implementation, all actions done on wake-up are implemented on exit of low power mode in `vPortSuppressTicksAndSleep()` and not in the timer callback.
- `LpTimerStart()` to start the low power timer before entering low power mode.
- `LpGetElapsedTime()` to return how long the system has been in low power mode. This is required to update the systick used by FreeRTOS with `vTaskStepTick()`.

The low power mode is entered with `LpEnter()`. The current implementation is based on the low power manager used in all BLE applications, whether they are based on FreeRTOS or not. The implementation of `LpEnter()` can be customized.

BLE

The number of functions to be called in the background depends upon the application, which also determines if each function is called from a dedicated or a single common task. The BLE architecture supports any combination.

Whatever the BLE application, there must be at least two functions to be called in a task:

- **hci_user_evt_proc()**: when `hci_notify_asynch_evt()` is called from the middleware, this function must be called in the background. `hci_user_evt_proc()` must not be called inside `hci_notify_asynch_evt()` as it can be called from the IPCC interrupt context. There is no timing constraint between the time `hci_notify_asynch_evt()` is called from the middleware and the time when `hci_user_evt_proc()` is called in the background. However, in some data throughput use cases, the performance is better when the time is short enough to read the events at the same rate they are notified. When several `hci_notify_asynch_evt()` are received, the `hci_user_evt_proc()` function needs to be called only once from the background. It does not hurt to call several times `hci_user_evt_proc()` from the background whereas there was only one or no notification with `hci_notify_asynch_evt()`.
- **shci_user_evt_proc()**: the requirement is the same as for `hci_user_evt_proc()` with the associated notification `shci_notify_asynch_evt()`. Note that there is no currently data throughput on this system channel.

As long as it is not possible to send a BLE command while there is already one pending, or a system command while there is already one pending, the middleware provides hook so that the application can implement a semaphore mechanism.

When `hci_cmd_resp_wait()` is called from the middleware, a semaphore must be taken and released on reception of `hci_cmd_resp_release()`. The application must not return from `hci_cmd_resp_wait()` until the semaphore is released.

Another semaphore must be used to handle the same mechanism on the system channel with `shci_cmd_resp_wait()/shci_cmd_resp_release()`.

4.10 Device information table

As soon as the System Ready Event is received from the CPU2, the device information table (DIT) can be read from the SRAM2A.

<i>uint32_t Safe Boot Version</i>	Safe boot version
<i>uint32_t FUS Version</i>	FUS version
<i>uint32_t FUS MemorySize</i>	Memory required by the FUS
<i>uint32_t FusInfo</i>	Reserved - Set to 0
<i>uint32_t Wireless Firmware Version</i>	Wireless firmware version
<i>uint32_t Wireless Firmware MemorySize</i>	Memory required by the wireless firmware
<i>uint32_t InfoStack</i>	Wireless firmware information

The DIT has a different mapping when filled by the FUS (see [\[6\]](#)) or by the wireless firmware.

The system command *SHCI_GetWirelessFwInfo()* can decode the two DIT mappings.

The DIT address can be found at the start of SRAM2 (+ IPCCDBA offset - user option byte). Unless modified by the user, IPCCDBA is always set to 0, hence the DIT address can be found at the first address of SRAM2A.

Figure 12. Format of version and memory information

Version	Version - Major								Version - Minor								Subversion								Branch				Build			
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Memory size	SRAM2b (no. of 1 KB sectors)								SRAM2a (no. of 1 KB sectors)								Reserved								Flash memory (no. of 4 KB sectors)							
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The Build information is always different from 0 for all official versions.

The Branch information is for internal use.

Only the InfoStack LSB is used, it provides the information on which wireless firmware is running on CPU2, namely:

- INFO_STACK_TYPE_BLE_STANDARD: 0x01
- INFO_STACK_TYPE_BLE_HCI: 0x02
- INFO_STACK_TYPE_BLE_LIGHT: 0x03
- INFO_STACK_TYPE_BLE_BEACON: 0x04
- INFO_STACK_TYPE_THREAD_FTD: 0x10
- INFO_STACK_TYPE_THREAD_MTD: 0x11
- INFO_STACK_TYPE_ZIGBEE_FFD: 0x30
- INFO_STACK_TYPE_ZIGBEE_RFD: 0x31
- INFO_STACK_TYPE_MAC: 0x40
- INFO_STACK_TYPE_BLE_THREAD_FTD_STATIC: 0x50
- INFO_STACK_TYPE_BLE_THREAD_FTD_DYNAMIC: 0x51
- INFO_STACK_TYPE_802154_LLD_TESTS: 0x60
- INFO_STACK_TYPE_802154_PHY_VALID: 0x61
- INFO_STACK_TYPE_BLE_PHY_VALID: 0x62
- INFO_STACK_TYPE_BLE_LLD_TESTS: 0x63
- INFO_STACK_TYPE_BLE_RLV: 0x64
- INFO_STACK_TYPE_802154_RLV: 0x65
- INFO_STACK_TYPE_BLE_ZIGBEE_FFD_STATIC: 0x70
- INFO_STACK_TYPE_BLE_ZIGBEE_RFD_STATIC: 0x71
- INFO_STACK_TYPE_BLE_ZIGBEE_FFD_DYNAMIC: 0x78
- INFO_STACK_TYPE_BLE_ZIGBEE_RFD_DYNAMIC: 0x79
- INFO_STACK_TYPE_RLV: 0x80

4.11 ECCD error management

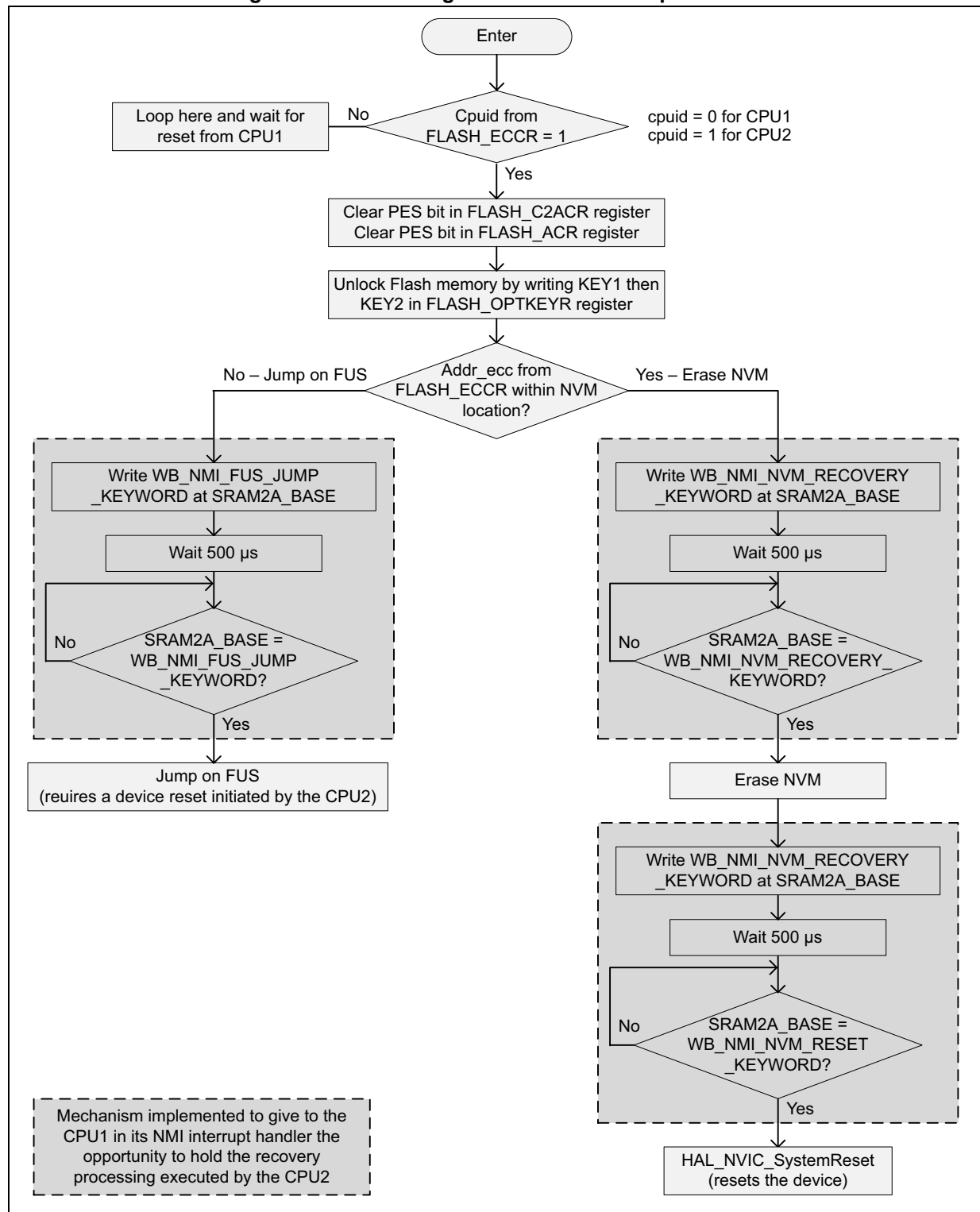
An NMI interrupt can be generated because of an ECCD flash memory error, either in the NVM data section, or in the code data section.

When the ECCD is generated from the NVM data section, the CPU2 can erase the NVM to remove the error.

When the ECCD is generated from the code section, the CPU2 must restart on the FUS to request a new wireless firmware install.

On ECCD error the NMI is generated to both CPUs. The algorithm shown in [Figure 13](#) (WB_NMI_NVM_RECOVERY_KEYWORD = 0xAFB449C9, WB_NMI_RESET_KEYWORD = 0x8518C6F2 and WB_NMI_FUS_JUMP_KEYWORD = 0x7E3FF448) describes the way the CPU2 manages the ECCD error and the mechanism to allow the CPU1 to hold CPU2 error processing.

Figure 13. ECC management in NMI interrupt handler



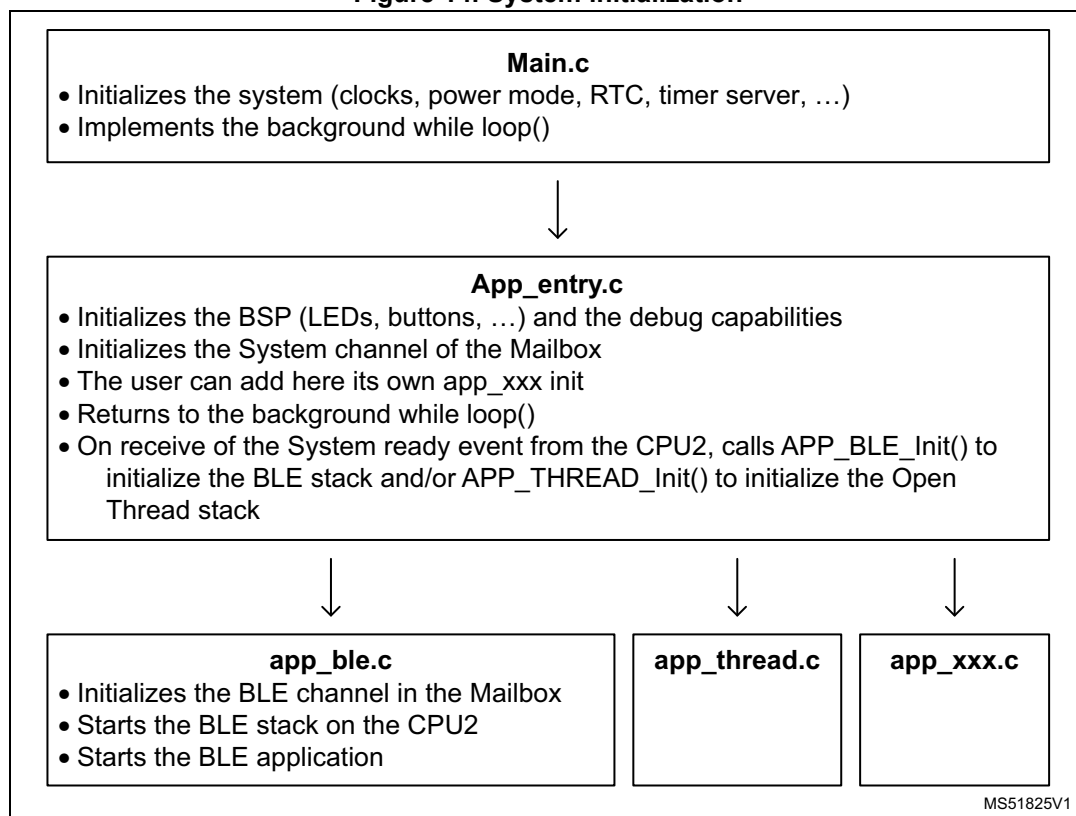
5 System initialization

5.1 General concepts

All applications begin with three sets of files (see [Figure 14](#)):

1. `main.c`: all HW configuration that is common to any application (the clock provided to the CPU2 must be always 32 MHz)
2. `app_entry.c`: all SW configuration and implementation that is common to any application
3. `app_ble.c` / `app_thread.c` / `app_xxx.c`: application dedicated files

Figure 14. System initialization



5.2 CPU2 startup

At startup, the CPU2 runs the minimal set of initializations to make available all supported system features. At the end of the initialization, the CPU2 reports the System ready event over the system channel to the CPU1. At this time, the CPU1 can send any System command to the CPU2, including all commands required to manage the keys in the customer key storage (CKS) in the secure CPU2 flash memory.

During the CPU2 startup phase, some shared resources are involved:

- The RNG peripheral fills a pool used when some random numbers are required for operation, so that these numbers are available without delay when the RNG IP is

already used by the CPU1. The RNG peripheral access requires getting Sem0. In addition, the CPU2 makes one attempt to get Sem5. If this is successful, it switches ON the HSI48 oscillator and configures the 48 MHz clock selection of the RNG IP to be HSI48. This assumes the RCC is configured to feed the RNG IP with a 48 MHz clock and not either LSI or LSE. In the latter case, the previous step is done anyway even though not relevant and the RNG operates on the selected clock. When Sem5 is busy, the CPU2 does not change anything in the RNG clock configuration and uses the current configuration. The HSI48 oscillator is switched OFF by the CPU2 when a wireless stack is started. This requires Sem5 to be available and only one attempt to take Sem5 is made.

- The NVM consistency is checked, and, if corrupted, it is reformatted. This operation requires to erase flash memory sectors. The access to the NVM at startup is compliant with the general rules to start any process on the memory. It first requires getting Sem2 to take the ownership of the flash memory, the CPU1 has the capability to hold any operation using Sem6.

All these steps must be completed before sending the System ready event so when any semaphore is required by the CPU2, it polls on it until it is free.

The CPU2 can execute its startup sequence until it reports the System ready event without the need for an external HSE or LSE oscillator.

Once the CPU2 has reported the System ready event, all system commands are supported without any external HSE or LSE oscillator.

HSE and LSE oscillators are required when a wireless protocol stack is started. If power consumption is not an issue it is possible to get rid of the external LSE oscillator and configure the device to use the HSE (32 MHz) / 1024 (= 32.768 kHz) instead.

Note: On STM32WB1x devices the HSI48 oscillator is not present, so when a wireless protocol stack is running, LSE or LSI oscillators must be active.

6 PLL management

6.1 How to switch the system clock between HSE and PLL

To switch the system clock between HSE and PLL clock, the C2HPRE divider in RCC_EXTCFGR register must be changed to provide to CPU2 a clock at 32 MHz.

6.1.1 Case 1: Before CPU2 is started

If CPU2 is not started, C2HPRE divider can be modified by CPU1. The algorithm shown in the next two figures describe how to switch the system clock between PLL and HSE and vice versa.

Figure 15. Algorithm to switch the system clock from PLL to HSE

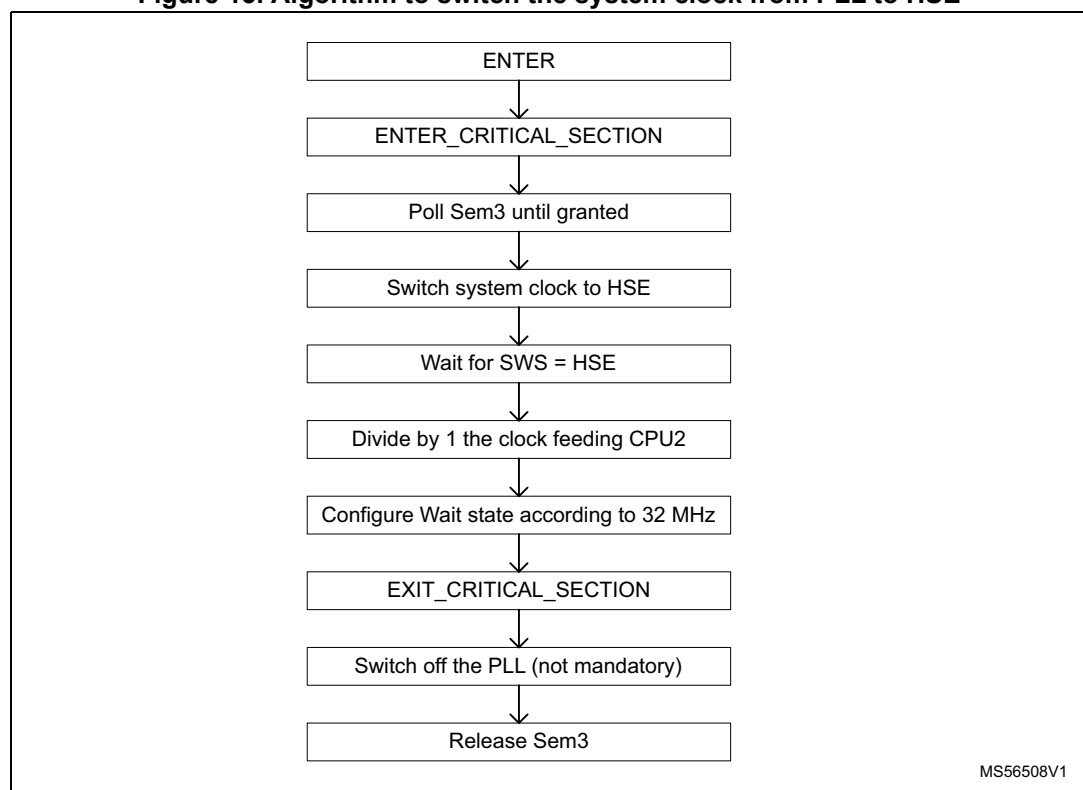
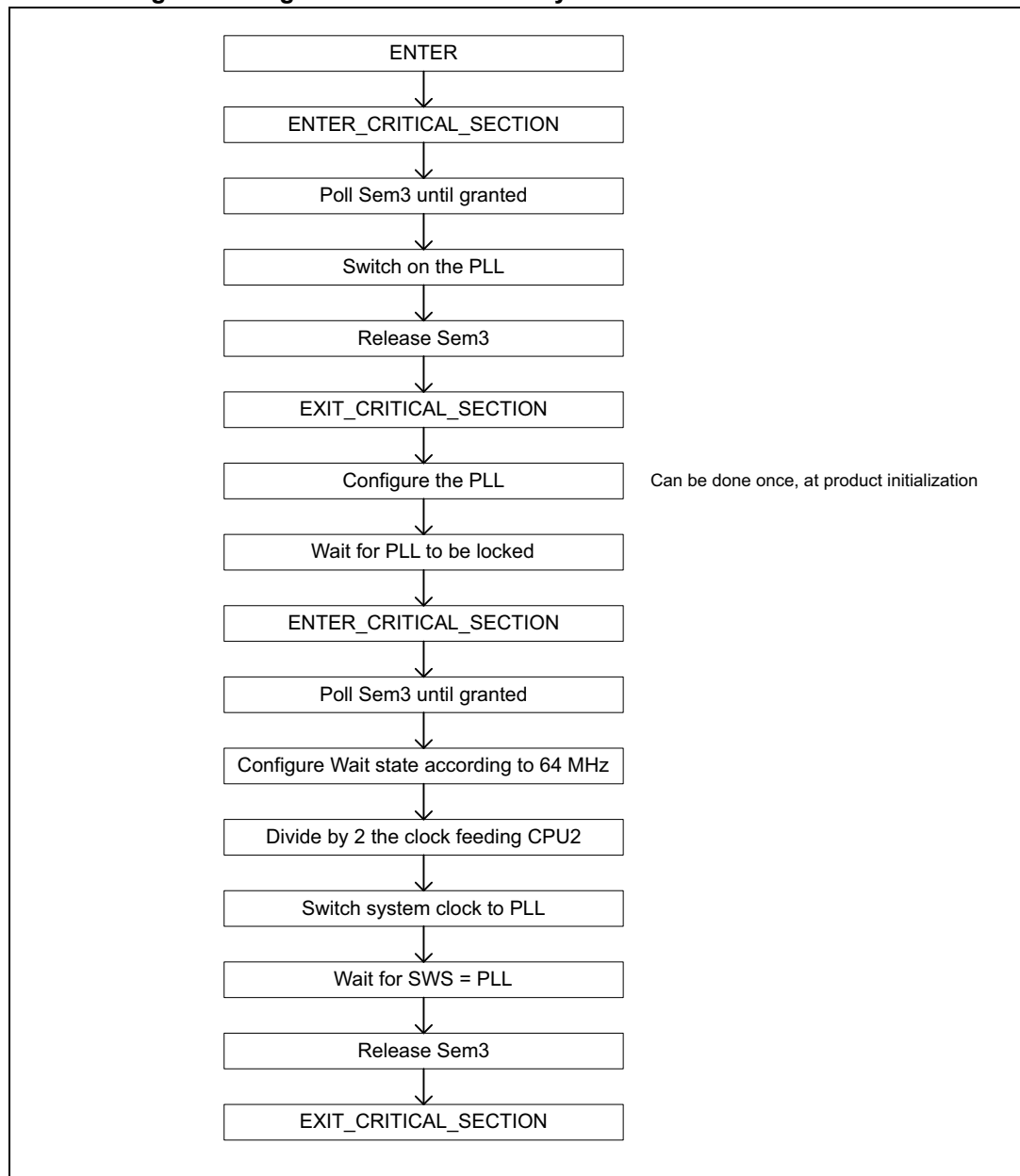


Figure 16. Algorithm to switch the system clock from HSE to PLL

6.1.2 Case 2: CPU2 is started

C2HPRE divider cannot be updated by CPU1, who must call the SHCI_C2_SetSystemClock system command to request CPU2 to manage the switch of the system clock

Once CPU2 is started, C2HPRE divider in RCC_EXTCFGR register must not be updated by the CPU1, only CPU2 can modify this divider. If CPU1 needs to switch the system clock from HSE to PLL and vice versa, it must use SHCI_C2_SetSystemClock system command.

The SHCI_C2_SetSystemClock system command has one parameter that can take the following values:

- SET_SYSTEM_CLOCK_HSE_TO_PLL
- SET_SYSTEM_CLOCK_PLL_ON_TO_HSE
- SET_SYSTEM_CLOCK_PLL_OFF_TO_HSE

To switch the system clock from HSE to PLL, the flow is:

1. CPU1 configures and starts the PLL (PLLON=1)
2. CPU1 does not need to wait for PLL to be ready
3. CPU1 calls the system command SHCI_C2_SetSystemClock(SET_SYSTEM_CLOCK_HSE_TO_PLL - 0x00). CPU2 is responsible to wait for PLL to be locked, set the FLASH LATENCY to 3 WS, set C2HPRE divide to divide the CPU2 clock by 2, and switch the system clock on PLL.
4. In return of the command, the system clock runs on the PLL.

To switch the system clock from PLL to HSE, the flow is:

1. CPU1 starts HSE (HSEON = 1)
2. CPU1 does not need to wait for HSE to be ready
3. CPU1 calls the system command SHCI_C2_SetSystemClock(SET_SYSTEM_CLOCK_PLL_ON_TO_HSE - 0x01) or SHCI_C2_SetSystemClock(SET_SYSTEM_CLOCK_PLL_OFF_TO_HSE - 0x02). CPU2 is responsible to switch the system clock on HSE, set C2HPRE divider to divide the CPU2 clock by 1 and set the FLASH LATENCY to 1 WS. The PLL is switched off if requested with parameter 0x02. Otherwise, CPU1 must switch it off.
4. On return of the command, the system clock is running on HSE.

Before entering in Stop mode (or Standby on STM32WB1x), the CPU1 must request the switch from PLL to HSE with the system command if the system clock used is the PLL. To call this command from critical section, the system command must be used in polling mode (see [How to use the system command in polling mode](#)). To reduce the critical section duration when this command is sent from critical section, the BLE command can be used in blocking mode (see [How to use the system command in polling mode](#)).

Note: *If the applications use only Sleep Mode, then it is good enough to switch on the PLL before CPU2 starts. When the device is in Sleep Mode, CPU2 does not make any change on the system clock. It is not required on CPU1 to switch on HSE before entering Sleep Mode. The requirement is there only for Stop Mode (and Standby for STM32WB1x devices).*

7 Step by step design of a BLE application

This section provides information and code examples on how to design and implement a BLE application on a STM32WB device.

7.1 Initialization phase

Several steps are mandatory to initialize the application.

- Initialize the device (HAL, reset device, clock and power configuration)
- Configure platform (buttons, LEDs)
- Configure hardware (UART, debug)
- Configure the BLE device public address (if used):
 - `aci_hal_write_config_data()` API
- Configure Tx Power
 - `aci_hal_set_tx_power_level()` API
- Init BLE GATT layer:
 - `aci_gatt_init()` API
- Init BLE GAP layer depending on the selected device role:
 - `aci_gap_init("role")` API
- Set the proper security I/O capability and authentication requirement (if BLE security is used):
 - `aci_gap_set_io_capability()` and `aci_gap_set_authentication_requirement()` APIs
- Define the required services, characteristics and characteristic descriptors if the device is a GATT server:
 - `aci_gatt_add_service()`, `aci_gatt_add_char()`, `aci_gatt_add_char_desc()` APIs
- Use sequencer to manage tasks and Low power

7.2 Advertising phase (GAP peripheral)

To establish a connection between a BLE GAP central (master) device and a BLE GAP peripheral (slave) device, the GAP discoverable mode must be initiated on the peripheral device. The APIs in [Table 6](#) can be used.

Table 6. Advertising phase API description

API name	Description
<code>aci_gap_set_discoverable()</code>	Sets the device in general discoverable mode. The device is discoverable until the device issues the <code>aci_gap_set_non_discoverable()</code> API.
<code>aci_gap_set_limited_discoverable()</code>	Sets the device in a limited discoverable mode. The device is discoverable for a maximum period TGAP (lim_adv_timeout) of 180 seconds. The advertising can be disabled at any time by calling <code>aci_gap_set_non_discoverable()</code> API.
<code>aci_gap_set_direct_connectable()</code>	Sets the device in direct connectable mode. The device is in this mode for 1.28 seconds, if no connection is established within this period, the device enters non-discoverable mode and advertising must be enabled again explicitly.
<code>aci_gap_set_non_connectable()</code>	Puts the device into non-connectable mode
<code>aci_gap_set_undirect_connectable()</code>	Puts the device into undirected connectable mode.

7.3 Discoverable and connectible phase (GAP central)

To create a connection between two devices, the GAP central can discover the remote and then initiate a connection to the target device. It is also possible to initiate a direct connection to the specified device.

The APIs that can be used for the GAP discovery procedure are listed in [Table 7](#).

Table 7. GAP central APIs

API	Description
<code>aci_gap_start_limited_discovery_proc ()</code>	Starts the limited discovery procedure. The controller starts the active scanning. Only the devices in limited discoverable mode are returned to the upper layers.
<code>aci_gap_start_general_discovery_proc ()</code>	Starts the general discovery procedure. The controller starts active scanning.
The following APIs be used in the procedure to establish the GAP connection	
<code>aci_gap_start_auto_connection_establish_proc ()</code>	Starts the auto connection procedure. The specified devices are added to the controller white list and initiate connection calls to the GAP controller using the initiator filter policy set to “use white list to determine which advertiser to connect to”.
<code>aci_gap_create_connection ()</code>	Starts the direct connection procedure. A create connection call is made to the controller by GAP with the initiator filter policy set to “ignore white list and process connectible advertising packets only for the specified device”.

Table 7. GAP central APIs (continued)

API	Description
<code>aci_gap_start_auto_connection_establish_proc()</code>	Starts the auto connection procedure. The specified devices are added to the controller white list and a create connection call is made to the controller by GAP with the initiator filter policy set to “use white list to determine which advertiser to connect to”.
<code>aci_gap_start_general_connection_establish_proc()</code>	Starts a general connection procedure. The device enables a controller scan with the scanner filter policy set to “accept all advertising packets” and from the scanning results, all the devices are sent to the upper layer using the event callback <code>hci_le_advertising_report_event()</code> .
<code>aci_gap_start_selective_connection_establish_proc()</code>	Starts a selective connection procedure. The GAP adds the specified device addresses into the white list and enables a controller scan with the scanner filter policy set to “accept packets only from devices in white list”. All the devices found are sent to the upper layer by the event callback <code>hci_le_advertising_report_event()</code> .
<code>aci_gap_terminate_gap_proc()</code>	Terminates the specified GAP procedure.

7.4 Services and characteristic configuration (GATT server)

To add a service and its related characteristics, a user application chooses from one of two profiles defined here:

- Standard profile defined by the Bluetooth SIG.
The user must follow the profile specification and services, and the characteristic specification documents to implement them using the related defined profile, services and characteristics 16-bit UUID (refer to Bluetooth SIG web page).
- Proprietary, non-standard profile.
The user must define custom services and characteristics. In this case, 128-bit UUIDs are required and must be generated by profile implementers (refer to UUID generator web page on www.famkruithof.net).

A service can be added using the following procedure:

```
aci_gatt_add_service(uint8_t Service_UUID_Type,
Service_UUID_t *Service_UUID,
uint8_t Service_Type,
uint8_t Max_Attribute_Records,
uint16_t *Service_Handle);
```

This procedure returns the pointer to the service handle (`Service_Handle`), which is used to identify the service within the user application. A characteristic can be added to this service using the following procedure:

```
aci_gatt_add_char(uint16_t Service_Handle,
uint8_t Char_UUID_Type,
Char_UUID_t *Char_UUID,
uint8_t Char_Value_Length,
uint8_t Char_Properties,
```

```
uint8_t Security_Permissions,
uint8_t GATT_Evt_Mask,
uint8_t Enc_Key_Size,
uint8_t Is_Variable,
uint16_t *Char_Handle);
```

This procedure returns the pointer to the characteristic handle (Char_Handle), which is used to identify the characteristic within the user application.

If the characteristic owner is in Notify or Indicate mode and enabled, the GATT server side must use the following API to send a notification or indication to the GATT client.

```
aci_gatt_update_char_value()
```

7.5 Service and characteristic discovery (GATT client)

Once two devices are connected, the application data exchange is based on GATT client-server architecture.

One device must implement the F and remove the GATT client.

The following APIs are used by the GATT client to discover services and characteristics, to enable/disable notification/indication to the GATT server, to write/read characteristics and to confirm GATT server Indication.

Table 8. GATT client APIs

API	Description
<code>aci_gatt_disc_all_primary_services ()</code>	<p>Starts the GATT client procedure to discover all primary services on the GATT server. It is used when a GATT client connects to a device and wants to find all the primary services provided on the device to determine what it can do.</p> <p>The procedure responses are given through the <code>aci_att_read_by_group_type_resp_event()</code> event callback</p>
<code>aci_gatt_disc_primary_service_by_uuid()</code>	<p>Starts the GATT client procedure to discover a primary service on the GATT server by using its UUID. It is used when a GATT client connects to a device and wants to find a specific service without the need for any other service.</p> <p>The procedure responses are given through the <code>aci_att_find_by_type_value_resp_event()</code> event callback</p>
<code>aci_gatt_find_included_services()</code>	<p>Starts the procedure to find all included services. It is used when a GATT client wants to discover secondary services once the primary services have been discovered.</p> <p>The procedure responses are given through the <code>aci_att_read_by_type_resp_event()</code> event callback.</p>

Table 8. GATT client APIs (continued)

API	Description
<code>aci_gatt_disc_all_char_of_service()</code>	Starts the GATT procedure to discover all the characteristics of a given service. The procedure responses are given through the <code>aci_att_read_by_type_resp_event()</code> event callback.
<code>aci_gatt_disc_char_by_uuid()</code>	Starts the GATT procedure to discover all the characteristics specified by a UUID. The procedure responses are given through the <code>aci_gatt_disc_read_char_by_uuid_resp_event()</code> event callback.
<code>aci_gatt_disc_all_char_desc()</code>	Starts the procedure to discover all characteristic descriptors on the GATT server. The responses are given through the <code>aci_att_find_info_resp_event()</code> event callback.

For all commands, the end of the procedure is indicated by `aci_gatt_proc_complete_event()` event callback.

7.6 Security (pairing and bonding)

The BLE security model includes five security features:

1. Pairing: process for creating one or more shared secret keys.
2. Bonding: act of storing the keys created during pairing for use in subsequent connections in order to form a trusted device pair.
3. Device authentication: verification to ensure two devices have the same keys.
4. Encryption: provides message confidentiality.
5. Message integrity: protects against message forgeries (4-byte message integrity check, or MIC)

BLE uses four pairing methods:

1. Just works
2. Out of band
3. Passkey entry
4. Numeric comparison (only secure connections) from Bluetooth 4.2

Method to determine computation of security keys:

- Legacy encryption - short temporary key (STK). STK is created to encrypt a connection. Then, if bonding, LTK will be used for subsequent connections.
- Secure connections - Long term key (LTK). LTK is created to encrypt the connection.

7.6.1 Security modes and level

LE security Mode 1 (Link layer):

- No security - level 1
- Unauthenticated pairing with encryption - level 2
- Authenticated pairing with encryption - level 3
- Authenticated LE secure connections pairing with encryption BT 4.2 - level 4

Authenticated pairing: pairing is performed with man In the middle (MITM) protection

Unauthenticated pairing: pairing is performed without MITM protection

LE security Mode 2 (ATT layer): not supported

- unauthenticated pairing with data signing
- authenticated pairing with data signing

7.6.2 Security commands

During the device initialization phase, the security properties can be initialized with the following commands:

aci_gap_set_io_capability()

Sets the IO capabilities of the device. This command must be given only when the device is not in a connected state.

aci_gap_set_authentication_requirement()

Sets the authentication requirements for the device. This command must be given only when the device is not in a connected state.

This command defines bonding mode information, MITM mode, LE secure connection support values, keypress notification support values, encryption key size, use or not of fixed pin, its value, and identity address type.

- SC_Support parameter defines the LE Secure connections support values.
 - 0x00: Secure connections pairing not supported (legacy pairing mode)
 - 0x01: Secure connections pairing supported but optional
 - 0x02: Secure connections pairing supported and mandatory (SC only mode)

Once the connection is established, the security procedure can be started:

- By the master with **aci_gap_set_pairing_req()**
 - Sends the SM pairing request to start a pairing process. The authentication requirements and IO capabilities must be set before issuing this command.
 - The **force_rebond** parameter value determines if the pairing request is sent even if the device was previously bonded.
- By the slave with **aci_gap_slave_security_req()**
 - Sends a slave security request to the master. This command must be issued to notify the master of the security requirements of the slave. The master can encrypt the link, initiate the pairing procedure, or reject the request.
 - **aci_gap_pairing_complete_event** is returned after the pairing process is completed.

Depending on the SC_Support parameter value, the device answers to security requests with one of the commands listed in [Table 9](#).

Table 9. Security commands

Command	Description
<code>aci_gap_pass_key_resp()</code>	This command must be sent by the host in response to <code>aci_gap_pass_key_req_event</code> . The command parameter contains the pass key used during the pairing process if no fixed pin.
<code>aci_gap_numeric_comparison_value_confirm_yesno()</code>	This command allows the user to confirm or not the numerical comparison value shown through the <code>aci_gap_numeric_comparison_value_event</code> . When devices are bonded, the keys are stored in the non-volatile memory area. This means that if devices are previously bonded, and one of devices is unplugged, keys are not lost. When <code>aci_gap_set_pairing_req()</code> command is sent, with <code>force_rebond</code> parameter set to no force rebond, the pairing completes without any other exchange.
To clear the security database:	
<code>aci_gap_clear_security_db()</code>	All the devices in the security database are be removed.

7.6.3 Security information commands

Table 10. Security information commands

Command	Description
<code>aci_gap_get_bonded_devices()</code>	This command gets the list of the devices which are bonded. It returns the number of addresses and the corresponding address types and values.
<code>aci_gap_is_device_bonded()</code>	This command determines whether the device, whose address is specified in the command, is bonded. If the device uses a resolvable private address and has been bonded, then the command returns <code>ble_status_success</code> .
<code>aci_gap_get_security_level()</code>	This command can be used to get the current security settings of the device.
If keypress notification is supported, use:	
<code>aci_gap_passkey_input()</code>	This command permits to tell the stack the input type detected during passkey input.

Table 10. Security information commands (continued)

Command	Description
If OOB is supported, use:	
<code>aci_gap_set_oob_data()</code>	This command is sent by the user to input the OOB data arrived via OOB communication.
<code>aci_gap_get_oob_data()</code>	This command is sent by the user to get (extract from the stack) the OOB data generated by the stack itself.

7.7 Privacy feature

The BLE privacy feature reduces the ability to track a device over a period of time by changing the device address on a frequent basis.

The address of a device using the privacy mode can be resolved using the IRK (identity resolving key), which is one of the encryption keys exchanged during the pairing process.

Devices need first to be initialized with privacy disabled, then connected and paired.

Then, for privacy enabled on both devices, send on both sides:

```
Hci_reset()
```

```
aci_gap_init() - privacy enabled
```

```
aci_gap_add_devices_to_resolving_list()
```

This command is used to add one device to the list of address translations used to resolve Resolvable Private Addresses in the controller

From central side, send:

```
aci_gap_create_connection()
```

or

```
aci_gap_start_auto_connection_establish_proc()
```

or

```
aci_gap_start_general_connection_establish_proc() (then
aci_gap_create_connection) : own_address_type = resolvable private address,
peer_address_type = public or random
```

From peripheral side, send:

```
aci_gap_set_discoverable()
```

or

```
aci_gap_set_direct_connectable()
```

or

```
aci_gap_set_undirected_connectable()
own_address_type = resolvable private address
```

When the connection is established, the LE enhanced connection complete event is generated.

7.8 How to use the 2 Mbps feature

During the device initialization phase, the preferred TX_PHYS, RX_PHYS values can be initialized with the following command:

Table 11. 2 Mbps feature commands

Command	Description
HCI_LE_Set_default_Phy()	Specifies the preferred PHY to implement (for RX and TX), not linked to a connection. By default, the preferred PHY is 2M.
Once a connection is established (1M), the preferred PHY for TX and RX can be sent by each device:	
HCI_LE_Set_Phy()	Makes it possible for the host to specify the preferred values for a connection.
During a connection, the RX or TX PHY used can be read:	
HCI_LE_Read_Phy()	Reads the current PHY TX and RX for a connection.

When the command HCI_LE_Set_Phy() is used, event hci_le_phy_update_complete is received by the master.

7.9 How to update connection parameters

When a connection is established, it is possible to update connection parameters.

Table 12. Proprietary connection data

Command	Description
When the master device (central) is initiator of the update:	
aci_gap_start_connection_update()	Starts the connection update (only when role is master). On completion of the procedure, an HCI_LE_CONNECTION_UPDATE_COMPLETE_EVENT event is returned to the upper layer.
When the slave device (peripheral) is initiator of the update:	
aci_l2cap_connection_parameter_update_req()	Sends an L2CAP connection parameter update request from the slave to the master. An HCI_L2CAP_CONNECTION_UPDATE_RESP_EVENT event is raised when the master responds to the request (accepts or rejects it).

7.10 Event and error code description

When a stack API is called, get the API return status and to monitor and track any potential error conditions.

BLE_STATUS_SUCCESS (0x00) is returned when the API is successfully executed.

All commands (HCI - ACI) are acknowledged by the hci_command_status_event().

This command status event is used to indicate that the command described by the Command_Opcode parameter has been received, and that the BLE stack controller is currently performing the task for this command.

For any problem, the Status event parameters contains the corresponding error code (see [\[7\]](#), v5.0, Vol. 2, part D).

On the GATT client side, the GATT discovery procedure can fail due to several reasons. The aci_gatt_error_resp_event() is generated when an error response is received from the GATT server. This does not mean that the procedure ended with an error, but this error is part of the procedure itself.

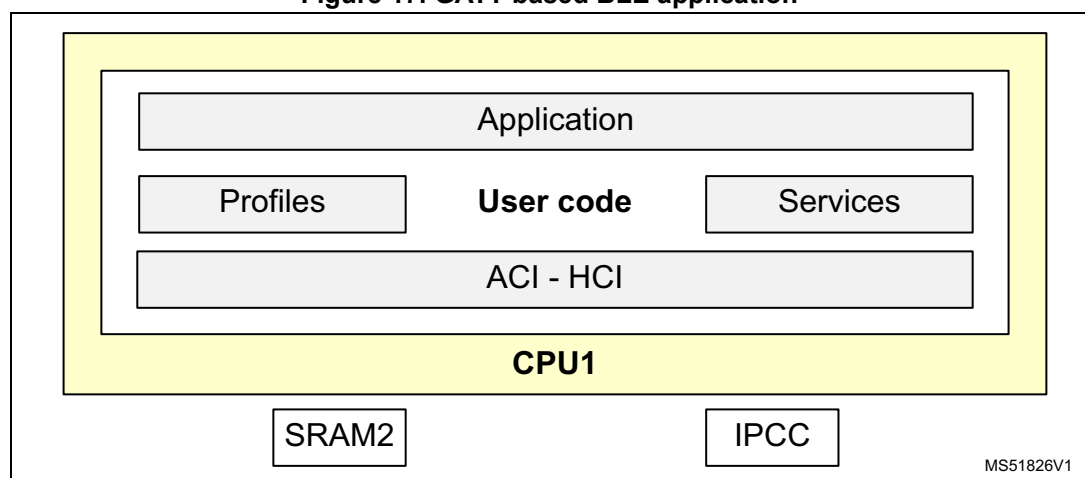
All GATT client procedures have to be completed with either a success or error on the aci_gatt_proc_complete_event(), so the GATT client can start a new procedure.

8 BT-SIG and proprietary GATT-based BLE application

This section describes the specification and implementation of the following applications running on CPU1:

- STMicroelectronics specific application
 - Transparent mode - Direct test mode
- BT-SIG GATT-based application
 - Heart rate sensor
- STMicroelectronics proprietary GATT-based application
 - P2P application (server / client)
 - FUOTA

Figure 17. GATT-based BLE application



8.1 Transparent mode - Direct test mode (DTM)

8.1.1 Purpose and scope

Among the set of HCI commands is a subset of commands used in order to enable direct test mode (DTM) as described in the Bluetooth Specification Core v5.0 Low Energy Controller Volume.

DTM is used to control the DUT and provides a report to the tester. According to the specification, the DTM must be set up using one of the two methods below:

1. Over HCI (the one implemented in STM32WB devices)
2. Through a 2-wire UART interface.

STM32WB supports the DTM as per Bluetooth Core Specification v5.0 [Vol. 6, Part F].

Here are the HCI test commands, which are fully compliant with the specifications:

- `HCI_LE_Transmitter_Test`
- `HCI_LE_Enhanced_Transmitter_Test`
- `HCI_LE_Receiver_Test`
- `HCI_LE_Enhanced_Receiver_Test`
- `HCI_LE_Test_End`

The number of the received test packets is a return value of the function `HCI_LE_Test_End`.

The additional available functions are listed in [Table 13](#).

Table 13. Direct test mode functions

Function	Description
<code>aci_hal_le_tx_test_packet_number</code>	This command provides the number of transmitted test packets during a DTM test.
<code>aci_hal_set_tx_power_level</code>	This command is used to set the TX output power. The ACI command set (see [3]) contains a complete description of the TX power level values.
<code>aci_hal_tone_start</code>	This command is used to generate a continuous waveform (CW) from the STM32WB radio.
<code>aci_hal_tone_stop</code>	This command is used to terminate the CW emission.

The command sequence below is a typical flow for enabling CW transmission from the STM32WB radio:

```
hci_reset
aci_hal_set_tx_power_level
aci_hal_tone_start
aci_hal_tone_stop
```

8.1.2 Transparent mode application principle

This firmware is used to:

- Receive commands on UART RX
- Transmit events on UART TX
- Communicate with the BLE stack via the IPCC.

No interpretation is done by CPU1 application firmware.

A set of commands/events must go through the STM32WB UART to control BLE stack via Transparent mode application.

Level shifter, VCP ST-LINK or applicative VCP can be used to manage the TX and RX.

8.1.3 Configuration

STM32WB is seen as a 2-wire UART interface (TXD, RXD). CPU1 application to be used is "Ble_TransparentMode". This firmware does not interpret the command and event but just

communicates to the wireless BLE stack via the IPCC and the selected UART interface (USART1 or LPUART1).

The UART interface and configuration selection is done using `app_conf.h`:

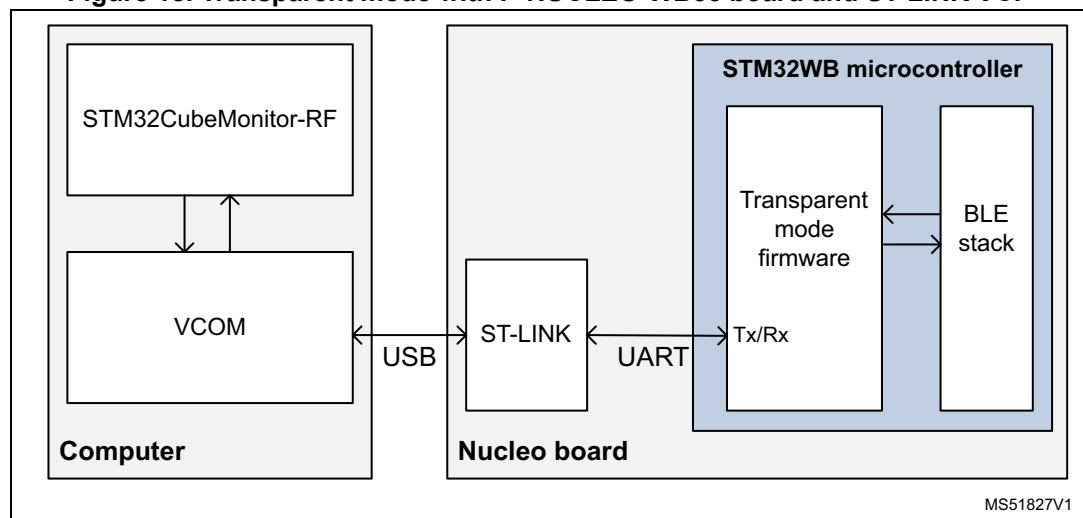
```
#define CFG_UART_GUI          hw_uart1
```

The P-NUCLEO-WB55 board includes the ST-LINK with Virtual COM port capability.

The following project is configured to communicate via the VCP of the ST-LINK:
`\Projects\ NUCLEO-WB55.Nucleo\Applications\BLE\Ble_TransparentMode`.

The UART1 (PB6, PB7) is connected to the P-NUCLEO-WB55 board ST-LINK VCP.

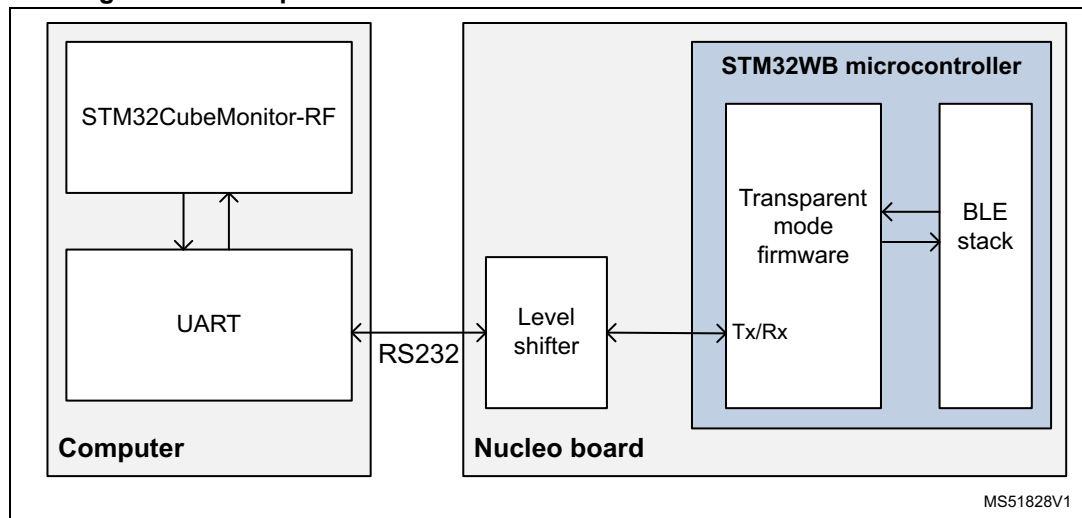
Figure 18. Transparent mode with P-NUCLEO-WB55 board and ST-LINK VCP



The P-NUCLEO-WB55 dongle board is provided without the ST-LINK. The project `\Projects\ NUCLEO-WB55.USB Dongle\Applications\BLE\BLE_TransparentModeVCP` includes Virtual COM port implementation in addition to the transparent mode feature.

It is also possible to connect the STM32WB UART interface directly to RS232 serial communication via a level shifter (not included on the NUCLEO-WB55RG board). This approach can be used to connect, among others, an RF tester.

Figure 19. Transparent mode with P-NUCLEO-WB55 board and level shifter

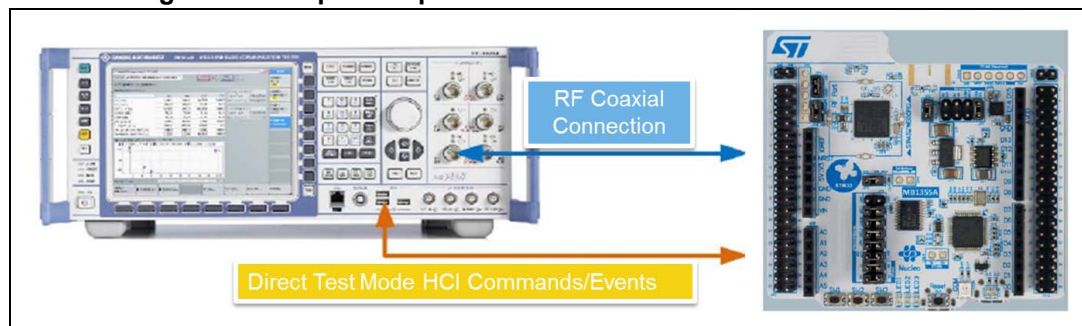


8.1.4 RF certification - Application implementation

The direct test mode (DTM) is specified by the Bluetooth SIG to provide a selection of different RF tests for BLE devices including remote control commands for the USB or RS232 interface.

BLE RF is placed in either continuous transmit or receive mode with or without modulation for RF evaluation. Figure 20 illustrates a simple setup.

Figure 20. Simple setup with BLE RF tester and P-NUCLEO board

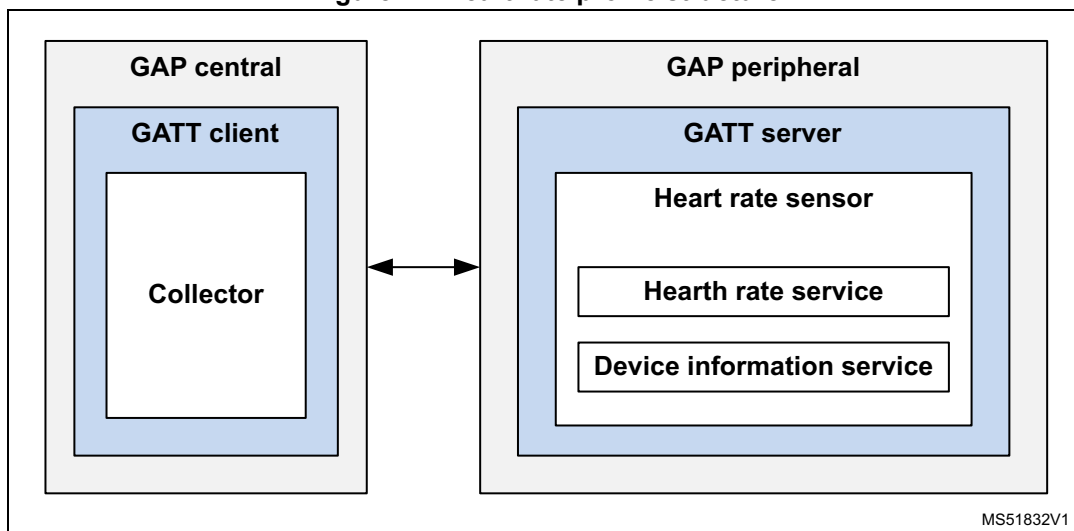


8.2 Heart rate sensor application

The heart rate profile is composed by two actions:

- Collector: GAP central and GATT client to receive heart rate measurement and other data
- Heart rate sensor: GAP peripheral and GATT server to provide heart rate measurement and other data.

Figure 21. Heart rate profile structure



The STM32WBCube_FW_WB_V1.0.0 release is provided with a heart rate sensor example.

This section describes the steps to create a Bluetooth SIG heart rate sensor application, aiming at transmitting heart rate from a sensor (for fitness application) each second and is composed of the following steps (illustrated in [Figure 22](#)):

- STM32WB user application initialization
- Heart rate service implementation - Middleware
- Heart rate sensor peripheral - User
- Heart rate sensor measurement update - User.

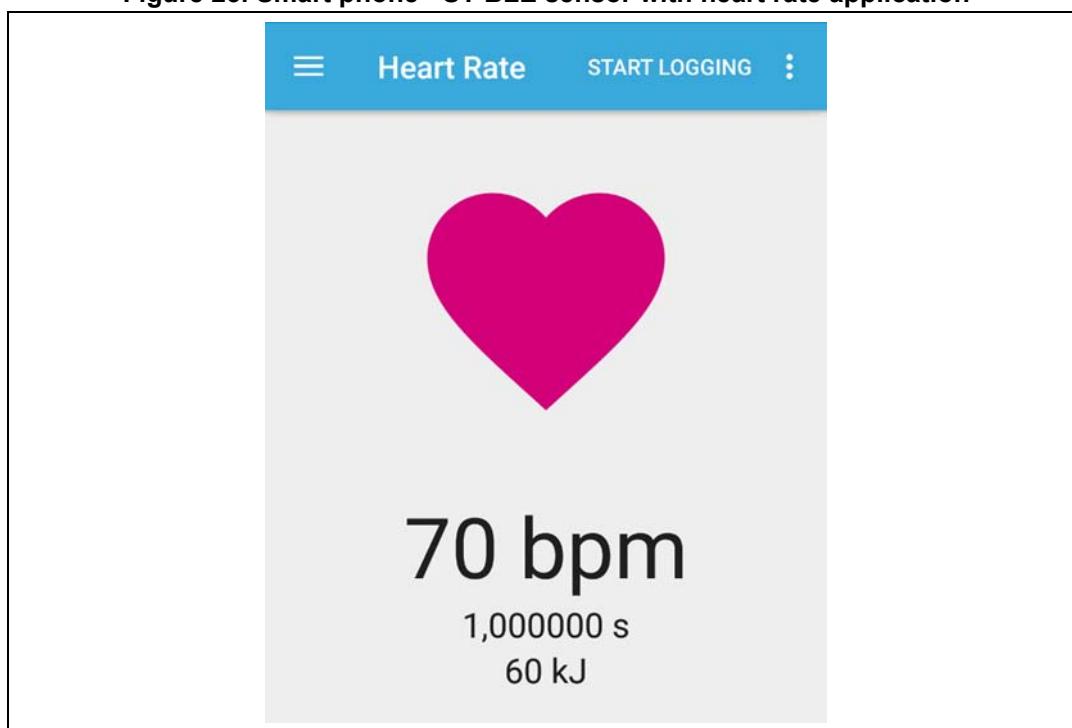
Figure 22. Simple setup with BLE RF tester and P-NUCLEO board



8.2.1 How to use STM32WB heart rate sensor application

- Open BLE_HeartRate project and follow readme.txt instructions
- Connect the ST BLE sensor mobile application to your heart rate application

Figure 23. Smart phone - ST BLE sensor with heart rate application



8.2.2 STM32WB heart rate sensor application - Middleware application

In Middlewares\STM32_WPAN\ble\core\Src\, the subfolder to insert BLE services is blesvc.

Warning: Do not modify the files in this folder.

- svc_ctl.c: Initializes the BLE stack and manages the services of the application (GATT events)
- hrs.c: used for the creation of:
 - A service and its characteristics for the application,
 - To update the service characteristics,
 - To receive the notification or write command and
 - to make a link between the BLE stack and the applicative.

For the application, the subfolder to create specific code is STM32_WPAN\app

- app_entry.c: Initializes the BLE Transport layer and the BSP (e.g. LEDs, buttons)
- app_ble.c: Initializes the GAP and manages the connection (e.g. advertising, scan)
- hrs_app.c: Initializes the GATT and manages the application

Heart rate service functionalities:
Middlewares\STM32_WPAN\ble\core\Src\blesvc\hrs.c

Table 14. Heart rate service functionalities

Function	Description
Service Init - HRS_Init()	<ul style="list-style-type: none"> – Registers the heart rate event handle to the service controller – Initializes the service UUID
aci_gatt_add_serv	<ul style="list-style-type: none"> – Adds the heart rate service as primary service – Initializes the heart rate measurement characteristic
aci_gatt_add_char	<ul style="list-style-type: none"> – Adds the heart rate characteristic – Initializes the body sensor location characteristic
aci_gatt_add_char	<ul style="list-style-type: none"> – Adds the body sensor location characteristic – Updates the heart rate measurement characteristic
aci_gatt_update_char_value	<ul style="list-style-type: none"> – Updates the requested characteristics within the specified value – Updates the body sensor location characteristic value
aci_gatt_update_char_value	<ul style="list-style-type: none"> – Updates the requested characteristics within the specified value
HeartRate_Event_Handler(void *Event)	<ul style="list-style-type: none"> – Manages the HCI Vendor Type Event
EVT_BLUE_GATT_WRITE_PERMIT_REQ	<ul style="list-style-type: none"> – Server receives a Write command – HR control point characteristic value – Resets energy expended command, then: Sends an aci_gatt_write_response() with an OK status. Notifies the HRS application to reset expended energy Or sends an aci_gatt_write_response() with an error.
EVT_BLUE_GATT_ATTRIBUTE_MODIFIED	<ul style="list-style-type: none"> – HR measurement characteristic description value – ENABLE or DISABLE notification – Notifies HRS application of the measurement notification

The purpose of the service implementation is to register the heart rate services and selected characteristics with the BLE stack GATT database.

```
/**
 * @brief Service Heart Rate initialization
 * @param None
 * @retval None
 */
void HRS_Init(void)
{
    REGISTER HEART RATE EVENT HANDLER
    ? SVCCTL_RegisterSvcHandler(HeartRate_Event_Handler);

    REGISTER HEART RATE SERVICE GATT DATABASE TO BLE STACK
    Add Heart Rate Service
    Add Heart Rate characteristics
    Measurement Value (mandatory)
```

```

Body sensor Location (Optional)
Heart rate control point(Optional)
Add Over The Air Reboot Request characteristic (Optional)
}

```

- Manage the GATT event dedicated to the HR service

```

/**
 * @brief Heart Rate Service Event handler
 * @param Event: Address of the buffer holding the Event
 * @retval Ack: Return whether the GATT Event has been managed or not
 */
static SVCCTL_EvtAckStatus_t HeartRate_Event_Handler(void *Event)
{
    MANAGE GATT EVENT FROM BLE STACK
    ? EVT_BLUE_GATT_WRITE_PERMIT_REQ
    ? EVT_BLUE_GATT_ATTRIBUTE_MODIFIED

    NOTIFY USER APPLICATION - HRS_Notification
    ? HRS_RESET_ENERGY_EXPENDED_EVT
    ? HRS_NOTIFICATION_ENABLED
    ? HRS_NOTIFICATION_DISABLED
    ? HRS_STM_BOOT_REQUEST_EVT
}

```

- Allow the application to update the characteristics to BLE stack GATT database

```

/**
 * @brief Characteristic update
 * @param UUID: UUID of the characteristic
 * @retval BodySensorLocationValue: The new value to be written
 */
tBleStatus HRS_UpdateChar(uint16_t UUID, uint8_t *pPayload)
{
    UPDATE BODY SENSOR LOCATION

    UPDATE HEART RATE MEASUREMENT VALUE
}

```

Service controller functionalities:**Middlewares\STM32_WPAN\ble\core\Src\blesvc\svc_ctl.c**

The SVCCTL_Init() has different function:

- Calls the initialization function of all the developed services
 - HR server - HRS_Init()
- Registers the service event handler
 - SVCCTL_RegisterSvcHandler()
 - Function receiving the GATT events from the svc_ctl.c and redirecting them to the application (hrs_app.c)
- Registers client event handler (not applicable to HR sensor project)
 - SVCCTL_RegisterClhHandler()

HR sensor application initialization:**Applications\BLE\BLE_HeartRate\STM32_WPAN\App\app_ble.c**

HR sensor peripheral initialization - APP_BLE_Init()

- Initializes the BLE stack on CPU2
 - SHCI_C2_BLE_Init()
- Initializes the HCI, GATT and GAP layers
 - Ble_Hci_Gap_Gatt_Init()
- Initializes the BLE Services
 - SVCCTL_Init()
- Calls heart rate server and device information application initialization
 - HRSAPP_Init()
 - DISAPP_Init()
- Configures and starts advertising: ADV parameters, Local name, UUID, ...
 - aci_gap_set_discoverable() - Sets the device in general discoverable mode
 - aci_gap_update_adv_data() - Adds information in advertising data packet
- Manages GAP Event - SVCCTL_App_Notification()
 - EVT_LE_CONN_COMPLETE

Provides the connection interval information, slave latency, Supervision timeout

- Provides the new information of the connection
 - EVT_LE_CONN_UPDATE_COMPLETE
- Informs the application about the link disconnection and the reason
 - EVT_DISCONN_COMPLETE
- Informs the application whether the link is encrypted
 - EVT_ENCRYPT_CHANGE

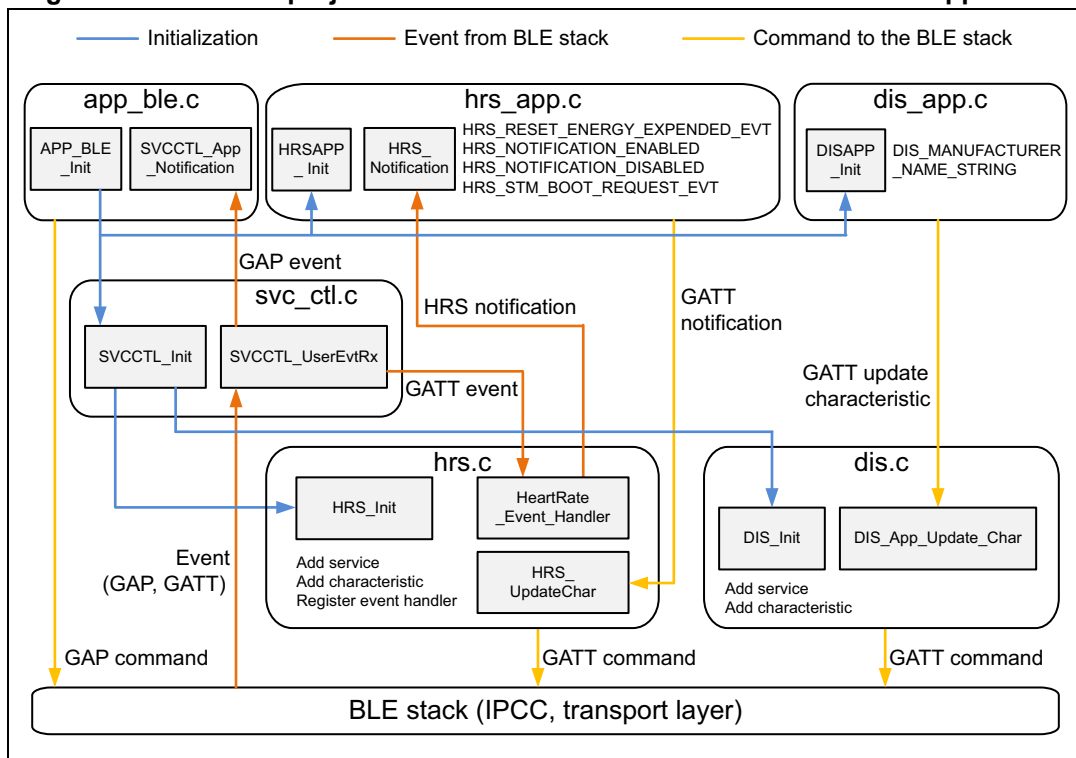
HR sensor application control:**Applications\BLE\BLE_HeartRate\STM32_WPAN\App\hrs_app.c**

The hrs_app.c file initializes the sensor application, creates timers

Table 15. HR sensor application control

Function	Description
HRSAPP_Init()	Receives and reacts to the internal events coming from the BLE stack at GATT level.
HRS_Notification()	Calls the service functions to update the characteristics (notify/write).
HRSAPP_Measurement()	-

Figure 24. Heart rate project - Interaction between middleware and user application



8.3 STMicroelectronics proprietary advertising

When the device is a peripheral, it advertises information such as Bluetooth address and advertising payload (0 to 31 bytes long).

The advertising information is represented by advertising data elements, standardized on the Bluetooth SIG:

- First byte: length of the element (excluding the length byte itself)
- Second byte: AD type - specifies what data is included in the element
- AD data: one or more bytes, the meaning of which is defined by AD type.

The AD type "0xFF" is used to provide manufacturer specific data.

The implementation of STMicroelectronics proprietary GATT-based applications such as P2P and FUOTA applications is proposed with the manufacturer specific AD type data. It is a way for the remote device (scanner) to filter the peripheral devices and access the requested application.

Table 16. AD structure according to the Bluetooth 5 Core specification Vol. 3 part C

Field name	Type	Len	Record size
TX_POWER_LEVEL	0x0A	2	3
COMPLETE_NAME	0x09	8	9
MANUF_SPECIFIC	0xFF	13	14
FLAGS	0x01	2	3

Table 17. STM32WB manufacturer specific data

Octet	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Name	Len	Type	Ver	DevID	Group A features		Group B features		Public device address (48 bits), optional					
Value	0x0D	0xFF	0x01	0xFF	RFU		0xFFFF		0XXXXXXXXXXXX					

Group B features

- Bit mask Thread: used to advertise the presence of the Thread switch characteristics.
- Bit mask OTA reboot request: used to advertise presence of the BLE reboot characteristics.

Table 18. Group B features - Bit mask

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	Thread support	OTA reboot request	RFU												

Table 19. Device ID Enum

ID	HW
0x00	Generic
0x83	STM32WB P2P server 1
0x84	STM32WB P2P server 2
0x85	STM32WB P2P router
0x86	STM32WB FUOTA

The advertising procedure is managed at user application level in the app_ble.c.

Here is an example for BLE_p2p Server project advertising startup.

```
/* Local name to be advertised */
static const char local_name[] = { AD_TYPE_COMPLETE_LOCAL_NAME, 'P', '2',
    'P', 'S', 'R', 'V', '1' };

/* manufacturer data & legacy data to be advertised */
uint8_t manuf_data[14] = {
    sizeof(manuf_data)-1, AD_TYPE_MANUFACTURER_SPECIFIC_DATA,
```

```

    0x01/*SKD version */ ,
    CFG_DEV_ID_P2P_SERVER1 /* STM32WB - P2P Server 1*/,
    0x00 /* GROUP A Feature */ ,
    0x00 /* GROUP A Feature */ ,
    0x00 /* GROUP B Feature */ ,
    0x00 /* GROUP B Feature */ ,
    0x00, /* BLE MAC start -MSB */
    0x00,
    0x00,
    0x00,
    0x00,
    0x00, /* BLE MAC stop */
};

/* Local device BD address*/
const uint8_t *bd_addr;
bd_addr = SVCCTL_GetBdAddress();

/* BLE MAC update for Advertising manufacturer data*/
manuf_data[ sizeof(manuf_data)-6] = bd_addr[5];
manuf_data[ sizeof(manuf_data)-5] = bd_addr[4];
manuf_data[ sizeof(manuf_data)-4] = bd_addr[3];
manuf_data[ sizeof(manuf_data)-3] = bd_addr[2];
manuf_data[ sizeof(manuf_data)-2] = bd_addr[1];
manuf_data[ sizeof(manuf_data)-1] = bd_addr[0];

/* Put the GAP peripheral in general discoverable mode:
Advertising_Type: ADV_IND(undirected scannable and connectable);
Advertising_Interval_Min;
Advertising_Interval_Max;
Own_Address_Type: PUBLIC_ADDR (public address: 0x00);
Adv_Filter_Policy: NO_WHITE_LIST_USE (no whit list is used);
Local_Name_Length
Local_Name:
Service_Uuid_Length: 0 (no service to be advertised);
Service_Uuid_List: NULL;
Slave_Conn_Interval_Min: 0 (Slave connection internal minimum value);
Slave_Conn_Interval_Max: 0 (Slave connection internal maximum value).
*/
result = aci_gap_set_discoverable(ADV_IND,
CFG_FAST_CONN_ADV_INTERVAL_MIN,
CFG_FAST_CONN_ADV_INTERVAL_MAX,
PUBLIC_ADDR,
NO_WHITE_LIST_USE, /* use white list */
sizeof(local_name), (uint8_t*) local_name,

```



```

0,
NULL,
0, 0);

/* Update Advertising data with manufacturer specific information*/
result = aci_gap_update_adv_data(sizeof(manuf_data), (uint8_t*)
manuf_data);

```

The result is always compared to BLE_STATUS_SUCCESS (0x00).

8.4 Proprietary P2P application

Three components can be used to demonstrate the different data communication types:

1. P2P server project
2. P2P client project
3. Smart phone application.

The combination of the different components results in the demonstrations shown in figures [Figure 25](#) and [Figure 26](#).

Figure 25. P2P server to client demonstration

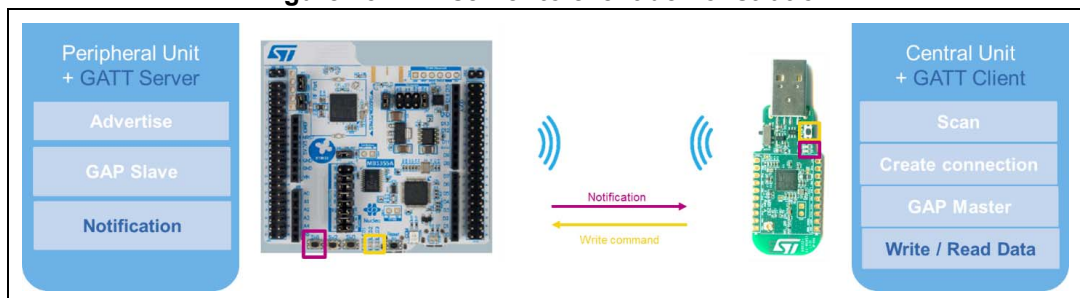
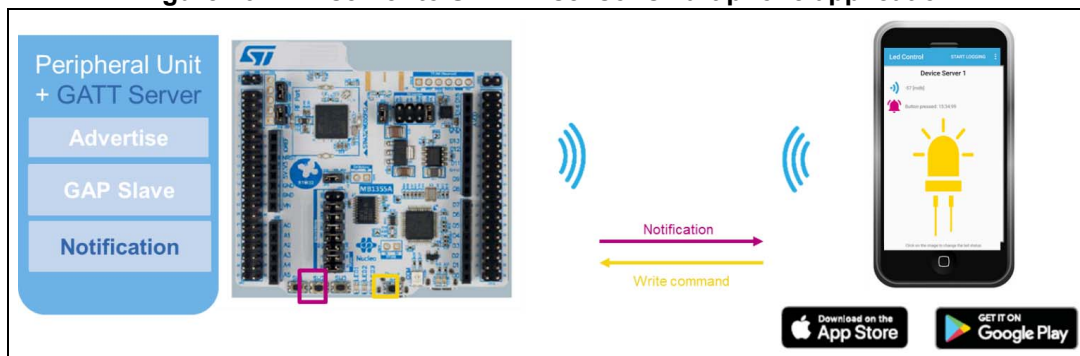


Figure 26. P2P server to ST BLE sensor smart phone application



8.4.1 P2P server specification

The P2P server application must be used to demonstrate point to point communication. It acts as a peripheral device with the following GATT service and characteristics.

Table 20. P2P service and characteristic UUIDs

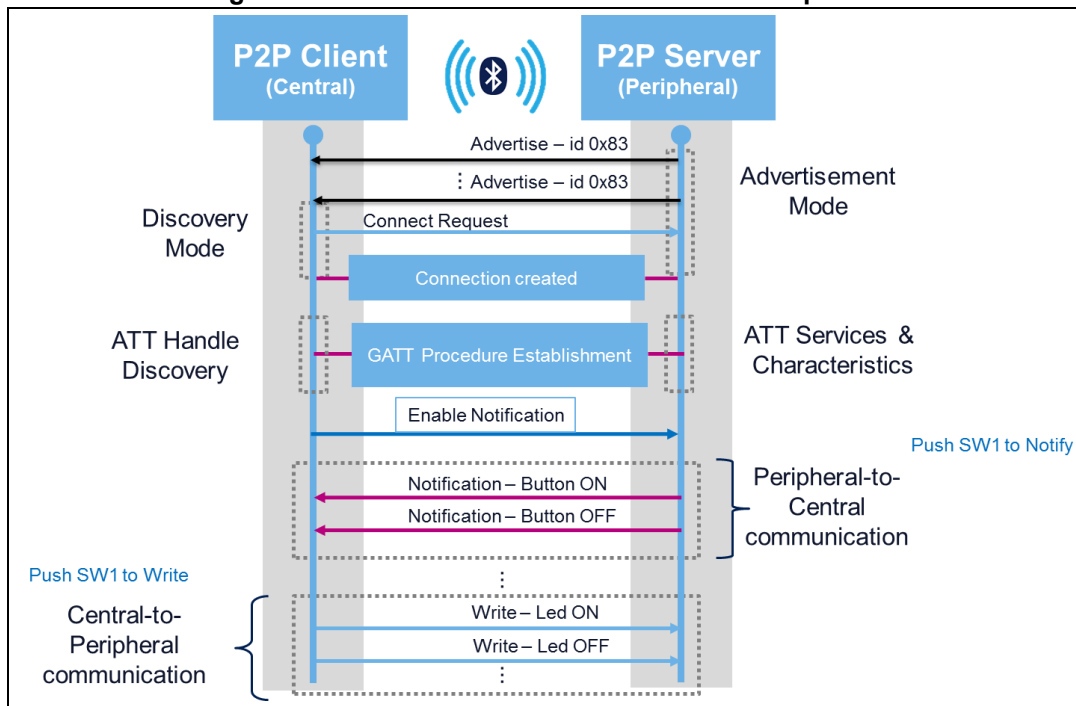
Groups	Service	Characteristic	Size	Mode	UUID
LED button control	P2P service	-	-	-	0000FE40-cc7a-482a-984a-7fed5b3e58f
	-	Write	2	Read / Write	0000FE41-8e22-4541-9d4c-21edae82ed19
	-	Notify	2	Notify	0000FE42-8e22-4541-9d4c-21edae82ed19

Table 21. P2P specification

Write	Octets LSB	0	1
	Name	Device selection	LED control
	Value	– 0x01: P2P server 1 – 0x02: P2P server 2 – 0x0x: P2P server x – 0x00: All	– 0x00 LED off – 0x01 LED on – 0x02 Thread switch
Notify	Octets LSB	0	1
	Name	Device selection	Button
	Value	– 0x01: P2P server 1 – 0x02: P2P server 2 – 0x0x: P2P server x	– 0x00 switch off – 0x01 switch on

To be used, a GAP central and GATT client device must discover and connect to the P2P server application. [Figure 27](#) explains the data exchange procedure.

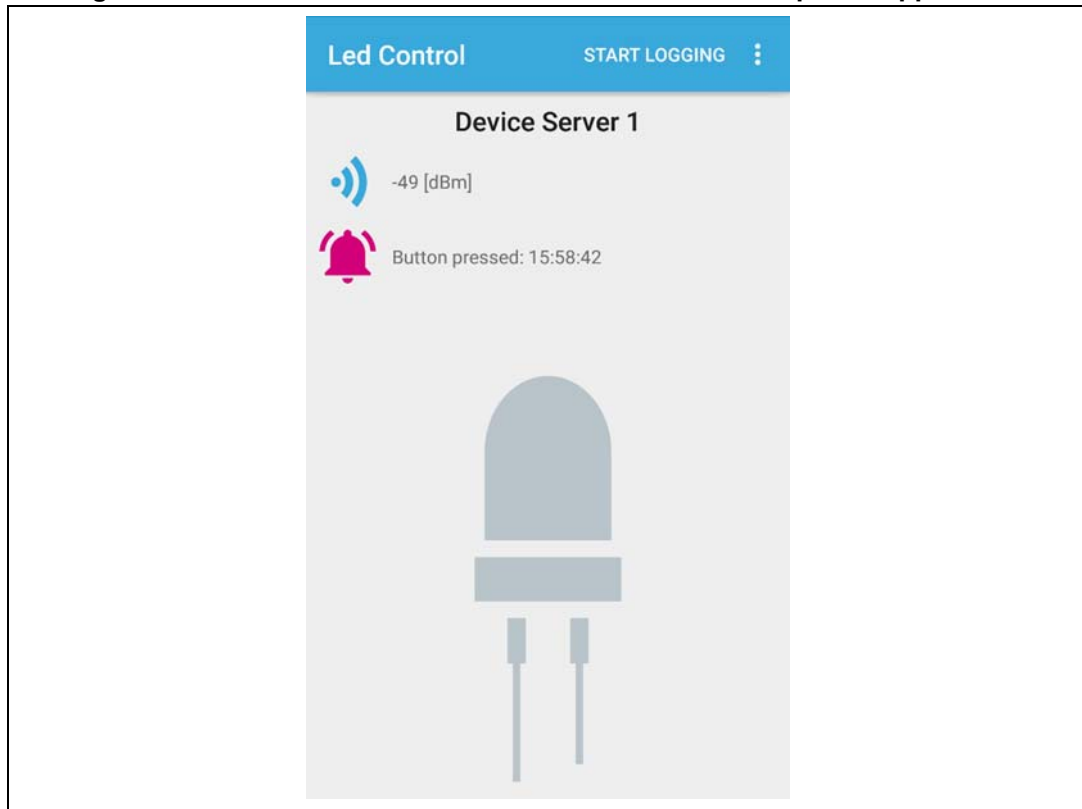
Figure 27. P2P server/client communication sequence



8.4.2 How to use the P2P server application

1. Flash your P-NUCLEO-WB55 board with the ble_P2P_Server project
2. Once flashed, connect the ST BLE sensor mobile application to your board, and use SW1 button to notify the smart phone.

Figure 28. P2P server connected to ST BLE sensor smart phone application



8.4.3 P2P server application - Middleware application

The P2P service and characteristics are created using the p2p_stm.c file.

p2p_stm.c: creates the service and the characteristics in the application to update the characteristics, to receive the notification or write command, and to make a link between the BLE wireless stack and the applicative part.

In the application, the subfolder to create specific code is "User"

- app_entry.c: initializes the BLE Transport layer and the BSP (e.g. LEDs, buttons)
- app_ble.c: initializes the GAP and manage the connection (e.g. advertising, scan)
- p2p_server_app.c: initializes the GATT and manage the application.

P2P service functionalities:

Middlewares\STM32_WPAN\ble\core\Src\blesvc\p2p_stm.c

Table 22. P2P service functionalities

Function	Description
Service Init - P2PS_STM_Init ()	<ul style="list-style-type: none"> – Register PeerToPeer_Event_Handler to Service controller – Initializes Service UUID aci_gatt_add_serv - adds P2P service as Primary services – Initializes P2P write characteristic aci_gatt_add_char - adds Write characteristic – Initializes P2P Notify characteristic aci_gatt_add_char - adds Notify characteristic – Updates notification characteristic - P2PS_STM_App_Update_Char() aci_gatt_update_char_value - updates the notify characteristics with a value aligned with the specification
PeerToPeer_Event_Handler (void *Event) - manages HCI Vendor Type Event:	<p>EVT_BLUE_GATT_ATTRIBUTE_MODIFIED</p> <ul style="list-style-type: none"> – Receives the configuration of the Descriptor value of the Notify characteristics – ENABLE or DISABLE notification – Informs the application, P2PS_STM__NOTIFY_ENABLED_EVT or P2PS_STM_NOTIFY_DISABLED_EVT – Receives data on the write characteristic and Informs P2P application P2PS_STM_WRITE_EVT – Receives data on the Reboot request characteristic and informs P2P application (to be used for FUOTA procedure) <p>P2PS_STM_BOOT_REQUEST_EVT</p>

P2P server application control:

Applications\BLE\BLE_p2pServer\STM32_WPAN\App\p2p_server_app.c

The p2p_server_app.c file

Initializes the P2P server application, Create Timers

P2PS_APP_Init()

Receives and reacts to the internal events coming from the BLE stack at GATT level.

P2PS_STM_App_Notification ()

```
void P2PS_STM_App_Notification(P2PS_STM_App_Notification_evt_t
*pNotification)
{
Switch (pNotification->P2P_Evt_Opcode)
{
case P2PS_STM__NOTIFY_ENABLED_EVT:
P2P_Server_App_Context.Notification_Status = 1;
APP_DBG_MSG("-- P2P APPLICATION SERVER : NOTIFICATION ENABLED\n");
APP_DBG_MSG(" \n\r");
break;
```

```

case P2PS_STM_NOTIFY_DISABLED_EVT:
    P2P_Server_App_Context.Notification_Status = 0;
. APP_DBG_MSG("-- P2P APPLICATION SERVER : NOTIFICATION DISABLED\n");
    APP_DBG_MSG(" \n\r");
    break;

case P2PS_STM_WRITE_EVT:
    if(pNotification->DataTransferred.pPayload[0] == 0x00){
.    if(pNotification->DataTransferred.pPayload[1] == 0x01)
        {
            BSP_LED_On(LED_BLUE);
            APP_DBG_MSG("-- P2P APPLICATION SERVER : LED1 ON\n");
            APP_DBG_MSG(" \n\r");
            P2P_Server_App_Context.LedControl.Led1=0x01;
        }
        if(pNotification->DataTransferred.pPayload[1] == 0x00)
        {
            BSP_LED_Off(LED_BLUE);
            APP_DBG_MSG("-- P2P APPLICATION SERVER : LED1 OFF\n");
            APP_DBG_MSG(" \n\r");
            P2P_Server_App_Context.LedControl.Led1=0x00;
        }
    }
}

```

Calls the service function to update the characteristic (notify).

```

P2PS_Send_Notification ()
void P2PS_Send_Notification(void)
{
    if(P2P_Server_App_Context.ButtonControl.ButtonStatus == 0x00){
        P2P_Server_App_Context.ButtonControl.ButtonStatus=0x01;
    } else {
        P2P_Server_App_Context.ButtonControl.ButtonStatus=0x00;
    }

    if(P2P_Server_App_Context.Notification_Status){
        APP_DBG_MSG("P2P APPLICATION SERVER : INFORM CLIENT BUTTON 1 PUSHED \n
");
        APP_DBG_MSG(" \n\r");
        P2PS_STM_App_Update_Char(P2P_NOTIFY_CHAR_UUID, (uint8_t *)
&P2P_Server_App_Context.ButtonControl);
    } else {
        APP_DBG_MSG("P2P APPLICATION SERVER : CAN'T INFORM CLIENT - NOTIFICATION
DISABLED\n ");
    }
}

```

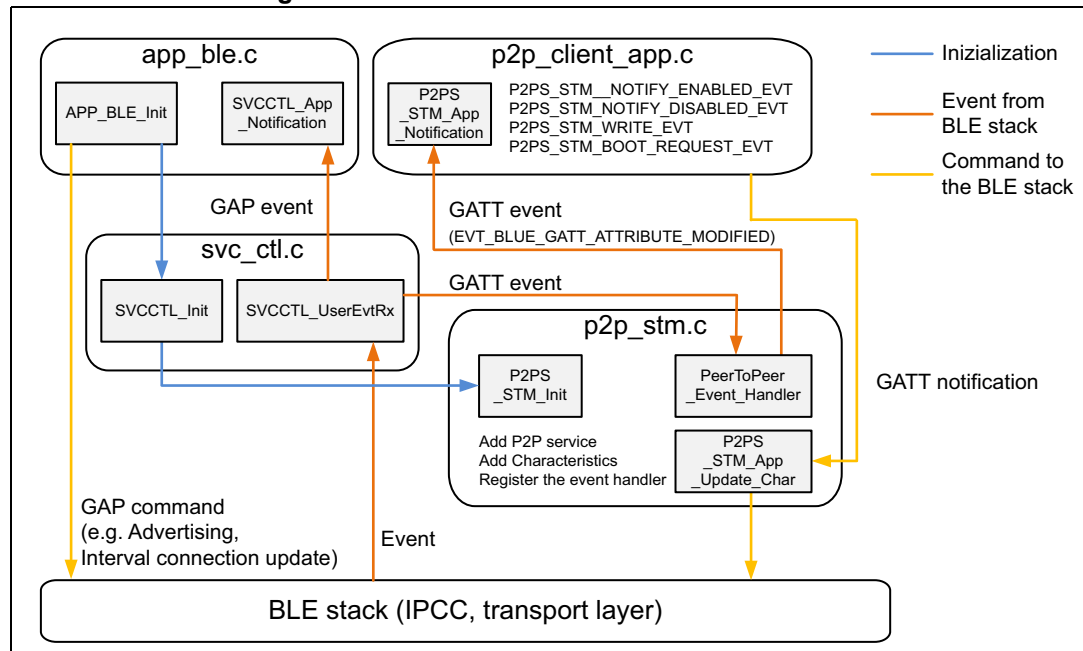
```

return;
}

```

8.4.4 P2P client application - Middleware application

Figure 29. P2P server software communication



There is no service created for the P2P client. It is only necessary to register the GATT client Handler `SVCCTL_RegisterClhHandler()` to be notified of any GATT events at application level.

In the application, the subfolder to create specific code is "User"

- `app_entry.c`: initializes the BLE transport layer and the BSP (e.g. LEDs, buttons)
- `app_ble.c`: initializes the GAP and manages the connection (scan and connect)
- `p2p_client_app.c`: initializes the GATT and manages the GATT client application.

P2P client - Scan and connect

The `app_ble.c` file

- Executes a scan to search for any P2P server advertising IDs:

```

static void Scan_Request( void )
{
    tBleStatus result;
    if (BleApplicationContext.Device_Connection_Status !=
        APP_BLE_CONNECTED_CLIENT)
    {
        BSP_LED_On(LED_BLUE);
        result = aci_gap_start_general_discovery_proc(SCAN_P, SCAN_L,
            PUBLIC_ADDR, 1);
    }
}

```

```

    if (result == BLE_STATUS_SUCCESS)
    {
        APP_DBG_MSG("\r\n\r** START GENERAL DISCOVERY (SCAN) ** \r\n\r");
    } else {
        APP_DBG_MSG("-- BLE_App_Start_Limited_Disc_Req, Failed \r\n\r");
        BSP_LED_On(LED_RED);
    }
}
return;
}

```

- Receives the ADV events report to be filtered to save the P2P server BD address:J'rric

```

case AD_TYPE_MANUFACTURER_SPECIFIC_DATA: // Manufactureur Specific
    if (adlength >= 7 && le_advertising_event->Advertising_Report[0].Data[k +
2] == 0x01) {

```

```

        APP_DBG_MSG("--- ST MANUFACTURER ID --- \n");
        switch (le_advertising_event->Advertising_Report[0].Data[k + 3]) {
            case CFG_DEV_ID_P2P_SERVER1:
                APP_DBG_MSG("-- SERVER DETECTED -- VIA MAN ID\n");
                BleApplicationContext.DeviceServerFound = 0x01;
                SERVER_REMOTE_BDADDR[0] = le_advertising_event-
>Advertising_Report[0].Address[0];
                SERVER_REMOTE_BDADDR[1] = le_advertising_event-
>Advertising_Report[0].Address[1];
                SERVER_REMOTE_BDADDR[2] = le_advertising_event-
>Advertising_Report[0].Address[2];
                SERVER_REMOTE_BDADDR[3] = le_advertising_event-
>Advertising_Report[0].Address[3];
                SERVER_REMOTE_BDADDR[4] = le_advertising_event-
>Advertising_Report[0].Address[4];
                SERVER_REMOTE_BDADDR[5] = le_advertising_event-
>Advertising_Report[0].Address[5];
                break;

```

- Initiates the connection to any detected P2P servers:

```

static void Connect_Request( void )
{
    tBleStatus result;
    APP_DBG_MSG("\r\n\r** CREATE CONNECTION TO SERVER ** \r\n\r");
    if (BleApplicationContext.Device_Connection_Status !=
APP_BLE_CONNECTED_CLIENT) {
        result = aci_gap_create_connection(
SCAN_P,
SCAN_L,
PUBLIC_ADDR, SERVER_REMOTE_BDADDR,
PUBLIC_ADDR,
CONN_P1,

```

```

CONN_P2,
0,
SUPERV_TIMEOUT,
CONN_L1,
CONN_L2);
• Starts "Services Discovery Procedure" once connection is established:
case EVT_LE_CONN_COMPLETE:
/**
 * The connection is established
 */
connection_complete_event = (hci_le_connection_complete_event_rp0 *)
meta_evt->data;
BleApplicationContext.BleApplicationContext_legacy.connectionHandle =
connection_complete_event->Connection_Handle;
BleApplicationContext.Device_Connection_Status = APP_BLE_CONNECTED_CLIENT;

APP_DBG_MSG("\r\n\r** CONNECTION EVENT WITH SERVER \n");
handleNotification.P2P_Evt_Opcode = PEER_CONN_HANDLE_EVT;
handleNotification.ConnectionHandle =
    BleApplicationContext.BleApplicationContext_legacy.connectionHandle;
P2PC_APP_Notification(&handleNotification);
result = aci_gatt_disc_all_primary_services(
BleApplicationContext.BleApplicationContext_legacy.connectionHandle);
if (result == BLE_STATUS_SUCCESS) {
    APP_DBG_MSG("\r\n\r** GATT SERVICES & CHARACTERISTICS DISCOVERY \n");
    APP_DBG_MSG("* GATT : Start Searching Primary Services \r\n\r");
}

```

At this step, all the GATT events is transferred to the GATT client events handler managed in the p2p_client_app.c.

P2P client - Application control - GATT client communication

The p2p_client_app.c file:

- Initializes the P2P client application and registers the Client Event Handler
 - P2PC_APP_Init()
 - SVCCTL_RegisterClhHandler()
- Starts discovery procedures and manages the remote P2P server characteristics
 - aci_gatt_disc_all_char_of_service()
 - aci_gatt_disc_all_char_desc()
 - aci_gatt_write_char_desc()

```

case APP_BLE_DISCOVER_SERVICES:
APP_DBG_MSG("P2P_DISCOVER_SERVICES\n");
break;
case APP_BLE_DISCOVER_CHARACS:

```



```

APP_DBG_MSG("** GATT : Discover P2P Characteristics\n");
aci_gatt_disc_all_char_of_service(ap2PClientContext[index].connHandle,
ap2PClientContext[index].P2PServiceHandle,
ap2PClientContext[index].P2PServiceEndHandle);
break;
case APP_BLE_DISCOVER_WRITE_DESC:
APP_DBG_MSG("** GATT : Discover Descriptor of TX - Write Characteritic\n");
aci_gatt_disc_all_char_desc(ap2PClientContext[index].connHandle,
ap2PClientContext[index].P2PWriteToServerCharHdle,
ap2PClientContext[index].P2PWriteToServerCharHdle+2);
break;
case APP_BLE_DISCOVER_NOTIFICATION_CHAR_DESC:
APP_DBG_MSG("** GATT : Discover Descriptor of Rx - Notification
Characteritic\n");
aci_gatt_disc_all_char_desc(ap2PClientContext[index].connHandle,
ap2PClientContext[index].P2PNotificationCharHdle,
ap2PClientContext[index].P2PNotificationCharHdle+2);
break;
case APP_BLE_ENABLE_NOTIFICATION_DESC:
APP_DBG_MSG("** GATT : Enable Server Notification\n");
aci_gatt_write_char_desc(ap2PClientContext[index].connHandle,
ap2PClientContext[index].P2PNotificationDescHandle,
2,
(uint8_t *)&enable);
ap2PClientContext[index].state = APP_BLE_CONNECTED_CLIENT;
break;
case APP_BLE_DISABLE_NOTIFICATION_DESC :
APP_DBG_MSG("** GATT : Disable Server Notification\n");
aci_gatt_write_char_desc(ap2PClientContext[index].connHandle,
ap2PClientContext[index].P2PNotificationDescHandle,
2,
(uint8_t *)&enable);
ap2PClientContext[index].state = APP_BLE_CONNECTED_CLIENT;
break;
•   Manages GATT Events to find and to register the remote device characteristics
    handles
    -   SVCCTL_EvtAckStatus_t Event_Handler()
uuid = UNPACK_2_BYTE_PARAMETER(&pr->Attribute_Data_List[idx]);
if(uuid == P2P_SERVICE_UUID){
APP_DBG_MSG("-- GATT : P2P_SERVICE_UUID FOUND - connection handle 0x%x \n",
ap2PClientContext[index].connHandle);
ap2PClientContext[index].P2PServiceHandle = UNPACK_2_BYTE_PARAMETER(&pr-
>Attribute_Data_List[idx-16]);
ap2PClientContext[index].P2PServiceEndHandle = UNPACK_2_BYTE_PARAMETER
(&pr->Attribute_Data_List[idx-14]);
ap2PClientContext[index].state = APP_BLE_DISCOVER_CHARACS ;

```

```

}
uuid = UNPACK_2_BYTE_PARAMETER(&pr->Handle_Value_Pair_Data[idx]);
/* store the characteristic handle not the attribute handle */
handle = UNPACK_2_BYTE_PARAMETER(&pr->Handle_Value_Pair_Data[idx-14]);
if(uuid == P2P_WRITE_CHAR_UUID){
APP_DBG_MSG("-- GATT : WRITE_UUID FOUND - connection handle 0x%x\n",
ap2PClientContext[index].connHandle);
ap2PClientContext[index].state = APP_BLE_DISCOVER_WRITE_DESC;
ap2PClientContext[index].P2PWriteToServerCharHdle = handle;
}
else if(uuid == P2P_NOTIFY_CHAR_UUID){
APP_DBG_MSG("-- GATT : NOTIFICATION_CHAR_UUID FOUND - connection handle
0x%x\n", ap2PClientContext[index].connHandle);
ap2PClientContext[index].state = APP_BLE_DISCOVER_NOTIFICATION_CHAR_DESC;
ap2PClientContext[index].P2PNotificationCharHdle = handle;
}

```

Once the P2P server services and characteristics handle are discovered, the application is able to:

- Control the remote device using the “Write” characteristic

```

tBleStatus Write_Char(uint16_t UUID, uint8_t Service_Instance, uint8_t
*pPayload){
tBleStatus ret = BLE_STATUS_INVALID_PARAMS;
uint8_t index;
index = 0;
while((index < BLE_CFG_CLT_MAX_NBR_CB) && (ap2PClientContext[index].state
!= APP_BLE_IDLE)){
switch(UUID){
case P2P_WRITE_CHAR_UUID:
ret =aci_gatt_write_without_resp(ap2PClientContext[index].connHandle,
ap2PClientContext[index].P2PWriteToServerCharHdle,
2, /* charValueLen */
(uint8_t *) pPayload);
break;

```

- Receive notifications via the “Notify” characteristics

```

void Gatt_Notification(P2P_Client_App_Notification_evt_t *pNotification){
switch(pNotification->P2P_Client_Evt_Opcode){
case P2P_NOTIFICATION_INFO_RECEIVED_EVT: {
P2P_Client_App_Context.LedControl.Device_Led_Selection=pNotification-
>DataTransferred.pPayload[0];
switch(P2P_Client_App_Context.LedControl.Device_Led_Selection) {
case 0x01 : {
P2P_Client_App_Context.LedControl.Led1=pNotification-
>DataTransferred.pPayload[1];
if(P2P_Client_App_Context.LedControl.Led1==0x00){
BSP_LED_Off(LED_BLUE);
APP_DBG_MSG(" -- P2P CLIENT : NOTIFICATION RECEIVED - LED OFF \n\r");

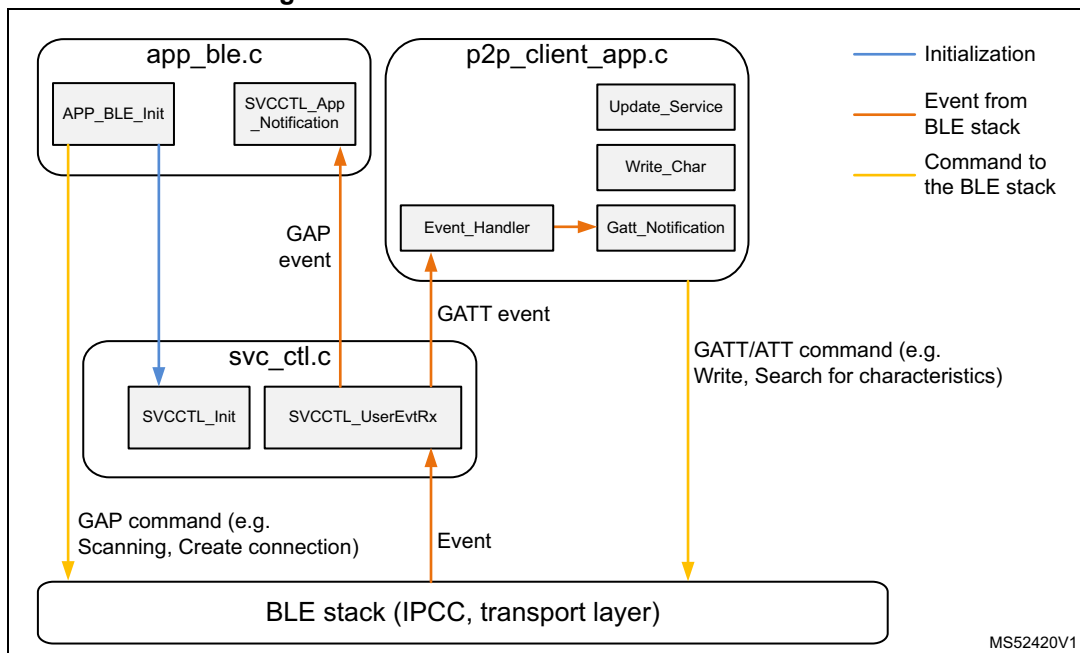
```

```

} else {
    BSP_LED_On(LED_BLUE);
    APP_DBG_MSG(" -- P2P CLIENT : NOTIFICATION RECEIVED - LED ON\n\r");
}
break;
}

```

Figure 30. P2P client software communication



8.5 FUOTA application

FUOTA is a standalone application able to install a BLE service to download the new CPU2 wireless stack, CPU1 application or configuration binaries:

- It requires that the first six flash memory sectors of the application (where the FUOTA application is written) are never deleted.
- The FUOTA application enables:
 - The update of the whole CPU1 application
 - To download CPU2 wireless firmware to be applied by the FUS
 - To download user data at any address in CPU1 user flash memory.

8.5.1 CPU1 user flash memory mapping

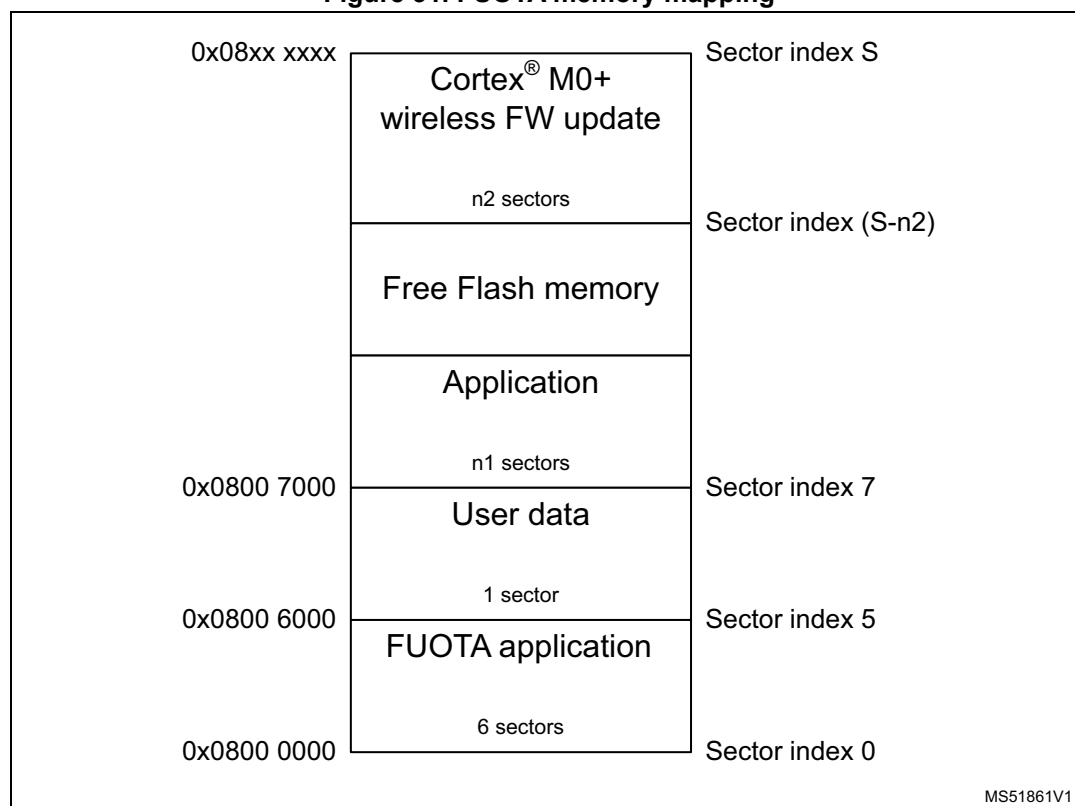
The FUOTA BLE application cannot update itself, but is able to:

- Jump on an existing application (Sector index 7)
- Run and install the STMicroelectronics proprietary FUOTA GATT service and characteristics to upload any data in specified area of a remote device.

The User data section can be used to update parts of the configuration for the application.

The application area contains the application standalone binary. It can be fully updated with FUOTA application.

Figure 31. FUOTA memory mapping



8.5.2 BLE FUOTA application startup

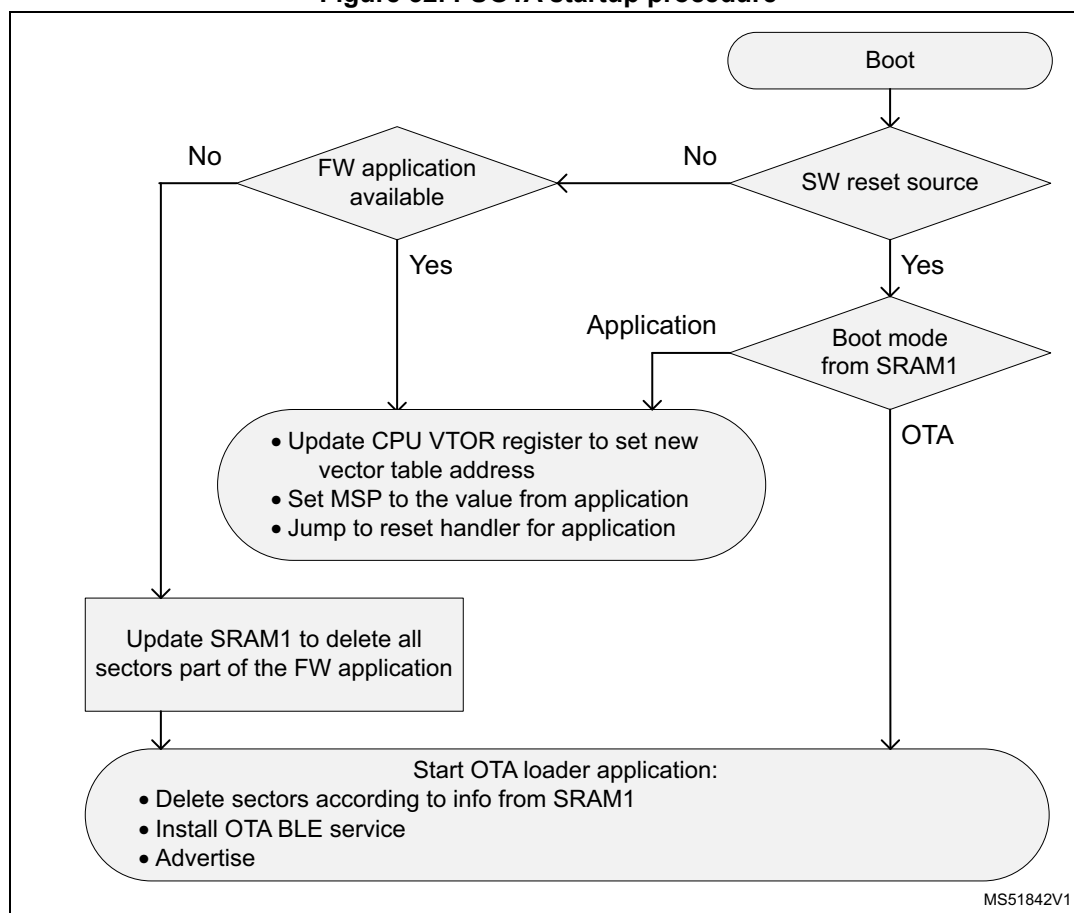
Once the project BLE_Ota is compiled and loaded, the application can:

- Either jump to the available application if binary code is present on application sectors (sector index 7)
 - no more activity related to BLE_Ota application
- Or start STMicroelectronics proprietary FUOTA GATT service and characteristics advertising:
 - the local name Advertising Data (AD) elements with “STM_OTA”
 - the manufacturer AD elements with Device ID “STM32WB FW Update OTA application”

The second possibility allows a remote device to upload a new binary (CPU2 wireless stack, CPU1 application or user data firmware update).

Note: *If only STMicroelectronics proprietary FUOTA GATT service and characteristics are used it is important to erase application sectors (from 7 onwards).*

Figure 32. FUOTA startup procedure



8.5.3 BLE FUOTA services and characteristics specification

The BLE FUOTA application (BLE_Ota project) is exported as a GATT service with the following characteristics:

- Base address to provide information where to store the new binary
- File upload reboot confirmation to confirm the reboot of the application after new binary file uploaded
- OTA raw data to transfer the data (binary file divided into packets).

Table 23. FUOTA service and characteristics UUID

Group	Service	Characteristic	Size	Mode	UUID
LED button control	OTA FW update	-	-	-	0000FE20-cc7a-482a-984a-7f2ed5b3e58f
	-	Base address	4 bytes	Write without response	0000FE22-8e22-4541-9d4c-21edae82ed19
	-	File upload reboot confirmation	1 byte	Indicate	0000FE23-8e22-4541-9d4c-21edae82ed19
	-	OTA raw data	20 bytes	Write without response	0000FE24-8e22-4541-9d4c-21edae82ed19

Table 24. Base address characteristics specification

Bit LSB	[0:7]	[8:31]
Name	Action	Address
Value	<ul style="list-style-type: none"> – 0x00: STOP all uploads – 0x01: START M0+ file upload – 0x02: START M4 file upload – 0x07: File upload finished – 0x08: Cancel upload 	0x007000

Table 25. File upload confirmation reboot request characteristics specification

Octets LSB	0
Name	Indication
Value	0x01 Reboot

Table 26. Raw data characteristics specification

Octets LSB	0	1	...	19
Name	Raw data			
Value	File data			

8.5.4 Flow description example to upload new CPU1 application binary

There are two types of procedures to upload a new binary:

- Only STMicroelectronics proprietary FUOTA GATT service and characteristics application is loaded (no application binary present on sector 7)
- Application is already running with the support of the Reboot request characteristic

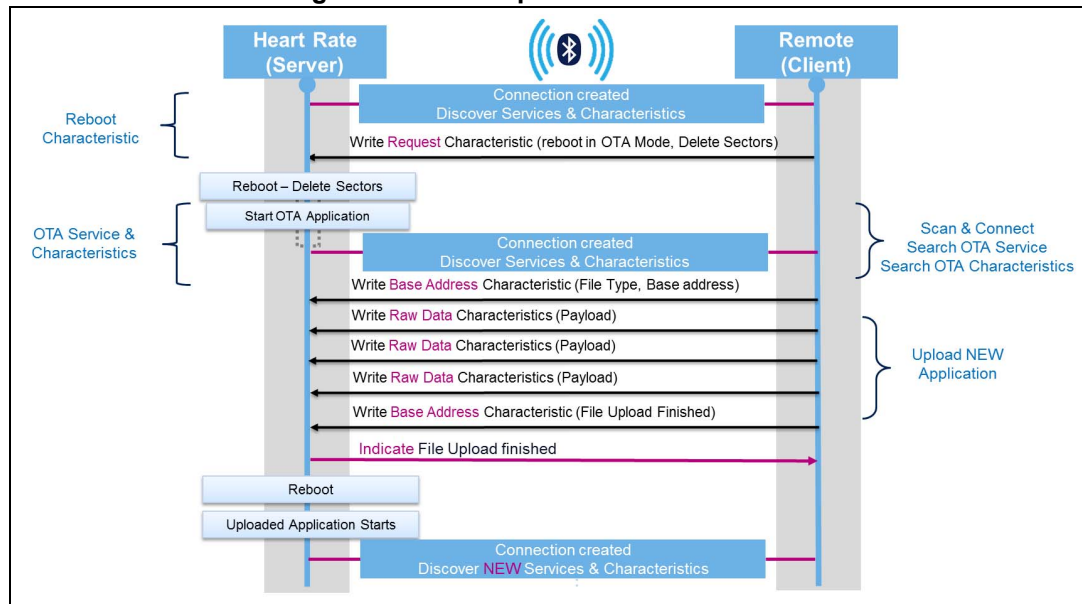
Table 27. Reboot request characteristics specification

Octets LSB	0	1	2
Name	Boot mode	Sector index	Number of sectors to erase
Value	<ul style="list-style-type: none"> – 0x00 application – 0x01 FUOTA application 	07 → 0x0800 7000	0x00 ... 0xFF

Starting from application including the reboot request characteristic, this is the flow to update CPU1 application:

1. BLE application includes the reboot characteristics.
2. Once the GAP connection is established, the remote GATT client device researches services and characteristics (reboot request characteristics detected).
3. Next, in order to switch to the FUOTA application, the remote device writes the Reboot request characteristics with information for boot mode option and sectors to erase.
4. At this stage the BLE Link is disconnected to reboot on STMicroelectronics proprietary FUOTA GATT service and characteristics application.
5. Application sectors are erased with the information provided by the reboot characteristics and STMicroelectronics proprietary FUOTA GATT service and characteristics application starts advertising.
6. New connection has to be established by the remote device to discover the FUOTA service and characteristics.
7. The base address characteristic is used to initiate the new binary upload.
8. All the data is transferred via the raw data characteristic and programmed directly to the flash memory once received.
9. The end of file transfer is confirmed by the base address characteristic.
10. Confirmation of the received file is indicate by the file upload confirmation characteristic.
11. At this stage, the FUOTA application checks the integrity of the new binary and reboots to start the new uploaded application.
12. If the application integrity is not ensured, the application sectors are erased to reboot on the FUOTA application.

Figure 33. FUOTA process with heart rate



8.5.5 Application example with smart phone

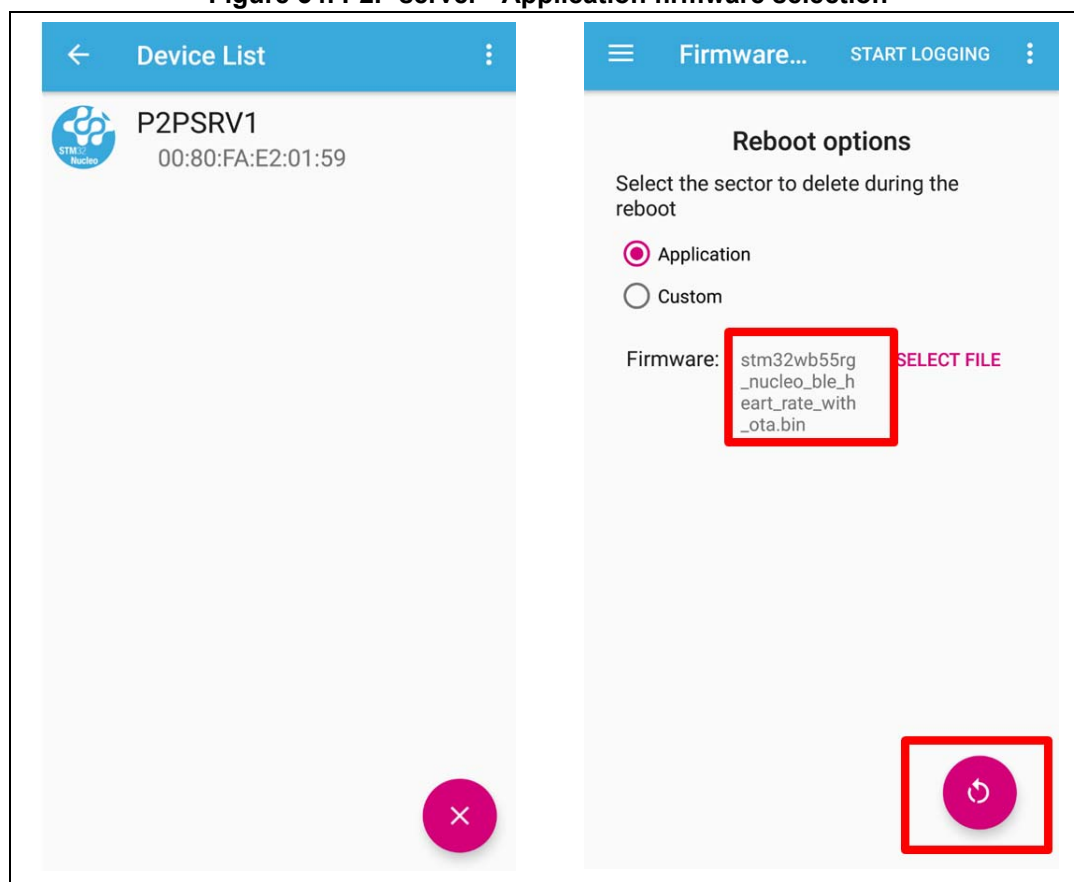
The reboot request characteristic is implemented in the projects BLE_HeartRate_ota and BLE_P2pServer_ota. Both include the OTA reboot request bit mask in their advertising element.

This is a way for the remote (scanner) to quickly acquire the information on the presence of the reboot request characteristic. The ST BLE sensor mobile application supports the detection of this reboot request characteristic.

For example, moving from a P2P server application to heart rate application.

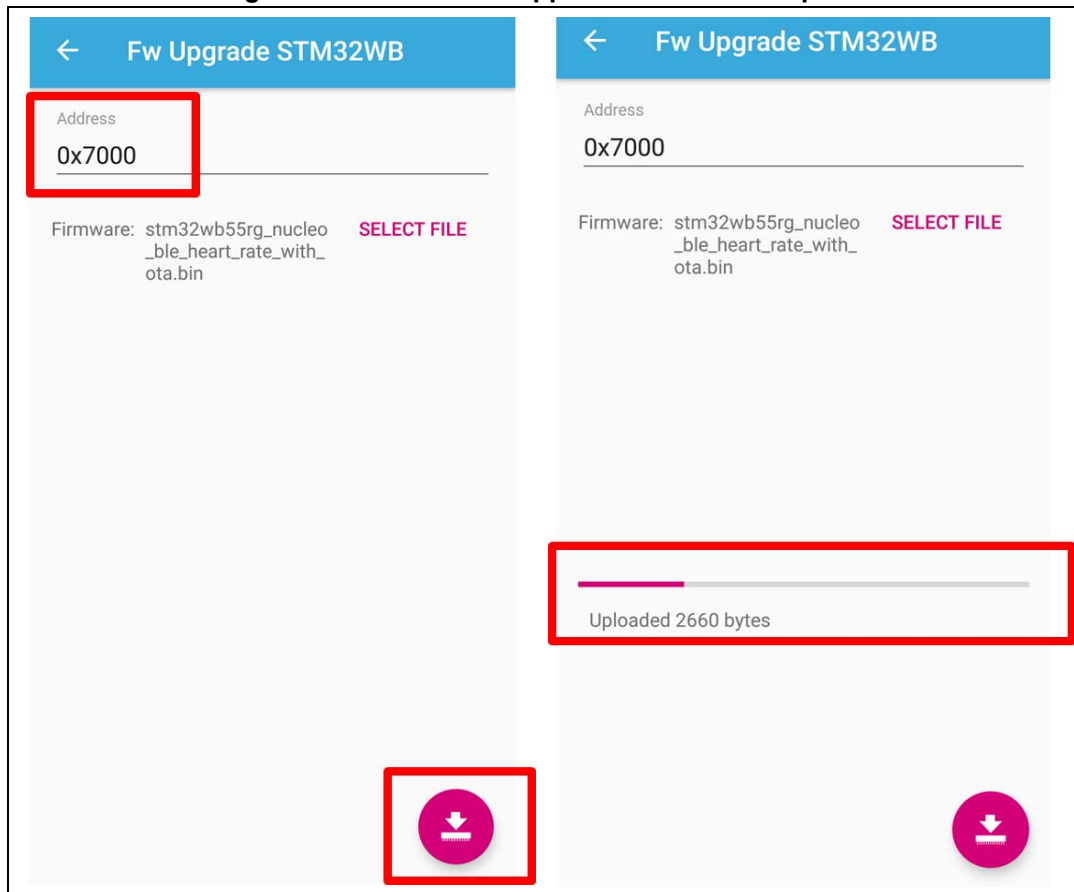
- Compile and load the BLE_Ota project to address 0x0800 0000
- Compile and load the BLE_p2pServer_ota project to address 0x0800 7000
- Reboot the device
 - At this stage, the P2P server advertises its presence.
- Discover and connect to the P2P server with ST BLE sensor mobile application
- Move to the reboot panel
- Select binary “BLE_HeartRate_ota” (copied before the demonstration to the smart phone memory)
- Click on upload
 - At this stage, the reboot request characteristic is used to provide information about sectors to erase and the next reboot phase (FUOTA application).

Figure 34. P2P server - Application firmware selection



Once rebooted, the address to upload the application binary file is selected. The default address is 0x7000 (sector 7 - application). Changing the binary file to upload is still possible at this stage, if needed.

Figure 35. P2P server - Application firmware update

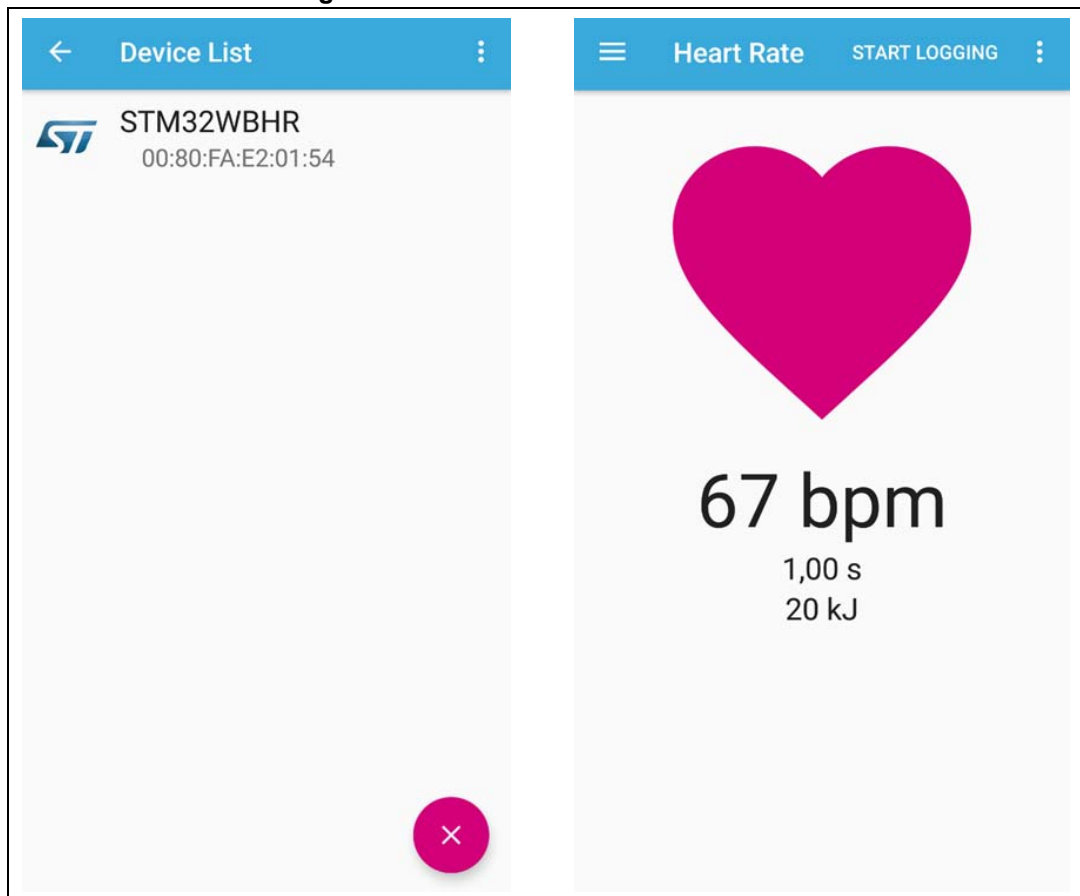


Once the upload is finished, the reboot procedure is executed to start the new application. Next, run a new scan procedure to discover the heart rate sensor advertising packets, and connect to it.

After connecting to the device, the heart rate measurement values are notified by the sensor.

Note: *The smart phone application associates the GATT database with remote Bluetooth address. To solve this issue the FUOTA application advertising address is increased by 1.*

Figure 36. Heart rate sensor notification



8.5.6 How to use the reboot request characteristics

Whatever application is used, the reboot request characteristics can be integrated into a service to reboot the application in FUOTA application mode.

The application must be loaded at the address 0x0800 0700 with “BLE_HeartRate_ota” and “BLE_p2pServer_ota” examples, the configuration is done as follows:

- ble_conf.h to define the OTA Reboot characteristics
- ```

/*****

* Over The Air Feature (OTA) - STM Proprietary

****/

#define BLE_CFG_OTA_REBOOT_CHAR 1 /**< REBOOT OTA MODE
CHARACTERISTIC */
• app_ble.c
/**
* Initialization of ADV - Ad Manufacturer Element - Support OTA Bit Mask
*/
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)

```

```

 manu_data[sizeof(manu_data)-8] = CFG_FEATURE_OTA_REBOOT;
#endif
• p2p_stm.c to add the characteristics (middleware)
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)
 /**
 * Add Boot Request Characteristic
 */
 aci_gatt_add_char(aPeerToPeerContext.PeerToPeerSvcHdle,
 BM_UUID_LENGTH,
 (Char_UUID_t *)BM_REQ_CHAR_UUID,
 BM_REQ_CHAR_SIZE,
 CHAR_PROP_WRITE_WITHOUT_RESP,
 ATTR_PERMISSION_NONE,
 GATT_NOTIFY_ATTRIBUTE_WRITE,
 10,
 0,
 &(aPeerToPeerContext.RebootReqCharHdle));
#endif
• p2p_stm.c to receive the request at GATT level and inform the application (middleware)
else if(attribute_modified->Attr_Handle ==
(aPeerToPeerContext.RebootReqCharHdle + 1))
{
 BLE_DBG_P2P_STM_MSG("-- GATT : REBOOT REQUEST RECEIVED\n");
 Notification.P2P_Evt_Opcode = P2PS_STM_BOOT_REQUEST_EVT;
 Notification.DataTransferred.Length=attribute_modified->Attr_Data_Length;
 Notification.DataTransferred.pPayload=attribute_modified->Attr_Data;
 P2PS_STM_App_Notification(&Notification);
• p2p_server_app.c to manage the reboot request (application)
void P2PS_STM_App_Notification(P2PS_STM_App_Notification_evt_t
*pNotification)
{
 switch(pNotification->P2P_Evt_Opcode)
 {
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)

 case P2PS_STM_BOOT_REQUEST_EVT:
 APP_DBG_MSG("-- P2P APPLICATION SERVER : BOOT REQUESTED\n");
 APP_DBG_MSG(" \n\r");

 (uint32_t)SRAM1_BASE = *(uint32_t*)pNotification-
>DataTransferred.pPayload;
 NVIC_SystemReset();
 break;
 }
#endif
}

```

## 8.5.7 Power failure recovery mechanism for CPU1 application

The BLE\_ota application provides a power failure recovery mechanism while updating the CPU1 application.

The two tags used to manage a power failure during CPU1 application firmware update are:

1. MagicKeywordAddress: must be mapped at 0x140 from start of the binary image to be loaded
2. MagicKeywordvalue: checked by the BLE\_ota application at MagicKeywordAddress.

While flashing the new application, if the link is dropped, the BLE\_ota application detects the failure and automatically erases the programmed sectors. This mechanism prevents a reboot on a wrong application.

```
/**
 * These are the two tags used to manage a power failure during CM4 Application OTA FW Update
 * The MagicKeywordAddress shall be mapped @0x140 from start of the binary image
 * The MagicKeywordvalue is checked in the ble_ota application
 */
PLACE_IN_SECTION("TAG_OTA_END") const uint32_t MagicKeywordValue = 0x94448A29 ;
PLACE_IN_SECTION("TAG_OTA_START") const uint32_t MagicKeywordAddress = (uint32_t)&MagicKeywordValue;

define region OTA_TAG_region = mem:(from (__ICFEDIT_region_ROM_start__ + 0x140) to (__ICFEDIT_region_ROM_start__ + 0x140 + 4));

define block CSTACK with alignment = 8, size = __ICFEDIT_size_cstack__ { };
define block HEAP with alignment = 8, size = __ICFEDIT_size_heap__ { };

initialize by copy { readwrite };
do not initialize { section .noinit,
 section MAPPING_TABLE,
 section MB_MEM1 };

place at address mem: __ICFEDIT_intvec_start__ { readonly section .intvec };

keep { section TAG_OTA_START };
keep { section TAG_OTA_END };
place in OTA_TAG_region { section TAG_OTA_START };
place in ROM_region { readonly, last section TAG_OTA_END };
```

## 8.6 Application tips

### 8.6.1 How to set Bluetooth device address

All Bluetooth devices must have an address that uniquely identifies them.

STM32WB devices support public and random (static or private) addresses.

Device addresses can be public or random, both are 48-bits long, and denoted as colon-delimited hex values (e.g. AA:BB:CC:DD:EE:FF).

The public device address must conform with the IEEE 802-2001 standard, using a valid organization unique identifier (OUI) obtained from the IEEE registration authority. Public device addresses are known as MAC addresses.

Customers willing to develop their own products (based on public device addresses) must obtain from the IEEE Registration Authority their own IEEE-assigned 48-bit universal LAN MAC address, and not use the one already assigned to ST Microelectronics.

For the details on how a BLE device can generate a random address refer to the Core Specification v5.0.

The STM32WB provides a 64-bit unique device identification

- 24-bit company ID (0x00 80 E1 for STMicroelectronics)
- 8-bit device ID (0x05 for STM32WB)
- 32-bit unique device number, to differentiate each individual device.

If the application needs to use a different public address, it must be obtained from the right organization, and then stored in a persistent memory location of the final product (either within the microcontroller flash memory/OTP, or in an external storage area).

During the STM32WB initialization phase, the application must configure this address.

The ACI command to set the public address is:

```
tBleStatus aci_hal_write_config_data(uint8_t offset, uint8_t len, const
uint8_t *val).
```

The parameters must be set as follows:

- Offset: 0x00
- Length: 0x06
- Value: pointer to the public address value, e.g. 0xaabbccddeeff (6-byte array).

The application microprocessor must send the command `aci_hal_write_config_data` to the wireless microprocessor before starting any BLE operations and after every power-up or reset, since the command `aci_hal_write_config_data` does not systematically save the data in the flash memory.

The following pseudo-code example illustrates how to set a MAC address from the application:

```
uint8_t bdaddr[] = {0xFF, 0xEE, 0xDD, 0xCC, 0xBB, 0xAA};
ret=aci_hal_write_config_data(0x00, 0x06, bdaddr);
if(ret) { PRINTF("Setting address failed.\n")}
```

BLE devices can also use random addresses. The address value can be read from the application using the `tBleStatus aci_hal_read_config_data(uint8_t offset, uint16_t data_len, uint8_t *data_len_out_p, uint8_t *data);` command with the parameter `offset` set to 0x80.

Alternatively, the application can set a random address from the external host processor using the `int hci_le_set_random_address(tBdAddr bdaddr)` command after each reset. If the random address is not set through the `hci_le_set_random_address` command, the address generation is handled independently by the stack as described above.

The 64-bit UID of the STM32WB device can be used to derive the unique BLE 48-bit device address. It is also possible to get the BLE-48-bit device address from the OTP register.

```
const uint8_t* BleGetBdAddress(void) {
 uint8_t *otp_addr;
 const uint8_t *bd_addr;
 uint32_t udn;
 uint32_t company_id;
 uint32_t device_id;
 udn = LL_FLASH_GetUDN();

 if(udn != 0xFFFFFFFF) {
 company_id = LL_FLASH_GetSTCompanyID();
 device_id = LL_FLASH_GetDeviceID();
```

```

 bd_addr_udn[0] = (uint8_t)(udn & 0x000000FF);
 bd_addr_udn[1] = (uint8_t)((udn & 0x0000FF00) >> 8);
 bd_addr_udn[2] = (uint8_t)((udn & 0x00FF0000) >> 16);
 bd_addr_udn[3] = (uint8_t)device_id;
 bd_addr_udn[4] = (uint8_t)(company_id & 0x000000FF);
 bd_addr_udn[5] = (uint8_t)((company_id & 0x0000FF00) >> 8);

 bd_addr = (const uint8_t *)bd_addr_udn;
}
else {
 otp_addr = OTP_Read(0);
 if(otp_addr) {
 bd_addr = ((OTP_ID0_t*)otp_addr)->bd_address;
 }
 else {
 bd_addr = M_bd_addr;
 }
}
return bd_addr;
}

```

### 8.6.2 How to set IR (Identity Root) and ER (Encryption Root)

The Identity Root (IR) key is used to generate IRK and DHK(Legacy), the Encryption Root (ER) key is used to generate LTK(Legacy) and CSRK. IRK and ERK must be generated as a random key on each unit, and stored in flash memory during device production. At startup, in the device application, the ER and IR values are read and transferred to the Cortex M0 side using `aci_hal_write_config_data`. The keys are different from device to device.

**Note:** *The generation of LTK using ER is applicable only when doing LE Legacy Pairing. The generation of CSRK using ER is applicable when doing LE Legacy Pairing and LE Secure Connections Pairing. If ER is changed, any previously distributed LTK or CSRK keys is no longer valid.*

*The generation of DHK using IR is applicable only when doing LE Legacy Pairing. The generation of IRK using IR is applicable when doing LE Legacy Pairing and LE Secure Connections Pairing. If IR is changed, any previously distributed IRK key is no longer valid.*

In STM32CubeWB examples, IR and ER are fixed values, written as described below:

Define:

`CFG_BLE_IR` and `CFG_BLE_ER`

Declare:

```

static const uint8_t BLE_CFG_IR_VALUE[16] = CFG_BLE_IR;
static const uint8_t BLE_CFG_ER_VALUE[16] = CFG_BLE_ER;

```

Write IR:

```

aci_hal_write_config_data(CONFIG_DATA_IR_OFFSET, CONFIG_DATA_IR_LEN,
(uint8_t*)BLE_CFG_IR_VALUE);

```

Write ER:

```
aci_hal_write_config_data(CONFIG_DATA_ER_OFFSET, CONFIG_DATA_ER_LEN,
(uint8_t*)BLE_CFG_ER_VALUE);
```

### 8.6.3 How to add a task to the sequencer

- Declare task ID - app\_conf.h -

```
/**< Add in that list all tasks that may send a ACI/HCI command */
typedef enum
{
 CFG_TASK_ADV_CANCEL_ID,
 CFG_TASK_SW1_BUTTON_PUSHED_ID,
 CFG_TASK_HCI_ASYNC_EVT_ID,
 CFG_LAST_TASK_ID_WITH_HCICMD, /**< Shall be LAST in the list */
} CFG_Task_Id_With_HCI_Cmd_t;

/**< Add in that list all tasks that never send a ACI/HCI command */
typedef enum
{
 CFG_FIRST_TASK_ID_WITH_NO_HCICMD = CFG_LAST_TASK_ID_WITH_HCICMD - 1,
 /**< Shall be FIRST in the list */
 CFG_TASK_SYSTEM_HCI_ASYNC_EVT_ID,
 CFG_LAST_TASK_ID_WITHO_NO_HCICMD
 /**< Shall be LAST in the list */
} CFG_Task_Id_With_NO_HCI_Cmd_t;
#define UTIL_SEQ_CONF_TASK_NBR CFG_LAST_TASK_ID_WITHO_NO_HCICMD
```

- Register task with callback function- "Cancel Advertising" - app\_ble.c  
SCH\_RegTask(CFG\_TASK\_ADV\_CANCEL\_ID, Adv\_Cancel);
- Start the Task with priority - app\_ble.c  
SCH\_SetTask(1 << CFG\_TASK\_ADV\_CANCEL\_ID, CFG\_SCH\_PRIO\_0);

### 8.6.4 How to use the timer server

- Create the timer with callback function
- ```
/**
 * Create timer to handle the Led Switch OFF
 */
HW_TS_Create(CFG_TIM_PROC_ID_ISR,
&(BleApplicationContext.SwitchOffGPIO_timer_Id), hw_ts_SingleShot,
Switch_OFF_GPIO);
```
- Start the timer with timeout
- ```
HW_TS_Start(BleApplicationContext.SwitchOffGPIO_timer_Id,
(uint32_t)LED_ON_TIMEOUT);
```
- Stop the timer
- ```
HW_TS_Stop(BleApplicationContext.SwitchOffGPIO_timer_Id);
```
- Callback function example

```
static void Switch_OFF_GPIO(){
    BSP_LED_Off(LED_GREEN);
}
```

8.6.5 How to start the BLE stack - SHCI_C2_BLE_Init()

```
SHCI_C2_Ble_Init_Cmd_Packet_t ble_init_cmd_packet =
{
    {{0,0,0}},           /**< Header unused */
    {0,                  /** pBleBufferAddress not used */
    0,                   /** BleBufferSize not used */
    CFG_BLE_NUM_GATT_ATTRIBUTES,
    CFG_BLE_NUM_GATT_SERVICES,
    CFG_BLE_ATT_VALUE_ARRAY_SIZE,
    CFG_BLE_NUM_LINK,
    CFG_BLE_DATA_LENGTH_EXTENSION,
    CFG_BLE_PREPARE_WRITE_LIST_SIZE,
    CFG_BLE_MBLOCK_COUNT,
    CFG_BLE_MAX_ATT_MTU,
    CFG_BLE_SLAVE_SCA,
    CFG_BLE_MASTER_SCA,
    CFG_BLE_LSE_SOURCE,
    CFG_BLE_MAX_CONN_EVENT_LENGTH,
    CFG_BLE_HSE_STARTUP_TIME,
    CFG_BLE_VITERBI_MODE,
    CFG_BLE_OPTIONS,
    0,
    CFG_BLE_MAX_COC_INITIATOR_NBR,
    CFG_BLE_MIN_TX_POWER,
    CFG_BLE_MAX_TX_POWER}
};
```

CFG_BLE_NUM_GATT_ATTRIBUTES

Maximum number of attribute records related to all the required characteristics (excluding the services) that can be stored in the GATT database, for the specific BLE user application.

For each characteristic, the number of attribute records goes from two to five depending on the characteristic properties:

- minimum of two (one for declaration and one for the value)
- add one more record for each additional property: notify or indicate, broadcast, extended property.

The total calculated value must be increased by 9, due to the records related to the standard attribute profile and GAP service characteristics, and automatically added when initializing GATT and GAP layers

- Min value: <number of user attributes> + 9
- Max value: depending on the GATT database defined by user application

CFG_BLE_NUM_GATT_SERVICES

Defines the maximum number of services that can be stored in the GATT database. Note that the GAP and GATT services are automatically added at initialization so this parameter must be the number of user services increased by two.

- Min value: <number of user service> + 2
- Max value: depending GATT database defined by user application

CFG_BLE_ATT_VALUE_ARRAY_SIZE

Size of the storage area for the attribute values.

To calculate CFG_BLE_ATT_VALUE_ARRAY_SIZE, it is necessary to add characteristics related to default GATT services.

By default, two services are present and must be included, with dedicated characteristics:

- Generic access service: service UUID 0x1800, with its three mandatory characteristics:
 - Device name UUID 0x2A00
 - Appearance UUID 0x2A01
 - Peripheral preferred connection parameters. UUID 0x2A04
- Generic attribute service. UUID 0x1801, with one optional characteristic:
 - Service changed UUID 0x2A05

Each characteristic contributes to the attrValueArrSize value as follows:

- Characteristic value length plus:
 - 5 bytes if characteristic UUID is 16 bits
 - 19 bytes if characteristic UUID is 128 bits
 - 2 bytes if characteristic has a server configuration descriptor
 - 2 bytes * CFG_BLE_NUM_LINK if the characteristic has a client configuration descriptor
 - 2 bytes if the characteristic has extended properties

Each descriptor contributes to the attrValueArrSize value as follows:

- Descriptor length

CFG_BLE_NUM_LINK

Maximum number of BLE links supported

- Min value: 1
- Max value: 8

CFG_BLE_DATA_LENGTH_EXTENSION

Disable/enable the extended packet length BLE 5.0 feature

- Disable: 0
- Enable: 1

CFG_BLE_PREPARE_WRITE_LIST_SIZE

Maximum number of supported “prepare write request”. The minimum required value can be calculated using the following DEFAULT_PREP_WRITE_LIST_SIZE macro:

```
#define DIVC(x, y)      (((x) + (y) - 1) / (y))
```

```

/**
 * DEFAULT_ATT_MTU: minimum mtu value that GATT must support.
 * 5.2.1 ATT_MTU, BLUETOOTH SPECIFICATION Version 4.2 [Vol 3, Part G]
 */
#define DEFAULT_ATT_MTU                (23)
/**
 * DEFAULT_MAX_ATT_SIZE: maximum attribute size.
 */
#define DEFAULT_MAX_ATT_SIZE            (512)

/**
 * PREP_WRITE_X_ATT(max_att): compute how many Prepare Write Request are
needed
 * to write a characteristic with size max_att when the used ATT_MTU value
is
 * equal to DEFAULT_ATT_MTU (23).
 */
#define PREP_WRITE_X_ATT(max_att)       (DIV_CEIL(max_att, DEFAULT_ATT_MTU
- 5U) * 2)
/**
 * DEFAULT_PREP_WRITE_LIST_SIZE: default minimum Prepare Write List size.
 */
#define DEFAULT_PREP_WRITE_LIST_SIZE
PREP_WRITE_X_ATT(DEFAULT_MAX_ATT_SIZE)

```

- Min value: see macros above
- Max value: a value higher than the minimum required can be specified, but it is not recommended

CFG_BLE_MBLOCK_COUNT

Number of allocated memory blocks for the BLE stack. The minimum required value can be calculated using the following MBLOCKS_CALC macros:

```

#define MEM_BLOCK_SIZE                (32)
/**
 * MEM_BLOCK_X_MTU (mtu): compute how many memory blocks are needed to
compose an ATT
 * Packet with ATT_MTU = mtu.
 * 7.2 FRAGMENTATION AND RECOMBINATION, BLUETOOTH SPECIFICATION Version 4.2
 * [Vol 3, Part A]
 */
#define MEM_BLOCK_X_TX (mtu)           (DIV_CEIL((mtu) + 4U,
MEM_BLOCK_SIZE) + 1U)
#define MEM_BLOCK_X_RX (mtu, n_link)  ((DIV_CEIL((mtu) + 4U,
MEM_BLOCK_SIZE) + 2U) * (n_link) + 1)
#define MEM_BLOCK_X_MTU (mtu, n_link) (MEM_BLOCK_X_TX(mtu) +
MEM_BLOCK_X_RX(mtu, (n_link)))

```

```

/**
 * Minimum number of blocks required for secure connections
 */
#define MBLOCKS_SECURE_CONNECTIONS      (4)

/**
 * MBLOCKS_CALC(pw, mtu, n_link): minimum number of buffers needed by the
stack.
 * This is the minimum recommended value and depends on:
 * - pw: size of Prepare Write List
 * - mtu: ATT_MTU size
 * - n_link: maximum number of simultaneous connections
 */
#define MBLOCKS_CALC(pw, mtu, n_link)    ((pw) + MAX(MEM_BLOCK_X_MTU(mtu,
n_link), (MBLOCKS_SECURE_CONNECTIONS)))

```

- Min value: see macro above
- Max value: a higher value can improve data throughput performance, but uses more memory.

CFG_BLE_MAX_ATT_MTU

Maximum ATT MTU size supported.

- Min value: 23
- Max value: 512

CFG_BLE_SLAVE_SCA

The sleep clock accuracy (ppm value) used in BLE connected slave mode to calculate the window widening (in combination with the sleep clock accuracy sent by master in CONNECT_REQ PDU), refer to BLE 5.0 specifications - Vol 6 - Part B - chap 4.5.7 and 4.2.2.

- Min value: 0
- Max value: 500 (worst possible admitted by specification)

CFG_BLE_MASTER_SCA

The sleep clock accuracy handled in master mode. It is used to determine the connection and advertising events timing. It is transmitted to the slave in CONNEC_REQ PDU used by the slave to calculate the window widening, see [CFG_BLE_SLAVE_SCA](#) and [\[7\]](#), v5.0 Vol 6 - Part B - chap 4.5.7 and 4.2.2.

Possible values:

- 251 ppm to 500 ppm: 0
- 151 ppm to 250 ppm: 1
- 101 ppm to 150 ppm: 2
- 76 ppm to 100 ppm: 3
- 51 ppm to 75 ppm: 4
- 31 ppm to 50 ppm: 5
- 21 ppm to 30 ppm: 6
- 0 ppm to 20 ppm: 7

CFG_BLE_LSE_SOURCE

Source for the 32 kHz slow speed clock.

- External crystal LSE: 0 - No calibration
- Internal RO (LSI): 1 - The accuracy of this oscillator can vary depending upon external conditions (temperature), hence it is calibrated every second to ensure correct behavior of timing sensitive BLE operations.

CFG_BLE_MAX_CONN_EVENT_LENGTH

This parameter determines the maximum duration of a slave connection event. When this duration is reached the slave closes the current connections event (whatever is the CE_length parameter specified by the master in HCI_CREATE_CONNECTION HCI command), expressed in units of 625 / 256 μ s (~2.44 μ s).

- Min value: 0 (if 0 is specified, the master and slave perform only a single TX-RX exchange per connection event).
- Max value: 1638400 (4000 ms). A higher value can be specified (max 0xFFFFFFFF) but results in a maximum connection time of 4000 ms as specified. In this case the parameter is not applied, and the predicted CE length calculated on slave is not shortened.

CFG_BLE_HSE_STARTUP_TIME

Startup time of the high speed (16 or 32 MHz) crystal oscillator, in units of 625/256 μ s (~2.44 μ s).

- Minimum: 0x148 (328 * 2.44 μ s = 800 μ s), default value
- Maximum: 0x334 (820 * 2.44 μ s = 2 ms)

Due to the design of the BLE radio, it is recommended to not modify the default value of this parameter.

CFG_BLE_VITERBI_MODE

Viterbi implementation in BLE LL reception

- 0: Enabled
- 1: Disabled

CFG_BLE_OPTIONS

This is an 8-bit parameter, each bit enables/disables an option:

- bit 0:
 - 1: LL only
 - 0: LL + host
- bit 1:
 - 1: No service change description
 - 0: With service change description
- bit 2:
 - 1: Device name read-only
 - 0: Device name R/W
- bit 3:
 - 1: Extended advertising supported
 - 0: Extended advertising not supported
- bit 4:
 - 1: CS Algo #2 supported
 - 0: CS Algo #2 not supported
- bits 5 and 6: Reserved (must be kept to 0)
- bit 7:
 - 1: LE power class 1
 - 0: LE power classes 2 and 3

8.6.6 BLE GATT DB and security record in NVM

The GATT record in NVM is composed:

- Per service:
 - 2 bytes for handle
 - 3 bytes if 16-bit UUID or 17 bytes if 128-bit UUID
- Per characteristic attribute:
 - 2 bytes for handle
 - 3 bytes if 16-bit UUID or 17 bytes if 128-bit UUID
 - 2 bytes for CCCD value (only if the attribute is a CCCD)

The total corresponds to size_of_gatt record. The security record in NVM is fixed, equal to 80 bytes, corresponding to size_of_sec record.

8.6.7 How to calculate the maximum number of bonded devices that can be stored in NVM

The computation of the number of bonded devices that can be stored in NVM is

$$N = (\text{total_size_of_nvm} - 1) / [(\text{size_of_sec_record} + 1) + (\text{size_of_gatt_record} + 1)]$$

with total_size_of_nvm = 507 words (of 4 bytes).

8.6.8 NVM write access

Refer to [Section 4.7.3](#).

8.6.9 How to maximize data throughput

The maximum data throughput is achieved when GATT server notification is used with the following link layer parameters:

- Connection interval: 400 ms
- Min_CE_Length = 0 and Max_CE_Length: x280 (400 ms)

Once the connection is established, the master device sends `aci_gatt_exchange_config` to get the `MAX_ATT_MTU` value.

Data exchange is limited to $(MAX_ATT_MTU - 3)$, which corresponds to the maximum notification length.

- If supported, set the link at 2M

To avoid fragmenting the LE data have, at maximum `PDU_length = 247` ($251 - 4$):

- Use `hci_le_set_data_length` command `hci_le_set_data_length(conn_handle, 251, 2120)`

To avoid fragmentation:

- If `MAX_ATT_MTU = 250` and `le_data_length = 251`, max data to transfer = 244 ($251 - 4 - 3$)
- If `MAX_ATT_MTU = 156` and `le_data_length = 251`, max data to transfer = 153 ($156 - 3$)

8.6.10 How to add a custom BLE service

In all BLE applications, it is possible to add a custom service in parallel to existing ones provided either in source code or library. All GAP/GATT events received by the CPU1 are going to the service controller (`svc_ctl.c` from `\Middlewares\ST\STM32_WPAN\ble\svc\Src`) responsible to initialize all BLE services and to forward GATT events to registered BLE services. An example of the flow is shown in [Figure 24](#).

Each service must have a `custom_xxx.c` and `custom_xxx.h`.

There must be only three public interfaces to provide to the user:

```
void Custom_xxx_Init( void )
```

This is implemented in `custom_xxx.c` and does the following:

- creates the services and add characteristics
- registers the callback to the service controller with the API `SVCCTL_RegisterSvcHandler()`

The function `SVCCTL_InitCustomSvc()` must be implemented in the application to call `Custom_xxx_Init()`.

The callback registered with `SVCCTL_RegisterSvcHandler()` is used to receive GATT events from the service controller. The type of the callback must be `SVCCTL_EvtAckStatus_t (*SVC_CTL_p_EvtHandler_t)(void *p_evt)`.

Depending upon the BLE service definition, the received GATT event can be processed only in the `custom_xxx.c` module or, most of the times, it must be forwarded to the

application with the notification `Custom_xxx_Notification()`. Each GATT event is relevant for only one BLE service. To avoid the service controller to call all registered BLE services to report the received event, the callback informs the service controller if the GATT has been processed or ignored.

Three values can be returned:

1. `SVCCTL_EvtNotAck`: means the GATT event was not relevant for that BLE service. The service controller keeps reporting this GATT event to other registered BLE services until it gets an ack. When a GATT event is not acknowledged by all registered BLE services, it is reported to the application with the notification `SVCCTL_App_Notification()`.
2. `SVCCTL_EvtAckFlowEnable`: means the GATT event has been processed and the service controller does not report it to either other registered BLE services or to the application.
3. `SVCCTL_EvtAckFlowDisable`: means the GATT event has been acknowledged and the service controller does not report it to either other registered BLE services or to the application. However, the GATT event has not been processed. The service controller notifies the transport layer that this event shall not be discarded. In that case, the transport layer does not report any more event until the command `hci_resume_flow()` has been called. As soon as the flow is resumed, the not acknowledged event is reported one more time. These are all BLE user hci events not reported anymore, not only those to the BLE service that did not acknowledge the GATT event.

```
tBleStatus Custom_xxx_UpdateChar( Custom_xxx_ChardId_t ChardId, uint8_t *
p_payload )
```

This API is used by the application to update the characteristic of the server. The mapping between the `ChardId` of the interface and the UUID to be sent to the BLE stack must be implemented in the BLE service.

```
void Custom_xxx_Notification( Custom_xxx_Notification_t *p_notification )
```

This API is used to report, when relevant, to the application a GATT event received by the BLE service.

8.6.11 How to use BLE commands in blocking mode

The two hooks `hci_cmd_resp_wait()` and `hci_cmd_resp_release()` are implemented as weak function in Middleware. With this implementation `hci_cmd_resp_wait()` polls on a flag set from the IPCC interrupt context by `hci_cmd_resp_release()`. This ensures not to enter in idle mode before receiving the command response. To use BLE commands in blocking mode, do not redefine these two hooks in the application.

9 Building a BLE application on top of the HCI layer interface

CPU2 can be used as a BLE HCI layer co-processor. In that case, the user must either implement its own HCI application, or use an existing open source BLE host stack.

Most BLE host stacks use a UART interface to communicate with a BLE HCI co-processor. The equivalent physical layer on the STM32WB device is the mailbox, as described in [Section 14.2: Mailbox interface](#).

The mailbox provides an interface for both the BLE and the System channel. The BLE host stack builds the command buffer to be sent over the BLE channel on the mailbox and must provide an interface to report the events received through the mailbox. In addition to the mailbox BLE host stack adaptation, the user must notify the mailbox driver when an asynchronous packet is released.

The system channel is not handled by a BLE host stack. The user must implement its own transport layer to build the System command buffer to be sent to the mailbox driver and to manage the event received from the mailbox (including the notification to release an asynchronous buffer to the mailbox driver), or use the mailbox extended driver (as described in [Section 14.3: Mailbox interface - Extended](#)), which provides an interface on top of the provided transport layer which builds the System command buffer and to manage the system asynchronous event.

The BLE_TransparentMode project can be used as an example to build an application on top of a BLE HCI layer co-processor using the mailbox as described in [Section 14.2: Mailbox interface](#).

10 Thread

10.1 Overview

The Thread stack embarked in CPU2 core is provided by OpenThread, an open-source implementation of the Thread networking protocol, and is released by Nest.

OpenThread provides several APIs that address different services at different levels inside the stack. All these APIs (documented in the STM32WB firmware package) are exported on CPU1 core and can be used directly by the application.

The STM32WB firmware package is provided with several examples demonstrating how to run simple Thread applications. To run these applications, the appropriate CPU2 firmware binary need to be downloaded.

There are three major MO firmwares available, as detailed in [Table 28](#).

Table 28. MO firmwares available for Thread

CPU2 firmware library	Features	Comment
stm32wb5x_Thread_FTD_fw.bin	FTD: Full Thread device	The device can support all Thread roles except border router (leader, router, end device, sleepy end device). Thread roles are described in Section 14.10.5 .
stm32wb5x_Thread_MTD_fw.bin	MTD: Minimal Thread device	The device can act only as 'end device' or 'sleepy end device'. The MTD configuration requires less memory than the FTD configuration.
stm32wb5x_BLE_Thread_static_fw.bin	Static concurrent mode	The device embeds the two stacks (BLE and Thread) in a single binary for static concurrent mode.
stm32wb5x_BLE_Thread_dynamic_fw.bin	Dynamic concurrent mode	The device embeds the two stacks (BLE and Thread) in a single binary for dynamic concurrent mode.

10.2 How to start

The easiest way to start with Thread is to use the two following applications:

- Thread_Cli_Cmd: shows how to control the Thread stack via CLI commands. The CLI (command line interface) commands are sent via an UART from a HyperTerminal (PC) to the board and can be used to create simple use-cases. This is the application used for running certification tests (Thread GRL test harness)
- Thread_Coap_Generic: requires two P-NUCLEO-WBxx boards. It shows a board exchanging CoAP messages with the other one. In this application, one device is acting as leader and the other one is acting as end device or router.

These two applications are provided in the STM32WB firmware package with an associated readme.txt file.

10.3 Thread configuration

Before starting any Thread application, the user must download the appropriate firmware (Thread MTD, Thread FTD or Thread Static mode), and use the correct option bytes.

Figure 37. User option bytes setting

User configuration option byte

<input checked="" type="checkbox"/> nBOOT0	<input checked="" type="checkbox"/> nRSTSHDW	<input checked="" type="checkbox"/> WWDGSW
<input checked="" type="checkbox"/> nBOOT1	<input checked="" type="checkbox"/> nRSTSTDBY	<input checked="" type="checkbox"/> IWDGSW
<input checked="" type="checkbox"/> nSWBOOT0	<input checked="" type="checkbox"/> nRSTSTOP	<input checked="" type="checkbox"/> IWDGSTDBY
<input checked="" type="checkbox"/> SRAM2RST	<input type="checkbox"/> PCROP_RDP	<input checked="" type="checkbox"/> IWDGSTOP
<input checked="" type="checkbox"/> SRAM2PE		

IPCCDBA 0x0000

Caution: The OpenThread stack provides several compilation flags to set different configurations. Nevertheless, since the stack inside the STM32WB is delivered as a binary, those flags are fixed and cannot be modified by the user. The selected flags can be seen in the files listed in [Table 29](#).

Table 29. Files for Thread configuration

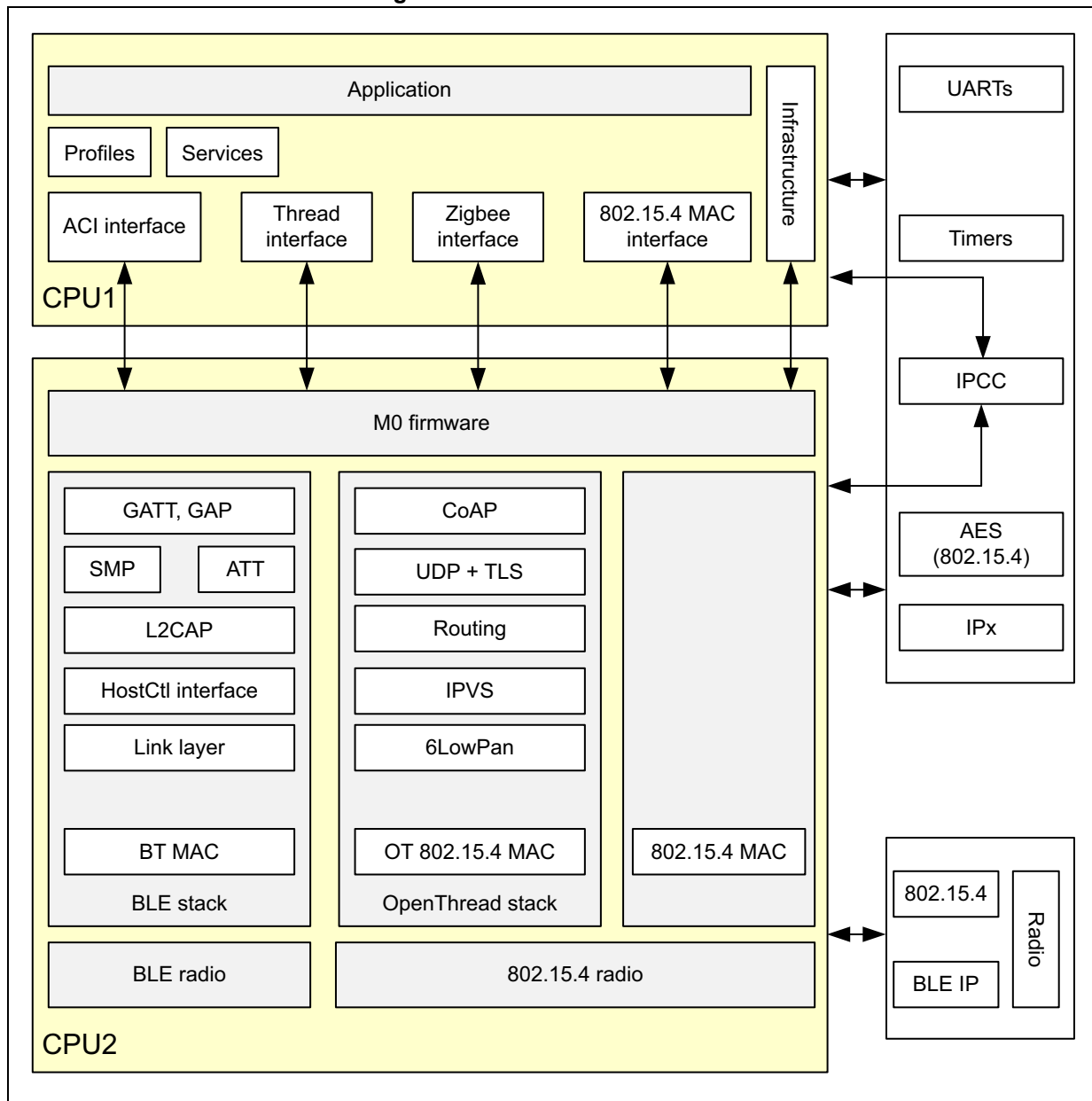
File name	Configuration
openthread_api_config_ftd.h	To be used with the Thread FTD CPU2 firmware
openthread_api_config_mtd.h	To be used with the Thread MTD CPU2 firmware
openthread_api_config_concurrent.h	To be used with Static concurrent mode CPU2 firmware

When building a Thread application, the appropriate configuration file must be used depending on the downloaded CPU2 firmware. The flags inside this configuration file are used to define which APIs are exported and available for CPU1 application. As mentioned before, these flags must not be modified by the user.

10.4 Architecture overview

[Figure 38](#) shows the overall software architecture with the two BLE and Thread stacks. All the code running on CPU2 is delivered as a binary library. The customer has only access to CPU1 core and sees the firmware running on CPU2 as a black box. Both the ACI and the Thread interfaces allow the user to access, respectively, the BLE and the Thread task.

Figure 38. Software architecture



10.5 Inter core communication

All OpenThread APIs are exposed to CPU1 and can be used to control the stack running on CPU2. The STM32WB middleware manages the communication between the two cores.

When the application calls an OpenThread function, a synchronous message is sent to CPU2 via IPCC. The parameters associated with this function are stored in shared memory.

OpenThread functions calls are put on hold until the command is completed to ensure that the overall system stays synchronized (see [Figure 39](#)). The application can register callbacks to be notified on specific events. These notifications are also put on hold, as shown in [Figure 40](#).

Figure 39. OpenThread functions calls

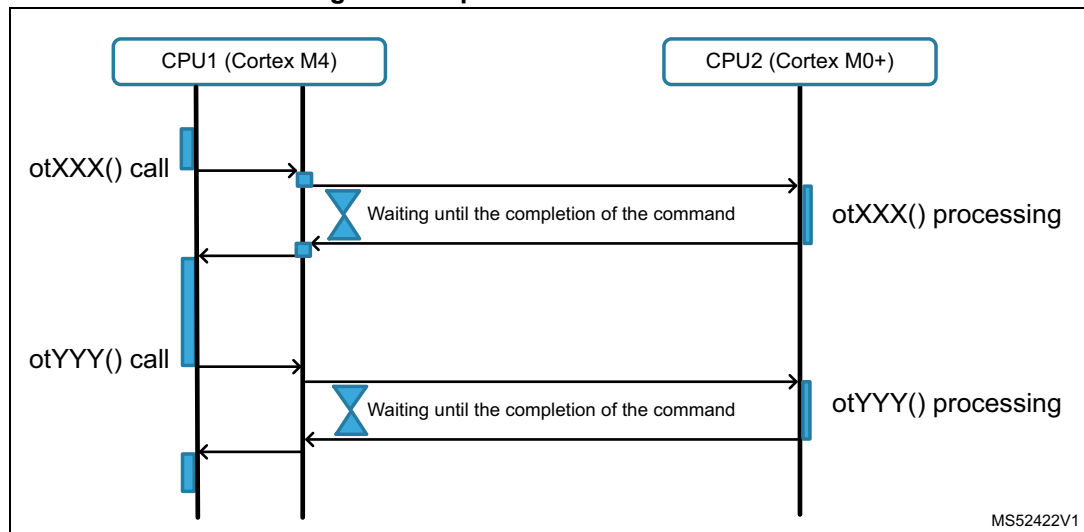
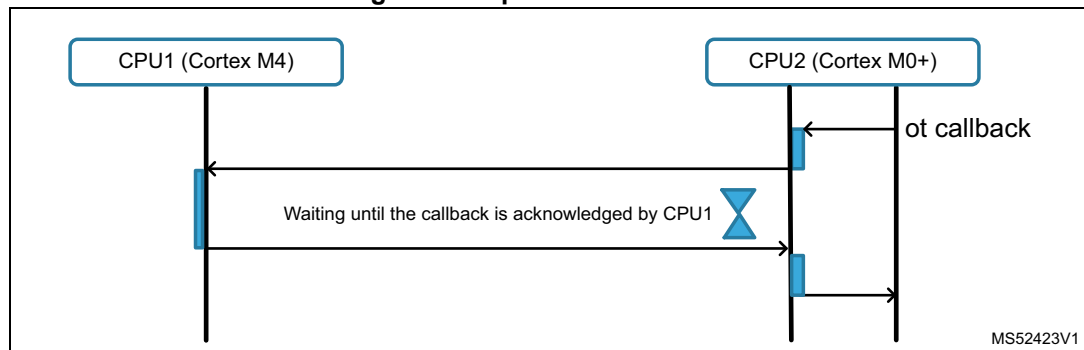


Figure 40. OpenThread callback



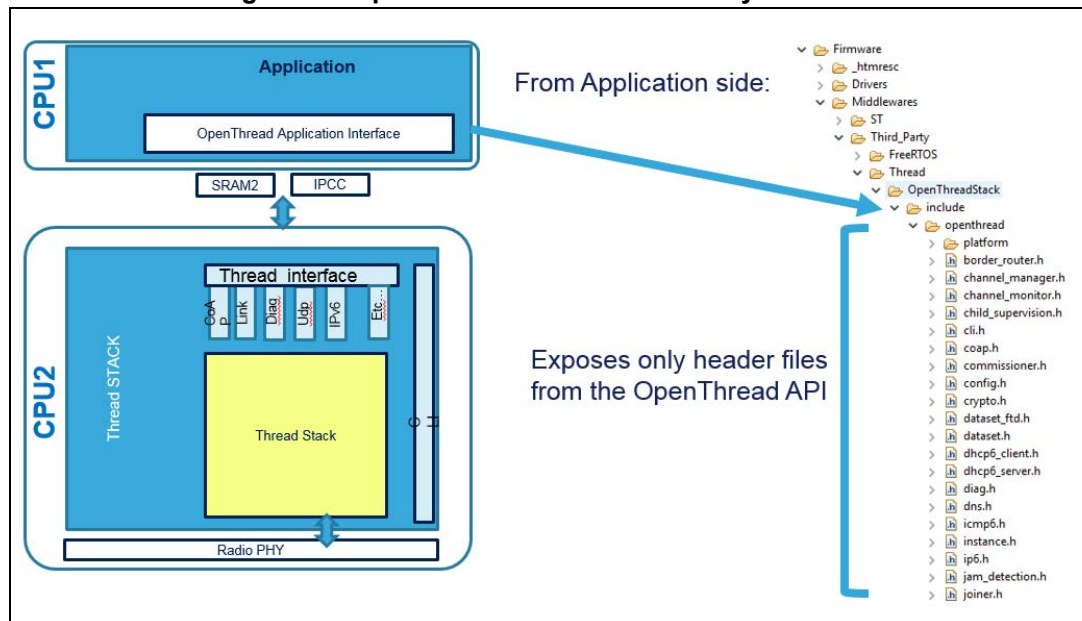
10.6 OpenThread API

OpenThread defines several APIs addressing different services at different level inside the stack:

- Functions used to manage CoAP services: `otCoapStart()`, `otCoapSendRequest()`
- Functions used to manage UDP datagrams: `otUdpOpen()` `otUdpConnect()`
- Functions that manage the radio configuration: `otLinkSetChannel()`
- Functions that manage the IPV6 addresses: `otIp6AddUnicastAddress()`

In total, there are more than 300 functions available. These APIs are described in the `STM32WBxx_OpenThread_API_User_Manual.chm` available in the STM32WB firmware package.

Figure 41. OpenThread stack API directory structure



10.7 Usage of the OpenThread APIs

The OpenThread APIs can be used as if the system is running on a single processor. The Thread interface hides all the multicore mechanisms (IPCC, shared memory), allowing the CPU1 to access to the OpenThread stack running on CPU2.

There are nevertheless two specificities linked to the way the STM32WB implements the OpenThread interface, described in the next subsections.

10.7.1 OpenThread instance

A lot of OpenThread APIs use the parameter `alInstance` as input, which defines the OpenThread instance, in bold in the example of the function `otThreadSetEnabled()` below:

otThreadSetEnabled(otInstance *aInstance, bool aEnabled)

In the STWM32WB Thread implementation, the OpenThread instance is directly allocated at the start of CPU2 firmware. CPU1 does not need to take care of this parameter, is always set to NULL (see the bold type in the code fragment below).

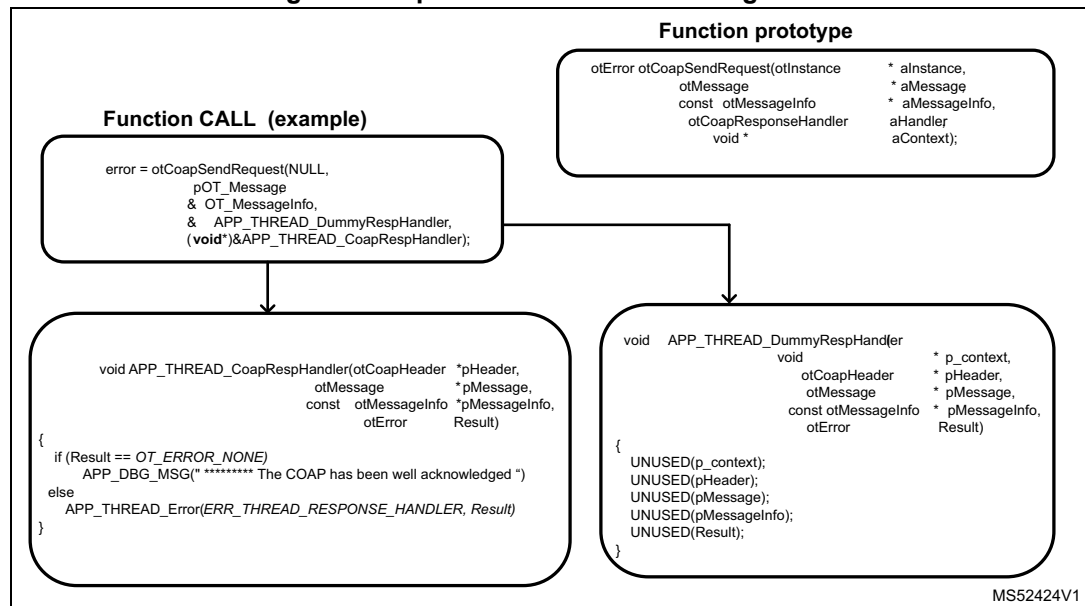
```
error = otThreadSetEnabled(NULL, true);
    if (error != OT_ERROR_NONE)
    {
        APP_THREAD_Error(ERR_THREAD_START, error);
    }
}
```

10.7.2 OpenThread call back management

In the STWM32WB thread implementation, the callbacks passed as parameters inside OpenThread functions do not follow the exact prototype of the standard OpenThread

function. This is due to the dual core architecture constraints. The application callback must be passed in the context parameter, as shown in [Figure 42](#).

Figure 42. OpenThread callback management



Note: The easiest way to see how OpenThread callbacks are managed is to refer to the different applications provided in the STM32WB firmware delivery.

10.8 System commands for Thread applications

Some commands can be called from the Thread applications:

- **SHCI_C2_THREAD_Init():** starts the Thread stack. Called at the end of initialization phase.
- **SHCI_C2_FLASH_StoreData():** stores the nonvolatile Thread data in the flash memory. It is the application that decides when data must be stored (e.g. after the commission phase, or after network configuration).

Note: This operation can take several seconds and must be called only when there is no Thread activity.

- **SHCI_C2_FLASH_EraseData():** Erases the nonvolatile Thread data from the flash memory.

Note: This operation can take several seconds and must be called only when there is no Thread activity.

- **SHCI_C2_CONCURRENT_SetMode():** enables or disables Thread activity on CPU2 for Concurrent mode.
- **SHCI_C2_RADIO_AllowLowPower():** allows or forbids the 802_15_4 radio IP from entering in Low-power mode.
- **SHCI_GetWirelessFwInfo():** reads the informations relative to the loaded wireless binary.

10.8.1 Non-volatile Thread data

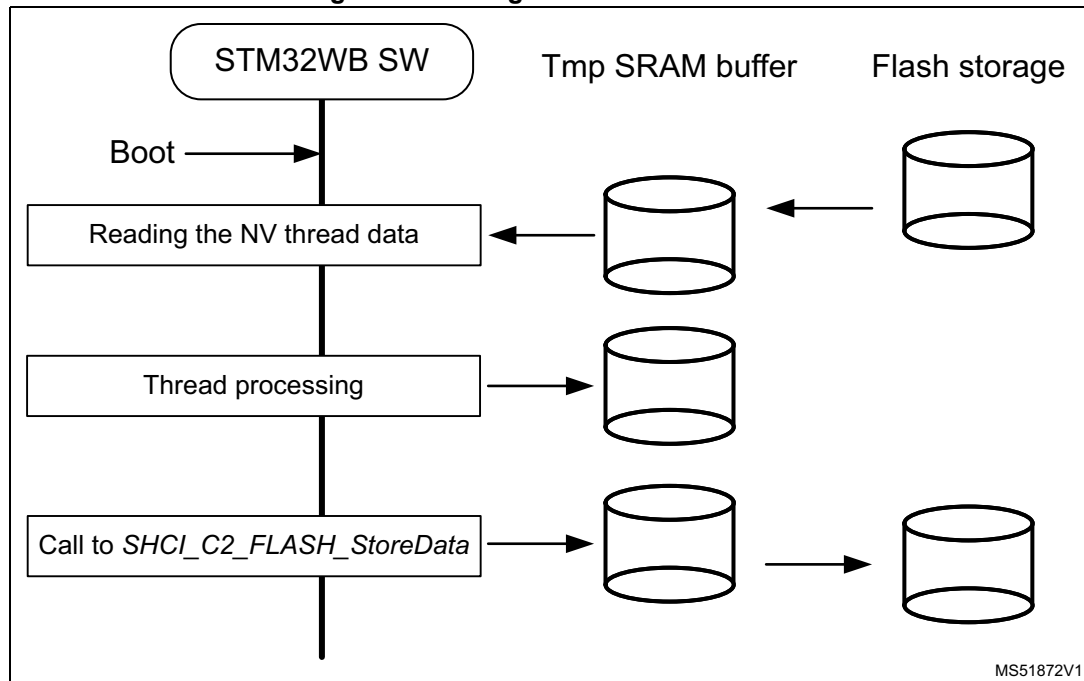
According to the Thread specification, several values must be stored in the flash memory to reuse them later on. These values concern the following entities:

1. Active operational dataset:
Written whenever a new active operational dataset is received. This only occurs when the commissioner or other external entity updates the active operational dataset.
2. Pending operational dataset:
Written whenever a new pending operational dataset is received. This only occurs when the commissioner or other external entity updates the pending operational dataset.
3. Network info:
Written whenever the device role changes (i.e. detached, child, router, leader). Written whenever the MAC and/or MLE frame counter increments beyond a specific threshold.
4. Parent info:
Written whenever a child attaches to a parent.
5. Child info:
Written whenever a child is added/removed from the child table.

After reset, the nonvolatile Thread dataset is automatically read from the flash memory. During run time, OpenThread regularly stores and updates this nonvolatile data in an internal SRAM buffer (see [Figure 43](#)). It is up to the application to force the copy of this nonvolatile data to the flash memory, using the function `SHCI_C2_FLASH_StoreData()`. This operation blocks the access to the flash memory (and so to the CPU), hence it must be done when there are no real time constraints (for instance after a Thread stop).

Note: The function `SHCI_C2_FLASH_StoreData()` is automatically triggered after a call to `otInstanceReset()` or `otInstanceFactoryReset()`.

Figure 43. Storage of non-volatile data



10.8.2 Low-power support

To reach the minimal power consumption, the device must be put in SED (sleepy end device) mode, waking up to poll for messages from its parent or to send data. For most of the time the device sleeps and enters automatically in Low-power mode. Low-power Thread devices can sleep and operate on battery power for years.

When the system is in Low-power mode, as soon as the application sends an otCmd, the system wakes up, executes the command and goes back in Low-power mode. If an application sends multiple otCmd sequentially, the system wakes up and goes back to sleep at high frequency. To avoid the risk of having multiple unneeded short wake-up/sleep cycles, the application allows or forbids the radio to enter Low-power mode via the function `SHCI_C2_RADIO_AllowLowPower()`. An example of this function is provided in the application named `Thread_SED_Coap_Multicast`.

11 Step by step design of an OpenThread application

This section provides information and code examples on how to design and implement an OpenThread application on a STM32WB device.

11.1 Initialization phase

Several steps are mandatory to initialize the STM32WB Thread application:

- Initialize device (HAL, reset device, clock and power configuration)
- Configure platform (e.g. button, LED)
- Configure hardware (e.g. UART, debug)
- Start CPU2, then have it send System notification to the application (on CPU1 side)
- On receiving the notification, the application starts the Thread configuration.

11.2 Set-up the Thread network

In the Thread applications provided in the firmware package, the setup of the Thread network is always done with the same function: `APP_THREAD_DeviceConfig()`

This function processes the following steps:

- Erase persistent Thread parameters to perform a clean start: `otInstanceErasePersistentInfo()`
- Register application callbacks that are called by the OpenThread stack when the role of the node is changed. (e.g. when the node becomes router). Operation performed using `otSetStateChangedCallback()`
- Set the channel: `otLinkSetChannel()`
- Set the PANID: `otLinkSetPanId()`
- Enable IPv6 communication: `otIp6SetEnabled()`
- Start the CoAP server: `otCoapStart()`
- Add a CoAP resource to the CoAP Server: `otCoapAddResource()`
- Start the Thread protocol operation: `otThreadSetEnabled()`

After these steps

- If the board is first one on this Thread network, the node becomes a leader (green LED on in Thread for example).
- If a leader is already present on the network, the node joins as a router or child (red LED on in Thread for example).

11.3 CoAP request

The constrained application protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and networks (such as low-power, lossy).

CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the web such as URIs and Internet media types.

Note: This section does not detail all the OpenThread APIs related to CoAP, but gives an overview of the main functions and abstractions available as part of STM32WB Thread examples.

11.3.1 Creating an otCoapResource

The Coap resource has the following structure:

```
typedef struct otCoapResource
{
    const char *          mUriPath; ///< The URI Path string
    otCoapRequestHandler mHandler; ///< The callback for handling a
    received request
    void *                mContext; ///< Application-specific context
    struct otCoapResource *mNext;   ///< The next CoAP resource in the list
} otCoapResource;
```

This structure can be initialized as shown in the different Thread Coap application examples provided inside the firmware package

```
#define C_RESSOURCE "light"
static otCoapResource OT_Ressource = {C_RESSOURCE,
APP_THREAD_CoapRequestHandler, "MyOwnContext", NULL};
```

Up to 100 different resources can be allocated in parallel per device.

11.3.2 Sending a CoAP request

The application Thread_Coap_Generic proposes an abstraction layer to facilitatesending CoAP requests. This is done using the following function:

```
static void APP_THREAD_CoapSendRequest(otCoapResource* pCoapRessource,
    otCoapType CoapType,
    otCoapCode CoapCode,
    const char *Address,
    uint8_t* Payload,
    uint16_t Size)
```

11.3.3 Receiving a CoAP request

It is called when the server receives a CoAP request.

The prototype of a CoAP request handler is as follows:

```
static void APP_THREAD_CoapRequestHandler(otCoapHeader * pHeader,
    otMessage * pMessage,
    const otMessageInfo * pMessageInfo)
```

In this function, the user can read the received message by calling the following function:

```
otMessageRead()
```

If the type of the message is confirmable (using otCoapHeaderGetType()), a CoAP response must be sent (see [Section 12.3: Thread_Coap_Generic](#) for examples).

11.4 Commissioning

Commissioning requires a device with the commissioner role, and another with the joiner role. The commissioner is either a Thread device in an existing network, or a device external to the Thread network (such as a mobile phone) that performs the role.

The joiner is the device wishing to join the Thread network.

A Thread commissioner is used to authenticate a device on the network. It does not transfer, nor have possession of Thread network credentials such as the master key.

This document covers basic, on-mesh commissioning without an external commissioner or border router.

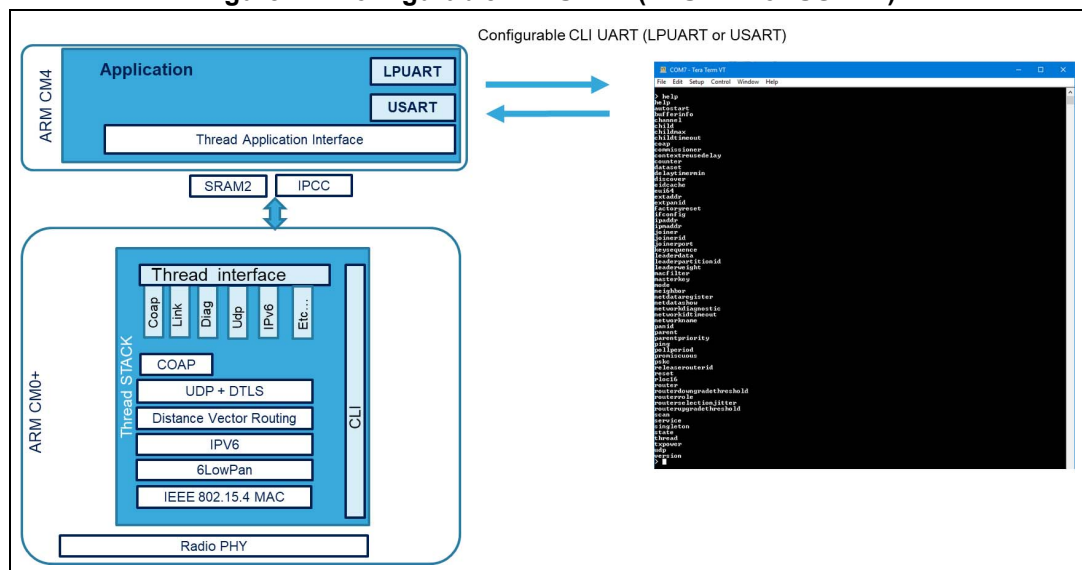
Thread_Commissioning application demonstrates a simple commissioning example.

11.5 CLI

The OpenThread stack exposes configuration and management APIs via a command line interface.

The certification environment (GRL test harness) uses the CLI to execute test cases.

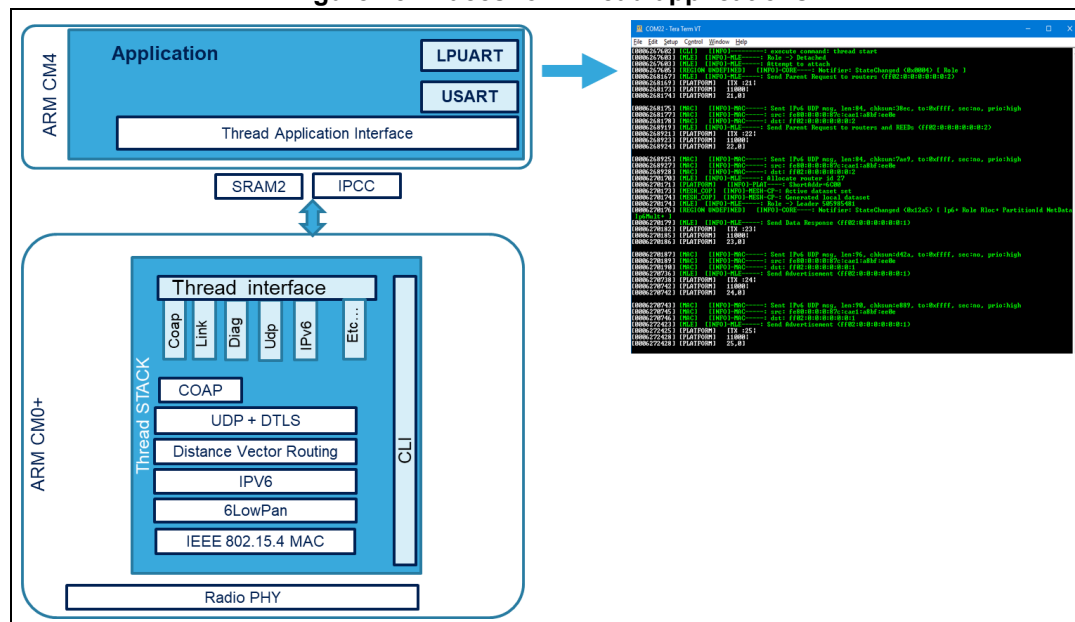
Figure 44. Configurable CLI UART (LPUART or USART)



11.6 Traces

Traces from CPU1 and CPU2 applications are routed to the UART (configured at compilation time using `app_conf.h` file).

Figure 45. Traces for Thread applications



OpenThread stack trace levels are dynamically configurable via OpenThread APIs using `otSetDynamicLogLevel()`.

12 STM32WB OpenThread application

12.1 Thread_Cli_Cmd

This application shows how to control the OpenThread stack via Command lines.

CLI commands are sent via a UART from an HyperTerminal (PC) to the STM32WBxx_NUCLEO board.

This is the application used for certification process.

12.2 Thread_Coap_DataTransfer

This application demonstrates how to transfer large blocks of data using CoAP messaging protocol.

In this application the mesh-local scope and multicast addressing types are used to probe the mesh-local IP addresses of the child devices to which the file is transferred.

Nodes are split into two forwarding roles: router or end device.

In this application, which uses two devices, one acting as a leader (router) and the other one as an end device (child mode).

After the reset of the two boards, one of them is in leader mode (green LED2 on) and the other one in child mode (red LED3 on).

Once the child mode is established for one of the devices, it starts the provisioning procedure in multicast mode to probe the IP address of the leader device.

Then this is used to start the file transfer procedure in unicast mode, the successful operation is signaled by the lightening of the blue LED.

12.3 Thread_Coap_Generic

This application demonstrates the use of CoAP messages. It provides the abstraction level to send CoAP multicast request.

This application requires two STM32WB boards. The objective is to demonstrate both boards exchanging CoAP messages with each other. In this application, one board is acting as leader and the other one is acting as end device or router.

12.4 Thread_Coap_Multiboard

This application shows how to use CoAP to send messages to multiple boards in a unicast way. It is designed to use from two to five STM32WBxx_NUCLEO boards.

The purpose of this application is to create a small Thread mesh network.

When the setup is correctly configured, a CoAP request is automatically and continuously transferred from one board to next in the following order:

- Board 1 → Board 2 → (...) → Board n → Board 1 → ...

To each board is associated a specific IPv6 address:

- fdde:ad00:beef:0:442f:ade1:3fc:1f3a for board number 1
- fdde:ad00:beef:0:442f:ade1:3fc:1f3b for board number 2
- fdde:ad00:beef:0:442f:ade1:3fc:1f3c for board number 3 if present
- fdde:ad00:beef:0:442f:ade1:3fc:1f3d for board number 4 if present
- fdde:ad00:beef:0:442f:ade1:3fc:1f3e for board number 5 if present

12.5 Thread_Commissioning

This application demonstrates the commissioning process between a commissioner and a joiner.

It shows a device distributing its Thread parameters (channel, panid, masterkey) to another device using to the commissioning process.

This application requires two STM32WBxx_NUCLEO boards.

One device acts as commissioner and the other one as joiner.

In this application, the commissioner accepts a newcomer in its Thread network.

12.6 Thread_FTD_Coap_Multicast

This application demonstrates the use of CoAP multicast message for Full Thread devices.

Note: To be used with Thread FTD CPU2 binary.

This application uses two devices, one device acts as a leader (router) and the other one as end device (child mode).

After the reset of the two boards (named respectively A and B), one board is in leader mode (green LED2 on), the other one in child mode (red LED3 on).

To send a CoAP command from board A to board B, press the SW1 pushbutton on board A. Board B receives the CoAP command to turn on its blue LED1. Pressing again the same push-button turns the blue LED1 off.

Same CoAP commands can be sent from board B to board A.

12.7 Thread_SED_Coap_Multicast

This application demonstrates the use of CoAP multicast message sent from a sleepy end device.

Note: To be used with Thread MTD CPU2 binary.

Two boards are needed for the proposed use case:

- One board acts as a leader (router) in FTD mode (board A)
- The other one acts as a SED in MTD mode (board B).

The board acting as a leader must be flashed with the FTD application: use the application "Thread_FTD_Coap_Multicast" + Thread FTD binary on CPU2.

The board acting as a SED must be flashed with the MTD application “Thread_SED_Coap_Multicast” + Thread MTD binary on CPU2.

After the two boards are reset, one board (A) automatically reaches the leader mode (green LED2 on) and the other one (B) the SED mode (red LED3 on) a few seconds later.

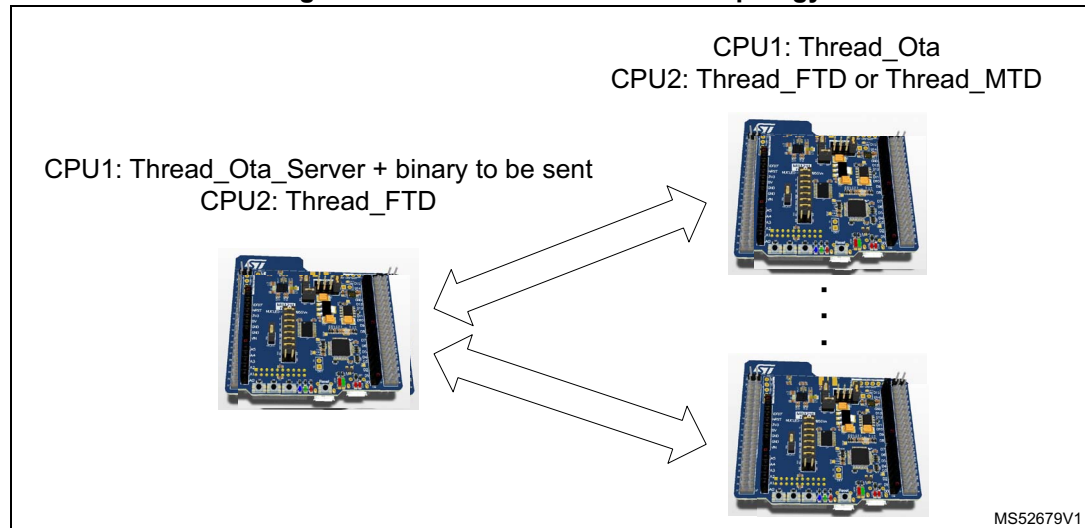
At this stage, these two boards belong to the same Thread network and device 2 sends a CoAP multicast request every second to device 1 to turn on/off its blue LED.

12.8 Thread FUOTA

12.8.1 Principle

The goal is to use Thread protocol to update CPU1 application binary or CPU2 wireless coprocessor binary on a remote device.

Figure 46. Thread FUOTA network topology



This thread requires at least two STM32WBxx boards (see [Figure 46](#)) running Thread protocol with specific applications:

- one board running Thread_Ota_Server application
- one or more boards running Thread_Ota application

FUOTA process can take place only on one device at a time.

The server initiates a FUOTA provisioning process and one client must respond to it. Multiple clients are updated one at the time.

12.8.2 Memory mapping

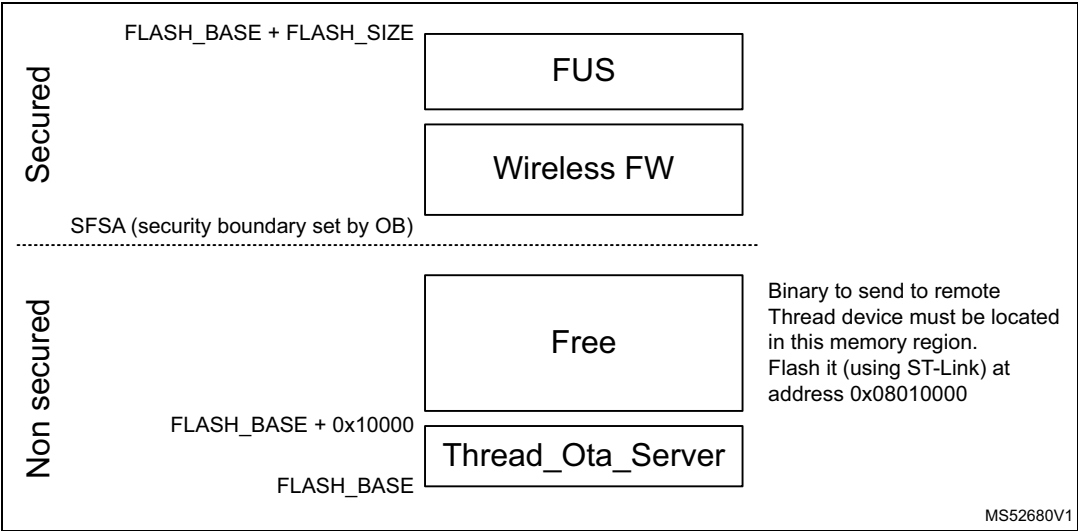
Server side

The binary file to be installed (either for CPU1 or for CPU2 update) on remote device has to be flashed first on the “FREE” memory region on the Server side (see [Figure 47](#)).

Maximum size of the binary to be transferred is equal to:

FREE region size = SFSA Address - (FLASH_BASE - 0x8010000)

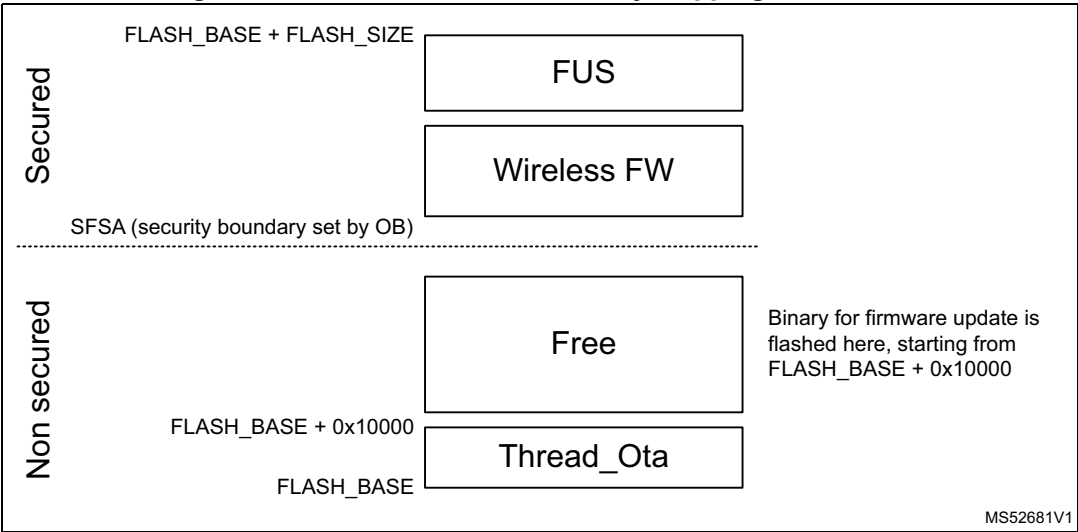
Figure 47. OTA server (Thread_Ota_Server) flash memory mapping



Client side

On the client side, before receiving the binary from the server, the flash memory is as shown in [Figure 48](#).

Figure 48. FUOTA client flash memory mapping initial state



After it has received the binary data from server side, the flash memory is updated as shown in [Figure 49](#) and [Figure 50](#), respectively, for CPU1 binary transfer and CPU2 binary transfer.

Figure 49. FUOTA server flash memory mapping after CPU1 binary transfer

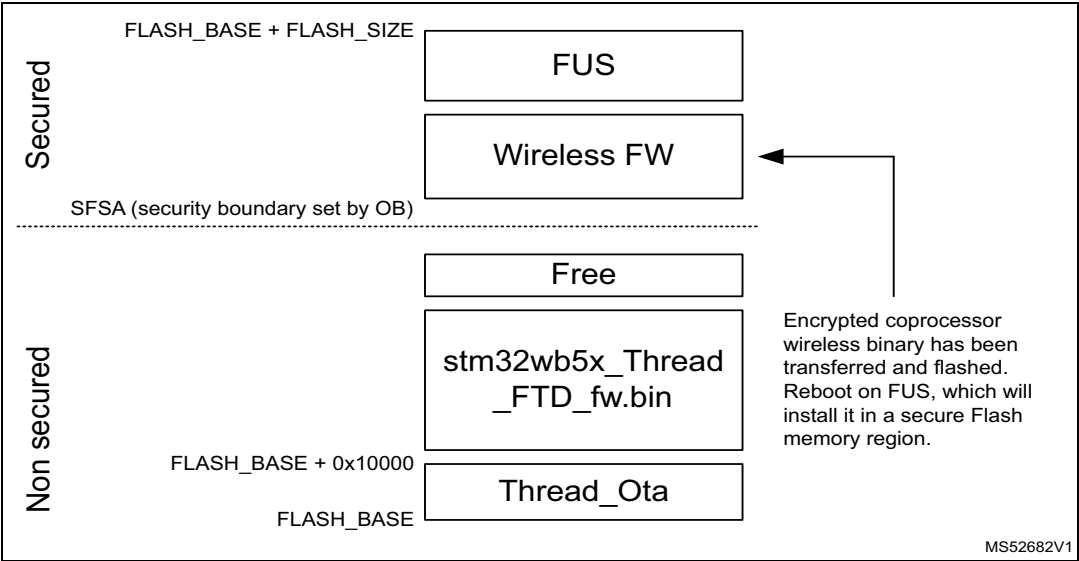
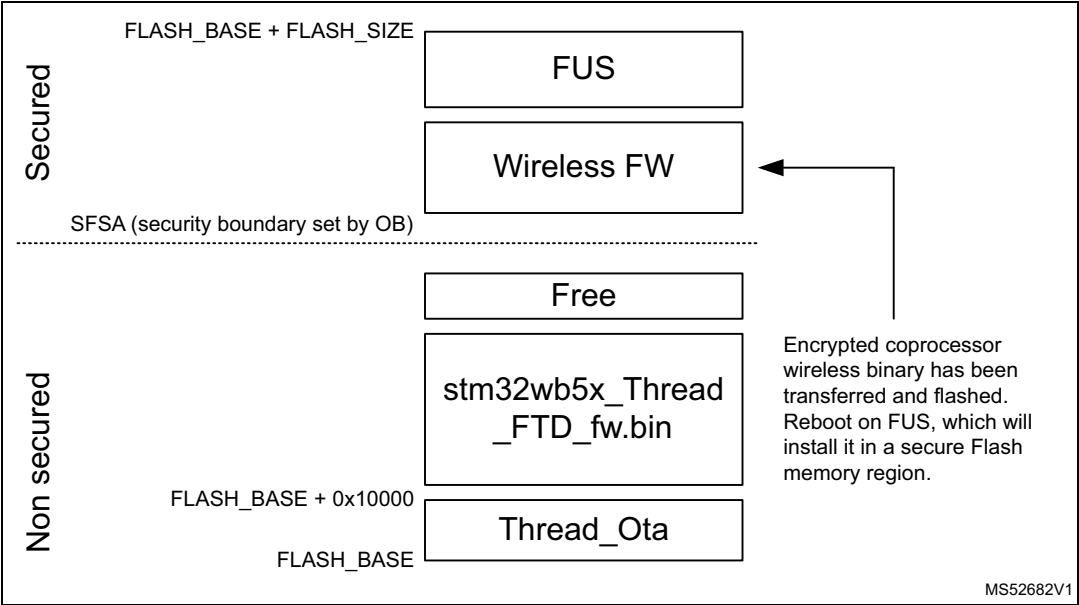


Figure 50. FUOTA server flash memory mapping after CPU2 binary transfer

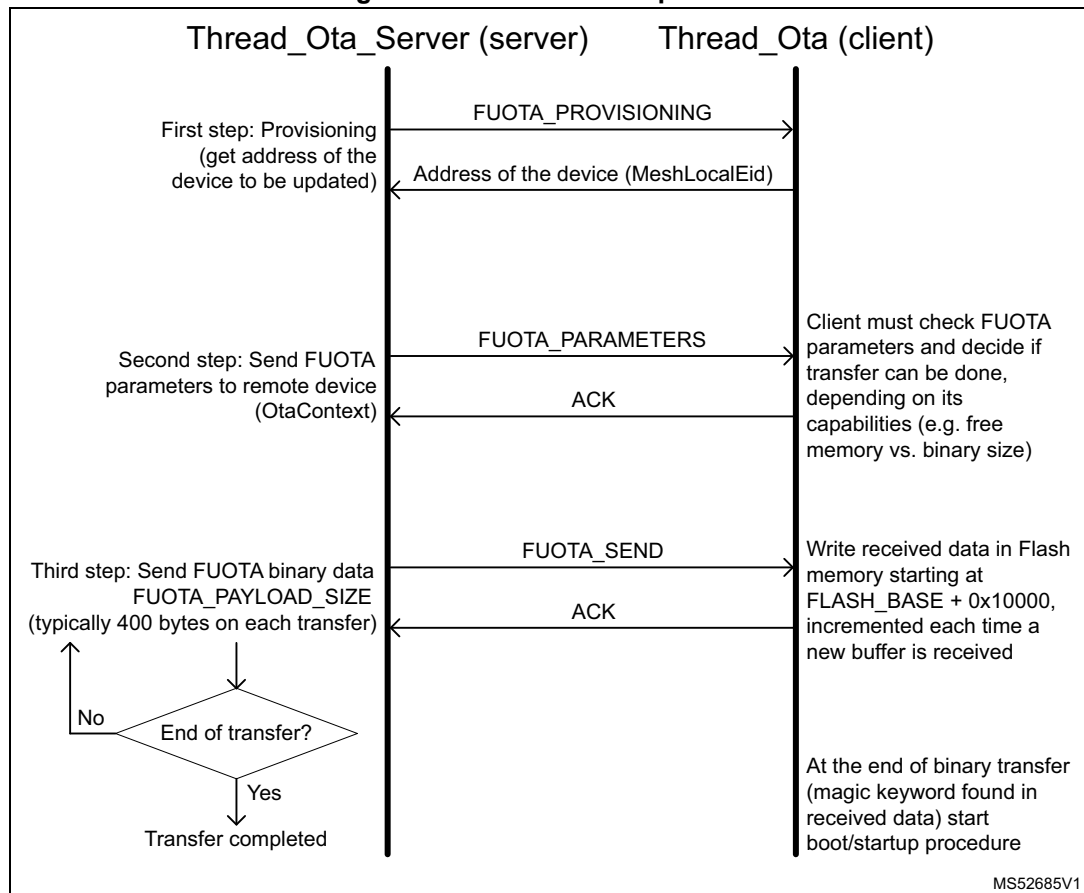


12.8.3 Thread FUOTA protocol

This is a STMicroelectronics proprietary protocol to update CPU2 wireless coprocessor binary or CPU1 FW application using Thread, based on CoAP request.

[Figure 51](#) details the steps to perform the firmware update transfer.

Figure 51. Thread FUOTA protocol



MS52685V1

1. Server sends a message to record the address of the remote device on which FUOTA is processed.
Server sends a multicast, Non-Confirmable, Get CoAP request on resource: "FUOTA_PROVISIONING"
The remote device answers with the Mesh Local Eid (Endpoint Identifier), which identifies a Thread interface, independent from the network topology.
2. `OtaContext` data structure is sent to the remote device. It contains:
 - File type: `FW_APP` update or `FW_COPRO_WIRELESS` update
 - Binary size: size in bytes of the binary to be transferred
 - Base address: start address in flash memory for remote device to copy binary data to
 - Magic keyword: keyword specifying end of the binary
3. Transfer binary to remote device.
The transfer is performed by buffers of `FUOTA_PAYLOAD_SIZE` (default 400 bytes). This is configurable on Server side.

For each transfer, Thread_Ota_Server waits for acknowledgment from the remote device before continuing the next buffer transfer.

On the remote side, each data buffer received is written to flash memory.

When magic keyword is found, it means that this is the last buffer to transmit.

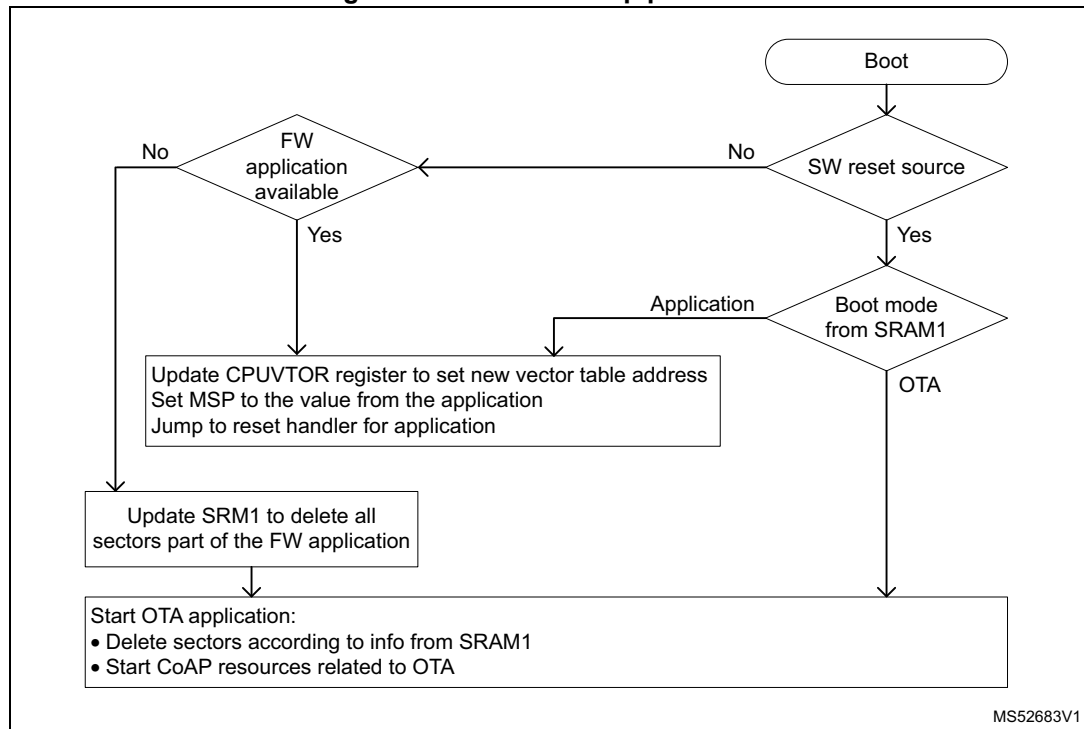
12.8.4 FUOTA application startup procedure

Once binary data has been transferred to the remote device (Thread FUOTA client), the startup procedure is different for the update of a CPU1 application or of CPU2 coprocessor wireless binary.

FUOTA for CPU1

On the client side after the binary transfer is completed, the process shown in [Figure 52](#) takes place to jump on OTA specific application (example: Thread_Coap_Generic_Ota):

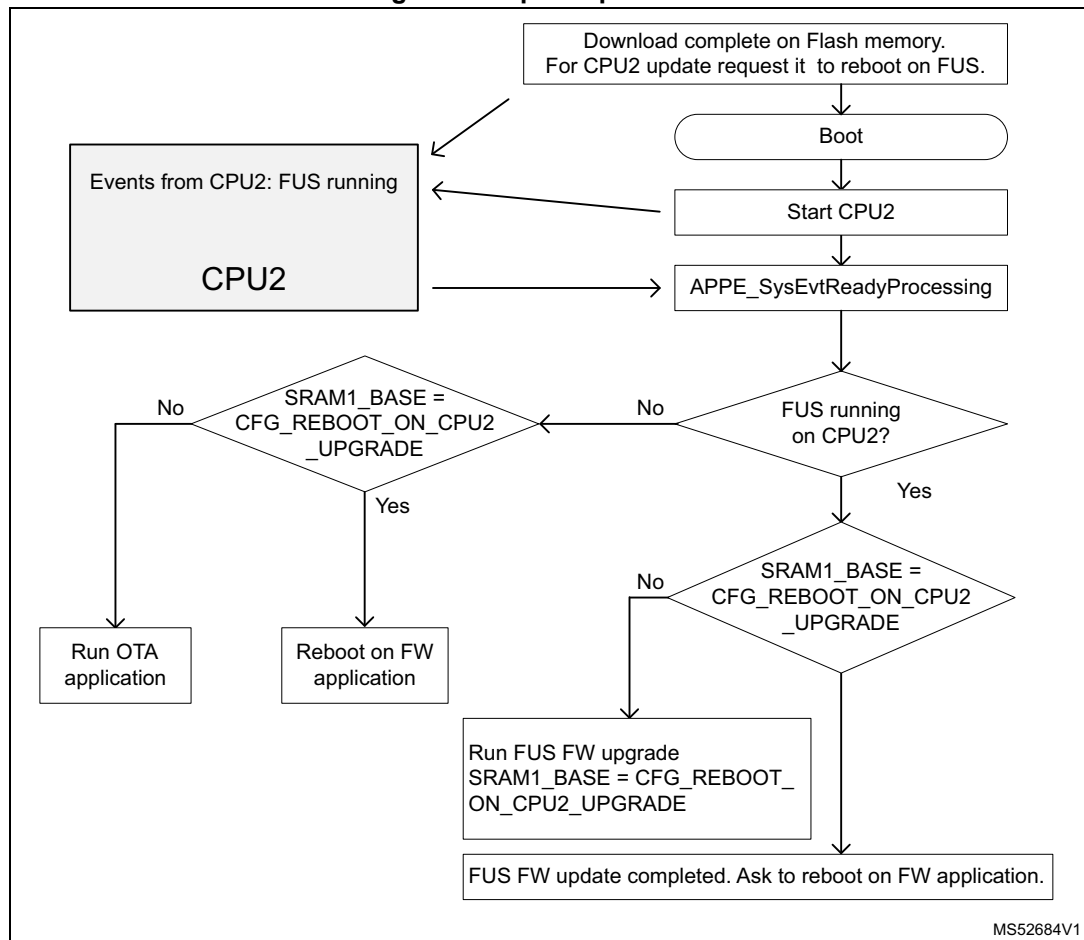
Figure 52. FUOTA startup procedure



FUOTA for CPU2

CPU2 update involves FUS (firmware upgrade service) software component, which is responsible to decrypt and install secure binary.

Figure 53. Update procedure



12.8.5 Applications

Thread_Ota_Server

This application must be loaded on STM32WB 1Nucleo board acting as FUOTA server.

Thread_Ota

This application must be loaded on STM32WB Nucleo board acting as FUOTA client.

Thread_Coap_Generic_Ota

This application is almost identical to Thread_Coap_Generic, the differences are:

- Use special tags (to manage end data transfer and data consistency):
 - TAG_OTA_END: the Magic keyword value is checked in the thread_ota application
 - TAG_OTA_START: the Magic keyword address shall be mapped at 0x140 from start of the binary image

Therefore, by reading memory content at 0x140 it must be equal to the Magic keyword value.

- Scatter file must be updated to place the sections above

Example for IAR:

```
Vector table and ROM start @ moved to 0x08010000:
define symbol __ICFEDIT_intvec_start__ = 0x08010000;
define symbol __ICFEDIT_region_ROM_start__ = 0x08010000;
define region OTA_TAG_region = mem:[from
(__ICFEDIT_region_ROM_start__ + 0x140) to
(__ICFEDIT_region_ROM_start__ + 0x140 + 4)];
keep { section TAG_OTA_START};
keep { section TAG_OTA_END };
place in OTA_TAG_region { section TAG_OTA_START };
place in ROM_region { readonly, last section TAG_OTA_END };
```

13 MAC IEEE Std 802.15.4-2011

13.1 Overview

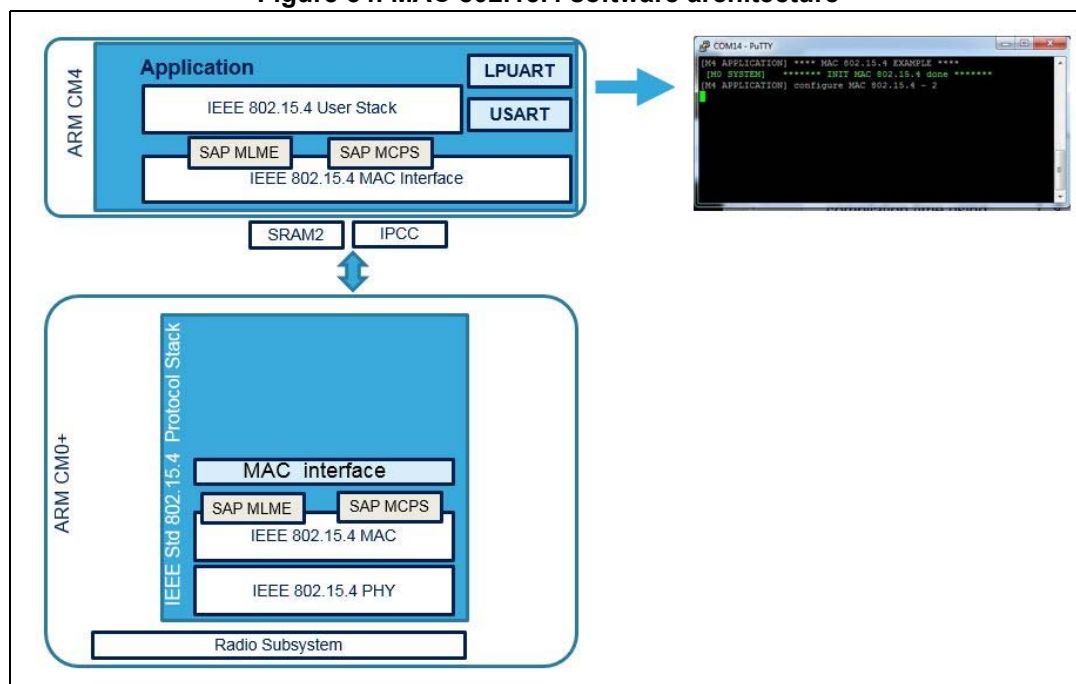
MAC IEEE Std 802.15.4-2011 layer is embedded by the MAC dedicated firmware running on CPU2 core (Radio protocol processor). The MAC layer relies on the PHY layer, which addresses the RF subsystem component.

As this implementation is provided in binary format and running on CPU2, the MAC API is exposed to CPU1 core to let user address MAC service access points. The user can then set up its STM32WB device as an FFD (full feature device, or coordinator), or as an RFD (reduced feature devices, or nodes) as described in IEEE Std 802.15.4-2011 specification document.

13.2 Architecture

Figure 54 shows the MAC software architecture used when the customer expects to implement an in house 802.15.4 network by integrating a custom solution or a third party solution on the application processor.

Figure 54. MAC 802.15.4 software architecture

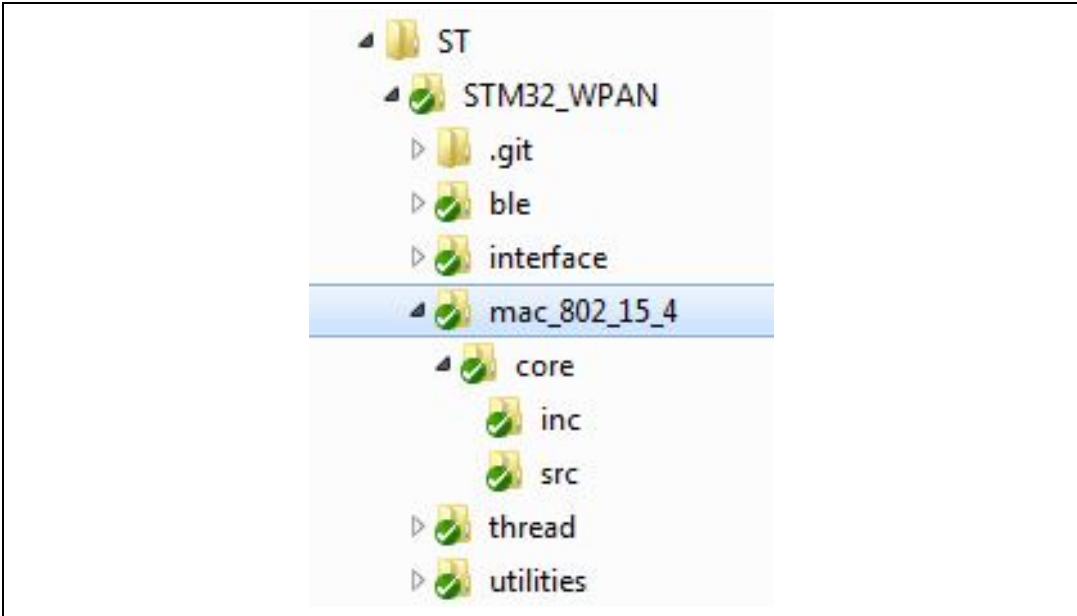


13.3 API

The MAC IEEE Std 802.15.4-2011 specification document defines an interface between 802.15.4 network layers and the medium access control layer. This API allows the user to address the MAC management entity service called MLME (MAC sub - Layer management entity) as the MAC data service called MCPS (MAC common part sub layer entity).

A MAC API dedicated to the application core with its associated implementation is available from the middleware provided under \Middlewares\ST\STM32_WPAN\mac_802_15_4 (see [Figure 55](#)).

Figure 55. MAC API dedicated to application core



This implementation is documented in STM32WBxx_MAC_802_15_4_User_Manual.chm available under Firmware\Middlewares\ST\STM32_WPAN\mac_802_15_4 directory of the STM32WB FW package. Detailed primitive descriptions are accessible through the IEEE Std 802.15.4-2011 document.

13.4 How to start

13.4.1 Board configuration

Ensure that the option bytes are set as in [Figure 56](#).

Figure 56. Option bytes configuration for MAC 802.15.4



13.4.2 MAC radio protocol processor CPU2 firmware

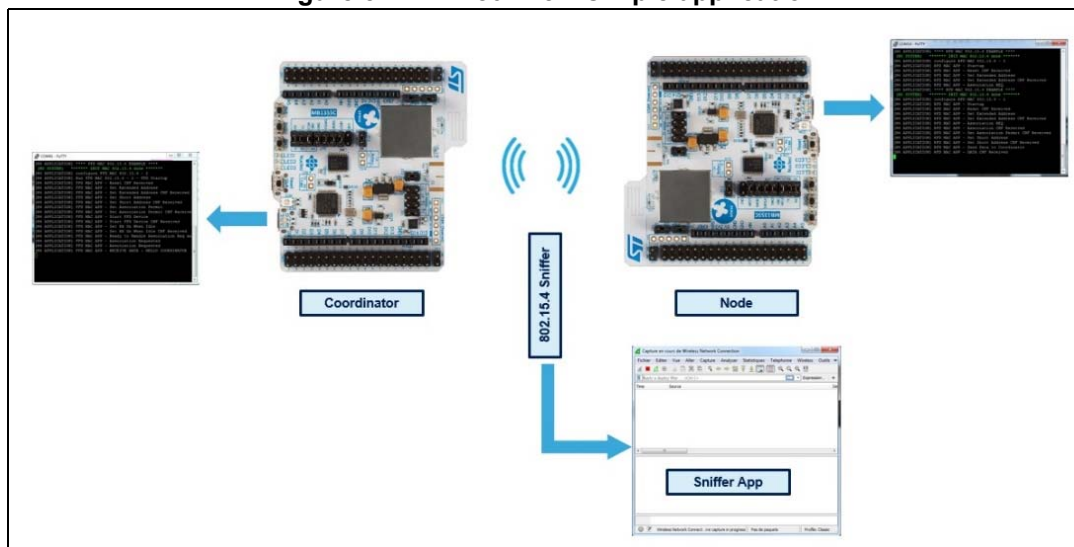
The user first needs to download the appropriate dedicated MAC firmware binary for CPU2 radio protocol core, see [Release_Notes.html](#) located in `Firmware\Projects\STM32WB_Copro_Wireless_Binaries` directory of the STM32WB FW package.

13.4.3 MAC application processor firmware

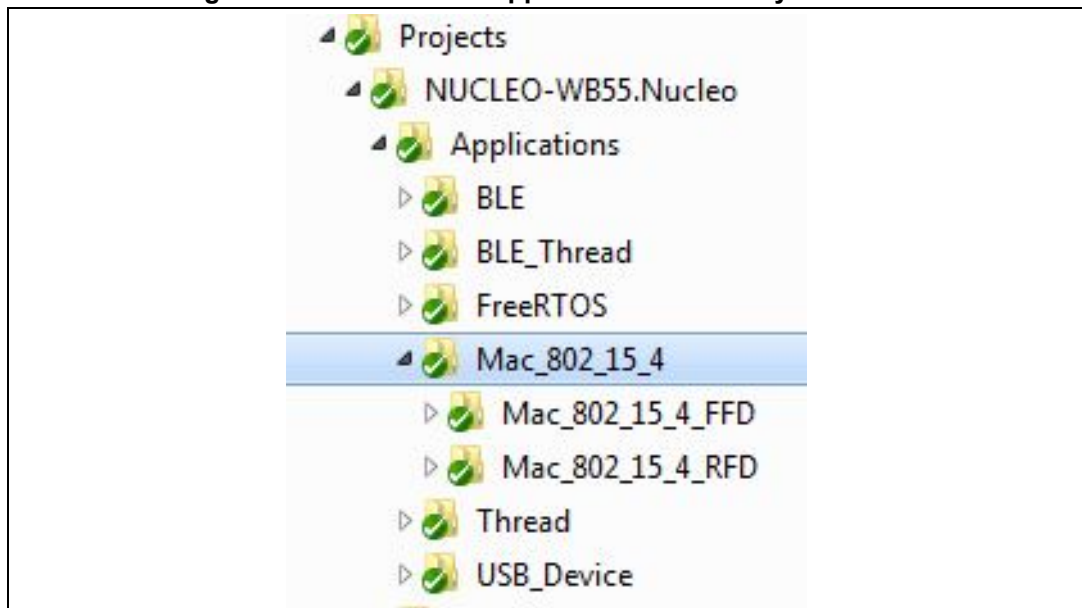
Before implementing a custom stack solution or integrating a third party stack provided for CPU1 application core MAC API, the user can ramp-up with the MAC application example referring to the two following applications that have to run simultaneously on two STM32WB boards:

- `Mac_802_15_4_FFD`: shows how to implement a simple 802.15.4 coordinator. This device manages the network as association request and gets or provides data on node demand.
- `Mac_802_15_4_RFD`: shows how to implement a simple 802.15.4 node. This device emits an association request to the coordinator. Once the addressed coordinator positively responds to the request, the node receives its new short address and then emits data to the coordinator.

Figure 57. MAC 802.15.4 simple application



Both applications, dedicated to Nucleo STM32WB boards, are available from `NUCLEO-WBxx.Nucleo application Mac_802_15_4` directories (see [Figure 58](#)).

Figure 58. MAC 802.15.4 applications - Directory structure

A readme.txt file describes the MAC sequence handled by each 802.15.4 devices. The files are available from each root projects.

13.4.4 Output

The user can use an OTA sniffer, on the right channel, to listen to the negotiation between the two boards during the association phase and to watch the data exchange once the node is registered in network managed by the coordinator.

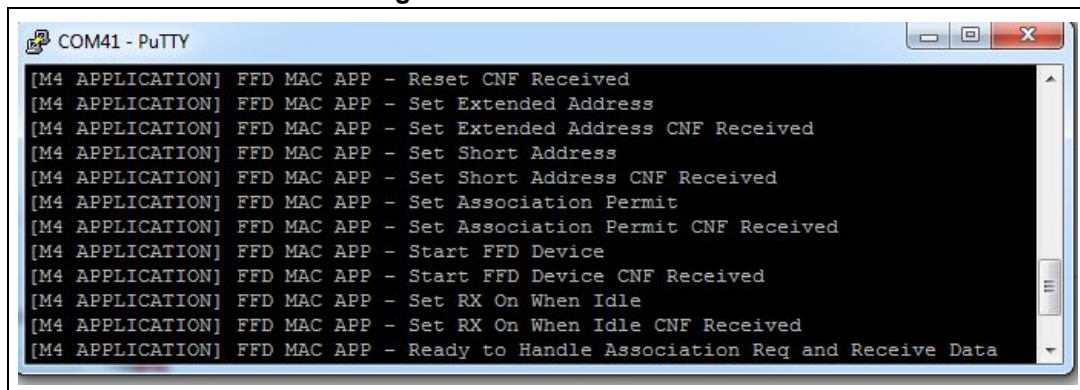
Application traces are routes to the UART. User can then start a HyperTerminal session, using a preferred terminal emulator, on each of the implemented Virtual COM port to check every MAC step.

The TTY Session configuration to connect console:

- Baud: 115200
- Data bits: 8
- Stop bits: 1
- Parity: None
- Flow control: XON/XOFF.

Running the two applications leads to the Hyper terminal shown in figures [59](#) to [61](#).

Figure 59. Coordinator start

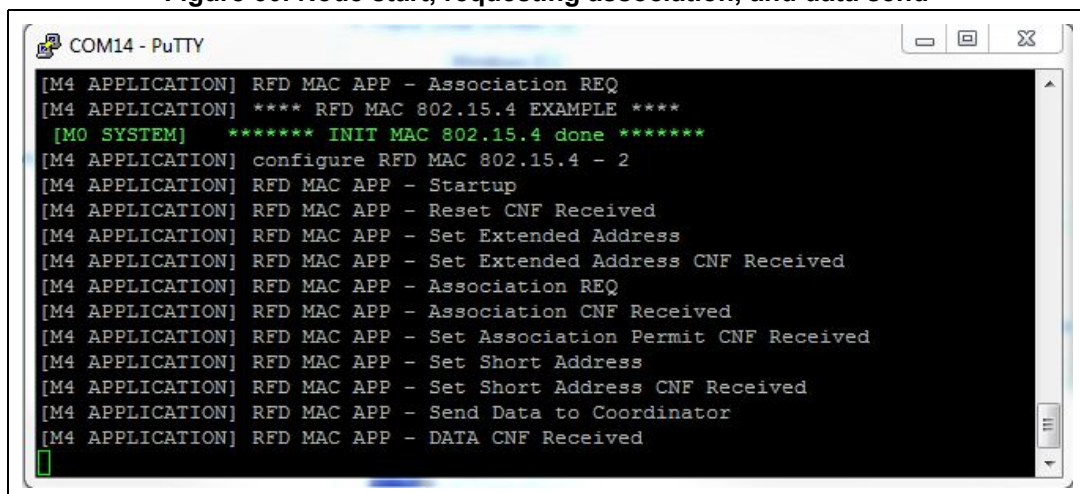


```

[M4 APPLICATION] FFD MAC APP - Reset CNF Received
[M4 APPLICATION] FFD MAC APP - Set Extended Address
[M4 APPLICATION] FFD MAC APP - Set Extended Address CNF Received
[M4 APPLICATION] FFD MAC APP - Set Short Address
[M4 APPLICATION] FFD MAC APP - Set Short Address CNF Received
[M4 APPLICATION] FFD MAC APP - Set Association Permit
[M4 APPLICATION] FFD MAC APP - Set Association Permit CNF Received
[M4 APPLICATION] FFD MAC APP - Start FFD Device
[M4 APPLICATION] FFD MAC APP - Start FFD Device CNF Received
[M4 APPLICATION] FFD MAC APP - Set RX On When Idle
[M4 APPLICATION] FFD MAC APP - Set RX On When Idle CNF Received
[M4 APPLICATION] FFD MAC APP - Ready to Handle Association Req and Receive Data

```

Figure 60. Node start, requesting association, and data send

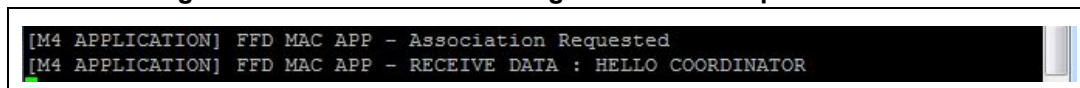


```

[M4 APPLICATION] RFD MAC APP - Association REQ
[M4 APPLICATION] **** RFD MAC 802.15.4 EXAMPLE ****
[M0 SYSTEM] ***** INIT MAC 802.15.4 done *****
[M4 APPLICATION] configure RFD MAC 802.15.4 - 2
[M4 APPLICATION] RFD MAC APP - Startup
[M4 APPLICATION] RFD MAC APP - Reset CNF Received
[M4 APPLICATION] RFD MAC APP - Set Extended Address
[M4 APPLICATION] RFD MAC APP - Set Extended Address CNF Received
[M4 APPLICATION] RFD MAC APP - Association REQ
[M4 APPLICATION] RFD MAC APP - Association CNF Received
[M4 APPLICATION] RFD MAC APP - Set Association Permit CNF Received
[M4 APPLICATION] RFD MAC APP - Set Short Address
[M4 APPLICATION] RFD MAC APP - Set Short Address CNF Received
[M4 APPLICATION] RFD MAC APP - Send Data to Coordinator
[M4 APPLICATION] RFD MAC APP - DATA CNF Received

```

Figure 61. Coordinator receiving association request and data



```

[M4 APPLICATION] FFD MAC APP - Association Requested
[M4 APPLICATION] FFD MAC APP - RECEIVE DATA : HELLO COORDINATOR

```

13.4.5 MAC IEEE Std 802.15.4-2011 system

This is currently the implemented MAC system command.

SHCI_C2_MAC_802_15_4_Init() starts the MAC layer and RF subsystem on radio processor (CPU2).

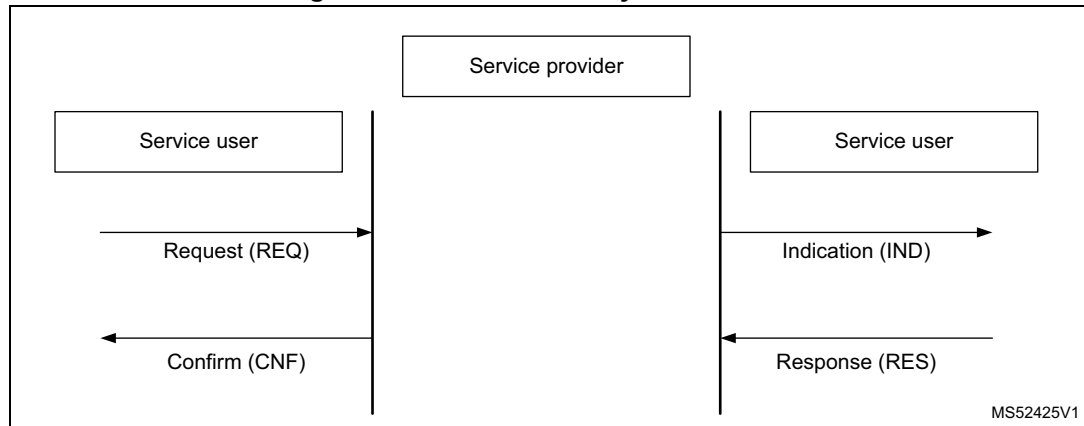
Non-volatile data are not guaranteed by the MAC layer. It is up to the application upper layer to ensure that these data are kept in the flash memory, and to restore those to be used later on.

Low power feature is not supported.

13.4.6 Integration recommendations

The MAC layer offers service primitives by implementing an abstraction layer. This abstraction layer, described in MAC IEEE Std 802.15.4-2011 specification documentation, is illustrated in [Figure 62](#).

Figure 62. MAC 802.15.4 layer abstraction



The proposed API lets the user call REQ and RES primitives with associated defined structure initialized from the upper layer. To get notification from the MAC layer, custom call function have to be implemented called MAC indication (IND) or MAC confirmation (CNF).

Request and Response examples

- Set the short address of the current device
- Call MAC_MLMESetReq with initialized SetReq structure storing the short address to set.

```
// Set Device Short Address
uint16_t shortAddr = 0x1122;
SetReq.PIB_attribute = g_MAC_SHORT_ADDRESS_c;
SetReq.PIB_attribute_valuePtr = (uint8_t*) &shortAddr;
MacStatus = MAC_MLMESetReq( &SetReq );
```

- Respond to an association indication

When an association is requested, a coordinator can respond by providing the response with a short address to the requester:

```
Call MAC_MLMEAssociateRes with initialized AssociateRes structure storing the attributed short address.

APP_DBG("Srv task : Response to Association Indication");
MAC_associateRes_t AssociateRes;
uint16_t shortAssociationAddr = 0x3344;

memcpy(AssociateRes.a_device_address, g_MAC_associateInd.a_device_address, 0x08);
memcpy(AssociateRes.a_assoc_short_address, &shortAssociationAddr, 0x08);
AssociateRes.security_level = 0x00;
AssociateRes.status = MAC_SUCCESS;

MacStatus = MAC_MLMEAssociateRes(&AssociateRes);
```

- Confirmation and indication examples

To be notified of confirmation or indication messages from lower MAC Layer, the user must register custom callbacks in `MAC_callbacks_t macCbConfig` (example provided in `app_ffd_mac_802_15_4.c`):

```
/* Mac Call Back Initialization */
macCbConfig.mlmeResetCnfCb = APP_MAC_mlmeResetCnfCb;
macCbConfig.mlmeScanCnfCb = APP_MAC_mlmeScanCnfCb;
macCbConfig.mlmeAssociateCnfCb = APP_MAC_mlmeAssociateCnfCb;
macCbConfig.mlmeAssociateIndCb = APP_MAC_mlmeAssociateIndCb;
...

```

- Action on data indication

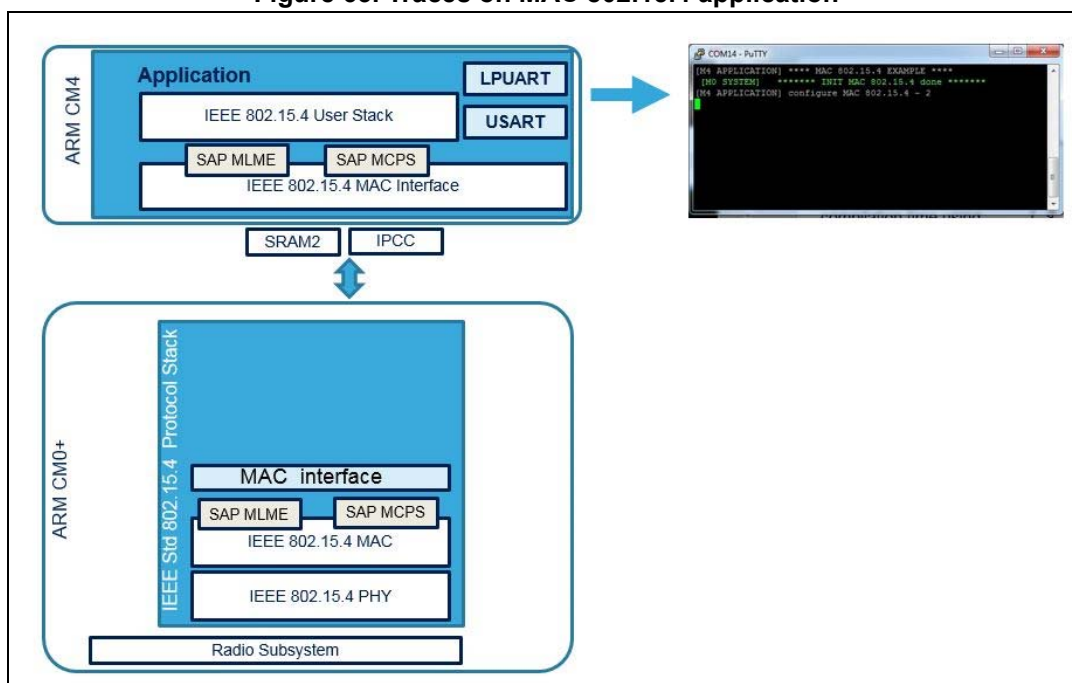
The user must implement custom callbacks to retrieve data from MAC services:

On data indication message from MAC layer, the `macCbConfig.mcpsDataIndCb` is used to call `APP_MAC_mcpsDataIndCb` callback, which can be implemented as follows to retrieve the indication data carried by `MAC_dataInd_t` structure (`app_mac_802-15-4_process.c`):

```
MAC_Status_t APP_MAC_mcpsDataIndCb( const MAC_dataInd_t * pDataInd )
{
    memcpy(&g_DataInd, pDataInd, sizeof(MAC_dataInd_t));
    return MAC_SUCCESS;
}

```

Figure 63. Traces on MAC 802.15.4 application

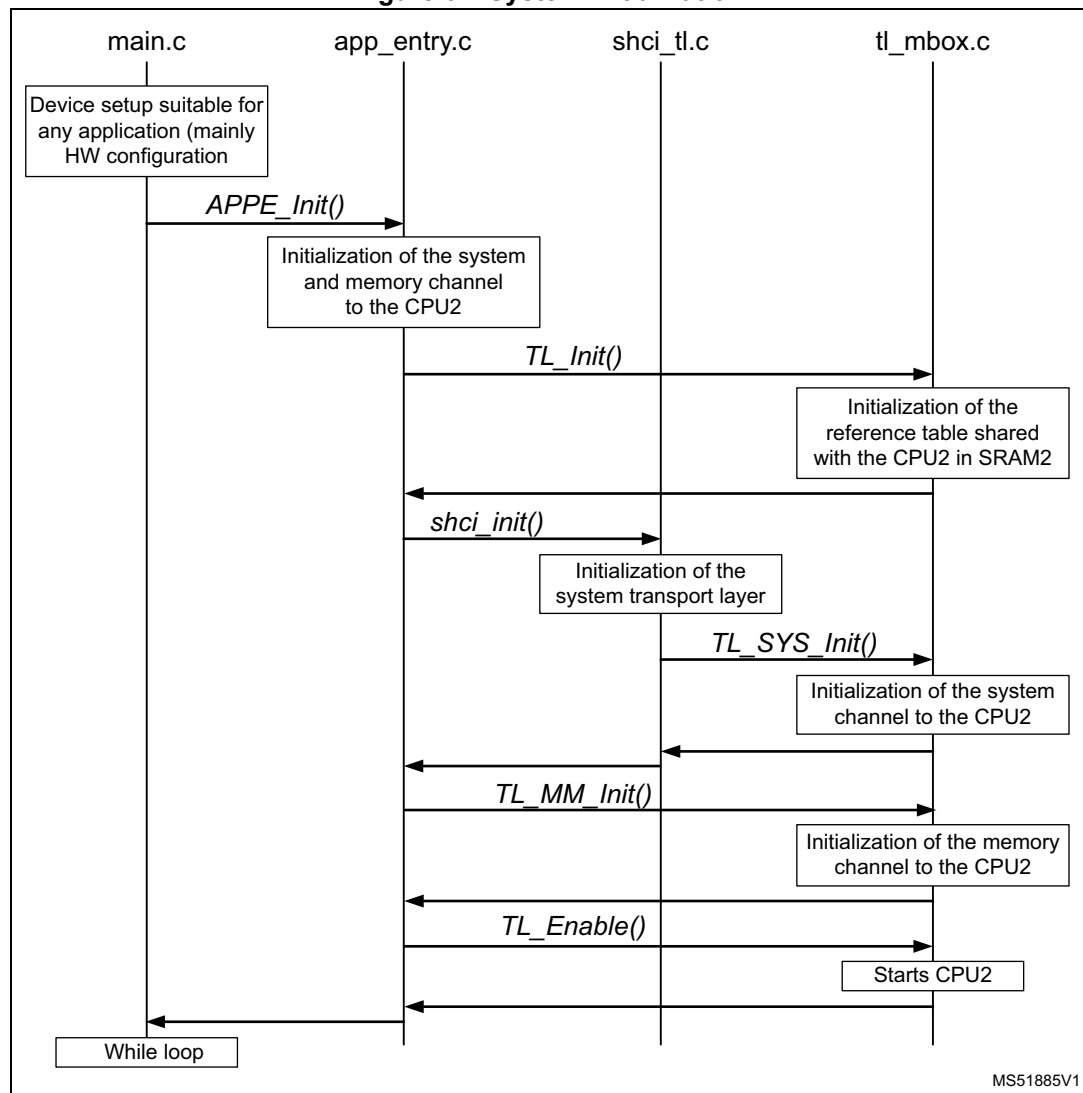


14 Annexes

14.1 Detailed flow of the device initialization

At startup, the device is first initialized and then the system channel to CPU2 is initialized. After this sequence is over, CPU1 returns to the background while loop and waits for the notification from CPU2 that it is ready to receive system command. CPU1 can run other application initializations that are not RF (CPU2) related. This startup is the same whether CPU2 is running a Full BLE host stack, HCI only interface or an OpenThread Stack.

Figure 64. System initialization



MS51885V1

When CPU2 is ready to receive system commands, it sends a notification to CPU1. Upon receiving the notification `shci_notify_async_evt()`, the user must call `shci_user_evt_proc()` to allow the system transport layer to process the event. The user application is notified with `APPE_SysUserEvtRx()` that a system event is received. As the

`shci_notify_async_evt()` is received in the IPCC interrupt handler context, the information is passed to the background so that the `shci_user_evt_proc()` is called from the background while loop (out of any interrupt context). This mechanism is the same whether CPU2 is running a Full BLE host stack, HCI only interface or an OpenThread Stack.

Figure 65. System ready event notification

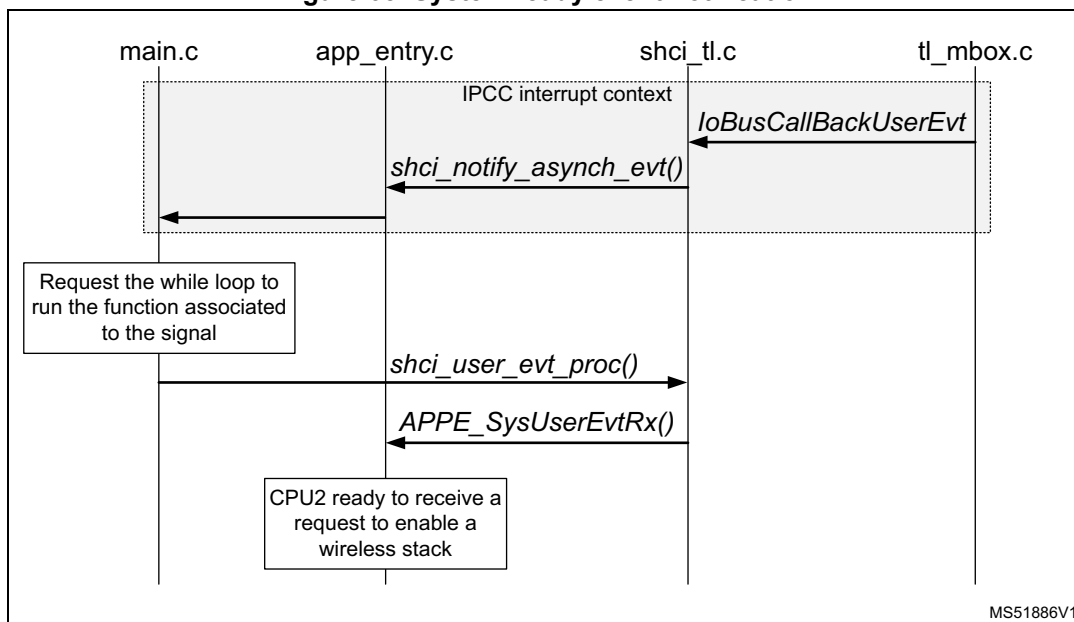
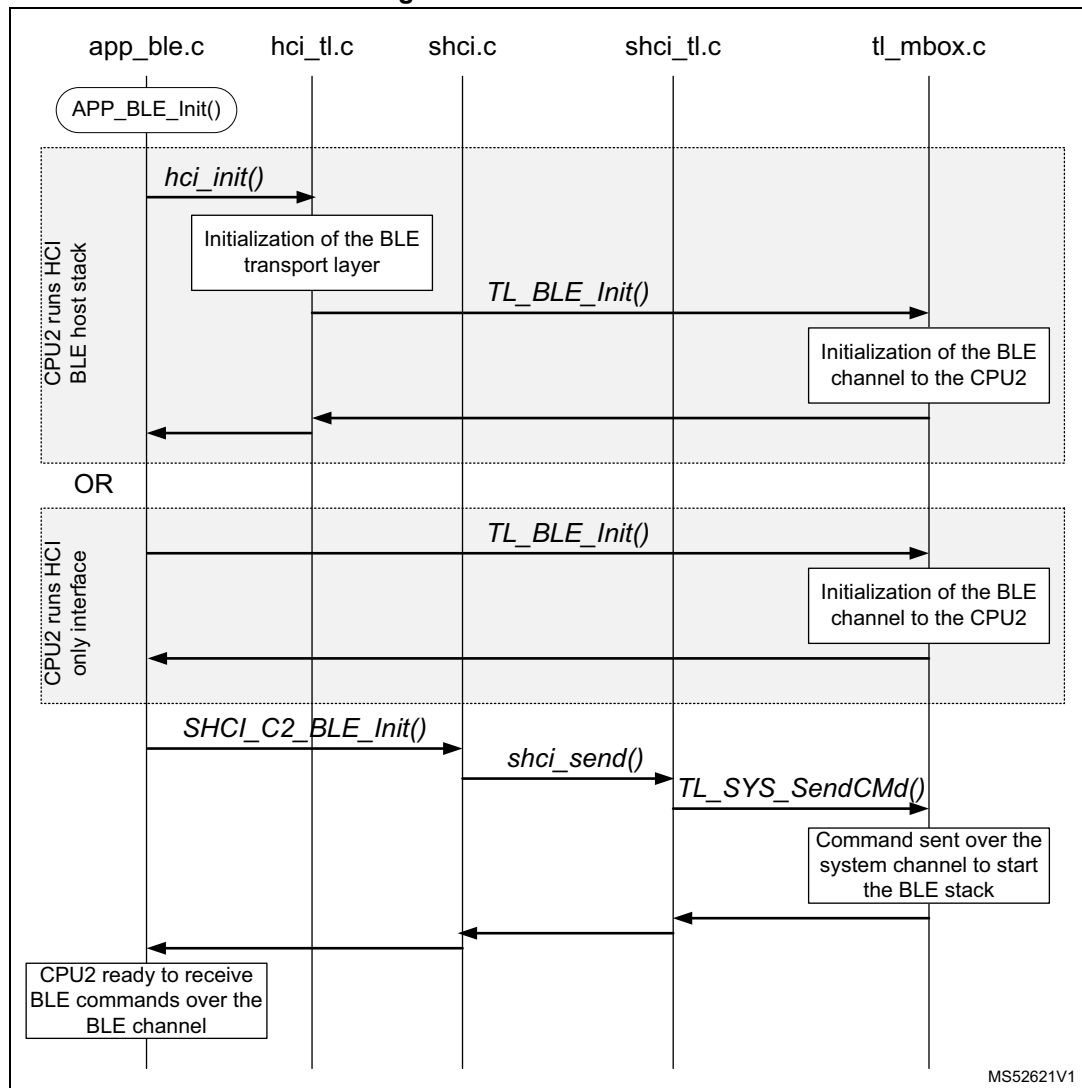


Figure 66. BLE initialization



On receiving a system event, the BLE transport layer is initialized and a system command is sent to CPU2 to start the BLE stack. As soon as the SHCI_C2_BLE_Init() system command has been sent to CPU2, CPUit is ready to receive BLE commands.

When CPU2 runs an HCI only interface, the BLE transport layer starts running in the Host BLE stack on CPU1. Therefore, the BLE transport layer provided must not be used and initialized.

14.2 Mailbox interface

This interface is the lowest level that must be used to send a command to the BLE controller. It is used in the transparent mode application and must be used when a BLE stack open source is used on top of the BT SIG HCI interface. All commands must be answered from CPU2 within 1 s.

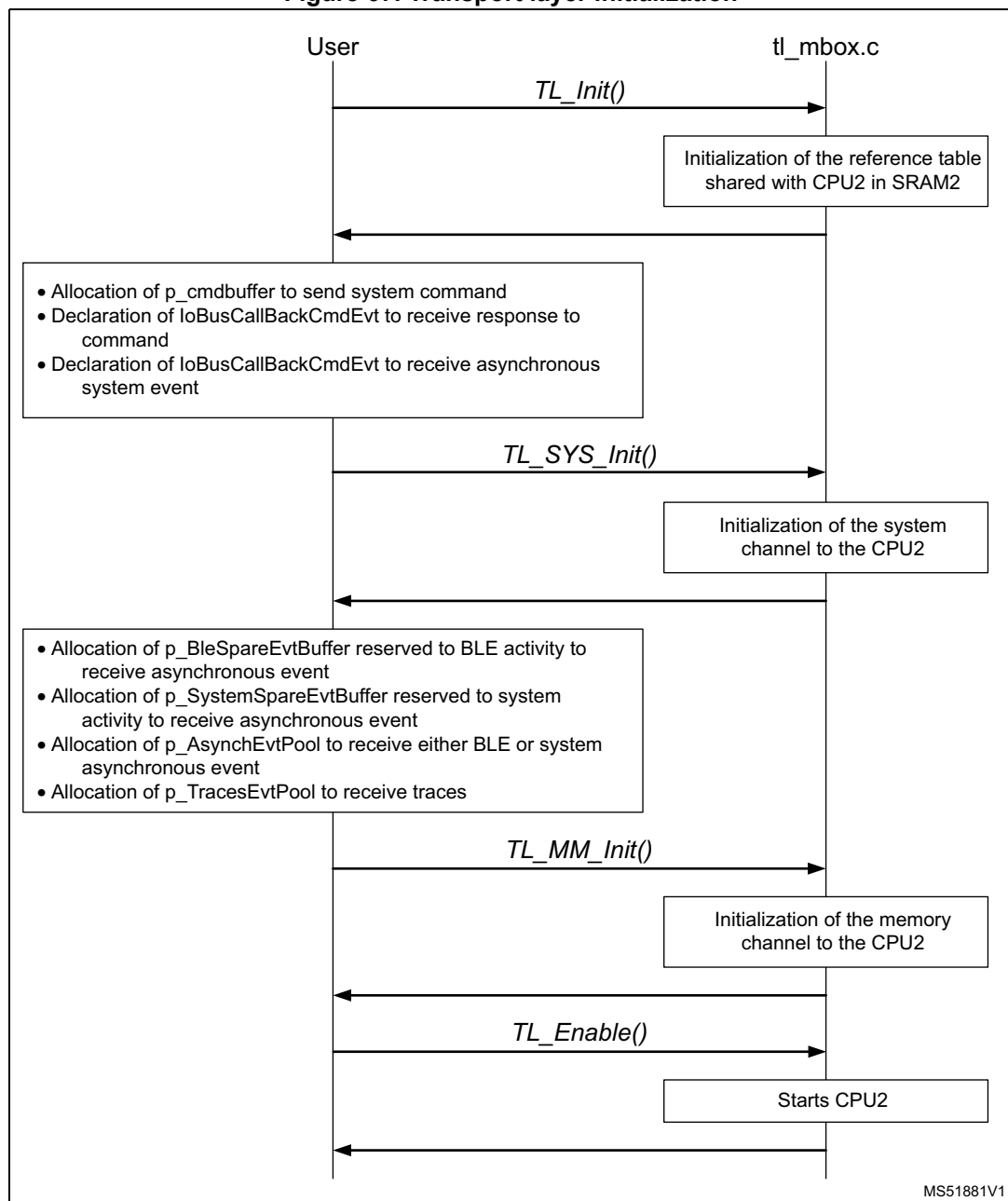
14.2.1 Interface API

Table 30. Interface APIs

Function	Description
<code>void TL_Init(void)</code>	Initializes the shared memory
<code>void TL_Enable(void)</code>	Enables the transport layer
<code>int32_t TL_SYS_Init(void* pConf)</code>	Initializes the system channel
<code>int32_t TL_SYS_SendCmd(uint8_t* buffer, uint16_t size)</code>	Sends a system command
<code>int32_t TL_BLE_Init(void* pConf)</code>	Initializes the BLE channel
<code>int32_t TL_BLE_SendCmd(uint8_t* buffer, uint16_t size)</code>	Sends a BLE command
<code>int32_t TL_BLE_SendAclData(uint8_t* buffer, uint16_t size)</code>	Sends ACL data packet
<code>void TL_MM_Init(TL_MM_Config_t *p_Config)</code>	Initializes the Memory channel
<code>void TL_MM_EvtDone(TL_EvtPacket_t * hcievt)</code>	Releases a buffer to the Memory channel

14.2.2 Detailed interface behavior

Figure 67. Transport layer initialization



void TL_Init(void):

This is the first command to be sent. It initializes the mailbox driver and shared memory.

int32_t TL_SYS_Init(void* pConf):

The user must first allocate the buffer to be used by the mailbox driver to send system command (p_cmdbuffer), the two callbacks to be used to receive the system command

response (IoBusCallBackCmdEvt) and the system asynchronous event (IoBusCallBackUserEvt).

The IoBusCallBackCmdEvt implements the new requirement where a new system command is only sent when the response of the previous one has been received.

This commands initializes the System channel in the mailbox driver.

```
void TL_MM_Init( TL_MM_Config_t *p_Config ):
```

The user must first allocate the buffer to be used by the mailbox driver to only report a BLE Asynchronous event (p_BleSpareEvtBuffer), the buffer to be used by the mailbox driver to only report a System Asynchronous event (p_SystemSpareEvtBuffer), the pool of memory (p_AsynchEvtPool) to be used by the BLE Controller to report either a BLE or System Asynchronous event and the pool of memory (p_TracesEvtPool) to be used by CPU2 for report traces.

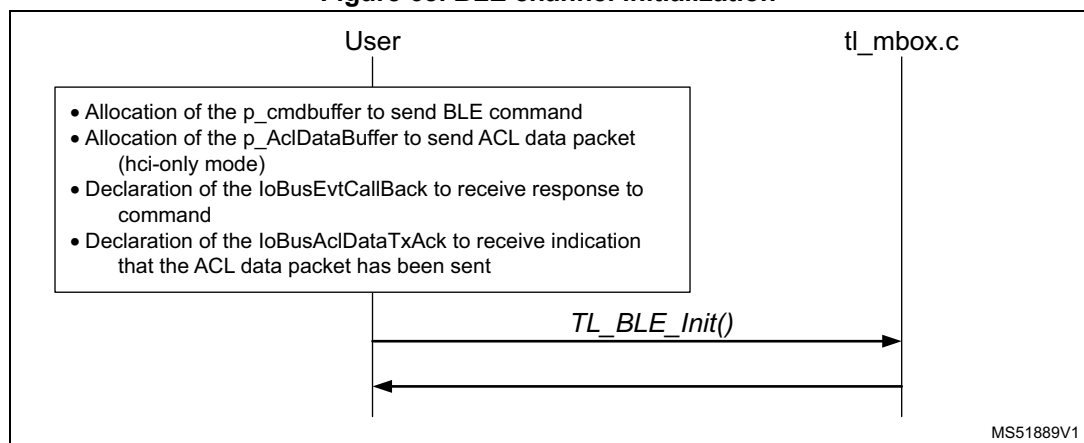
The p_BleSpareEvtBuffer and p_SystemSpareEvtBuffer buffers are used to guarantee that even if the memory pool p_AsynchEvtPool is empty, CPU2 is always able to report either BLE or System events.

This commands initializes the Memory channel in the mailbox driver.

```
void TL_Enable( void ):
```

When the mailbox driver is fully initialized, this command is sent to start CPU2.

Figure 68. BLE channel initialization



```
int32_t TL_BLE_Init( void* pConf ):
```

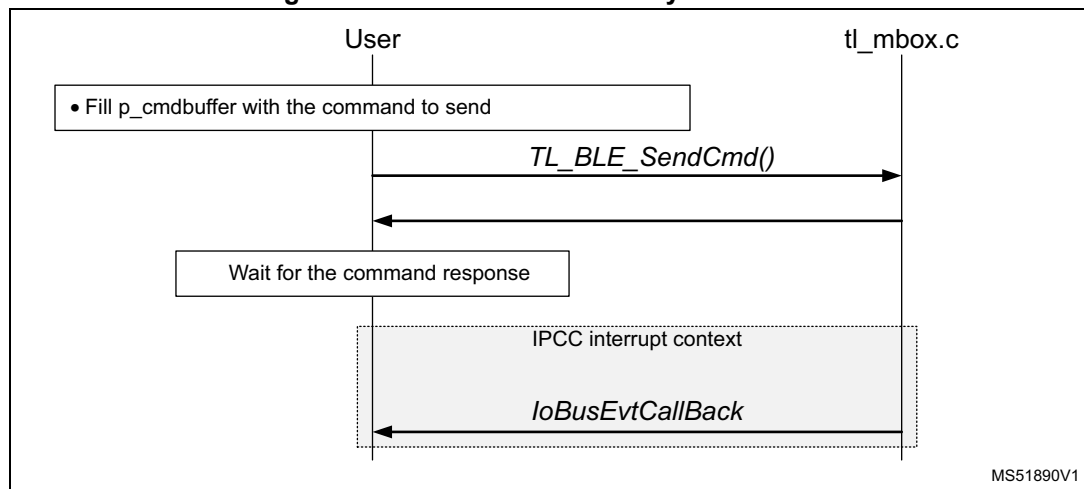
The user must first allocate the buffer to be used by the mailbox driver to send a BLE command (p_cmdbuffer), the buffer to be used by the mailbox driver to send ACL data packet (p_AclDataBuffer), and the two callbacks to be used to receive a BLE event (IoBusEvtCallBack) and the acknowledge of the ACL Data packet (IoBusAclDataTxAck).

The IoBusEvtCallBack must be used to comply to the requirement that a new BLE command can be sent only when the command flow (as specified by the BT SIG) allows it.

When not in HCI only mode, both p_AclDataBuffer and IoBusAclDataTxAck are not used and must be set to 0.

This commands initializes the BLE controller.

Figure 69. BLE command sent by the mailbox

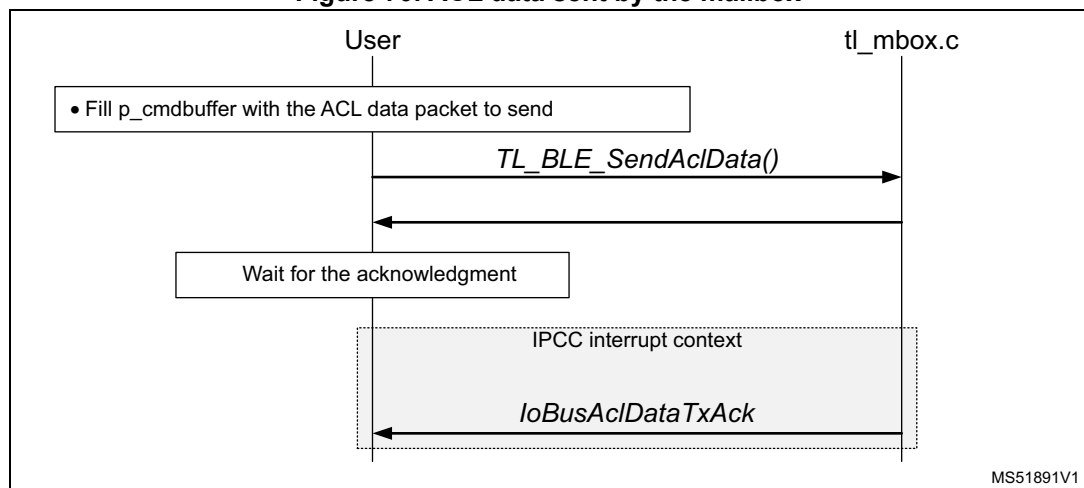


```
int32_t TL_BLE_SendCmd( uint8_t* buffer, uint16_t size ):
```

The user must first fill the buffer p_cmdbuffer with the command to be sent. The parameter buffer and size are not used.

The user must wait for the command response received with IoBusEvtCallBack to check the flow command control in the response packet to understand if a new command can be sent or not. The IoBusEvtCallBack is generated asynchronously in the IPCC interrupt context. It is recommended, depending on the processing load, to implement a background mechanism to decode the received packet (out of the IPCC interrupt context).

Figure 70. ACL data sent by the mailbox



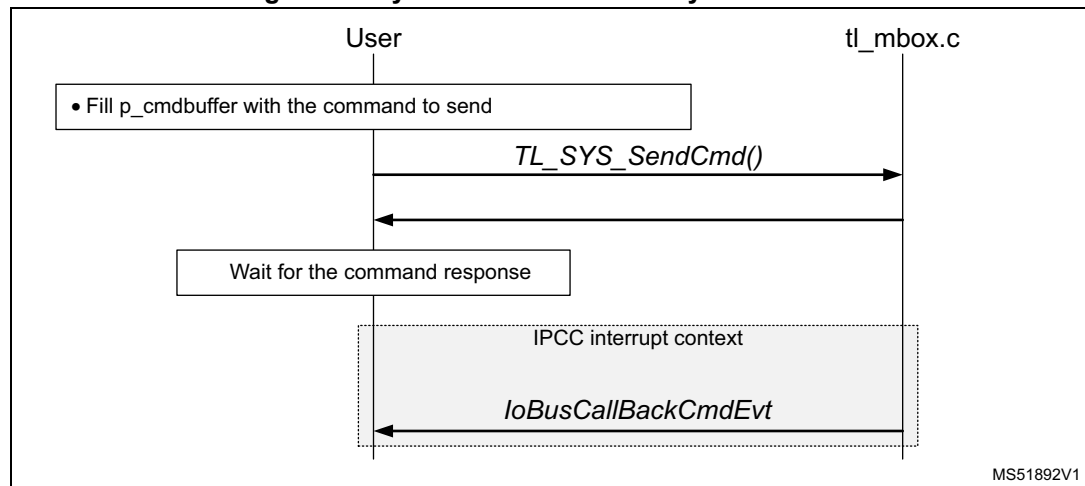
```
int32_t TL_BLE_SendAclData( uint8_t* buffer, uint16_t size ):
```

The user must first fill the p_AclDataBuffer buffer with the ACL data packet to be sent. The parameter buffer and size are not used.

The user must wait for the acknowledge received with `IoBusAclDataTxAck` before sending a new ACL data packet. The `IoBusAclDataTxAck` is generated asynchronously in the IPCC interrupt context. It is recommended, depending on the processing load, to implement a background mechanism to handle the acknowledgment (out of the IPCC interrupt context).

The ACL data packet interface is supported only in HCI Mode. When supported, it is possible to send ACL data packets while a BLE command is pending. The BLE command and ACL data packets do not share resources.

Figure 71. System command sent by the mailbox

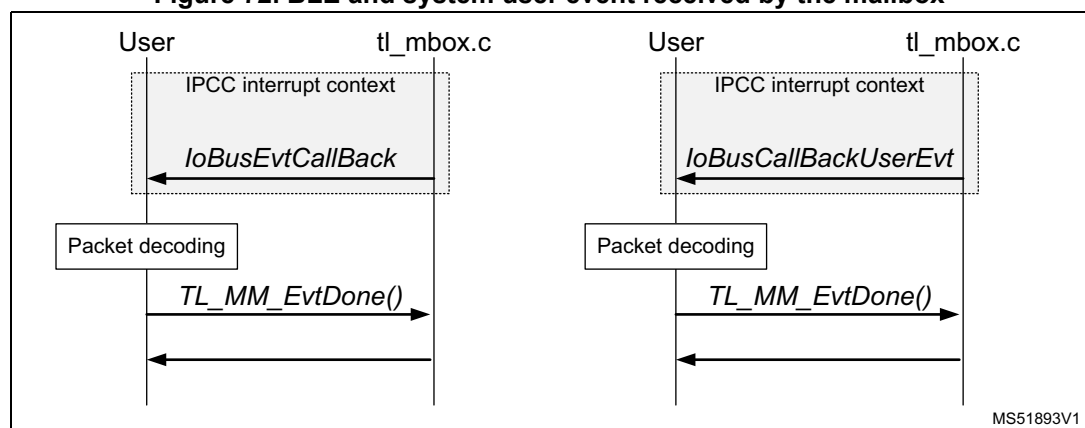


```
int32_t TL_SYS_SendCmd( uint8_t* buffer, uint16_t size )
```

The user must first fill the `p_cmdbuffer` buffer with the command to be sent. The parameter `buffer` and `size` are not used.

The user must wait for the command response received with `IoBusCallBackCmdEvt` before sending a new command. The `IoBusCallBackCmdEvt` is generated asynchronously in the IPCC interrupt context. It is recommended, depending on the processing load, to implement a background mechanism to decode the received packet (out of the IPCC interrupt context).

Figure 72. BLE and system user event received by the mailbox



```
void TL_MM_EvtDone( TL_EvtPacket_t * hcievt ):
```

This API must be called to return the packet to the Memory manager running on CPU2 in the following cases

- For each packet received with IoBusEvtCallBack (User BLE event callback) that is not a BLE command response
- For each packet received with IoBusCallBackUserEvt (User System event callback).

14.3 Mailbox interface - Extended

The mailbox interface is suitable when the command to be sent is built by the user into a buffer to be sent by the mailbox. In the same way, the user must decode the event packet received and manage the command flow control to check if a new command can be sent.

This is the case when using a BLE host stack running on CPU1 on top of the HCI interface. In this case, CPU2 is used in HCI only mode.

However, the BLE host stack does not support the system channel required to initialize CPU2. Therefore, when using only the mailbox interface, the user must build the system command packet to be sent to CPU2 and must manage the event received from CPU2.

It is possible to mix a simple BLE mailbox interface with a higher level shci interface to encode/decode the system packet when connecting to the system mailbox interface. This is the purpose of the "Mailbox interface - Extended"

14.3.1 Interface API

The BLE and memory interface is identical to the simple mailbox interface.

To use the higher shci interface (from shci.h), the shci transport layer must be initialized and connected to the Mailbox driver.

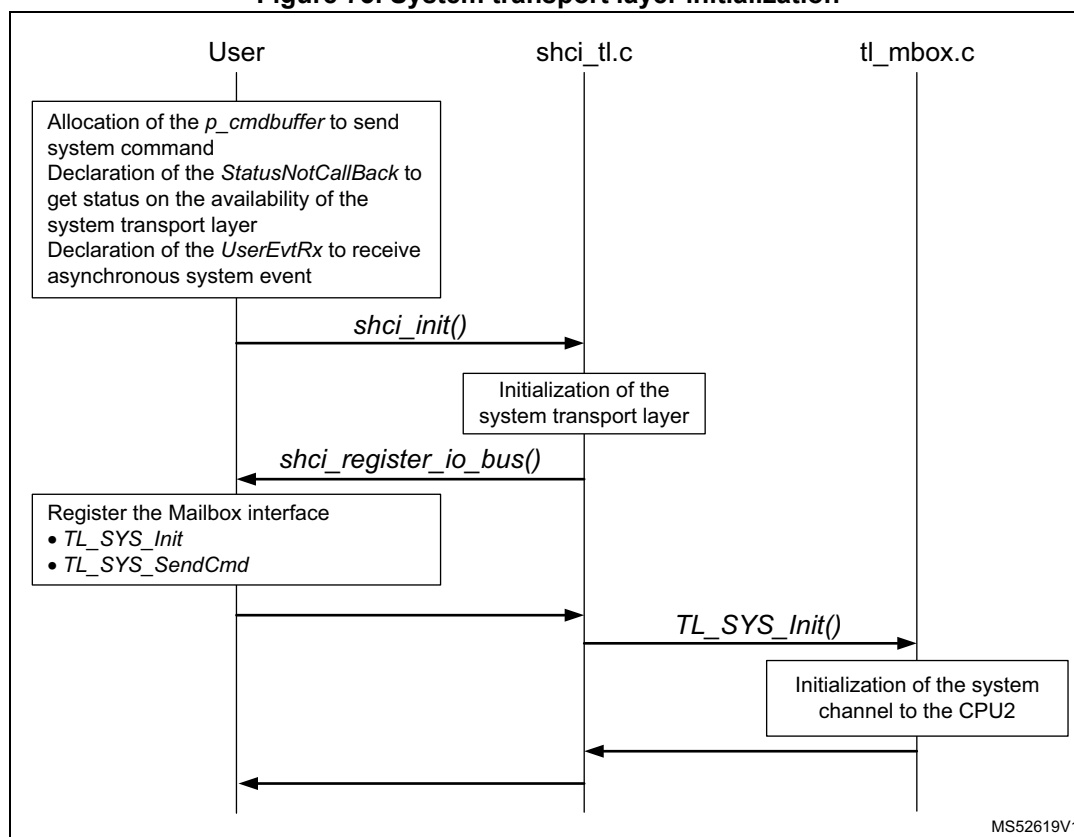
The two API TL_SYS_Init() and TL_SYS_SendCmd() along with the two callbacks IoBusCallBackCmdEvt and IoBusCallBackUserEvt are used and implemented in the transport layer and cannot be used individually any longer.

Table 31. Interface APIs

Function	Description
<code>void shci_init(void(* UserEvtRx)(void* pData), void* pConf)</code>	Initializes the system transport layer.
<code>void shci_register_io_bus(tSHciIO* fops)</code>	Registers the mailbox interface to the system transport layer.
<code>void shci_notify_asynch_evt(void* pdata)</code>	Requests the user to call shci_user_evt_proc
<code>void shci_resume_flow(void)</code>	Resumes the asynchronous user event reporting when stopped by the user.
<code>void shci_cmd_resp_wait(uint32_t timeout)</code>	Waits for a command response.
<code>void shci_cmd_resp_release(uint32_t flag)</code>	Notifies that a command response has been received.
<code>void shci_user_evt_proc(void)</code>	Processes the received asynchronous user event and calls UserEvtRx.

14.3.2 Detailed interface and behavior

Figure 73. System transport layer initialization



MS52619V1

```
void shci_init(void(* UserEvtRx)(void* pData), void* pConf):
```

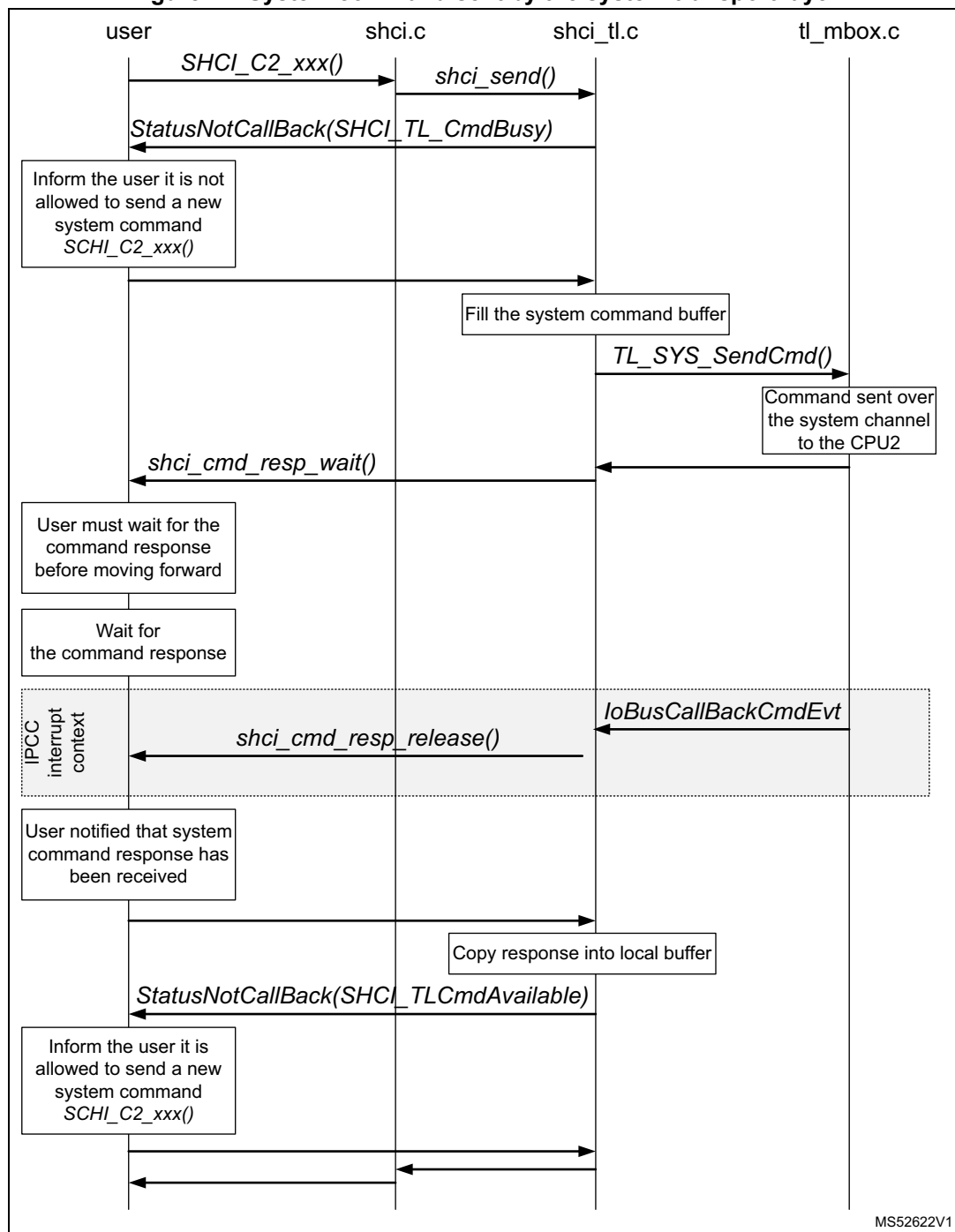
The user must first allocate the buffer to be used by the mailbox driver to send a system command (`p_cmdbuffer`), the two callbacks that will receive a user asynchronous system event (`UserEvtRx`) and the transport layer availability notification (`StatusNotCallBack`).

This command initializes the System channel in the Transport layer and the mailbox driver.

```
void shci_register_io_bus(tSHciIO* fops);:
```

This command registers the Mailbox driver to the system transport layer.

Figure 74. System command sent by the system transport layer



SHCI_C2_xxx()

The list of supported system commands that can be used by the application is in the file shci.h.

void StatusNotCallBack(SHCI_TL_CmdStatus_t status):

This is the registered callback in shci_init() to acknowledge if a system command can be sent. It must be used in a multi-thread application where system commands can be sent from different threads.

When status = SHCI_TL_CmdBusy, the system transport layer is busy and no new system command are be sent.

void shci_cmd_resp_wait(uint32_t timeout):

The application must not return from this command until the shci_cmd_resp_wait() is called to notify the response has been received.

The parameter is meaningless.

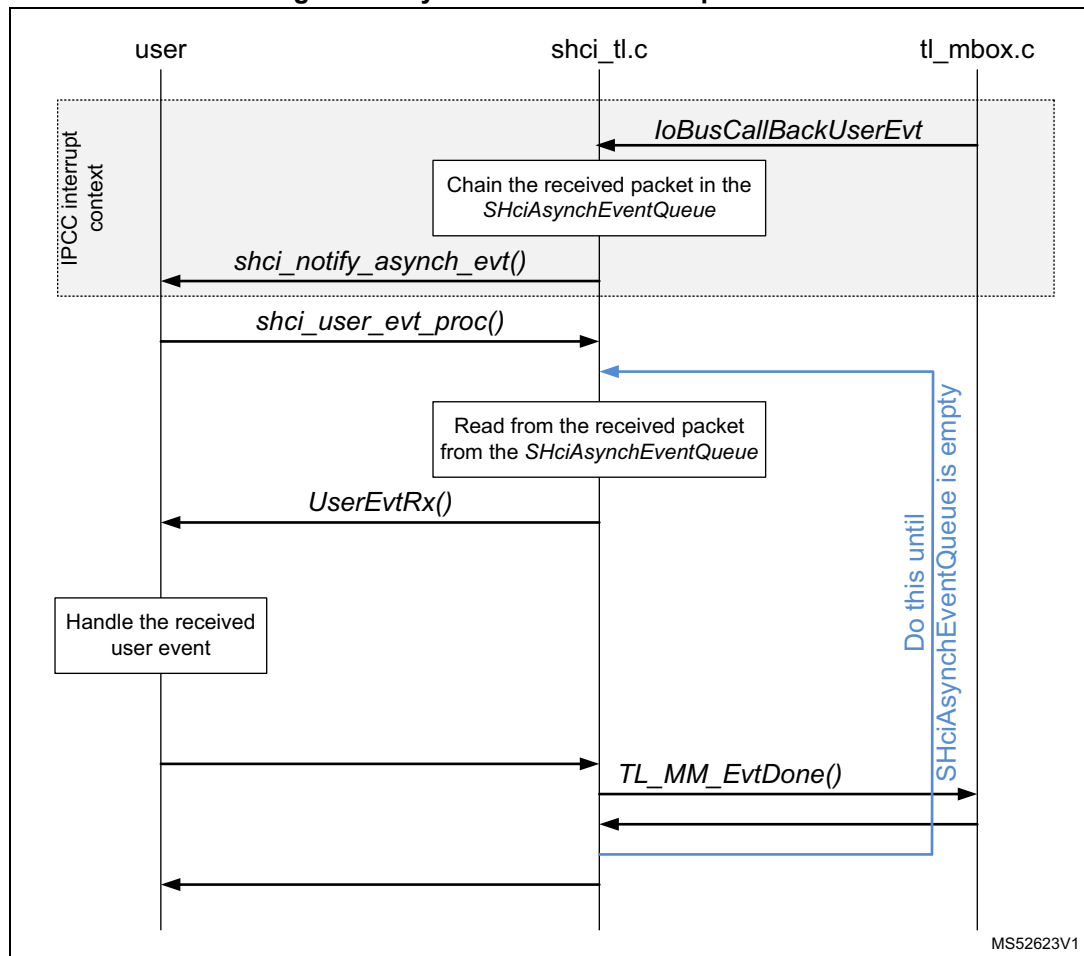
void shci_cmd_resp_release(uint32_t flag):

This function informs the user that the response of the pending system command has been received.

It is called in the IPCC interrupt context. When moving out from this API, the application can return from the API shci_cmd_resp_wait().

The parameter is meaningless.

Figure 75. System user event reception flow



MS52623V1

void shci_notify_asynch_evt(void* pdata):

This API notifies the user that a system user event has been received. The user must call the `shci_user_evt_proc()` to process the notification in the system transport layer. As the `shci_notify_asynch_evt()` notification is called from the IPCC interrupt context, it is strongly recommended to implement a background mechanism to call `shci_user_evt_proc()` (out of IPP Interrupt context).

`pdata` holds the address of the `SHciAsynchEventQueue`.

void shci_user_evt_proc(void):

This function reports the received event to the user with `UserEvtRx()`. As the received event queue `SHciAsynchEventQueue` is filled within the IPCC interrupt context, new events can be stored in the queue while the user is processing an event. `UserEvtRx()` is called for each event retrieved from the queue. The `shci_user_evt_proc()` process frees the buffer to CPU2 memory manager for each return of `UserEvtRx()`.

```
void UserEvtRx (void * pData):
```

This reports to the user the received system event. The buffer holding the received event is freed on return of this function.

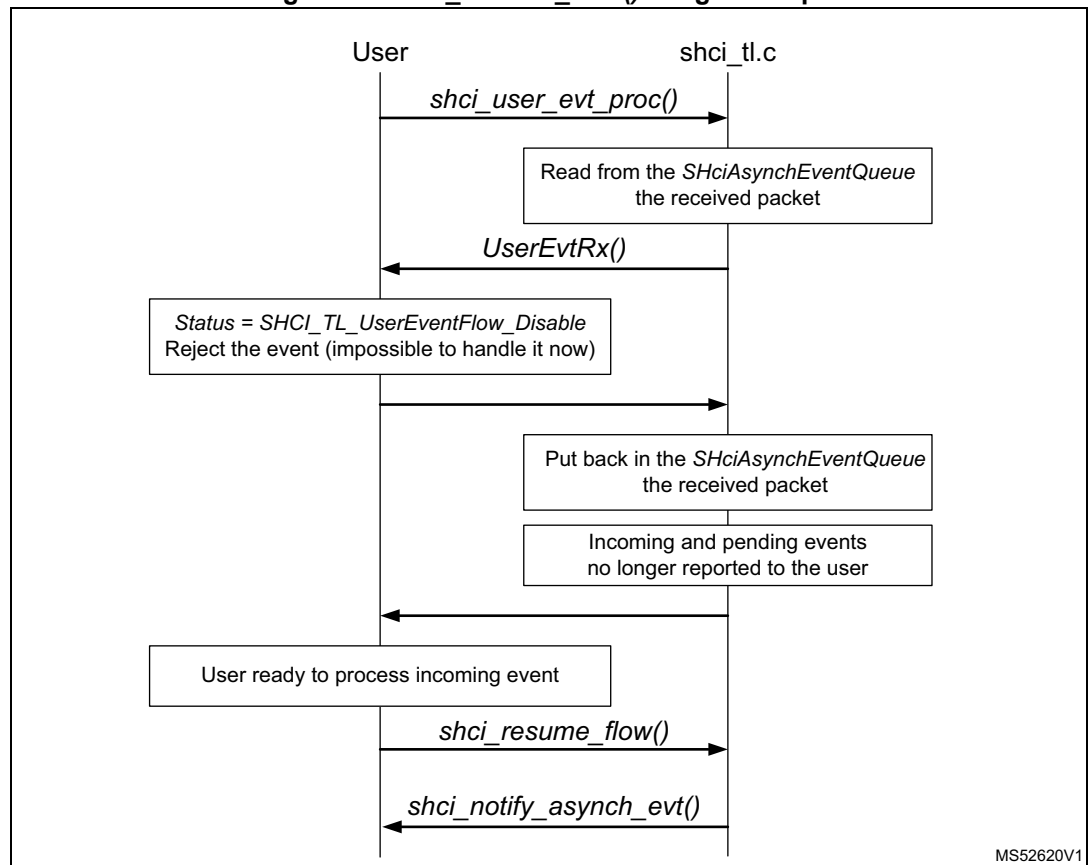
pData is the address of a structure holding the following parameters:

```
typedef struct
{
    SHCI_TL_UserEventFlowStatus_t status;
    TL_EvtPacket_t *pkt;
} tSHCI_UserEvtRxParam;
```

pkt: holds the address of the received event.

status: provides a way for user to notify the system transport layer that the received packet has not been processed and must not be thrown away. When not filled in by the user on return of UserEvtRx(), this parameter is set to SHCI_TL_UserEventFlow_Enable, which means the user has processed the received event.

Figure 76. shci_resume_flow() usage example



MS52620V1

```
void shci_resume_flow( void ):
```

When the user is not able to process incoming event, it must set the status parameter to SHCI_TL_UserEventFlow_Disable before returning from UserEvtRx(). In that case, the

system transport layer does not release the system event and does not report any new incoming events.

When the user is ready to process a system event, it must send the `shci_resume_flow()` that informs the system transport layer to restart reporting of system event.

14.4 ACI interface

This is the interface to the BLE stack running on CPU2. It provides a full set of APIs to use all features in the BLE layers (GATT, GAP, HCI LE).

The ACI commands are sent over the HCI transport.

The interfaces to access all BLE layers (GATT, GAP) are located in the folder `\Middlewares\ST\STM32_WPAN\ble\core\Inc\core`.

When using the ACI interface, the BLE controller must be set in Full stack mode. The HCI transport layer must be implemented in the application to send and receive command from the ACI interface to the mailbox.

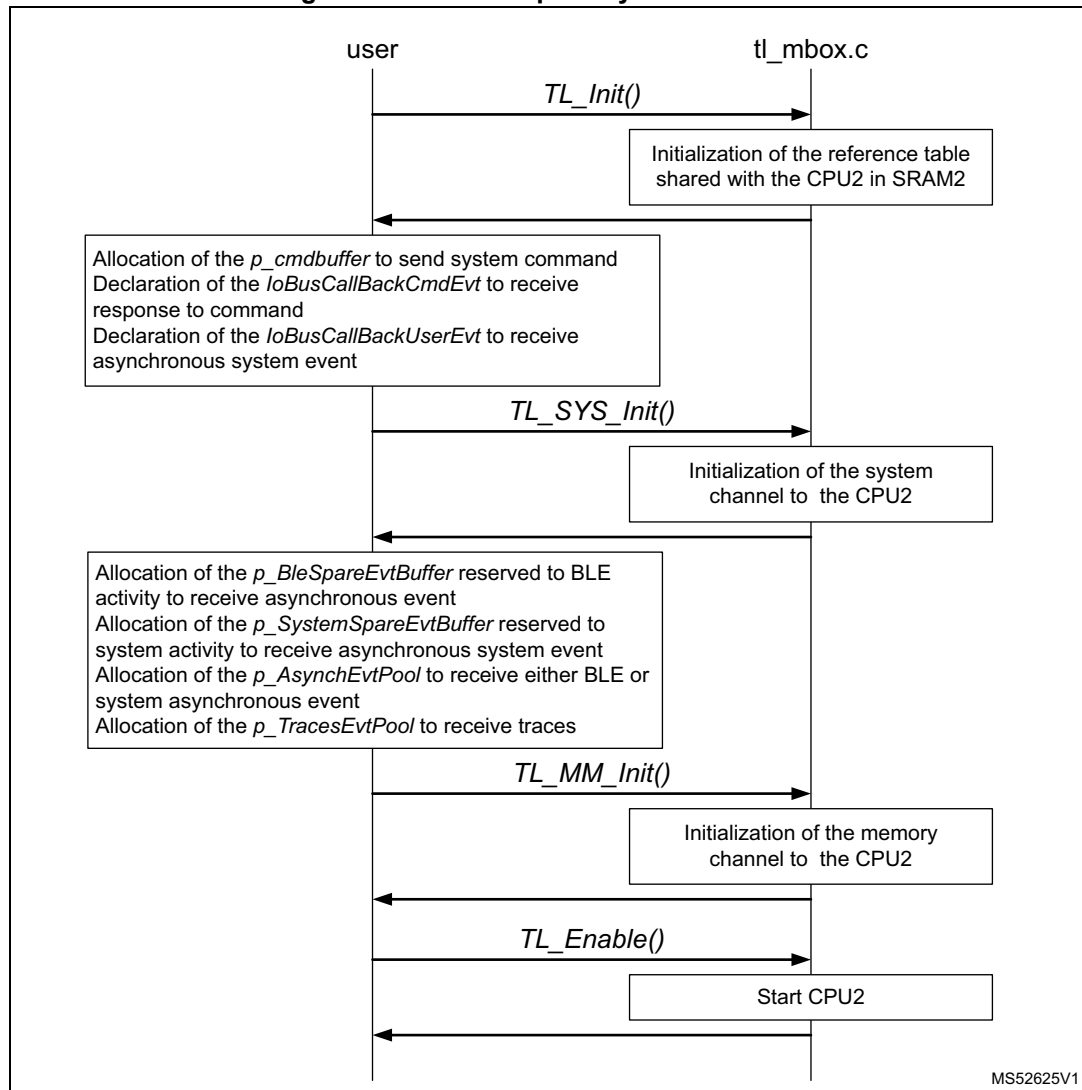
The new interface is basically the [Mailbox interface - Extended](#) where the HCI transport layer is implemented. When the ACI interface is used, the application no longer uses the low level mailbox interface.

Table 32. BLE transport layer interfaces

Function	Description
<code>void hci_init(void(* UserEvtRx)(void* pData), void* pConf);</code>	Initializes the BLE transport layer
<code>void hci_register_io_bus(tHciIO* fops);</code>	Registers the mailbox interface to the BLE transport layer
<code>void hci_notify_asynch_evt(void* pdata);</code>	Requests the user to call <code>hci_user_evt_proc</code>
<code>void hci_resume_flow(void)</code>	Resumes the asynchronous user event reporting when stopped by the user
<code>void hci_cmd_resp_wait(uint32_t timeout)</code>	Waits for a command response
<code>void hci_cmd_resp_release(uint32_t flag)</code>	Notifies that a command response has been received
<code>void hci_user_evt_proc(void)</code>	Process the received asynchronous user event and call <code>UserEvtRx</code>

14.4.1 Detailed interface and behavior

Figure 77. BLE transport layer initialization



```
void hci_init(void(* UserEvtRx)(void* pData), void* pConf);:
```

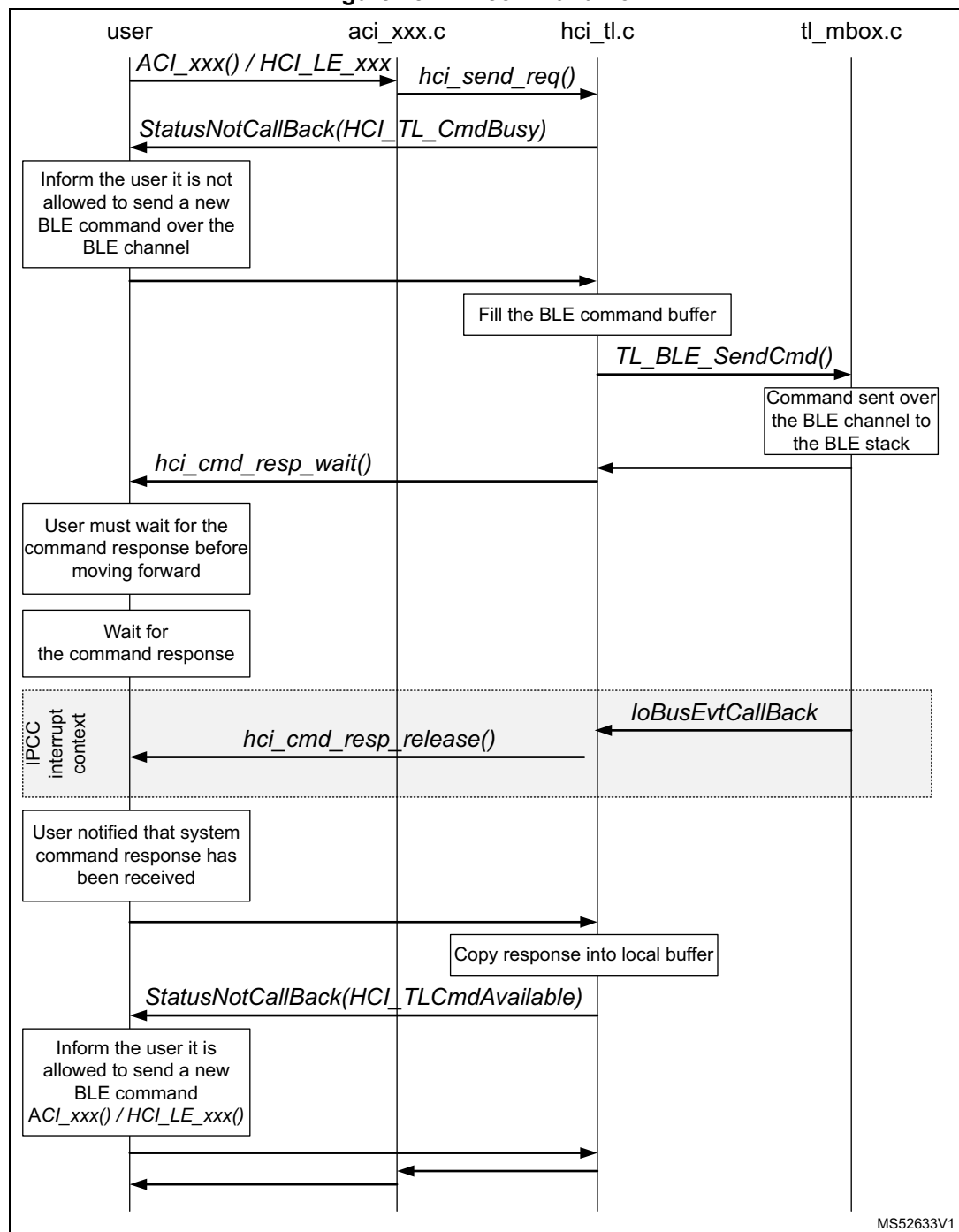
The user must first allocate the buffer to be used by the mailbox driver to send a BLE command (*p_cmdbuffer*), the two callbacks to be used to receive a user asynchronous system event (*UserEvtRx*) and the transport layer availability notification (*StatusNotCallBack*).

This commands initializes the BLE Channel in the HCI Transport layer and the mailbox driver.

```
void hci_register_io_bus(tSHciIO* fops);:
```

This commands registers the mailbox driver to the HCI transport layer.

Figure 78. ACI command flow



ACI_xxx() / HCI_LE_xxx()

The list of supported system commands that can be used by the application is in the folder `\Middlewares\ST\STM32_WPAN\ble\core\inc\core`.

```
void StatusNotCallBack(HCI_TL_CmdStatus_t status):
```

This is the registered callback in `hci_init()` to acknowledge if a BLE command can be sent. To be used in a multi-thread application where the BLE commands can be sent from different threads.

When `status = HCI_TL_CmdBusy`, the HCI transport layer is busy and no new BLE command can be sent.

```
void hci_cmd_resp_wait(uint32_t timeout):
```

The application must not return from this command until the `hci_cmd_resp_wait()` is called to notify the response has been received

The parameter is meaningless.

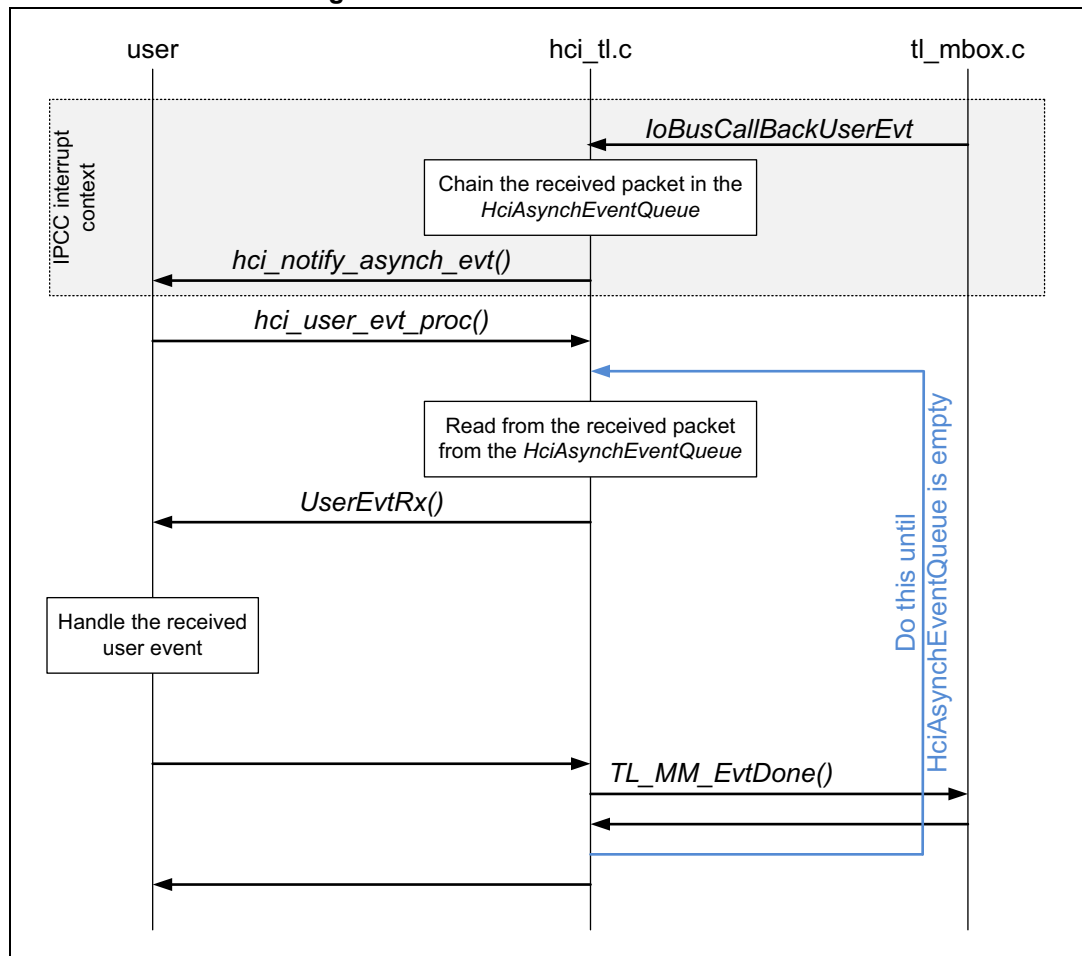
```
void hci_cmd_resp_release(uint32_t flag):
```

This function informs the user that the response of the BLE command pending has been received.

It is called in the IPCC interrupt context. When moving out from this API, the application can return from the API `hci_cmd_resp_wait()`.

The parameter is meaningless.

Figure 79. BLE user event receive flow



```
void hci_notify_asynch_evt(void* pdata):
```

This API notifies the user a BLE user event has been received. The user must then call the `hci_user_evt_proc()` to process the notification in the HCI transport layer. As the `hci_notify_asynch_evt()` notification is called from the IPCC interrupt context, it is strongly recommended to implement a background mechanism to call `hci_user_evt_proc()` (out of IPP Interrupt context)

`pdata` holds the address of the `HciCmdEventQueue`.

```
void hci_user_evt_proc(void):
```

This function reports the received event to the user through `UserEvtRx()`. As the received event queue `HciCmdEventQueue` is filled within the IPCC interrupt context, new events can be stored in the queue while the user is processing an event. `UserEvtRx()` is called for each event retrieved from the queue. The `hci_user_evt_proc()` process is responsible for freeing the buffer to the CPU2 memory manager on each `UserEvtRx()` return.

```
void UserEvtRx (void * pData):
```

This reports the received BLE user event. The buffer holding the received event is freed on return of this function.

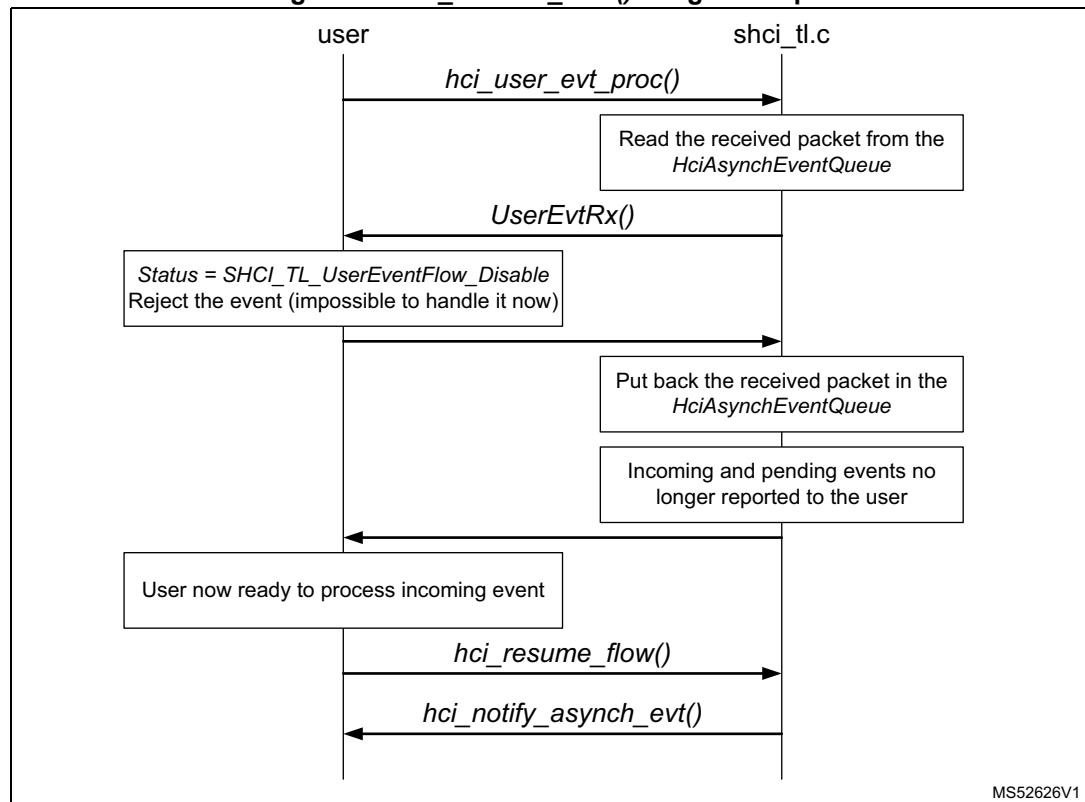
pData is the address of a structure holding the following parameters:

```
typedef struct
{
  HCI_TL_UserEventFlowStatus_t status;
  TL_EvtPacket_t *pkt;
} tHCI_UserEvtRxParam;
```

pkt: holds the address of the received event

status: provides a way for the user to notify the HCI transport layer that the received packet has not been processed and must not be thrown away. When not filled by the user on return of UserEvtRx(), this parameter is set to HCI_TL_UserEventFlow_Enable which means the user has processed the received event.

Figure 80. hci_resume_flow() usage example



MS52626V1

```
void hci_resume_flow( void ):
```

When the user is not able to process an incoming event, it must set the status parameter to HCI_TL_UserEventFlow_Disable before returning from UserEvtRx(). In that case, the HCI transport layer does not release the BLE user event and does not report any new incoming events.

When the user is ready to process BLE user event, it must send the `hci_resume_flow()` that informs the HCI transport layer to resume BLE user event reporting.

14.5 Vendor specific HCI commands for controller

These are local commands for an external device (like STM32CubeMonitor-RF tool) to CPU1.

Table 33. Vendor specific HCI commands

Command	Code	Description	Parameters
LHCI_C1_Write_Register()	0xFD60	Requests CPU1 to write a register (atomic write)	Byte[0]: Bus size access – 1: 8 bits – 2: 16 bits – 4: 32 bits Bytes[1:4]: Mask for bit selection Bytes[5:8]: Address Bytes[9:12]: Value
LHCI_C1_Read_Register()	0xFD61	Requests CPU1 to read a register (atomic read)	Byte[0]: Bus size access – 1: 8 bits – 2: 16 bits – 4: 32 bits Bytes[1:4]: Address
LHCI_C1_Read_Device_Information()	0xFD62	Returns device information	None

The result of these commands is a *Command_complete* event with 0xFDOE code, and according to the command, the following result bytes for:

LHCI_C1_Write_Register:

Byte[0]: Status

LHCI_C1_Read_Register:

Byte[0]: Status

Byte[1:4]: Value

LHCI_C1_Read_Device_Information:

Byte[0]: Status

Byte[1:2]: Revision ID (from DBGMCU_ICODE register)

Bytes[3:4]: Device code ID (from DBGMCU_ICODE register)

Byte[5]: Package type (from package data register)

Byte[6]: Device type ID (from FLASH UID64)

Bytes[7:10]: ST company ID (from FLASH UID64)

Bytes[11:14]: UID64 (from FLASH UID64)
Bytes[15:18]: UID96_0 (from Unique Device ID register)
Bytes[19:22]: UID96_1 (from Unique Device ID register)
Bytes[23:26]: UID96_2 (from Unique Device ID register)
Bytes[27:30]: Safe boot information (from CPU2 in SRAM2)
Bytes[31:42]: RSS information (from CPU2 in SRAM2)
Bytes[31:34]: Version
Bytes[35:38]: Memory size
Bytes[39:42]: RSS
Bytes[43:58]: CPU2 wireless FW information (from CPU2 in SRAM2)
Bytes[43:46]: Version
Bytes[47:50]: Memory size
Bytes[51:54]: Stack information
Bytes[55:58]: Reserved
Bytes[59:62]: CPU1 FW information (hard coded in CPU1 user flash)
Version bytes details:

- bits[0:3] = Build: 0: untracked, x: tracked version
- bits[4:7] = Branch: 0: cut 2.1, 1: cut 2.0
- bits[8:15] = Subversion
- bits[16:23] = Version (minor)
- bits[24:31] = Version (major)

Memory size bytes details:

- bits[0:7] = Flash (number of 4K sector)
- bits[8:15] = Reserved (set to 0, can be used as flash extension)
- bits[16:23] = SRAM2b (number of 1K sector)
- bits[24:31] = SRAM2a (number of 1K sector)

Stack information byte[51] detail:

- INFO_STACK_TYPE_BLE_STANDARD = 0x01
- INFO_STACK_TYPE_BLE_HCI = 0x02
- INFO_STACK_TYPE_BLE_LIGHT = 0x03
- INFO_STACK_TYPE_THREAD_FTD = 0x10
- INFO_STACK_TYPE_THREAD_MTD = 0x11
- INFO_STACK_TYPE_ZIGBEE_FFD = 0x30
- INFO_STACK_TYPE_ZIGBEE_RFD = 0x31
- INFO_STACK_TYPE_MAC = 0x40
- INFO_STACK_TYPE_BLE_THREAD_FTD_STATIC = 0x50
- INFO_STACK_TYPE_802154_LLD_TESTS = 0x60
- INFO_STACK_TYPE_802154_PHY_VALID = 0x61
- INFO_STACK_TYPE_BLE_PHY_VALID = 0x62
- INFO_STACK_TYPE_BLE_LLD_TESTS = 0x63
- INFO_STACK_TYPE_BLE_RLV = 0x64
- INFO_STACK_TYPE_802154_RLV = 0x65
- INFO_STACK_TYPE_BLE_ZIGBEE_FFD_STATIC = 0x70

14.6 STM32WB system commands and events

14.6.1 Commands

Table 34. System interface commands⁽¹⁾

Command	Code	Description
SHCI_C2_FUS_GetState()	0xFC52	Refer to [6] . The following commands are supported by both the FUS and the wireless firmware: <ul style="list-style-type: none"> – SHCI_C2_FUS_StoreUsrKey – SHCI_C2_FUS_LoadUsrKey – SHCI_C2_FUS_LockUsrKey – SHCI_C2_FUS_UnloadUsrKey
SHCI_C2_FUS_FwUpgrade()	0xFC54	
SHCI_C2_FUS_FwDelete()	0xFC55	
SHCI_C2_FUS_UpdateAuthKey()	0xFC56	
SHCI_C2_FUS_LockAuthKey()	0xFC57	
SHCI_C2_FUS_StoreUsrKey() ⁽²⁾	0xFC58	
SHCI_C2_FUS_LoadUsrKey()	0xFC59	
SHCI_C2_FUS_StartWs()	0xFC5A	
SHCI_C2_FUS_LockUsrKey()	0xFC5D	
SHCI_C2_FUS_UnloadUsrKey()	0xFC5E	
SHCI_C2_FUS_ActivateAntiRollback()	0xFC5F	
SHCI_C2_BLE_Init()	0xFC66	Sends the BLE Init parameters. Must be sent before any ACI command. Refer to Section 8.6.9: How to maximize data throughput for more details.
SHCI_C2_THREAD_Init()	0xFC67	Refer to Section 10.8: System commands for Thread applications .

Table 34. System interface commands⁽¹⁾ (continued)

Command	Code	Description
SHCI_C2_DEBUG_Init	0xFC68	Enables the traces on both CPU1 and CPU2 and the GPIO debug configuration on CPU2.
SHCI_C2_FLASH_EraseActivity	0xFC69	Notifies CPU2 that flash memory erase operation can be requested by CPU1. This allows CPU2 to enable timing protection versus the erase operation.
SHCI_C2_CONCURRENT_SetMode()	0xFC6A	Refer to Section 10.8: System commands for Thread applications .
SHCI_C2_FLASH_StoreData()	0xFC6B	
SHCI_C2_FLASH_EraseData()	0xFC6C	
SHCI_C2_RADIO_AllowLowPower()	0xFC6D	
SHCI_C2_MAC_802_15_4_Init ()	0xFC6E	Refer to Section 13.4.5: MAC IEEE Std 802.15.4-2011 system .
SHCI_C2_Reinit() ⁽²⁾	0xFC6F	Requests CPU2 to restart its initialization phase on receiving an event generated by an SEV instruction on CPU1. This is expected to be used by the SBSFU when no RF activity has been started.
SHCI_GetWirelessFwInfo()	-	Returns the version and memory footprint of the wireless stack and FUS running on CPU2.
SHCI_C2_ZIGBEE_Init()	0xFC70	Initializes the ZigBee® protocol stack on CPU2.
SHCI_C2_ExtpaConfig()	0xFC72	Sends to CPU2 the GPIO to be used and its configuration to drive the enable/disable pin of an external PA.
SHCI_C2_SetFlashActivityControl()	0xFC73	Requests CPU2 to use either the PESD bit or Semaphore 7 to protects its timing versus flash memory operation. When the command is not sent, CPU2 uses PESD.
SHCI_C2_BLE_LLD_Init	0xFC74	Initializes the BLE LLD interface on CPU2.
SHCI_C2_Config	0xFC75	Sends the system configuration to the CPU2. Not mandatory.
SHCI_C2_CONCURRENT_GetNextBleEvtTime	0xFC76	Get the next BLE event date (relative time).
SHCI_C2_CONCURRENT_EnableNext_802154_EvtNotification	0xFC77	Activate the next 802.15.4 event notification (one shot).
SHCI_C2_802_15_4_DeInit	0xFC78	Deinitialize 802.15.4 layer (to be used before entering Standby mode).
SHCI_C2_SetSystemClock	0xFC79	Request CPU2 to change system clock.

1. More details on the system command description can be found in the header file shci.h in the STM32WB firmware package.

2. The detailed description is provided in the following part of this section.

SHCI_C2_Reinit()

On reset, the CPU2 is started with the *TL_Enable()* command and when it has finalized its initialization, it reports the SHCI_SUB_EVT_CODE_READY system event. Once it has started, it cannot be reset to restart its startup sequence.

As result, when an application is started from a primary boot application that has already started the CPU2, the application never receives the SHCI_SUB_EVT_CODE_READY system event because it has already been reported to the primary boot application.

The *SHCI_C2_Reinit()* must be sent by the primary boot application (if it has started the CPU2) just before jumping to the main application that is expected to receive the SHCI_SUB_EVT_CODE_READY system event.

When the CPU2 receives the *SHCI_C2_Reinit()* command, the following steps are executed

- Execute both *__SEV()* and *__WFE()* instructions to set and clear the event register
- Enable the EXTI rising edge for the line41 (C1SEV interrupt to CPU2)
- Send response of the command to the CPU1
- Execute WFE and wait for the CPU1 event
- On wake-up from WFE, restart startup code

This command does not reset the hardware, but requires the CPU2 to restart from its reset vector.

SHCI_C2_FUS_StoreUsrKey()

To store user keys in flash memory, one or more SHCI_C2_FUS_StoreUsrKey commands need to be called. Since the request comes from CPU1, the Sem2 shall be taken to take the ownership of flash IP before calling the command and release once all commands have been sent.

How to use the system command in polling mode

CPU1 must set CH2S in IPCC_C1SCR when the transmit buffer has been filled with the command to send to CPU2. Then, it must poll on CH2F in IPCC_C1TOC2SR until it is cleared. Once cleared, the transmit buffer has been filled with the command response from CPU2.

For the system asynchronous event, CPU1 must poll on CH2F in IPCC_C2TO1CSR until it is set. Once set, a buffer can be read from the list. The CPU1 must set CH2C in IPCC_C1SCR to release the list to CPU2 once reading is completed.

Note: All asynchronous buffers must be given back to the CPU2. To do so, CPU1 must poll on CH4F in IPCC_C1TOC2SR until it is cleared. Once cleared, the buffer must be pushed in the free list, and CPU1 must set CH4S in IPCC_C1SCR to notify CPU2 this.

14.6.2 Events

The events listed in [Table 35](#) can be enabled/disabled with the *SHCI_C2_Config()* system command

Table 35. User system events

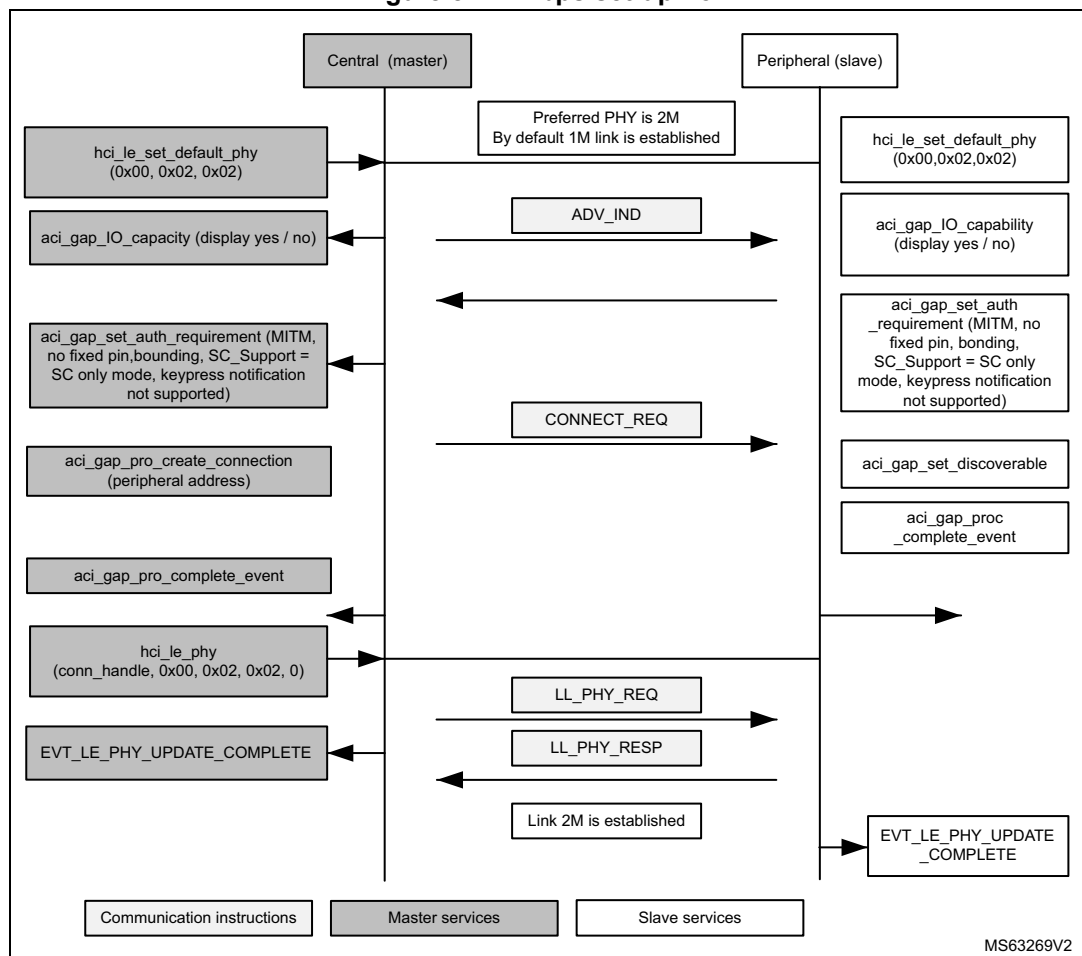
Event	Code	Description
SHCI_SUB_EVT_CODE_READY	0x9200	Returned as soon as the CPU2 has been started and is ready to receive commands.
SHCI_SUB_EVT_ERROR_NOTIF	0x9201	Reports error(s) from CPU2.
SHCI_SUB_EVT_BLE_NVM_RAM_UPDATE	0x9202	Returned when the CPU2 has written the BLE NVM data into the SRAM when requested with the <i>SHCI_C2_Config()</i> command.
SHCI_SUB_EVT_THREAD_NVM_RAM_UPDATE	0x9203	Returned when the CPU2 has written the THREAD NVM data into the SRAM when requested with the <i>SHCI_C2_Config()</i> command.
SHCI_SUB_EVT_NVM_START_WRITE	0x9204	Returned when the CPU2 starts a flash memory write procedure on CPU2.
SHCI_SUB_EVT_NVM_END_WRITE	0x9205	Returned when the CPU2 has successfully written data in flash memory on CPU2.
SHCI_SUB_EVT_NVM_START_ERASE	0x9206	Returned when the CPU2 starts a flash memory erase procedure on CPU2.
SHCI_SUB_EVT_NVM_END_ERASE	0x9207	Returned when the CPU2 has successfully erased data in flash memory on CPU2.

14.7 BLE - Set 2 Mbps link

During the device initialization phase, the preferred TX_PHYS, RX_PHYS values can be initialized.

After the connection at 1 Mbps, it is possible to change the PHY to 2 Mbps for this link, as detailed in [Figure 81](#).

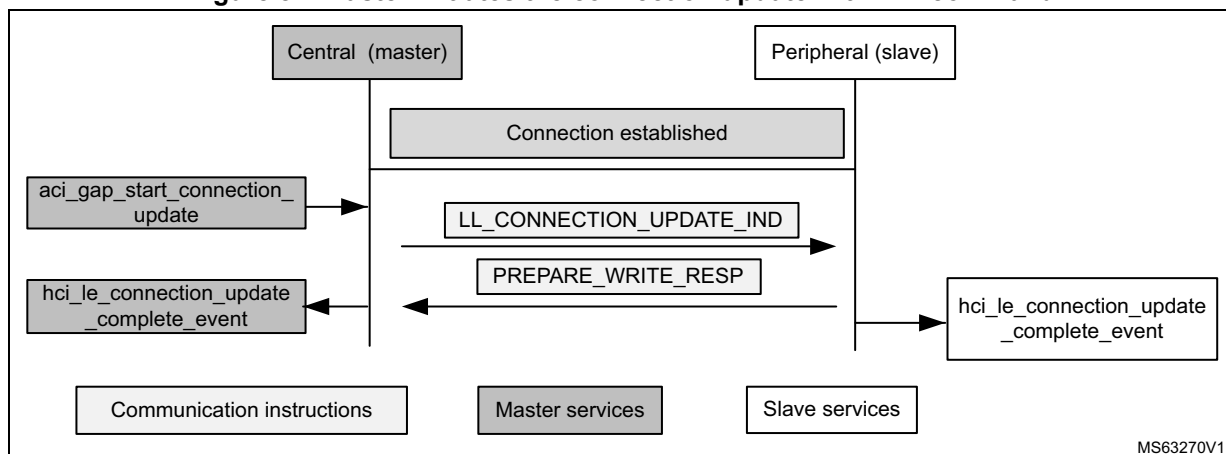
Figure 81. 2 Mbps set-up flow



14.8 BLE - Connection update procedure

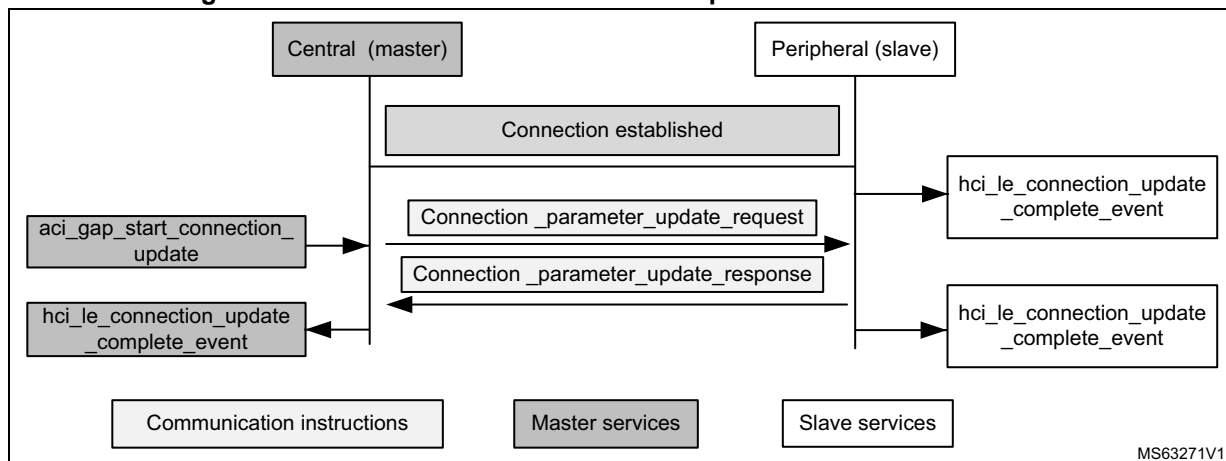
Once a connection is established, it is possible for the master to update the connection parameters with the `aci_gap_start_connection_update` command.

Figure 82. Master initiates the connection update with HCI command



Once a connection is established, it is possible for the slave to update the connection parameters with `aci_l2cap_connection_parameter_update_req` command.

Figure 83. Slave initiates the connection update with L2CAP command



14.9 BLE - Link layer data packet

The BLE has a single packet format used for both the advertising and data channel packets.

Figure 84. Data packet breakdown

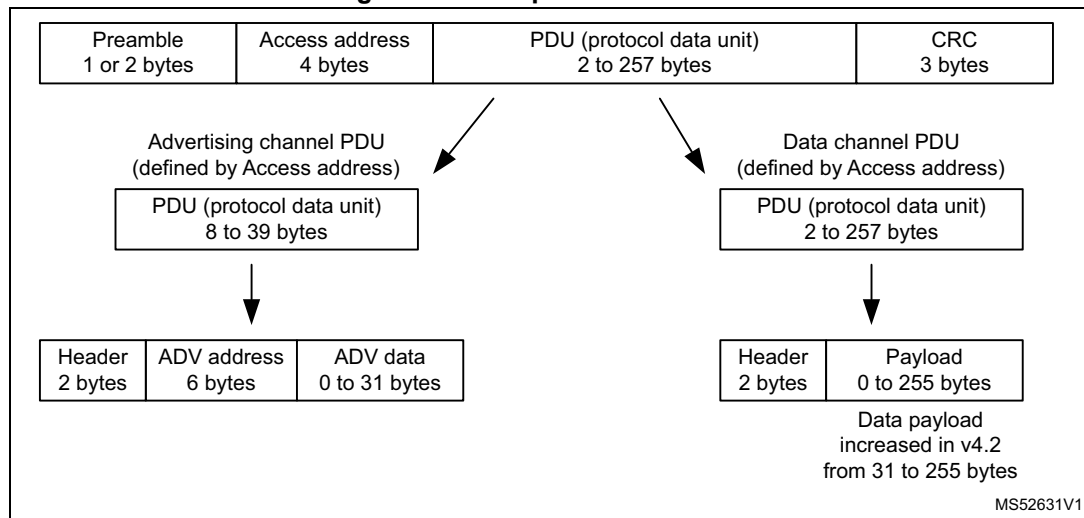
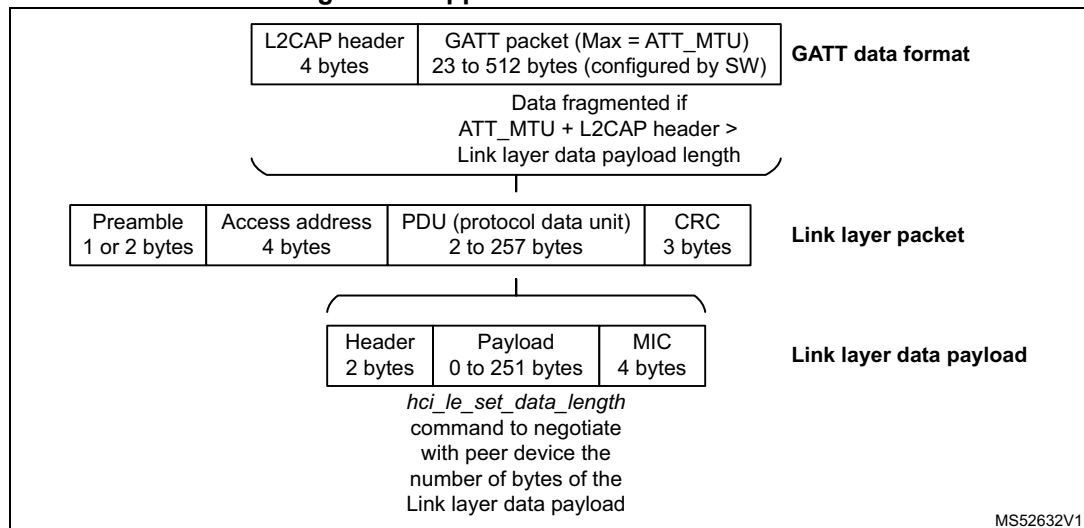


Figure 85. Application GATT data format



14.10 Thread overview

14.10.1 Introduction

The Thread stack is an open standard for reliable, cost-effective, low-power, wireless D2D communication. It is designed specifically for connected home applications where IP-based networking is desired and a variety of application layers can be used on the stack.

The full specification ([9]) is available on <http://threadgroup.org/>.

This standard is based on the IEEE 802.15.4 [IEEE802154] PHY (physical) and MAC (media access control) layers operating at 250 kbps in the 2.4 GHz band.

14.10.2 Main characteristics

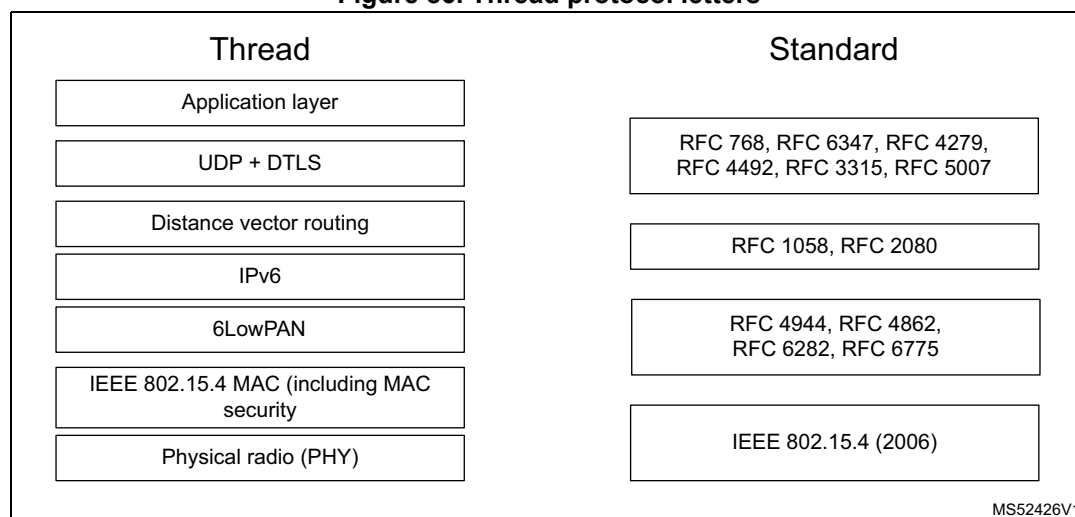
Thread targets smart home applications like environment control, thermostat, alarms, energy management, smart lockers and smart lighting devices. One of the main advantages of this standard is that it is based on IPv6, hence any Thread network can easily be connected to any other IPV6 application. Another big advantage is that it is based on a real mesh network. Once deployed, this network is supposed to be very robust and reliable. For instance, when a route fails, the system is able to auto-reconfigure itself by finding a new route to the destination. Through a mesh network, devices can communicate with each other across much longer distances.

Thread does not really define any application layer. Nevertheless, most of the Thread applications use CoAP to transfer data. CoAP is widely deployed and is already used natively inside Thread in address resolution management, for instance. On STM32WB devices the CoAP layer is exposed to the customer.

14.10.3 Layers

Thread is based on mature and well proven standards, as shown in [Figure 86](#).

Figure 86. Thread protocol letters



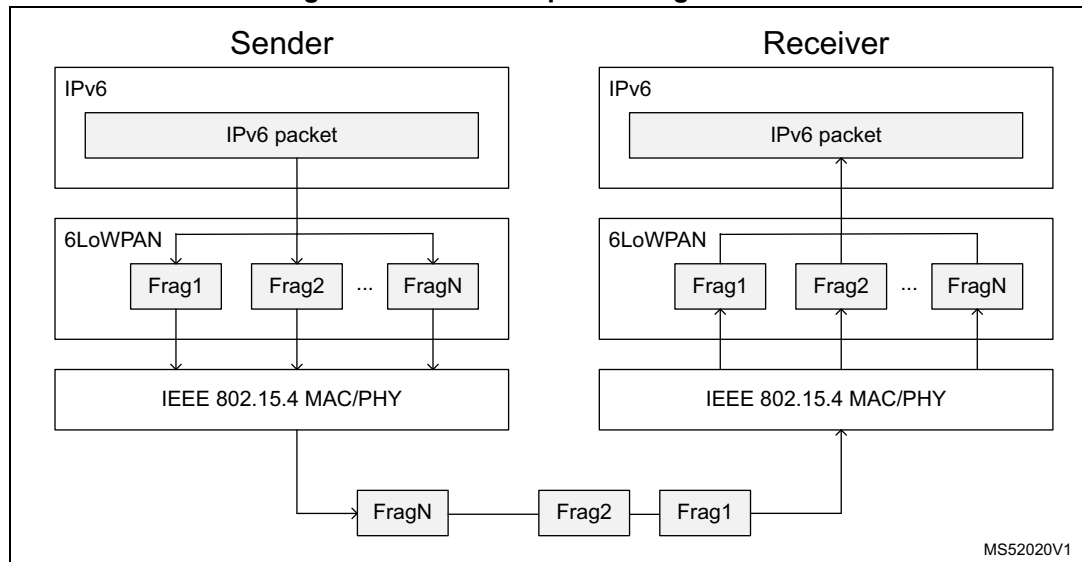
Starting from the lower layer:

- MAC layer, based only on a subset of IEEE 802.15.4 specification from 2006. It supports a rate of 250 kbps in the 2.4 GHz band. There are 16 channels available, ranging from channel 11 to channel 26. Inside a Thread network, only one single channel is used in real time. The MAC radio layer uses the CSMA mechanism to send frames. If the transmission medium is busy it is delayed by a random value. This mechanism reduces the probability of transmission collision between two nodes. On the STM32WB, transmission delay by a random value is managed directly by the hardware.
- 6LoWPAN layer: 6LoWPAN stands for “IPV6 over low power wireless personal area networks”. On Ethernet, an IPV6 packet of 1280 bytes is easily sent as a single “monolithic” frame. On the MAC layer, this not possible because the maximum packet

size is limited to 127 bytes. For this, Thread uses the 6LoWPan layer. This layer implements two techniques:

- Fragmentation (splitting the packets in small TX pieces, and reassembling them in RX)
- Header compression (in some cases, a header of 48 bytes can be compressed into a header of only 6 bytes).

Figure 87. 6LoWPAN packet fragmentation



- IPv6 (Internet Protocol version 6), intended to replace IPv4.
IPv4 uses 32-bit addressing, IPv6 uses 128-bit addressing, which gives billions of possibilities. In addition to a larger addressing space, IPv6 provides other technical benefits, in particular it facilitates the routing procedure.
In Thread there are several addresses defined:
 - MeshLocal64: the address is “topology independent”, it means that it is stable and will never change, even if a device becomes router or end device. The MeshLocal64 address is usually the address used when pinging from one device to another.
 - MeshLocal16: even if the application uses the mMeshLocal64, at low level, the stack will use the mMeshLocal16 address to perform the routing. The Mesh local contains the RLOC field (routing locator). This address is topology dependent (depending upon the network and the link quality). A child can decide to select a new parent, hence a new router, and gets a new address. A child can also become a router, depending on the use case.
 - MeshLinkLocal64: the address starts with 0xFE80, and ends with the MAC extended address with the universal/local bit inverted. It is used for direct point to point link and for the MLE messages.
- Routing: All the mesh network management is based on MLE (mesh link establishment) messages. These messages are used to detect neighboring devices, to configure the system and to maintain routing cost all over the network. Thread claims to

be very robust and is able to manage dynamic routing adaptation. Routers periodically send advertisement messages, which contain the following parameters:

- Link quality between the sender and its neighbors
- Route cost to access to all routers in the Thread network partition.

All routers contain a table with the link quality in UL and DL with all its neighboring routers and the routing cost for all the routers present inside the mesh network.

In this table, there are also the “next hops” defining how to travel all over the network and the so called “age” value, which represents the elapsed time since the latest advertisement reception.

The quality link is a value comprised between 0 and 3, 3 being the best quality (when the signal strength recorded is above 20 dB). Having only four possible link quality values minimizes the overhead of communicating the link quality with neighbors. Routers acting as leaders maintain an additional database for tracking router ID assignments and the extended address associated with each router.

- Application layer: Thread supports CoAP, and this protocol acts as the application layer in our design. CoAP can be considered as a very light version of http protocol. It requires far less resources than http and has a very low overhead. Like http, CoAP is based on the REST model: servers make resources available under a URL, and clients access these resources using requests such as Get, Put, Post, and Delete. The URL (uniform resource locator) specifies the resources and the way to access them. There are four types of messages:
 - Non confirmable message
 - Confirmable message
 - Acknowledgment message
 - Reset message.

14.10.4 Mesh topology

Thread supports Mesh network. As shown in [Figure 88](#), devices inside a Thread network can have two main roles:

- Router
 - Forwards packets for network devices
 - Provides secure commissioning services for devices trying to join the network
 - Keeps its transceiver enabled at all times
- End device
 - Communicates primarily with a single router
 - Does not forward packets for other network devices
 - Can disable its transceiver to reduce power

Amongst all routers, one is always promoted as ‘leader’. The Thread leader is a router manages a the set of routers in a Thread network.

Among all end devices, there can be sleepy end devices, REEDs or standard end devices’.

- REED (router eligible end device) is an end device that can be promoted router if needed
- Sleepy end device is normally disabled, it wakes up occasionally to poll for messages from its parent or to send data.

The size of the Mesh network is configurable. There is a maximum of 32 active routers. Each router can be connected to different child devices. Each child ID is coded on 9 bits, resulting in a theoretical maximum number of 511 children per router. Because of memory constraints on STM32WB the number of child per router is limited to 10.

Figure 88. Thread network topology

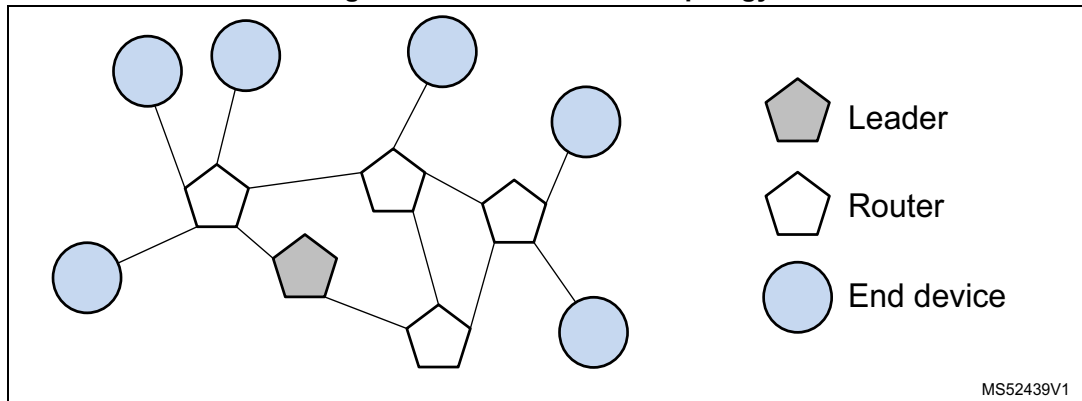
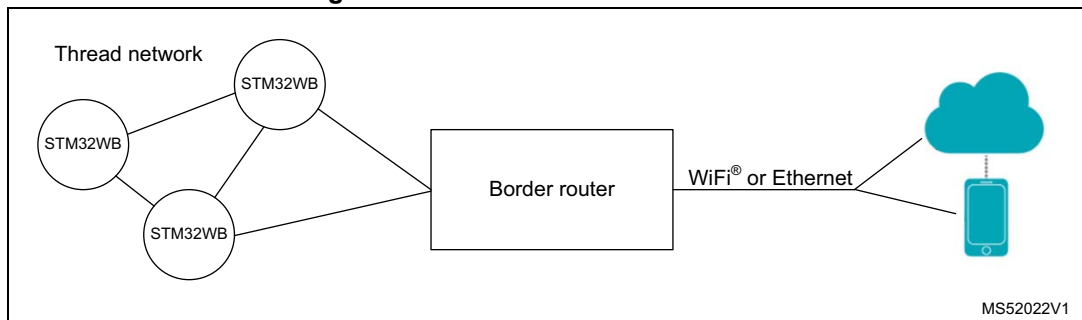


Figure 89. Link with the external world



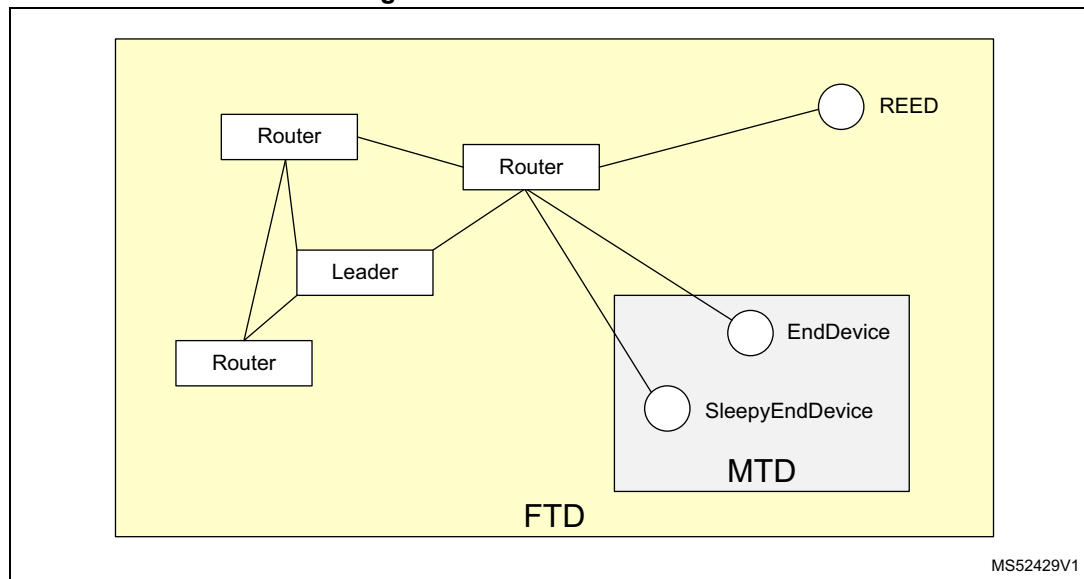
14.10.5 Thread configuration

When compiling OpenThread, several options can be set depending on the targeted use case. For STM32WB two cases are considered:

1. Full Thread device (FTD): the device can act as a simple sleepy end device but also as router and leader inside the Thread network.
2. Minimal Thread device (MTD): the device acts only as end device or sleepy end device

The MTD configuration needs less memory (both RAM and flash) vs. the FTD configuration. On the other hand, an MTD only acts as end device or sleepy end device.

Figure 90. Thread device roles



15 Conclusion

Bluetooth Low Energy (BLE) or 802.15.4 applications based on for STM32WB series microcontrollers require the understanding of dedicated software protocols and architectures.

This document details how the developer must create the embedded application software, a key element is to follow the correct procedure for the system initialization.

16 Revision history

Table 36. Document revision history

Date	Revision	Changes
18-Jun-2019	1	Initial release.
26-Sep-2019	2	Updated Introduction , Section 4.2: Memory mapping , Section 4.3: Shared peripherals , Section 10.2: How to start , Section 10.6: OpenThread API , Section 12.1: Thread_Cli_Cmd , Section 12.4: Thread_Coap_Multiboard , Section 12.5: Thread_Commissioning , Section 13.3: API and Section 13.4.3: MAC application processor firmware . Added Section 12.8: Thread FUOTA and its subsections. Updated Figure 4: Memory mapping . Updated Table 2: Semaphores .
23-Mar-2020	3	Updated Section 4.3: Shared peripherals and Section 8.6.1: How to set Bluetooth device address . Added Section 4.7: Flash memory management , Section 4.8: Debug information from CPU , Section 4.9: FreeRTOS low power and their subsections. Updated Table 2: Semaphores , Table 34: System interface commands and Table 35: User system events . Updated Figure 10: Algorithm to write/erase data in the flash memory , Figure 24: Heart rate project - Interaction between middleware and user application and Figure 29: P2P server software communication . Minor text edits across the whole document.
20-Oct-2020	4	Updated Section 4.3: Shared peripherals , Section 5: System initialization , Section 8.6.5: How to start the BLE stack - SHCI_C2_BLE_Init() , Section 11.3.1: Creating an otCoapResource and Section 14.6.1: Commands . Updated Figure 10: Algorithm to write/erase data in the flash memory . Updated Table 34: System interface commands and Table 35: User system events . Added Section 4.10: Device information table and Section 5.2: CPU2 startup .
20-Apr-2021	5	Updated Introduction , Section 1: References , Section 4.3: Shared peripherals , Section 8.6.5: How to start the BLE stack - SHCI_C2_BLE_Init() , Section 8.6.9: How to maximize data throughput and Section 14.2: Mailbox interface . Updated Figure 1: Protocols supported by STM32WB series microcontrollers , Figure 38: Software architecture and Figure 79: BLE user event receive flow . Updated Table 1: Stacks supported by STM32WB series microcontrollers , Table 2: Semaphores , Table 9: Security commands and Table 28: MO firmwares available for Thread . Added Section 4.11: ECCD error management . Removed former Section 6.10: Write or read long local or distant values , Section 13.8: BLE - Security procedure and their subsections.

Table 36. Document revision history (continued)

Date	Revision	Changes
14-Dec-2021	6	<p>Added Section 7.6.2: How to set IRK (identity root key) and ERK (encryption root key), Section 8.6.6: BLE GATT DB and security record in NVM and Section 8.6.7: How to calculate the maximum number of bonded devices that can be stored in NVM.</p> <p>Updated Section 8.6.1: How to set Bluetooth device address, CFG_BLE_ATT_VALUE_ARRAY_SIZE, Section 14.2: Mailbox interface, SHCI_C2_xxx(), and ACI_xxx() / HCI_LE_xxx().</p> <p>Updated Table 31: Interface APIs.</p>
15-Jul-2022	7	<p>Updated Section 4.3: Shared peripherals, Section 4.6: Low power manager, and Section 8.2.2: STM32WB heart rate sensor application - Middleware application.</p> <p>Added Section 8.6.8: NVM write access.</p> <p>Updated Figure 24: Heart rate project - Interaction between middleware and user application.</p> <p>Minor text edits across the whole document.</p>
24-Nov-2022	8	<p>Updated CFG_BLE_OPTIONS.</p> <p>Added Section 7.6.11: How to switch from 32 to 64 MHz and Section 7.6.12: How to re-enable the PLL when exiting low power mode.</p> <p>Minor text edits across the whole document.</p>
11-Apr-2023	9	<p>Added Section 14.5: Vendor specific HCI commands for controller.</p> <p>Minor text edits across the whole document.</p>
18-Jul-2023	10	<p>Updated document title.</p> <p>Updated Security attack, CFG_BLE_HSE_STARTUP_TIME, and SHCI_C2_xxx().</p> <p>Updated Table 34: System interface commands.</p> <p>Minor text edits across the whole document.</p>
18-Aug-2023	11	<p>Updated Table 2: Semaphores.</p> <p>Updated Section 4.3: Shared peripherals.</p> <p>Minor text edits across the whole document.</p>
25-Sep-2023	12	<p>Updated Section 8.6.2: How to set IR (Identity Root) and ER (Encryption Root).</p>
24-Oct-2023	13	<p>Updated Section 4.7.3: Conflict between RF activity and flash memory management and Section 8.6.8: NVM write access.</p> <p>Updated Table 34: System interface commands.</p> <p>Added note in Section 5.2: CPU2 startup, Section 6: PLL management and its subsections, Section 8.6.11: How to use BLE commands in blocking mode, and How to use the system command in polling mode.</p> <p>Removed former Section 7.6.11: How to switch from 32 to 64 MHz and Section 7.6.12: How to re-enable the PLL when exiting low power mode.</p>
27-Nov-2023	14	<p>Updated Section 14.6.1: Commands.</p> <p>Updated Table 34: System interface commands.</p>
04-Dec-2023	15	<p>Updated Table 23: FUOTA service and characteristics UUID and Table 24: Base address characteristics specification.</p>

IMPORTANT NOTICE – READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST's terms and conditions of sale in place at the time of order acknowledgment.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2023 STMicroelectronics – All rights reserved