AN5406

Application note

How to build a LoRa® application with STM32CubeWL

Introduction

This application note guides the user through all the steps required to build specific LoRa® applications based on STM32WL Series microcontrollers.

LoRa® is a type of wireless telecommunication network designed to allow long-range communications at a very-low bitrate and to enable long-life battery-operated sensors. LoRaWAN® defines the communication and security protocol that ensures the interoperability with the LoRa® network.

The firmware in the STM32CubeWL MCU Package is compliant with the LoRa Alliance® specification protocol named LoRaWAN® and has the following main features:

- Application integration ready
- Easy add-on of the low-power LoRa® solution
- Extremely low CPU load
- No latency requirements
- Small STM32 memory footprint
- Low-power timing services

The firmware of the STM32CubeWL MCU Package is based on the STM32Cube HAL drivers.

This document provides customer application examples on the STM32WL Nucleo boards NUCLEO_WL55JC (order codes NUCLEO-WL55JC1 for high-frequency band and NUCLEO-WL55JC2 for low-frequency band).

To fully benefit from the information in this application note and to create an application, the user must be familiar with the STM32 microcontrollers, the LoRa® technology, and understand system services such as low-power management and task sequencing.

1 General information

The STM32CubeWL runs on STM32WL Series microcontrollers based on the Arm® Cortex®-M processor.

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

Table 1. Acronyms and terms

Acronym	Definition
ABP	Activation by personalization
ADR	Adaptive data rate
BSP	Board support package
HAL	Hardware abstraction layer
IPCC	Inter-processor communication controller
IRQ	Interrupt request
LoRa	Long range radio technology
LoRaWAN	LoRa wide-area network
LPWAN	Low-power wide-area network
MAC	Media access control
MCPS	MAC common part sublayer
MIB	MAC information base
MLME	MAC sublayer management entity
MSC	Message sequence chart
OTAA	Over-the-air activation
PER	Packet error rate
PRBS mode	Pseudo-random binary sequence
SWD	Serial-wire debug
<target>	STM32WL Nucleo boards (NUCLEO-WL55JC)

Reference documents

[1] LoRaWAN 1.0.3 Specification by LoRa Alliance Specification Protocol– 2018, January

[2] Application note *LoRaWAN AT commands for STM32CubeWL* (AN5481)

[3] User manual *Description of STM32WL HAL and low-layer drivers* (UM2642)

[4] IEEE Std 802.15.4TM - 2011. Low-Rate Wireless Personal Area Networks (LR-WPANs)

[5] Application note *GFSK long packet with STM32CubeWL* (AN5687)

[6] Application note *Integration guide of SBSFU on STM32CubeWL* (AN5544)

[7] Application note *How to secure LoRaWAN and Sigfox with STM32CubeWL* (AN5682)

LoRa standard

Refer to document [1] for more details on LoRa and LoRaWAN recommendations.

2 STM32CubeWL architecture

2.1 STM32CubeWL overview

The firmware of the STM32CubeWL MCU Package includes the following resources (see Figure 1):

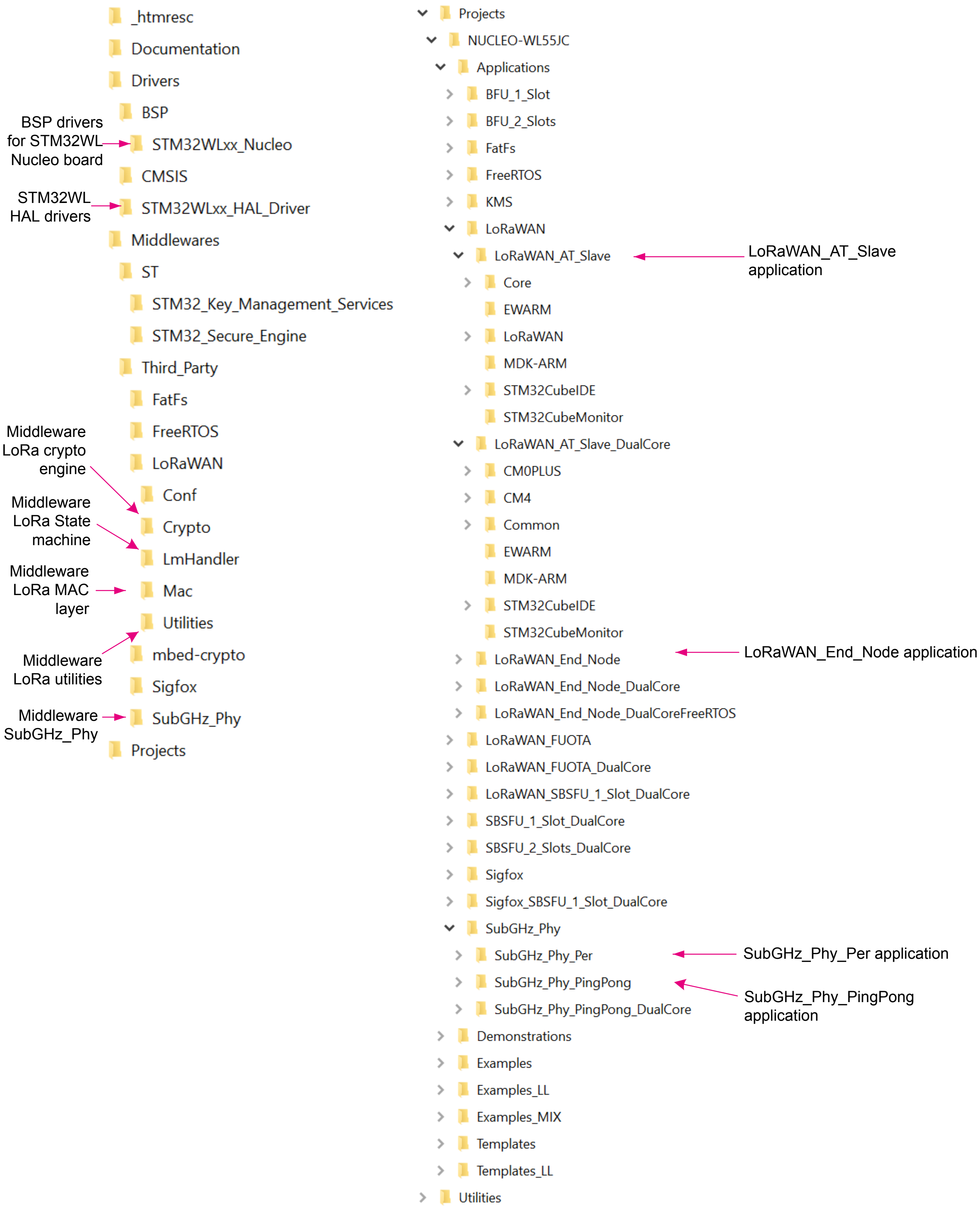
- Board support package: STM32WLxx_Nucleo drivers
- STM32WLxx_HAL_Driver
- Middleware:
  - LoRaWAN containing:
    - LoRaWAN layer
    - LoRa utilities
    - LoRa software crypto engine
    - LoRa state machine
  - SubGHz_Phy layer middleware containing the radio and radio_driver interfaces
- LoRaWAN applications:
  - LoRaWAN_AT_Slave (SingleCore and DualCore)
  - LoRaWAN_End_Node (SingleCore, DualCore and DualCore with FreeRTOS)
- SubGHz_Phy application:
  - SubGHz_Phy_PingPong (SingleCore and DualCore)
  - SubGHz_Phy_Per (SingleCore)

In addition, this application provides an efficient system integration with the following:

- a sequencer to execute the tasks in background and to 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

For more details refer to Section 7 Utilities description.

Figure 1. Project file structure



BSP drivers for STM32WL Nucleo board

STM32WL HAL drivers

Middleware LoRa crypto engine

Middleware LoRa State machine

Middleware LoRa MAC layer

Middleware LoRa utilities

Middleware SubGHz_Phy

LoRaWAN_AT_Slave application

LoRaWAN_End_Node application

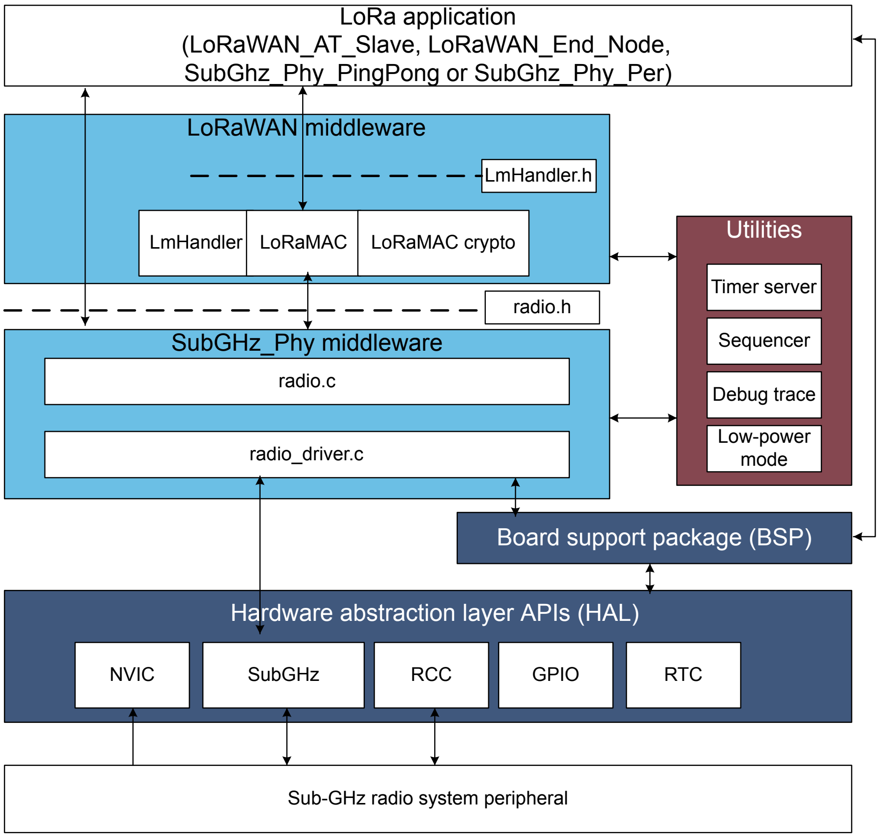
SubGHz_Phy_Per application

SubGHz_Phy_PingPong application

2.2 Static LoRa architecture

The figure below describes the main design of the firmware for the LoRa application.

Figure 2. Static LoRa architecture



The HAL uses STM32Cube APIs to drive the MCU hardware required by the application. Only specific hardware is included in the LoRa middleware as it is mandatory to run a LoRa application.

The RTC provides a centralized time unit that continues to run even in low-power mode (Stop 2 mode). The RTC alarm is used to wake up the system at specific timings managed by the timer server.

The SubGHz_Phy middleware uses the HAL SubGHz to control the radio (see the above figure). For more details, refer to Section 5

The MAC controls the SubGHz_Phy using the 802.15.4 model. The MAC interfaces with the SubGHz_Phy driver and uses the timer server to add or remove timed tasks.

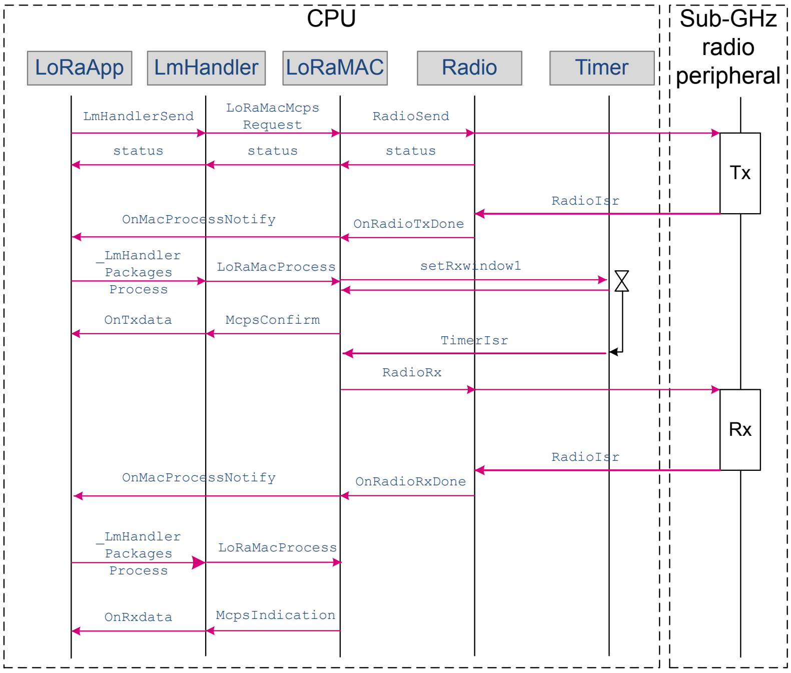
Since the state machine that controls the LoRa Class A is sensitive, an intermediate level of software is inserted (`LmHandler.c`) between the MAC and the application (refer to LoRaMAC driver in the above figure). With a limited set of APIs, the user is free to implement the Class A state machine at application level. For more details, refer to Section 6 .

The application, built around an infinite loop, manages the low-power mode, runs the interrupt handlers (alarm or GPIO) and calls the LoRa Class A if any task must be done.

2.3 Dynamic view

The MSC (message sequence chart) shown in the figure below depicts a Class A device transmitting an application data and receiving application data from the server.

Figure 3. Class A Tx and Rx processing MSC



Once the radio has completed the application data transmission, an asynchronous RadioIRQ wakes up the system. The RadioIsr here calls `txDone` in the handler mode.

All RadioIsr and MAC timer call a `LoRaMacProcessNotify` callback to request the application layer to update the LoRaMAC state and to do further processing when needed.

For instance, at the end of the reception, `rxDone` is called in the ISR (handler), but all the Rx packet processing including decryption must not be processed in the ISR. This case is an example of call sequence. If no data is received into the Rx1 window, then another Rx2 window is launched..

3 SubGHz HAL driver

This section focuses on the SubGHz HAL (other HAL functions such as timers or GPIO are not detailed).

The SubGHz HAL is directly on top of the sub-GHz radio peripheral (see Figure 2. Static LoRa architecture).

The SubGHz HAL driver is based on a simple one-shot command-oriented architecture (no complete processes). Therefore, no LL driver is defined.

This SubGHz HAL driver is composed the following main parts:

- Handle, initialization and configuration data structures
- Initialization APIs
- Configuration and control APIs
- MSP and event callbacks
- Bus I/O operation based on the SUBGHZ_SPI (Intrinsic services)

As the HAL APIs are mainly based on the bus services to send commands in one-shot operations, no functional state machine is used except the RESET/READY HAL states.

3.1 SubGHz resources

The following HAL SubGHz APIs are called at the initialization of the radio:

- Declare a SUBGHZ_HandleTypeDef handle structure.
- Initialize the sub-GHz radio peripheral by calling the `HAL_SUBGHZ_Init(&hUserSubghz)` API.
- Initialize the SubGHz low-level resources by implementing the `HAL_SUBGHZ_MspInit()` API:
  - PWR configuration: Enable wakeup signal of the sub-GHz radio peripheral.
  - NVIC configuration:
    - Enable the NVIC radio IRQ interrupts.
    - Configure the sub-GHz radio interrupt priority.

The following HAL radio interrupt is called in the s`tm32wlxx_it.c` file:

- `HAL_SUBGHZ_IRQHandler` in the SUBGHZ_Radio_IRQHandler.

3.2 SubGHz data transfers

The **Set** command operation is performed in polling mode with the `HAL_SUBGHZ_ExecSetCmd();` API.

The **Get Status** operation is performed using polling mode with the `HAL_SUBGHZ_ExecGetCmd();` API.

The read/write register accesses are performed in polling mode with following APIs:

- `HAL_SUBGHZ_WriteRegister();`
- `HAL_SUBGHZ_ReadRegister();`
- `HAL_SUBGHZ_WriteRegisters();`
- `HAL_SUBGHZ_ReadRegisters();`
- `HAL_SUBGHZ_WriteBuffer();`
- `HAL_SUBGHZ_ReadBuffer();`

4 BSP STM32WL Nucleo boards

This BSP driver provides a set of functions to manage radio RF services, such as RF switch settings and control, TCXO settings, and DCDC settings.

Note: The radio middleware (SubGHz_Phy) interfaces the radio BSP via `radio_board_if.c/h` interface file. When a custom user board is used, it is recommended to perform one of the following:

- *First option*
  - *Copy the `BSP/STM32WLxx_Nucleo/` directory.*
  - *Rename and update the user BSP APIs with:*
    - *user RF switch configuration and control (such as pin control or number of port)*
    - *user TCXO configuration*
    - *user DC/DC configuration*
  - *replace in the IDE project the STM32WLxx_Nucleo BSP files by the user BSP files.*
- *Second option*
  - *Disable `USE_BSP_DRIVER` in `Core/Inc/platform.h` and implement the BSP functions directly into `radio_board_if.c`.*

4.1 Frequency band

Two types of Nucleo board are available on the STM32WL Series:

- NUCLEO-WL55JC1: high-frequency-band, tuned for frequency between 865 MHz and 930 MHz
- NUCLEO-WL55JC2: low-frequency-band, tuned for frequency between 470 MHz and 520 MHz

If the user tries to run a firmware compiled at 868 MHz on a low-frequency-band board, very poor RF performances are expected.
The firmware does not check the band of the board on which it runs.

4.2 RF switch

The STM32WL Nucleo board embeds an RF 3-port switch (SP3T) to address, with the same board, the following modes:

- high-power transmission
- low-power transmission
- reception

Table 2. **BSP radio switch**

Function	Description
`int32_t BSP_RADIO_Init(void)`	Initializes the RF switch.
`BSP_RADIO_ConfigRFSwitch(BSP_RADIO_Switch_TypeDef Config)`	Configures the RF switch.
`int32_t BSP_RADIO_DeInit (void)`	De-initializes the RF switch.
`int32_t BSP_RADIO_GetTxConfig(void)`	Returns the board configuration: high power, low power or both.

The RF states versus the switch configuration are given in the table below.

Table 3. **RF states versus switch configuration**

RF state	FE_CTRL1	FE_CTRL2	FE_CTRL3
High-power transmission	Low	High	High
Low-power transmission	High	High	High
Reception	High	Low	High

4.3 RF wakeup time

The sub-GHz radio wakeup time is recovered with the following API.

Table 4. **BSP radio wakeup time**

Function	Description
`uint32_t BSP_RADIO_GetWakeUpTime(void)`	Returns `RF_WAKEUP_TIME` value.

The user must start the TCXO by setting the command `RADIO_SET_TCXOMODE` with a timeout depending of the application.
The timeout value can be updated in `radio_conf.h`. Default template value is the following:

```
#define RF_WAKEUP_TIME                      1U
```

4.4 TCXO

Various oscillator types can be mounted on the user application. On the STM32WL Nucleo boards, a TCXO (temperature compensated crystal oscillator) is used to achieve a better frequency accuracy.

Table 5. **BSP radio TCXO**

Function	Description
`uint32_t BSP_RADIO_IsTCXO (void)`	Returns `IS_TCXO_SUPPORTED` value.

The TCXO mode is defined by the STM32WL Nucleo BSP by selecting `USE_BSP_DRIVER` in `Core/Inc/platform.h`.
If the user wants to update this value (no NUCLEO board compliant), or if the BSP is not present, the TXCO mode can be updated in `radio_board_if.h`. Default template value is the following:

```
#define IS_TCXO_SUPPORTED                   1U
```

4.5 Power regulation

Depending on the user application, a LDO or an SMPS (also named DCDC) is used for power regulation. An SMPS is used on the STM32WL Nucleo boards.

Table 6. **BSP radio SMPS**

Function	Description
`uint32_t BSP_RADIO_IsDCDC (void)`	Returns `IS_DCDC_SUPPORTED` value.

The DCDC mode is defined by the STM32WL Nucleo BSP by selecting `USE_BSP_DRIVER` in `Core/Inc/platform.h`.

If the user wants to update this value (no NUCLEO board compliant), or if the BSP is not present, the DCDC mode can be updated in `radio_board_if.h`. Default template value is defined below:

```
#define IS_DCDC_SUPPORTED                         1U
```

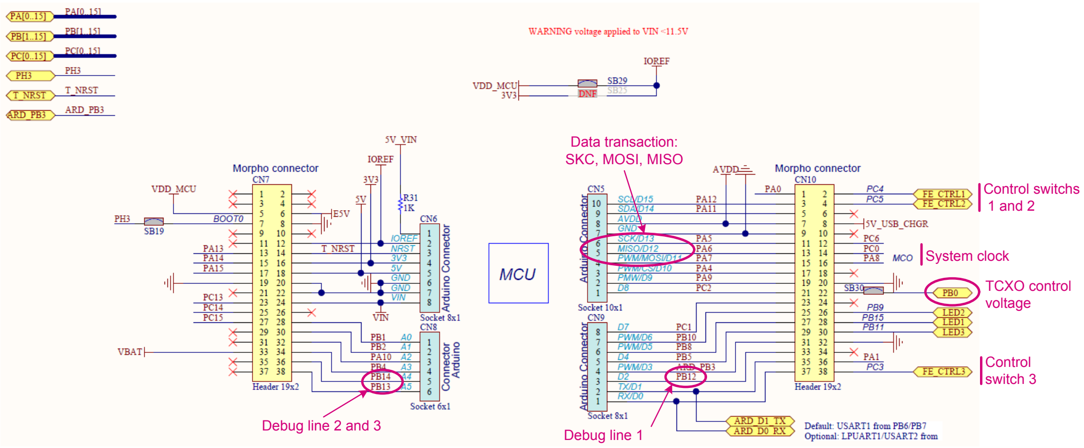
The SMPS on the board can be disabled by setting `IS_DCDC_SUPPORTED = 0`.

4.6 STM32WL Nucleo board schematic

The figure below details the STM32WL Nucleo board (MB1389 reference board) schematic, highlighting some useful signals:

- control switches on PC4, PC5 and PC3
- TCXO control voltage pin on PB0
- debug lines on PB12, PB13 and PB14
- system clock on PA8
- SCK on PA5
- MISO on PA6
- MOSI on PA7

Figure 4. NUCLEO-WL55JC schematic



Data transaction: SKC, MOSI, MISO

Control switchs 1 and 2

System clock

TCXO control voltage

Control switch 3

Debug line 2 and 3

Debug line 1

5 SubGHz_Phy layer middleware description

The radio abstraction layer is composed of two layers:

- high-level layer (`radio.c`)

  It provides a high-level radio interface to the stack middleware. It also maintains radio states, processes interrupts and manages timeouts. It records callbacks and calls them when radio events occur.
- low-level radio drivers

  It is an abstraction layer to the RF interface. This layer knows about the register name and structure, as well as detailed sequence. It is not aware about hardware interface.

The SubGHz_Phy layer middleware contains the radio abstraction layer that interfaces directly on top of the hardware interface provided by BSP (refer Section 4).

The SubGHz_Phy middleware directory is divided in two parts:

- `radio.c`: contains a set of all radio generic callbacks, calling radio_driver functions. This set of APIs is meant to be generic and identical for all radios.
- `radio_driver.c`: low-level radio drivers

`radio_conf.h` contains radio application configuration like RF_WAKEUP_TIME, DCDC dynamic settings, XTAL_FREQ.

5.1 Middleware radio driver structure

A radio generic structure (*struct Radio_s Radio {};*) is defined to register all the callbacks, with the fields detailed in the table below.

Table 7. **Radio_s structure callbacks**

Callback	Description
`radio_conf.h`	TCXO_WAKE_UP_TIME XTAL freq
`RadioInit`	Initializes the radio.
`RadioGetStatus`	Returns the current radio status.
`RadioSetModem`	Configures the radio with the given modem.
`RadioSetChannel`	Sets the channel frequency.
`RadioIsChannelFree`	Checks if the channel is free for the given time.
`RadioRandom`	Generates a 32-bit random value based on the RSSI readings.
`RadioSetRxConfig`	Sets the reception parameters.
`RadioSetTxConfig`	Sets the transmission parameters.
`RadioCheckRfFrequenc`	Checks if the given RF frequency is supported by the hardware.
`RadioTimeOnAir`	Computes the packet time on air (in ms), for the given payload.
`RadioSend`	Prepares the packet to be sent and starts the radio in transmission.
`RadioSleep`	Sets the radio in Sleep mode.
`RadioStandby`	Sets the radio in Standby mode.
`RadioRx`	Sets the radio in reception mode for the given time.
`RadioStartCad`	Starts a CAD (channel activity detection).
`RadioSetTxContinuousWave`	Sets the radio in continuous-wave transmission mode.
`RadioRssi`	Reads the current RSSI value.
`RadioWrite`	Writes the radio register at the specified address.
`RadioRead`	Reads the radio register at the specified address.
`RadioSetMaxPayloadLength`	Sets the maximum payload length.
`RadioSetPublicNetwork`	Sets the network to public or private, and updates the sync byte.
`RadioGetWakeUpTime`	Gets the time required for the radio to exit Sleep mode.
`RadioIrqProcess`	Processes radio IRQ.
`RadioRxBoosted`	Sets the radio in reception mode with max LNA gain for the given time.
`RadioSetRxDutyCycle`	Sets the Rx duty-cycle management parameters.
`RadioTxPrbs`	Sets the transmitter in continuous PRBS mode.
`RadioTxCw`	Sets the transmitter in continuous unmodulated carrier mode.

5.2 Radio IRQ interrupts

The possible sub-GHz radio interrupt sources are detailed in the table below.

Table 8. **Radio IRQ bit mapping and definition**

Bit	Source	Description	Packet type	Operation
0	`txDone`	Packet transmission finished	LoRa and GFSK	Tx
1	`rxDone`	Packet reception finished		Rx
2	`PreambleDetected`	Preamble detected		
3	`SyncDetected`	Synchronization word valid	GFSK	
4	`HeaderValid`	Header valid	LoRa	
5	`HeaderErr`	Header error		
6	`Err`	Preamble, sync word, address, CRC or length error	GFSK	
	`CrcErr`	CRC error	LoRa	
7	`CadDone`	Channel activity detection finished		CAD
8	`CadDetected`	Channel activity detected		
9	`Timeout`	Rx or Tx timeout	LoRa and GFSK	Rx and Tx

For more details, refer to the product reference manual.

6 LoRaWAN middleware description

The LoRa stack middleware is split into the following modules:

- LoRaMAC layer module (in `Middlewares\Third_Party\LoRaWAN\Mac`)
- LoRa utilities module (in `Middlewares\Third_Party\LoRaWAN\Utilities`)
- LoRa crypto module (in `Middlewares\Third_Party\LoRaWAN\Crypto`)
- LoRa LmHandler module (in `Middlewares\Third_Party\LoRaWAN\LmHandler`)

6.1 LoRaWAN middleware features

- Compliant with the specification for the LoRa Alliance protocol, named LoRaWAN
- On-board LoRaWAN Class A, Class B and Class C protocol stack
- EU 868 MHz ISM band ETSI compliant
- EU 433 MHz ISM band ETSI compliant
- US 915 MHz ISM band FCC compliant
- KR 920 MHz ISM band defined by Korean government
- RU 864 MHz ISM band defined by Russian regulation
- CN 779 MHz and CN470Mhz ISM band defined by Chinese government
- AS 923 MHz ISM band defined by Asian governments
- AU 915 MHz ISM band defined by Australian government
- IN 865 MHz ISM band defined by Indian government
- End-device activation either through OTAA or through activation-by-personalization (ABP)
- Adaptive data-rate support
- LoRaWAN test application for certification tests included
- Low-power optimized

6.2 LoRaWAN middleware initialization

The initialization of the LoRaMAC layer is done through the `LoRaMacInitialization` API, that initializes both the preamble run time of the LoRaMAC layer and the callback primitives of the MCPS and MLME services (see the table below).

Table 9. **LoRaWAN middleware initialization**

Function	Description

`LoRaMacStatus_t LoRaMacInitialization`
`(LoRAMacPrimitives_t *primitives,`
`LoRaMacCallback_t *callback,`
`LoRaMacRegion_t region)`
	Initializes the LoRaMAC layer module
	(see Section 6.4 Middleware MAC layer callbacks)

6.3 Middleware MAC layer APIs

The provided APIs follow the definition of “primitive” defined in IEEE802.15.4-2011 (see document [4]).

The interfacing with the LoRaMAC is made through the request-confirm and the indication-response architecture. The application layer can perform a request that the LoRaMAC layer confirms with a confirm primitive. Conversely, the LoRaMAC layer notifies an application layer with the indication primitive in case of any event.

The application layer may respond to an indication with the response primitive. Therefore, all the confirm or indication are implemented using callbacks.

The LoRaMAC layer provides the following services:

- **MCPS** services
  In general, the LoRaMAC layer uses the MCPS services for data transmissions and data receptions.

Table 10. **MCPS services**

Function	Description
`LoRaMacStatus_t LoRaMacMcpsRequest (McpsReq_t* mcpsRequest, bool allowDelayedTx)`	Requests to send Tx data.

- **MLME** services
  The LoRaMAC layer uses the MLME services to manage the LoRaWAN network.

Table 11. **MMLE services**

Function	Description
`LoRaMacStatus_t LoRaMacMlmeRequest (MlmeReq_t *mlmeRequest )`	Generates a join request or requests for a link check.

- **MIB** services
  The MIB stores important runtime information (such as MIB_NETWORK_ACTIVATION or MIB_NET_ID) and holds the configuration of the LoRaMAC layer (for example the MIB_ADR, MIB_APP_KEY).

Table 12. **MIB services**

Function	Description
`LoRaMacStatus_t LoRaMacMibSetRequestConfirm (MibRequestConfirm_t *mibSet)`	Sets attributes of the LoRaMAC layer.
`LoRaMacStatus_t LoRaMacMibGetRequestConfirm (MibRequestConfirm_t *mibGet )`	Gets attributes of the LoRaMAC layer.

6.4 Middleware MAC layer callbacks

The LoRaMAC user event functions primitives (also named callbacks) to be implemented by the application are the following:

- **MCPS**

Table 13. **MCPS primitives**

Function	Description
`void (*MacMcpsConfirm ) (McpsConfirm_t *McpsConfirm)`	Response to a McpsRequest
`Void (*MacMcpsIndication) (McpsIndication_t *McpsIndication)`	Notifies the application that a received packet is available.

- **MLME**

Table 14. **MLME primitive**

Function	Description
`void ( *MacMlmeConfirm ) ( MlmeConfirm_t *MlmeConfirm )`	Manages the LoRaWAN network.

- **MIB**
  No available functions.

6.5 Middleware MAC layer timers

- **Delay Rx window**

Table 15. **Delay Rx window**

Function	Description
`void OnRxWindow1TimerEvent (void)`	Sets the RxDelay1 (ReceiveDelayX - RADIO_WAKEUP_TIME).
`void OnRxWindow2TimerEvent (void)`	Sets the RxDelay2.

- **Delay for Tx frame transmission**

Table 16. **Delay for Tx frame transmission**

Function	Description
`void OnTxDelayedTimerEvent (void)`	Sets the timer for Tx frame transmission.

- **Delay for Rx frame**

Table 17. **Delay for Rx frame**

Function	Description
`void OnAckTimeoutTimerEvent (void)`	Sets timeout for received frame acknowledgment.

6.6 Middleware LmHandler application function

The interface to the MAC is done through the MAC interface `LoRaMac.h` file, in one of the following modes:

- Standard mode

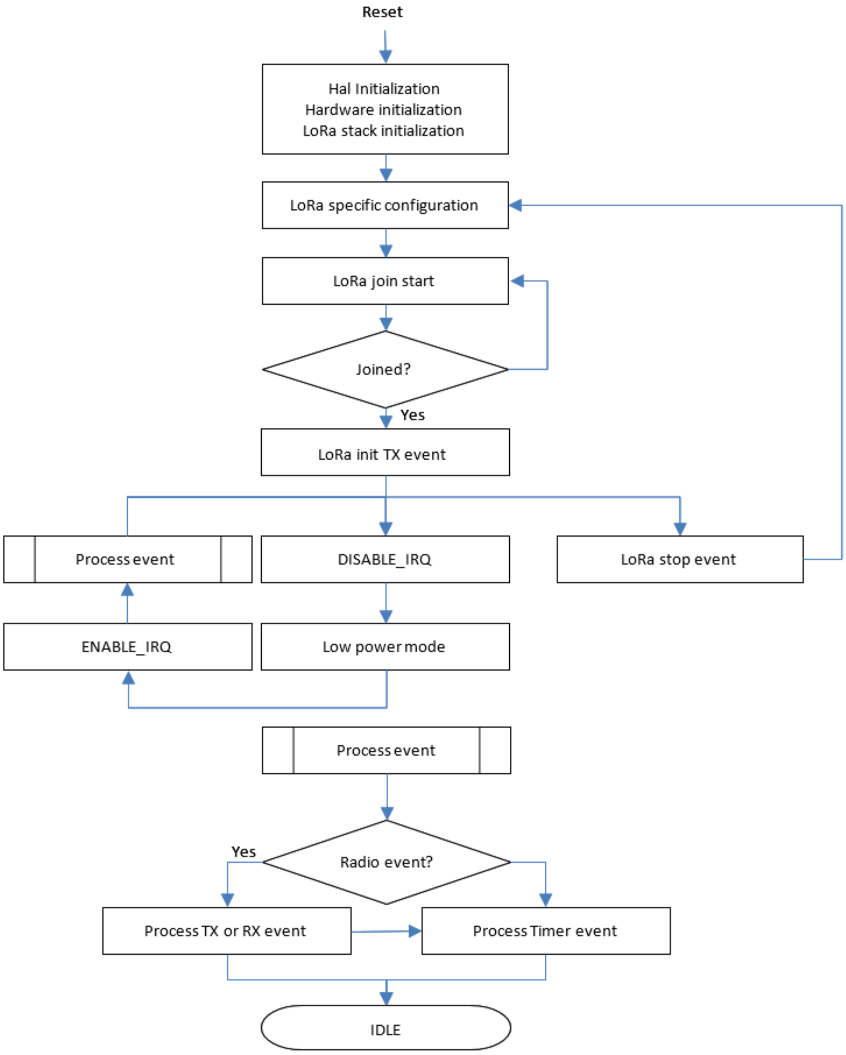
  An interface file (LoRaMAC driver, see Figure 2) is provided to let the user start without worrying about the LoRa state machine. This file is located in `Middlewares\Third_Party\LoRaWAN\LmHandler\LmHandler.c` and implements:
  - a set of APIs to access to the LoRaMAC services
  - the LoRa certification test cases that are not visible to the application layer
- Advanced mode

  The user accesses directly the MAC layer by including the MAC in the user file.

6.6.1 Operation model

The operation model proposed for the LoRaWAN_End_Node is based on ‘event-driven’ paradigms including ‘time-driven’ (see the figure below). The behavior of the LoRa system is triggered either by a timer event or by a radio event plus a guard transition.

Figure 5. Operation model



The next sections detail the LoRaWAN_End_Node and LoRaWAN_AT_Slave APIs used to access the LoRaMAC services. The corresponding interface files are located in
`Middlewares\Third_Party\LoRaWAN\LmHandler\LmHandler.c`
The user must implement the application with these APIs.

An example of LoRaWAN_End_Node application is provided in
`\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.c`.

An example of LoRaWAN_AT_Slave application is provided in
`\Projects\<target>\Applications\LoRaWAN\LoRaWAN_AT_Slave\LoRaWAN\App\lora_app.c`.

6.6.2 Main application functions definition

Table 18. **LoRa initialization**

Function	Description
`LmHandlerErrorStatus_t LmHandlerInit (LmHandlerCallbacks_t *handlerCallbacks)`	Initialization of the LoRa finite state machine

Table 19. **LoRa configuration**

Function	Description
`LmHandlerErrorStatus_t LmHandlerConfigure (LmHandlerParams_t *handlerParams)`	Configuration of all applicative parameters

Table 20. **LoRa join request entry point**

Function	Description
`void LmHandlerJoin (ActivationType_t mode)`	Join request to a network either in OTAA or ABP mode.

Table 21. **LoRa stop**

Function	Description
`void LmHandlerStop (void)`	Stops the LoRa process and waits a new configuration before a rejoin action.

Table 22. **LoRa request class**

Function	Description
`LmHandlerErrorStatus LmHandlerRequestClass (DeviceClass_t newClass)`	Requests the MAC layer to change LoRaWAN class.

Table 23. **Send an uplink frame**

Function	Description
`LmHandlerErrorStatus_t LmHandlerSend (LmHandlerAppData_t *appData, LmHandlerMsgTypes_t isTxConfirmed) TimerTime_t *nextTxIn, bool allowDelayedTx)`	Sends an uplink frame. This frame can be either an unconfirmed empty frame or an unconfirmed/ confirmed payload frame.

6.7 Application callbacks

Callbacks in the tables below are used for both LoRaWAN_End_Node and LoRaWAN_AT_Slave applications.

Table 24. **Current battery level**

Function	Description
`uint8_t GetBatteryLevel (void)`	Gets the battery level.

Table 25. Current temperature

Function	Description
`uint16_t GetTemperature (void)`	Gets the current temperature (in °C) of the device in q7.8 format.

Table 26. Board unique ID

Function	Description
`void GetUniqueId (uint8_t *id)`	Gets the board 64-bit unique ID.

Table 27. Device address

Function	Description
`uint32 GetDevAddr (void)`	Gets the board 32-bit unique ID (LSB).

Table 28. LmHandler process

Function	Description
`void OnMacProcess (void)`	Calls LmHandler Process when a Radio IRQ is received.

Table 29. NVM Data changed

Function	Description
`void OnNvmDataChange (LmHandlerNvmContextStates_t state, uint16_t size)`	Notifies the upper layer that the NVM context has changed.

Table 30. Network parameters changed

Function	Description
`void OnNetworkParametersChange ( CommissioningParams_t *params )`	Notifies the upper layer that network parameters have been set.

Table 31. Join status

Function	Description
`void OnJoinRequest (LmHandlerJoinParams_t *params)`	Notifies the upper layer that a network has been joined.

Table 32. Tx frame done

Function	Description
`void OnTxData (LmHandlerTxParams_t *params)`	Notifies the upper layer that a frame has been transmitted

Table 33. **Rx frame received**

Function	Description
`void OnRxData ( LmHandlerAppData_t *appData, LmHandlerRxParams_t *params)`	Notifies the upper layer that an applicative frame has been received.

Table 34. **Class updated**

Function	Description
`void OnClassChange ( DeviceClass_t deviceClass )`	Confirms the LoRaWAN device class change.

Table 35. **Beacon status changed**

Function	Description
`void OnBeaconStatusChange ( LmHandlerBeaconParams_t *params )`	Notifies the upper layer that the beacon status has changed.

Table 36. **System time updated**

Function	Description
`void OnSysTimeUpdate ( void )`	Notifies the upper layer that the system time has been updated.

6.8 Extended application functions

These callbacks are used for both LoRaWAN_End-Node and LoRaWAN_AT-Slave applications.

Table 37. **Getter/setter functions**

Function	Description
`LmHandlerErrorStatus_t LmHandlerGetCurrentClass( DeviceClass_t *deviceClass)`	Gets the current LoRaWAN class.
`LmHandlerErrorStatus_t LmHandlerGetDevEUI( uint8_t *devEUI)`	Gets the LoRaWAN device EUI.
`LmHandlerErrorStatus_t LmHandlerSetDevEUI( uint8_t *devEUI)`	Sets the LoRaWAN device EUI (if OTAA).
`LmHandlerErrorStatus_t LmHandlerGetAppEUI( uint8_t *appEUI)`	Gets the LoRaWAN App EUI.
`LmHandlerErrorStatus_t LmHandlerSetAppEUI( uint8_t *appEUI)`	Sets the LoRaWAN App EUI.
`LmHandlerErrorStatus_t LmHandlerGetNetworkID( uint32_t *networkId)`	Gets the LoRaWAN Network ID.
`LmHandlerErrorStatus_t LmHandlerSetNetworkID uint32_t networkId)`	Sets the LoRaWAN Network ID.
`LmHandlerErrorStatus_t LmHandlerGetDevAddr( uint32_t *devAddr)`	Gets the LoRaWAN device address.
`LmHandlerErrorStatus_t LmHandlerSetDevAddr( uint32_t devAddr)`	Sets the LoRaWAN device address (if ABP).
`LmHandlerErrorStatus_t LmHandlerGetAppKey( uint8_t *appKey)`	Gets the LoRaWAN Application Root Key.
`LmHandlerErrorStatus_t LmHandlerSetAppKey( uint8_t *appKey)`	Sets the LoRaWAN Application Root Key.
`LmHandlerErrorStatus_t LmHandlerGetNwkKey( uint8_t *nwkKey )`	Gets the LoRaWAN Network Root Key.
`LmHandlerErrorStatus_t LmHandlerSetNwkKey( uint8_t *nwkKey )`	Sets the LoRaWAN Network Root Key.
`LmHandlerErrorStatus_t LmHandlerGetNwkSKey( uint8_t *nwkSKey )`	Gets the LoRaWAN Network Session Key.
`LmHandlerErrorStatus_t LmHandlerSetNwkSKey( uint8_t *nwkSKey)`	Sets the LoRaWAN Network Session Key.
`LmHandlerErrorStatus_t LmHandlerGetAppSKey( uint8_t *appSKey )`	Gets the LoRaWAN Application Session Key.
`LmHandlerErrorStatus_t LmHandlerSetAppSKey( uint8_t *appSKey)`	Sets the LoRaWAN Application Session Key.
`LmHandlerErrorStatus_t LmHandlerGetActiveRegion( LoRaMacRegion_t *region)`	Gets the active region.
`LmHandlerErrorStatus_t LmHandlerSetActiveRegion( LoRaMacRegion_t region)`	Sets the active region.
`LmHandlerErrorStatus_t LmHandlerGetAdrEnable( bool *adrEnable)`	Gets the adaptive data rate state.
`LmHandlerErrorStatus_t LmHandlerSetAdrEnable( bool adrEnable)`	Sets the adaptive data rate state.
`LmHandlerErrorStatus_t LmHandlerGetTxDatarate( int8_t *txDatarate)`	Gets the current Tx data rate.
`LmHandlerErrorStatus_t LmHandlerSetTxDatarate( int8_t txDatarate)`	Sets the Tx data rate (if adaptive DR disabled).

Function	Description
`LmHandlerErrorStatus_t LmHandlerGetDutyCycleEnable ( bool *dutyCycleEnable)`	Gets the current Tx duty cycle state.
`LmHandlerErrorStatus_t LmHandlerSetDutyCycleEnable ( bool dutyCycleEnable)`	Sets the Tx duty cycle state.
`LmHandlerErrorStatus_t LmHandlerGetRX2Params ( RxChannelParams_t *rxParams)`	Gets the current Rx2 data rate and frequency conf.
`LmHandlerErrorStatus_t LmHandlerSetRX2Params ( RxChannelParams_t *rxParams)`	Sets the Rx2 data rate and frequency conf.
`LmHandlerErrorStatus_t LmHandlerGetTxPower( int8_t *txPower)`	Gets the current Tx power value.
`LmHandlerErrorStatus_t LmHandlerSetTxPower( int8_t txPower)`	Sets the Tx power value.
`LmHandlerErrorStatus_t LmHandlerGetRx1Delay( uint32_t *rxDelay)`	Gets the current Rx1 delay (after Tx window).
`LmHandlerErrorStatus_t LmHandlerSetRx1Delay( uint32_t rxDelay)`	Sets the Rx1 delay (after Tx window).
`LmHandlerErrorStatus_t LmHandlerGetRx2Delay( uint32_t *rxDelay)`	Gets the current Rx2 delay (after Tx window).
`LmHandlerErrorStatus_t LmHandlerSetRx2Delay( uint32_t rxDelay)`	Sets the Rx2 delay (after Tx window).
`LmHandlerErrorStatus_t LmHandlerGetJoinRx1Delay( uint32_t *rxDelay)`	Gets the current Join Rx1 delay (after Tx window).
`LmHandlerErrorStatus_t LmHandlerSetJoinRx1Delay( uint32_t rxDelay)`	Sets the Join Rx1 delay (after Tx window).
`LmHandlerErrorStatus_t LmHandlerGetJoinRx2Delay( uint32_t *rxDelay)`	Get the current Join Rx2 delay (after Tx window)
`LmHandlerErrorStatus_t LmHandlerSetJoinRx2Delay( uint32_t rxDelay)`	Sets the Join Rx2 delay (after Tx window).
`LmHandlerErrorStatus_t LmHandlerGetPingPeriodicity ( uint8_t pingPeriodicity)`	Gets the current Rx Ping Slot periodicity (If `LORAMAC_CLASSB_ENABLED`)
`LmHandlerErrorStatus_t LmHandlerSetPingPeriodicity ( uint8_t pingPeriodicity)`	Sets the Rx Ping Slot periodicity (If `LORAMAC_CLASSB_ENABLED`)
`LmHandlerErrorStatus_t LmHandlerGetBeaconState ( BeaconState_t *beaconState)`	Gets the beacon state (If `LORAMAC_CLASSB_ENABLED`)
`LmHandlerErrorStatus_t LmHandlerDeviceTimeReq( void )`	Requests network server time update.
`LmHandlerErrorStatus_t LmHandlerLinkCheckReq( void )`	Requests Link connectivity check.

7 Utilities description

Utilities are located in the \Utilities directory.

Main APIs are described below. Secondary APIs and additional information can be found on the header files related to the drivers.

7.1 Sequencer

The sequencer provides a robust and easy framework to execute tasks in the background and enters low-power mode when there is no more activity. The sequencer implements a mechanism to prevent race conditions.

In addition, the sequencer provides an event feature allowing any function to wait for an event (where particular event is set by interrupt) and MIPS and power to be easily saved in any application that implements “run to completion” command.

The utilities_def.h file located in the project sub-folder is used to configure the task and event IDs. The ones already listed must not be removed.

The sequencer is not an OS. Any task is run to completion and can not switch to another task like a RTOS can do on RTOS tick unless a task suspends itself by calling UTIL_SEQ_WaitEvt. Moreover, one single-memory stack is used. The sequencer is an advanced ‘while loop’ centralizing task and event bitmap flags.

The sequencer provides the following features:

- Advanced and packaged while loop system
- Support up to 32 tasks and 32 events
- Task registration and execution
- Waiting event and set event
- Task priority setting
- Race condition safe low-power entry

To use the sequencer, the application must perform the following:

- Set the number of maximum of supported functions, 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.

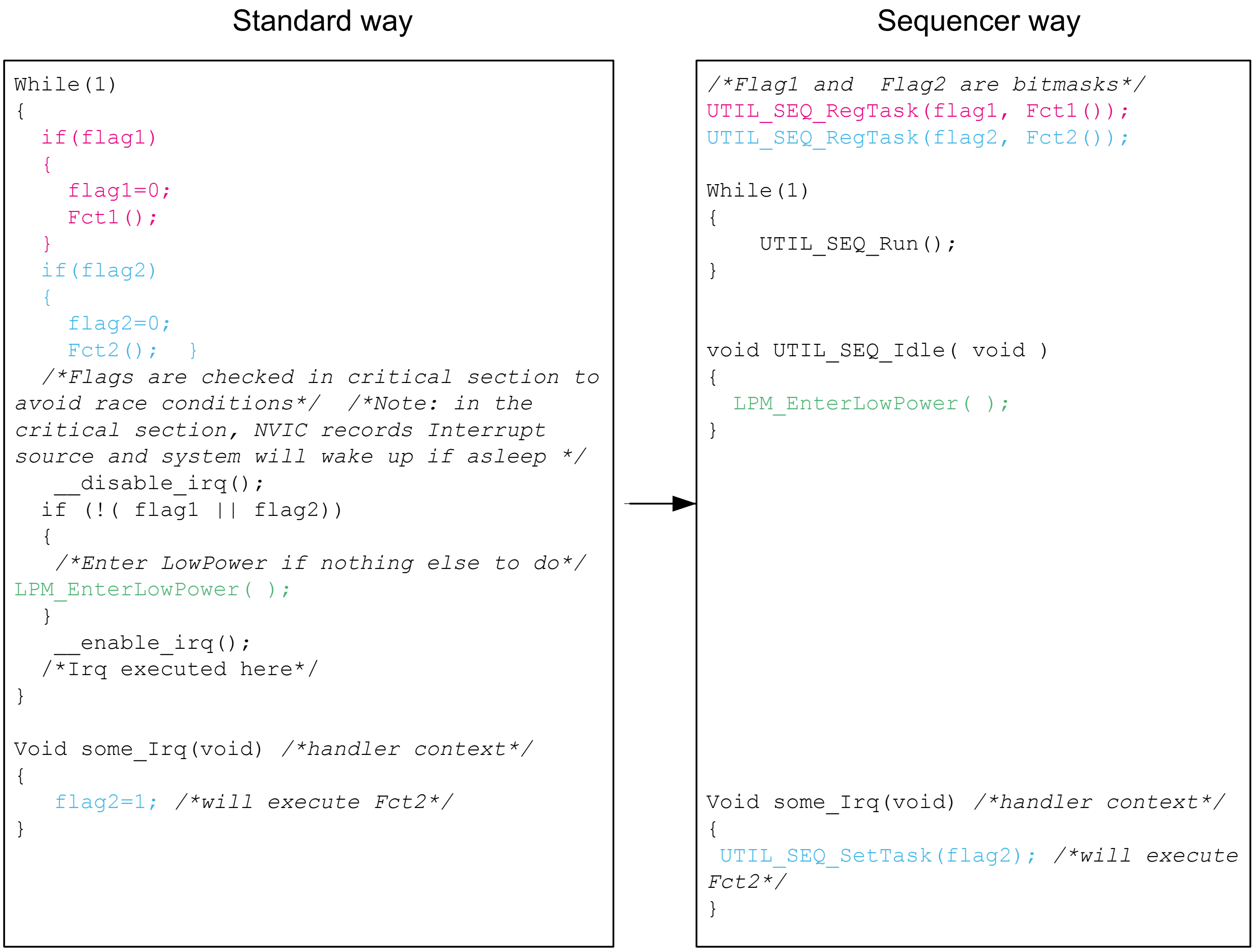
The sequencer utility is located in Utilities\sequencer\stm32_seq.c.

Table 38. Sequencer APIs

Function	Description
void UTIL_SEQ_Idle(void)	Called (in critical section - PRIMASK) when there is nothing to execute.
void UTIL_SEQ_Run(UTIL_SEQ_bm_t mask_bm)	Requests the sequencer to execute functions that are pending and enabled in the mask mask_bm.
void UTIL_SEQ_RegTask(UTIL_SEQ_bm_t task_id_bm, uint32_t flags, void (*task)(void))	Registers a function (task) associated with a signal (task_id_bm) in the sequencer. The task_id_bm must have a single bit set.
void UTIL_SEQ_SetTask(UTIL_SEQ_bm_t taskId_bm, uint32_t task_Prio)	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.
void UTIL_SEQ_SetEvt(UTIL_SEQ_bm_t EvtId_bm);	Waits for a specific event to be set.
void UTIL_SEQ_SetEvt(UTIL_SEQ_bm_t EvtId_bm);	Sets an event that waits with UTIL_SEQ_WaitEvt().

The figure below compares the standard while-loop implementation with the sequencer while-loop implementation.

Figure 6. While-loop standard vs. sequencer implementation



Standard way

```
While(1)
{
  if(flag1)
  {
    flag1=0;
    Fct1();
  }
  if(flag2)
  {
    flag2=0;
    Fct2();  }
  /*Flags are checked in critical section to
avoid race conditions*/  /*Note: in the
critical section, NVIC records Interrupt
source and system will wake up if asleep */
   __disable_irq();
  if (!( flag1 || flag2))
  {
   /*Enter LowPower if nothing else to do*/
LPM_EnterLowPower( );
  }
   __enable_irq();
  /*Irq executed here*/
}

Void some_Irq(void) /*handler context*/
{
   flag2=1; /*will execute Fct2*/
}
```

Sequencer way

```
/*Flag1 and  Flag2 are bitmasks*/
UTIL_SEQ_RegTask(flag1, Fct1());
UTIL_SEQ_RegTask(flag2, Fct2());

While(1)
{
    UTIL_SEQ_Run();
}


void UTIL_SEQ_Idle( void )
{
  LPM_EnterLowPower( );
}

Void some_Irq(void) /*handler context*/
{
 UTIL_SEQ_SetTask(flag2); /*will execute
Fct2*/
}
```

7.2 Timer server

The timer server allows the user to request timed-tasks execution. As the hardware timer is based on the RTC, the time is always counted, even in low-power modes.

The timer server provides a reliable clock for the user and the stack. The user can request as many timers as the application requires.

The timer server is located in `Utilities\timer\stm32_timer.c`.

Table 39. Timer server APIs

Function	Description
`UTIL_TIMER_Status_t UTIL_TIMER_Init( void )`	Initializes the timer server.
`UTIL_TIMER_Status_t UTIL_TIMER_Create ( UTIL_TIMER_Object_t *TimerObject, uint32_t PeriodValue, UTIL_TIMER_Mode_t Mode, void ( *Callback ) ( void *), void *Argument)`	Creates the timer object and associates a callback function when timer elapses.
`UTIL_TIMER_Status_t UTIL_TIMER_SetPeriod(UTIL_TIMER_Object_t *TimerObject, uint32_t NewPeriodValue)`	Updates the period and starts the timer with a timeout value (milliseconds).
`UTIL_TIMER_Status_t UTIL_TIMER_Start ( UTIL_TIMER_Object_t *TimerObject )`	Starts and adds the timer object to the list of timer events.
`UTIL_TIMER_Status_t UTIL_TIMER_Stop ( UTIL_TIMER_Object_t *TimerObject )`	Stops and removes the timer object from the list of timer events.

7.3 Low-power functions

The `low-power` utility centralizes the low-power requirement of separate modules implemented by the firmware, and manages the low-power entry when the system enters idle mode. For example, when the DMA is in use to print data to the console, the system must not enter a low-power mode below Sleep mode because the DMA clock is switched off in Stop mode

The APIs presented in the table below are used to manage the low-power modes of the core MCU. The `low-power` utility is located in `Utilities\lpm\tiny_lpm\stm32_lpm.c`.

Table 40. Low-power APIs

Function	Description
`void UTIL_LPM_EnterLowPower( void )`	Enters the selected low-power mode. Called by idle state of the system
`void UTIL_LPM_SetStopMode( UTIL_LPM_bm_t lpm_id_bm, UTIL_LPM_State_t state );`	Sets Stop mode. `id` defines the process mode requested: `UTIL_LPM_ENABLE` or `UTIL_LPM_DISABLE`.(1)
`void UTIL_LPM_SetOffMode( UTIL_LPM_bm_t lpm_id_bm, UTIL_LPM_State_t state );`	Sets Stop mode. `id` defines the process mode requested: `UTIL_LPM_ENABLE` or `UTIL_LPM_DISABLE`.
`UTIL_LPM_Mode_t UTIL_LPM_GetMode( void )`	Returns the currently selected low-power mode.

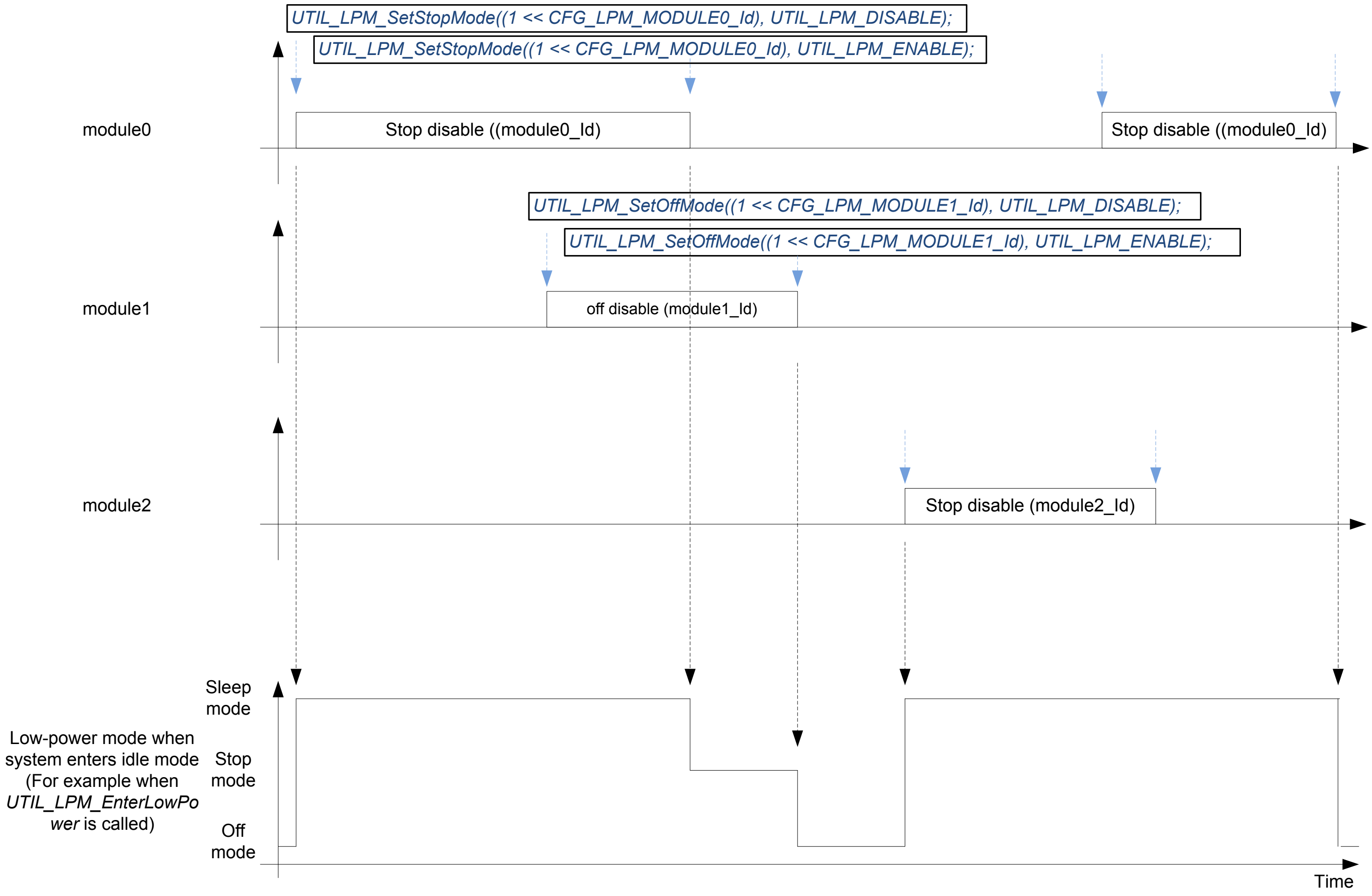
1. *Bitmaps for which the shift values are defined in `utilities_def.h`.*

The default low-power mode is Off mode, that may be Standby or Shutdown mode (defined in `void PWR_EnterOffMode (void)` from Table 41):

- If Stop mode is disabled by at least one firmware module and low-power is entered, Sleep mode is selected.
- If Stop mode is not disabled by any firmware module, Off mode is disabled by at least one firmware module, and low-power is entered. Stop mode is selected.
- If Stop mode is not disabled by any firmware module, Off mode is not disabled by any firmware module, and low-power is entered. Off mode is selected.

The figure below depicts the behavior with three different firmware modules setting dependently their low-power requirements and low-power mode, selected when the system enters low-power mode.

Figure 7. Example of low-power mode dynamic view


UTIL_LPM_SetStopMode((1 << CFG_LPM_MODULE0_Id), UTIL_LPM_DISABLE);
UTIL_LPM_SetStopMode((1 << CFG_LPM_MODULE0_Id), UTIL_LPM_ENABLE);
module0
Stop disable ((module0_Id)
Stop disable ((module0_Id)
UTIL_LPM_SetOffMode((1 << CFG_LPM_MODULE1_Id), UTIL_LPM_DISABLE);
UTIL_LPM_SetOffMode((1 << CFG_LPM_MODULE1_Id), UTIL_LPM_ENABLE);
module1
off disable (module1_Id)
module2
Stop disable (module2_Id)
Sleep mode
Stop mode
Off mode
Low-power mode when system enters idle mode (For example when UTIL_LPM_EnterLowPower is called)
Time

Low-level APIs must be implemented to define what the system must do to enter/exit a low-power mode. These functions are implemented in `stm32_lpm_if.c` of project sub-folder.

Table 41. **Low-level APIs**

Function	Description
`void PWR_EnterSleepMode (void)`	API called before entering Sleep mode
`void PWR_ExitSleepMode (void)`	API called on exiting Sleep mode
`void PWR_EnterStopMode (void)`	API called before Stop mode
`void PWR_ExitStopMode (void)`	API called on exiting Stop mode
`void PWR_EnterOffMode (void)`	API called before entering Off mode
`void PWR_ExitOffMode(void)`	API called on exiting Off mode

In Sleep mode, the core clock is stopped. Each peripherals clocks can be gated or not. The power is maintained on all peripherals.

In Stop 2 mode, most peripheral clocks are stopped. Most peripheral supplies are switched off. Some registers of the peripherals are not retained and must be reinitialized on Stop 2 mode exit. Memory and core registers are retained.

In Standby mode, all clocks are switched off except LSI and LSE. All peripheral supplies are switched off (except BOR,backup registers, GPIO pull, and RTC), with no retention (except additional SRAM2 with retention), and must be reinitialized on Standby mode exit. Core registers are not retained and must be reinitialized on Standby mode exit.

Note: The sub-GHz radio supply is independent from the rest of the system. See the product reference manual for more details.

7.4 System time

The MCU time is referenced to the MCU reset. The system time is able to record the UNIX® epoch time.

The APIs presented in the table below are used to manage the system time of the core MCU. The `systime` utility is located in `Utilities\misc\stm32_systime.c`.

Table 42. **System time functions**

Function	Description
`void SysTimeSet (SysTime_t sysTime)`	Based on an input UNIX epoch in seconds and sub-seconds, the difference with the MCU time is stored in the backup register (retained even in Standby mode).[1]
`SysTime_t SysTimeGet (void)`	Gets the current system time.[1]
`uint32_t SysTimeMkTime (const struct tm* localtime)`	Converts local time into UNIX epoch time. [2]

`void SysTimeLocalTime`
`(const uint32_t timestamp, struct tm *localtime)`
	Converts UNIX epoch time into local time.[2]

[1] *The system time reference is UNIX epoch starting January 1st 1970.*

[2] `SysTimeMkTime` *and* `SysTimeLocalTime` *are also provided in order to convert epoch into* `tm` *structure as specified by the* `time.h` *interface.*

To convert UNIX time to local time, a time zone must be added and leap seconds must be removed. In 2018, 18 leap seconds must be removed. In Paris summer time, there are two hours difference from Greenwich time. Assuming time is set, local time can be printed on terminal with the code below.

```
{
SysTime_t UnixEpoch = SysTimeGet();
struct tm localtime;
UnixEpoch.Seconds-=18; /*removing leap seconds*/
UnixEpoch.Seconds+=3600*2; /*adding 2 hours*/
SysTimeLocalTime(UnixEpoch.Seconds, & localtime);
PRINTF ("it's %02dh%02dm%02ds on %02d/%02d/%04d\n\r",
localtime.tm_hour, localtime.tm_min, localtime.tm_sec,
localtime.tm_mday, localtime.tm_mon+1, localtime.tm_year + 1900);
}
```

7.5 Trace

The trace module enables to print data on a COM port using DMA. The APIs presented in the table below are used to manage the trace functions.

The `trace` utility is located in `Utilities\trace\adv_trace\stm32_adv_trace.c`.

Table 43. Trace functions

Function	Description

`UTIL_ADV_TRACE_Status_t`
`UTIL_ADV_TRACE_Init( void )`
	`TraceInit` must be called at the application initialization. Initializes the com or vcom hardware in DMA mode and registers the callback to be processed at DMA transmission completion.

`UTIL_ADV_TRACE_Status_t`
`UTIL_ADV_TRACE_COND_FSend(uint32_t VerboseLevel,`
`uint32_t Region,`
`uint32_t TimeStampState, const char *strFormat, ...)`
	Converts string format into a buffer and posts it to the circular queue for printing.

`UTIL_ADV_TRACE_Status_t`
`UTIL_ADV_TRACE_COND_Send(uint32_t VerboseLevel, uint32_t Region, uint32_t TimeStampState, const uint8_t *pdata, uint16_t length)`
	Posts data of length = `len` and posts it to the circular queue for printing.

`UTIL_ADV_TRACE_Status_t`
`UTIL_ADV_TRACE_COND_ZCSend_Allocation(uint32_t VerboseLevel, uint32_t Region, uint32_t TimeStampState, uint16_t length,uint8_t **pData, uint16_t *FifoSize, uint16_t *WritePos)`
	Writes user formatted data directly in the FIFO (Z-Cpy).

The status values of the trace functions are defined in the structure `UTIL_ADV_TRACE_Status_t` as follows.

```
typedef enum {
  UTIL_ADV_TRACE_OK                    = 0,     /*Operation terminated successfully*/
  UTIL_ADV_TRACE_INVALID_PARAM         = -1,     /*Invalid Parameter*/
  UTIL_ADV_TRACE_HW_ERROR              = -2,     /*Hardware Error*/
  UTIL_ADV_TRACE_MEM_ERROR             = -3,     /*Memory Allocation Error*/
  UTIL_ADV_TRACE_UNKNOWN_ERROR         = -4,     /*Unknown Error*/
  UTIL_ADV_TRACE_GIVEUP                = -5,     /*!< trace give up*/
  UTIL_ADV_TRACE_REGIONMASKED          = -6      /*!< trace region masked*/
} UTIL_ADV_TRACE_Status_t;
```

The UTIL_ADV_TRACE_COND_FSend (..) function can be used:

- in polling mode when no real time constraints apply: for example, during application initialization

```
#define APP_PPRINTF(...)  do{ } while( UTIL_ADV_TRACE_OK \
!= UTIL_ADV_TRACE_COND_FSend(VLEVEL_ALWAYS, T_REG_OFF, TS_OFF, __VA_ARGS__) )
/* Polling Mode */
```

- in real-time mode: when there is no space left in the circular queue, the string is not added and is not printed out in com port

```
#define APP_LOG(TS,VL,...)do{
{UTIL_ADV_TRACE_COND_FSend(VL, T_REG_OFF, TS, __VA_ARGS__);} }while(0);)
```

where:

  - VL is the VerboseLevel of the trace.
  - TS allows a timestamp to be added to the trace (TS_ON or TS_OFF).

The application verbose level is set in Core\Inc\sys_conf.h with:

```
#define VERBOSE_LEVEL <VLEVEL>
```

where VLEVEL can be VLEVEL_OFF, VLEVEL_L, VLEVEL_M or VLEVEL_H.

UTIL_ADV_TRACE_COND_FSend (..) is displayed only if VLEVEL ≥ VerboseLevel.

The buffer length can be increased in case it is saturated in Core\Inc\utilities_conf.h with:

```
#define UTIL_ADV_TRACE_TMP_BUF_SIZE 256U
```

The utility provides hooks to be implemented in order to forbid the system to enter Stop or lower mode while the DMA is active:

-
  ```
  void UTIL_ADV_TRACE_PreSendHook (void)
  { UTIL_LPM_SetStopMode((1 << CFG_LPM_UART_TX_Id) , UTIL_LPM_DISABLE ); }
  ```

-
  ```
  void UTIL_ADV_TRACE_PostSendHook (void)
  { UTIL_LPM_SetStopMode((1 << CFG_LPM_UART_TX_Id) , UTIL_LPM_ENABLE );}
  ```

8 LoRaWAN_End_Node application

This application measures the battery level and the temperature of the MCU. These values are sent periodically to the LoRa network using the LoRa radio in Class A at 868 MHz.

To launch the LoRaWAN_End_Node project, go to `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node` and choose the favorite toolchain folder (in the IDE environment). Select the LoRa project from the proper target board.

Focus on the configuration described below to setup the application.

8.1 Device configuration

8.1.1 Activation methods and keys

There are two ways to activate a device on the network, either by OTAA or by ABP.

The global variable “ActivationType” in the application must be adjusted to activate the device with the selected mode.

```
static ActivationType_t ActivationType = LORAWAN_DEFAULT_ACTIVATION_TYPE;
```

in `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.c`

and

```
#define LORAWAN_DEFAULT_ACTIVATION_TYPE ACTIVATION_TYPE_OTAA
```

in `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.h`

where `ActivationType_t enum` is defined as follows:

```
typedef enum eActivationType {
    ACTIVATION_TYPE_NONE = 0, /* None */
    ACTIVATION_TYPE_ABP = 1, /* Activation by personalization */
    ACTIVATION_TYPE_OTAA = 2, /* Over the Air Activation */
```

`\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\se-identity.h` file contains commissioning data useful for device activation.

8.1.2 LoRa Class activation

By default, Class A is defined. To change the class activation (Class A, Class B, or Class C), the user must:

- set the code

  ```
  #define LORAWAN_DEFAULT_CLASS CLASS_B;
  ```

  in `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.h`

- set the code

  ```
  #define LORAMAC_CLASSB_ENABLED 1
  ```

  in `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lorawan_conf.h`

8.1.3 Tx trigger

There are two ways to generate an uplink action, with the EventType global variable in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.c:

- by timer
- by an external event

with the code

```
static TxEventType_t EventType = TX_ON_TIMER;
```

where TxEventType_t enum is defined as follows:

```
typedef enum TxEventType_e {
    TX_ON_TIMER = 0, /* App data transmission issue based on timer */
    TX_ON_EVENT = 1, /* App data transmission by external event */
}TxEventType_t;
```

The TX_ON_EVENT feature uses the button 1 as event in the LoRaWAN_End_Node application.

8.1.4 Duty cycle

The duty cycle value (in ms) to be used for the application is defined in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.h, with the code below (for example):

```
#define APP_TX_DUTYCYCLE                            10000 /* 10s duty cycle */
```

8.1.5 Application port

The application port to be used for the application is defined in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.h, with the code below (for example):

```
#define LORAWAN_APP_PORT                            2
```

Note: LORAWAN_APP_PORT must not use port 224 that is reserved for certification.

8.1.6 Confirm/unconfirmed mode

The confirm/unconfirmed mode to be used for the application is defined in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.h, with the code below:

```
#define LORAWAN_DEFAULT_CONFIRMED_MSG_STATE          LORAMAC_HANDLER_UNCONFIRMED_MSG
```

8.1.7 Data buffer size

The size of the buffer sent to the network is defined in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.h, with the code below:

```
#define LORAWAN_APP_DATA_BUFFER_MAX_SIZE                           242
```

8.1.8 Adaptive data rate (ADR)

The ADR is enabled in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.h, with the code below:

```
#define LORAWAN_ADR_STATE                                   LORAMAC_HANDLER_ADR_ON
```

When the ADR is disabled, the default rate is set in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.h, with the code below:

```
#define LORAWAN_DEFAULT_DATA_RATE                           DR_0
```

8.1.9 Ping periodicity

If the device is able to switch in Class B, the default Rx Ping slot periodicity must be enabled in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.h with the code below.

```
#define LORAWAN_DEFAULT_PING_SLOT_PERIODICITY 4
```

where the expected value must be in the 0-7 range.
The resulting period time is defined by:

```
period = 2^LORAWAN_DEFAULT_PING_SLOT_PERIODICITY
```

8.1.10 LoRa band selection

The region and its corresponding band selection are defined in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\Target\lorawan_conf.h with the code below:

```
#define REGION_AS923
#define REGION_AU915
#define REGION_CN470
#define REGION_CN779
#define REGION_EU433
#define REGION_EU868
#define REGION_KR920
#define REGION_IN865
#define REGION_US915
#define REGION_RU864
```

Note: *Several regions can be defined on the same application.*

Depending on the region, the default active region must be defined in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\Core\Inc\sys_conf.h with the code (example for Europe)

```
#define ACTIVE_REGION LORAMAC_REGION_EU868
```

8.1.11 Debug switch

The debug mode is enabled in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\Core\Inc\sys_conf.h with the code below:

```
#define DEBUGGER_ENABLED    1  /* ON=1, OFF=0 */
```

The debug mode enables the SWD pins, even when the MCU goes in low-power mode.
The probe-pin mode is enabled in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\Core\Inc\sys_conf.h with the code below:

```
#define PROBE_PINS_ENABLED 1 /* ON=1, OFF=0 */
```

The probe-pin mode enables PROBE_GPIO_WRITE, PROBE_GPIO_SET_LINE and PROBE_GPIO_RST_LINE macros, as well as the debugger mode, even when the MCU goes in low-power mode.

Note: *In order to enable a true low-power, #define DEBUGGER_ENABLED must be reset.*

8.1.12 Low-power switch

When the system is in idle, it enters the low-power Stop 2 mode.
This entry in Stop 2 mode can be disabled in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\Core\Inc\sys_conf.h with the code below:

```
#define LOW_POWER_DISABLE    0 /* Low power enabled = 0, Low power disabled = 1 */
```

where:

- Low power enabled = 0 means the MCU enters to Stop 2 mode
  Stop 2 is a Stop mode with low-power regulator and V_{DD12I} interruptible digital core domain supply OFF. Less peripherals are activated than in low-power Stop 1 mode to reduce power consumption. See the document [3] for more details
- Low power disabled = 1 means the MCU enters only in Sleep mode.

8.1.13 Trace level

The trace mode is enabled in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\Core\Inc\sys_conf.h with the code below:

```
#define APP_LOG_ENABLED    1
```

The trace level is selected in \Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\Core\Inc\sys_conf.h with the code below :

```
#define VERBOSE_LEVEL VLEVEL_M
```

The following trace levels are proposed:

- VLEVEL_OFF: all traces disabled
- VLEVEL_L: functional traces enabled
- VLEVEL_M: debug traces enabled
- VLEVEL_H: all traces enabled

8.2 Device configuration summary for LoRaWAN_End_Node application

Table 44. Switch options for LoRaWAN_End_Node application configuration

Project module	Detail	Switch option	Definition	Location
LoRa stack	Identification	STATIC_DEVICE_EUI	Static or dynamic end-device identifying	se-identity.h
	Address	STATIC_DEVICE_ADDRESS	Static or dynamic end-device address	
	Supported regions	REGION_EU868	Regions supported by the device	lorawan_conf.h
		REGION_EU433		
		REGION_US915		
		REGION_AS923		
		REGION_AU915		
		REGION_CN470		
		REGION_CN779		
		REGION_IN865		
		REGION_RU864		
		REGION_KR920		
	Limited channels	HYBRID_ENABLED	Limits the number of usable channels by default for AU915, CN470 and US915 regions.	
	Read keys	KEY_EXTRACTABLE	Defines the read access of the keys in the memory.	
	Optional class	LORAMAC_CLASSB_ENABLED	End-device Class B capability	
Application	Tx trigger	EventType = TX_ON_TIMER	Tx trigger method	lora_app.c
	Class choice	LORAWAN_DEFAULT_CLASS	Sets class of the device.	lora_app.h
	Duty cycle	APP_TX_DUTYCYCLE	Time period between two Tx sent	
	App port	LORAWAN_USER_APP_PORT	LoRa port used by the Tx data frame	
	Confirmed mode	LORAWAN_DEFAULT_CONFIRMED _MSG_STATE	Confirmed mode selection	
	Adaptive data rate	LORAWAN_ADR_STATE	ADR selection	
	Default data rate	LORAWAN_DEFAULT_DATA_RATE	Data rate if ADR is disabled	
	Maximum data buffer size	LORAWAN_APP_DATA_BUFFER _MAX_SIZE	Buffer size definition	
	Ping period	LORAWAN_DEFAULT_PING _SLOT_PERIODICITY	Rx ping slot period	
	Network Join activation	LORAWAN_DEFAULT _ACTIVATION_TYPE	Activation procedure default choice	
	Initial region	ACTIVE_REGION	Region used at device startup	
	Debug	DEBUGGER_ENABLED	Enables SWD pins.	sys_conf.h
	Probe pins	PROBE_PINS_ENABLED	Enables four pins usable as probe signals by the middleware radio layer.	

Project mo- dule	Detail	Switch option	Definition	Location
Application	Low power	`LOW_POWER_DISABLE`	Disables low-power mode	`sys_conf.h`
	Trace enable	`APP_LOG_ENABLED`	Enables the trace mode.	
	Trace level	`VERBOSE_LEVEL`	Enables the trace level.	

9 LoRaWAN_AT_Slave application

The purpose of this example is to implement a LoRa modem controlled though the AT command interface over UART by an external host.

The external host can be a host microcontroller embedding the application and the AT driver, or simply a computer executing a terminal.

This application targets the STM32WL Nucleo board (NUCLEO-WL55JC).

The LoRaWAN_AT_Slave example implements the LoRaWAN stack driving the built-in LoRa radio. The stack is controlled through the AT command interface over UART. The modem is always in Stop 2 mode unless it processes an AT command from the external host.

To launch the LoRaWAN_AT_Slave project, the user must go to `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_AT_Slave` and follow the same procedure as for the LoRaWAN_End_Node project to launch the preferred toolchain.

The document [2] gives the list of AT commands and their description.

The table below summarizes the main options for the LoRaWAN_AT_Slave application configuration.

Table 45. Switch options for LoRaWAN_AT_Slave application configuration

Project module	Detail	Switch option	Definition	Location
LoRa stack	Identification	STATIC_DEVICE_EUI	Static or dynamic end-device identifying	se-identity.h
	Address	STATIC_DEVICE_ADDRESS	Static or dynamic end-device address	
	Supported regions	REGION_EU868	Regions supported by the device	lorawan_conf.h
		REGION_EU433		
		REGION_US915		
		REGION_AS923		
		REGION_AU915		
		REGION_CN470		
		REGION_CN779		
		REGION_IN865		
		REGION_RU864		
		REGION_KR920		
	Limited channels	HYBRID_ENABLED	Limits the number of usable channels by default for AU915, CN470 and US915 regions.	
	Read keys	KEY_EXTRACTABLE	Defines the read access of the keys in the memory.	
	Optional class	LORAMAC_CLASSB_ENABLED	End-device Class B capability	
Application	Adaptive data rate	LORAWAN_ADR_STATE	ADR selection	lora_app.h
	Default data rate	LORAWAN_DEFAULT_DATA_RATE	Data rate if ADR is disabled	
	Ping period	LORAWAN_DEFAULT_PING SLOT_PERIODICITY	Rx ping slot period	
	Initial region	ACTIVE_REGION	Region used at device startup	
	Debug	DEBUGGER_ENABLED	Enables SWD pins.	sys_conf.h

Project module	Detail	Switch option	Definition	Location
Application	Probe pins	PROBE_PINS_ENABLED	Enables four pins usable as probe signals by the middleware radio layer .	sys_conf.h
	Low power	LOW_POWER_DISABLE	Disables low-power mode.	
	Trace enable	APP_LOG_ENABLED	Enables the trace mode.	
	Trace level	VERBOSE_LEVEL	Enables the trace level.	

10 SubGhz_Phy_PingPong application

This application shows a simple Rx/Tx RF link between the two PingPong devices (one called Ping, the other called Pong).

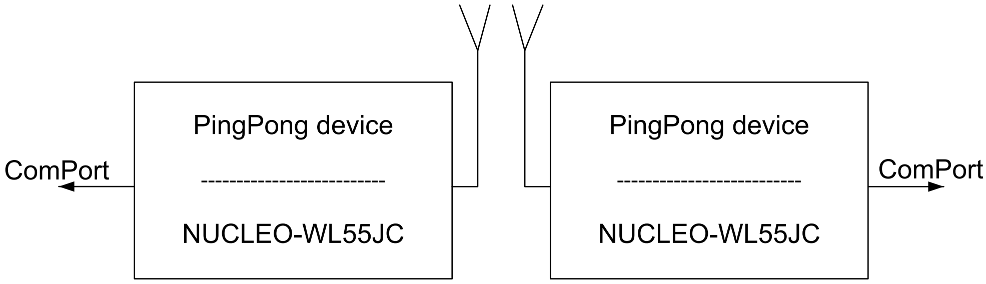
By default, each PingPong device starts as a master, transmits a 'Ping' message, and waits for an answer. At startup, each PingPong device has its two LEDs blinking. When the boards are synchronized (Tx window of one board aligned with Rx window of the other board), the Ping device (board receiving 'Ping' message) makes the green LED blinking, and the Pong device (board receiving 'Pong' message) makes the red LED blinking. The first PingPong device that receives a 'Ping' message becomes a slave and answers with a 'Pong' message to the master.

To launch the SubGhz_Phy_PingPong project, the user must go to `\Projects\<target>\Applications\SubGHz_Phy\SubGHz_Phy_PingPong` and follow the same procedure as for the LoRaWAN_End_Node project to launch the preferred toolchain.

10.1 SubGhz_Phy_PingPong hardware/software environment setup

To setup the STM32WL Nucleo board (NUCLEO-WL55JC), connect this board to the computer with a USB Type-A to Mini-B cable to the ST-LINK connector (CN1), as shown in the figure below.

Figure 8. SubGhz_Phy_PingPong application setup



10.2 Device configuration summary for SubGhz_Phy_PingPong application

Table 46. Switch options for SubGhz_Phy_PingPong application configuration

Project module = Application

Detail	Switch option	Definition	Location
Rx/Tx configuration	`RX_TIMEOUT_VALUE`	Rx window timeout	`subghz_phy_app.c`
	`TX_TIMEOUT_VALUE`	Tx window timeout	
	`MAX_APP_BUFFER_SIZE`	Max data buffer size	
	`RX_TIME_MARGIN`	Time between the end of Rx and start of Tx	
	`FSK_AFC_BANDWIDTH`	AFC bandwidth (in Hz)	
	`LED_PERIOD_MS`	LED blink period	
Modulation configuration	`USE_MODEM_LORA`	LoRa modem selected	`subghz_phy_app.h`
	`USE_MODEM_FSK`	FSK modem selected	
LoRa/FSK common parameters	`REGION_XXyyy`	Active LoRa region: AS923, AU915, CN470, CN779, EU433, EU868, KR920, IN865, US915, or RU864.	
	`RF_FREQUENCY`	Frequency used by the transceiver	

Detail	Switch option	Definition	Location
LoRa/FSK common parameters	TX_OUTPUT_POWER	RF output power: -17 to 22 dBm	subghz_phy_app.h
	PAYLOAD_LEN	Data buffer size	
LoRa specific parameters	LORA_BANDWIDTH	Bandwidth: • 0: 125 kHz • 1: 250 kHz • 2: 500 kHz • 3: Reserved	
	LORA_SPREADING_FACTOR	Spreading factor: SF7 to SF12	
	LORA_CODINGRATE	Coding rate: • 1: 4/5 • 2: 4/6 • 3: 4/7 • 4: 4/8	
	LORA_PREAMBLE_LENGTH	Length of Tx/Rx preamble	
	LORA_SYMBOL_TIMEOUT	Number of symbols checked before timeout	
	LORA_FIX_LENGTH_PAYLOAD_ON	Fix/dynamic length payload option	
	LORA_IQ_INVERSION_ON	IQ inversion option	
FSK specific parameters	FSK_FDEV	Frequency deviation (in Hz)	
	FSK_DATARATE	Data rate (in bit/s)	
	FSK_BANDWIDTH	Bandwidth (in Hz)	
	FSK_PREAMBLE_LENGTH	Length of Tx/Rx preamble	
	FSK_FIX_LENGTH_PAYLOAD_ON	Fix/dynamic length payload option	
Debug	DEBUGGER_ENABLED	Enables SWD pins.	sys_conf.h
Probe pins	PROBE_PINS_ENABLED	Enables four pins usable as probe signals by the middleware radio layer.	
Low power	LOW_POWER_DISABLE	Disables low-power mode.	
Trace enable	APP_LOG_ENABLED	Enables the trace mode.	
Trace level	VERBOSE_LEVEL	Enables the trace level.	

11 SubGhz_Phy_Per application

The SubGHz_Phy_Per application is a packet-error-rate test with IBM® whitening between one Tx device and one Rx device.

Tx device

Update `#define TEST_MODE to RADIO_TX` in `/SubGHz_Phy/App/subghz_phy_app.c`. Compile and load.

The packet content is `preamble | sync | payload length | payload | crc` where:

- `crc` is calculated using `payload length` and `payload`.
- Whitening is calculated over `payload length | payload | crc`.

The transmission starts forever in GFSK 50 Kbit/s with a payload of 64 bytes. The user button 1 increments `packet length` by 16 bytes. The user button 2 increments `packet length` by 1 byte. The user button 3 toggles packet `payload mode` from ramp (0x00, 0x01..) to prbs9.

The blue LED is on while radio in Tx.

Rx device

Update `#define TEST_MODE to RADIO_RX` in `/SubGHz_Phy/App/subghz_phy_app.c`. Compile and load.

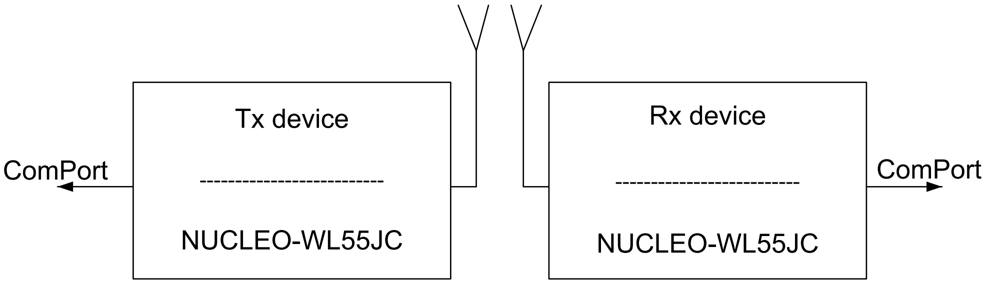
The green LED is on when Rx is OK. The red LED is on when Rx is KO.

To launch the SubGhz_Phy_Per project, the user must go to `\Projects\<target>\Applications\SubGHz_Phy\SubGHz_Phy_Per` and follow the same procedure as for the LoRaWAN_End_Node project to launch the preferred toolchain.

11.1 SubGhz_Phy_Per hardware/software environment setup

To setup the STM32WL Nucleo board (NUCLEO-WL55JC), connect this board to the computer with a USB Type-A to Mini-B cable to the ST-LINK connector (CN1), as shown in the figure below.

Figure 9. SubGhz_Phy_Per application setup



11.2 Device configuration summary for SubGhz_Phy_Per application

Table 47. Switch options for SubGhz_Phy_Per application configuration

Project module = Application

Detail	Switch option	Definition	Location
Rx/Tx configuration	RX_TIMEOUT_VALUE	Rx window timeout	subghz_phy_app.c
	TX_TIMEOUT_VALUE	Tx window timeout	
	RX_CONTINUOUS_ON	RX mode continuous or with timeout	
	TEST_MODE	Device mode: • RADIO_TX: send packet indefinitely • RADIO_RX: receive packet indefinitely	
	APP_LONG_PACKET	Long packet option[1]	
	MAX_APP_BUFFER_SIZE	Max data buffer size	
Modulation configuration	USE_MODEM_FSK	FSK modem selected	subghz_phy_app.h
LoRa/FSK common parameters	REGION_XXyyy	Active LoRa region: AS923, AU915, CN470, CN779, EU433, EU868, KR920, IN865, US915, or RU864.	
	RF_FREQUENCY	Frequency used by the transceiver	
	TX_OUTPUT_POWER	RF output power: -17 to 22 dBm	
	PAYLOAD_LEN	Data buffer size	
FSK specific parameters	FSK_FDEV	Frequency deviation (in Hz)	
	FSK_DATARATE	Data rate (in bit/s)	
	FSK_BANDWIDTH	Bandwidth (in Hz)	
	FSK_PREAMBLE_LENGTH	Length of Tx/Rx preamble	
	FSK_FIX_LENGTH_PAYLOAD_ON	Fix/dynamic length payload option	
Debug	DEBUGGER_ENABLED	Enables SWD pins.	sys_conf.h
Probe pins	PROBE_PINS_ENABLED	Enables four pins usable as probe signals by the middleware radio layer.	
Low power	LOW_POWER_DISABLE	Disables low-power mode.	
Trace enable	APP_LOG_ENABLED	Enables the trace mode.	
Trace level	VERBOSE_LEVEL	Enables the trace level.	

1. Refer to the document [5] for more details.

12 Dual-core management

The STM32WL5x devices embed two Cortex:

- Cortex-M4 (named CPU1)
- Cortext-M0+ (named CPU2)

In the dual-core applications, the application part mapped on CPU1 is separated from the stack and firmware low layers mapped on CPU2.

In a dual-core proposed model, two separated binaries are generated: CPU1 binary is placed at 0x0800 0000 and CPU2 binary is placed at 0x0802 0000.

A function address from one binary is not known from the other binary: this is why a communication model must be put in place. The aim of that model is that the user can change the application on CPU1 without impacting the core stack behavior on CPU2. However, ST still provides the implementation of the two CPUs in open source.

The interface between cores is done by the IPCC peripheral (interprocessor communication controller) and the inter-core memory, as described in Section 12.1 .

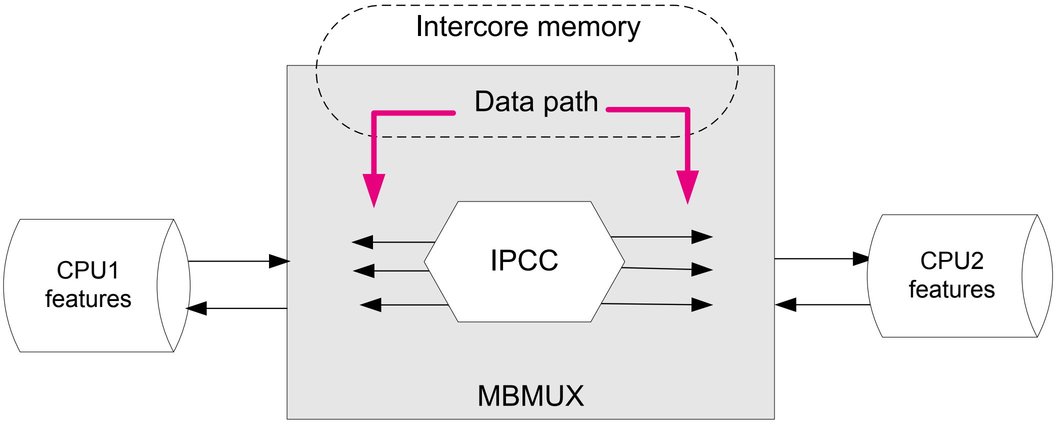
This dual-core implementation has been designed to behave the same way as the single-core program execution, thanks to a message blocking handling through a mailbox mechanism.

12.1 Mailbox mechanism

The mailbox is a service implementing a way to exchange data between the two processors. As shown in the figure below, the mailbox is built over two resources:

- **IPCC**: This hardware peripheral is used to trigger an interrupt to the remote CPU, and to receive an interrupt when it has completed the notification. The IPCC is highly configurable and each interrupt notification may be disabled/enabled. There is no memory management inside the IPCC.
- **Intercore memory**: This shared memory can be read/written by both CPUs. It is used to store all buffers that contain the data to be exchanged between the two CPUs.

Figure 10. **Mailbox overview**



The mailbox is specified to allow changes of the buffer definition to some extend, without breaking the backward compatibility.

12.1.1 Mailbox multiplexer (MBMUX)

As described in Figure 11, the data to be exchanged need to communicate via the 12 available IPCC channels (six for each direction). This is done via the MBMUX (mailbox multiplexer) that is a firmware component in charge to route the messages. These channels are identified from 1 to 6. An additional channel 0 is dedicated to the system feature.

The data type has been divided in groups called features. Each feature interfaces with the MBMUX via its own MBMUXIF (MBUX interface).

The mailbox is used to abstract a function executed by another core.

12.1.2 Mailbox features

In STM32WL5x devices, the MBMUX has the following features:

- **System**, supporting all communications related to the system

  This includes messages that are either related to one of the supported stacks, or related to none of them. The CPU1 channel 0 is used to notify the CPU2 that a command has been posted, and to receive the response of that command from the CPU2. The CPU2 channel 0 is used to notify CPU1 that an asynchronous event has been posted.

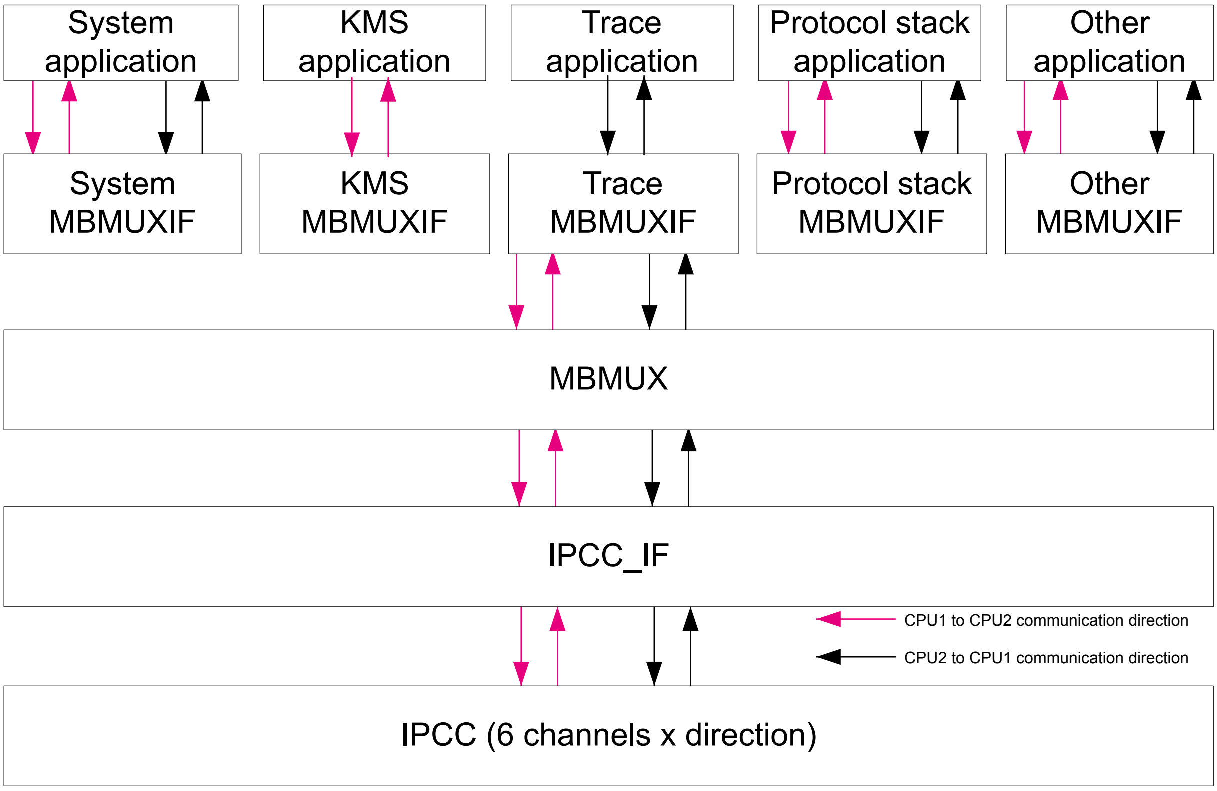
  The following services are mapped on system channel:
  - system initialization
  - IPCC channels versus feature registration
  - information exchanged on feature attributes and capabilities
  - possible additional system channels for high-priority operations (such as RTC notifications)
- **Trace**

  The CPU2 fills a circular queue for information or debug that is sent to the CPU1 via the IPCC. The CPU1 is in charge to handle this information, by outputting it on the same channel used for CPU1 logs (such as the USART).
- **KMS** (key management services)
- **Radio**

  The sub-GHz radio can be interfaced directly without passing by the CPU2 stack. A dedicated mailbox channel is used.
- **Protocol stack**

  This channel is used to interface all protocol stack commands (such as Init or request), and events (response/indication) related to the stack implemented protocol.

Figure 11. **MBMUX - Multiplexer between features and IPCC channels**



In order to use the MBMUX, a feature needs to be registered (except the system feature that is registered by default and always mapped on IPCC channel 0). The registration dynamically assigns to the feature, the requested number of IPCC channels: typically one for each direction (CPU1 to CPU2 and CPU2 to CPU1).

In the following cases, the feature needs just a channel in one direction:

- Trace feature is only meant to send debug information from CPU2 to CPU1.
- KMS is only used by CPU1 to request functions execution to CPU2.

Note:

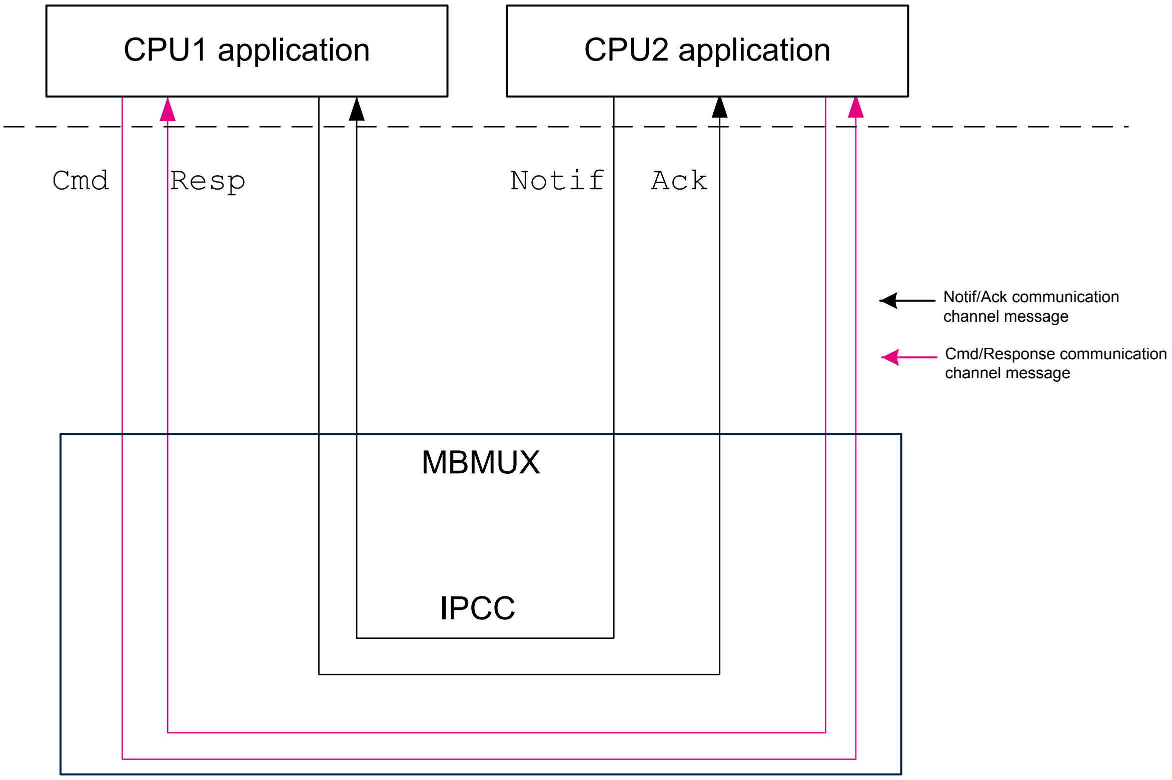
- *The RTC alarm A transfers the interrupt using one IPCC IRQ, not considered as a feature.*
- *The user must consider adding KMS wrapper to be able to use it as a feature.*

12.1.3 MBMUX messages

The mailbox uses the following types of messages:

- `Cmd` command sent by CPU1 to CPU2, composed of:
  - `Msg ID` identifies a function called by CPU1 but implemented on CPU2.
  - `Ptr buffer params` points to the buffer containing the parameters of the above function
  - `Number of params`
- `Resp`, response sent by CPU2 to CPU1, composed of:
  - `Msg ID` (same value as `Cmd Msg ID`)
  - `Return value` contains the return value of the above function.
- `Notif`, notification sent by CPU2 to CPU1, composed of:
  - `Msg ID` identifies a callback function called by CPU2 but implemented on CPU1.
  - `Ptr buffer params` points to the buffer containing the parameters of the above function.
  - `Number of params`
- `Ack`, acknowledge sent by CPU1 to CPU2, composed of:
  - `Msg ID` (same value as `Notif Msg ID`)
  - `Return value` contains the return value of the above callback function.

Figure 12. Mailbox messages through MBMUX and IPCC channels


CPU1 application
CPU2 application
Cmd
Resp
Notif
Ack
Notif/Ack communication channel message
Cmd/Response communication channel message
MBMUX
IPCC

12.2 Intercore memory

The intercore memory is a centralized memory accessible by both cores, and used by the cores to exchange data, function parameters, and return values.

12.2.1 CPU2 capabilities

Several CPU2 capabilities must be known by the CPU1 to detail its supported features (such as protocol stack implemented on the CPU2, version number of each stack, or regions supported).

These CPU2 capabilities are stored in the *features_info* table. Data from this table are requested at initialization by the CPU1 to expose CPU2 capabilities, as shown in RAM mapping.

The *features_info* table is composed of:

- `Feat_Info_Feature_Id`: feature name
- `Feat_Info_Feature_Version`: feature version number used in current implementation

MB_MEM2 is used to store these CPU2 capabilities.

12.2.2 Mailbox sequence to execute a CPU2 function from a CPU1 call

When the CPU1 needs to call a CPU2 `feature_func_X()`, a `feature_func_X()` with the same API must be implemented on the CPU1:

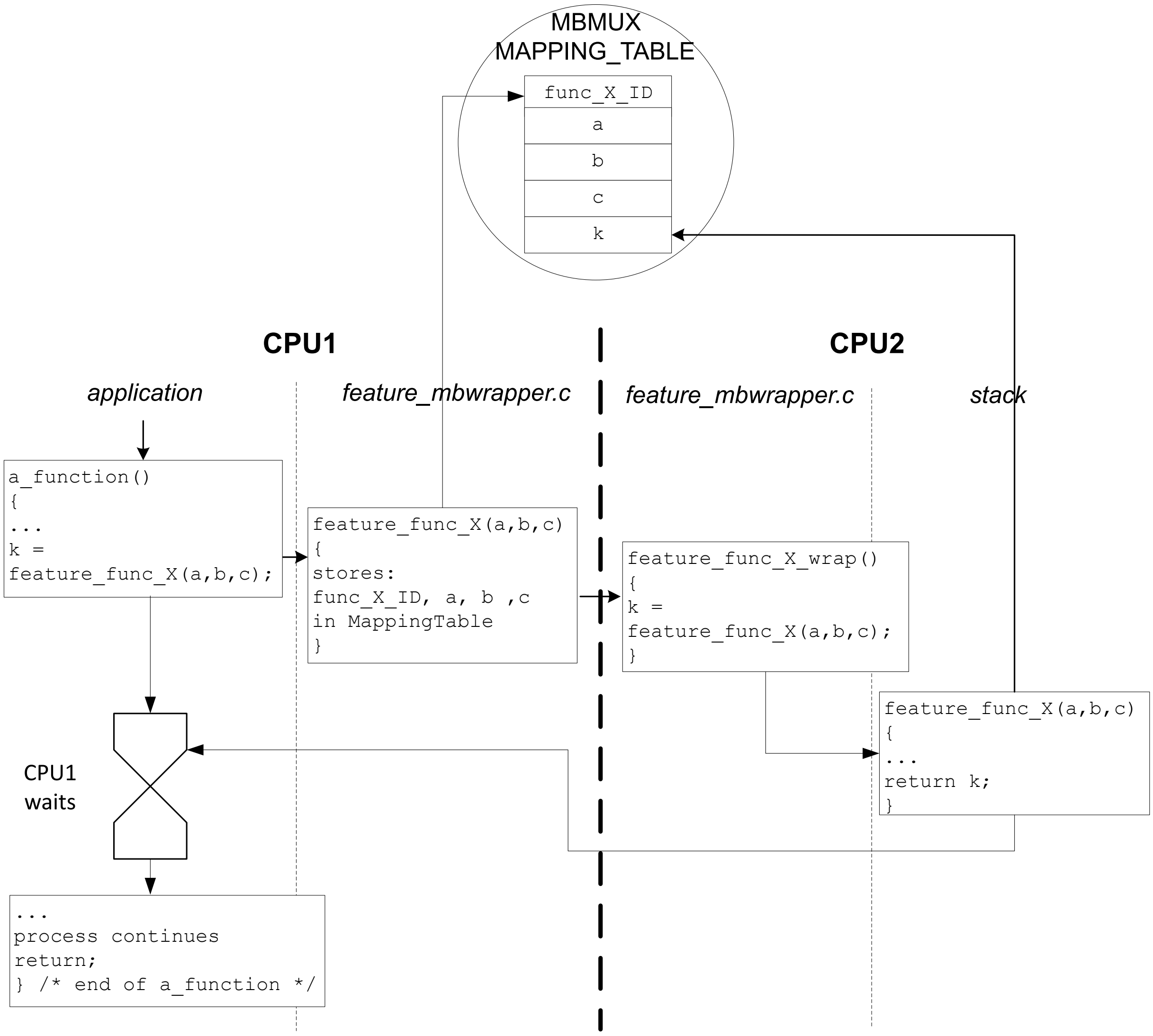
1. The CPU1 sends a **command** containing `feature_func_X()` parameters in the *Mapping* table:
   a. `func_X_ID` that was associated to `feature_func_X()` at initialization during registration, is added in the *Mapping* table. `func_X_ID` has to be known by both cores: this is fixed at compilation time.
   b. The CPU1 waits the CPU2 to execute the `feature_func_X()` and goes in low-power mode.
   c. The CPU2 wakes up if it was in low-power mode and executes the `feature_func_X()`.
2. The CPU2 sends a **response** and fills the *Mapping* table with the return value:
   a. The IPCC interrupt wakes up the CPU1.
   b. The CPU1 retrieves the return value from the *Mapping* table.

Conversely, when the CPU2 needs to call a CPU1 `feature_func_X_2()`, a `feature_func_X_2()` with the same API must be implemented on the CPU2:

1. The CPU2 sends a **notification** containing `feature_func_X_2()` in the *Mapping* table.
2. The CPU1 sends an **acknowledge** and fills the *Mapping* table with the return value.

The full sequence is shown in the figure below.

Figure 13. **CPU1 to CPU2 feature_func_X() process**

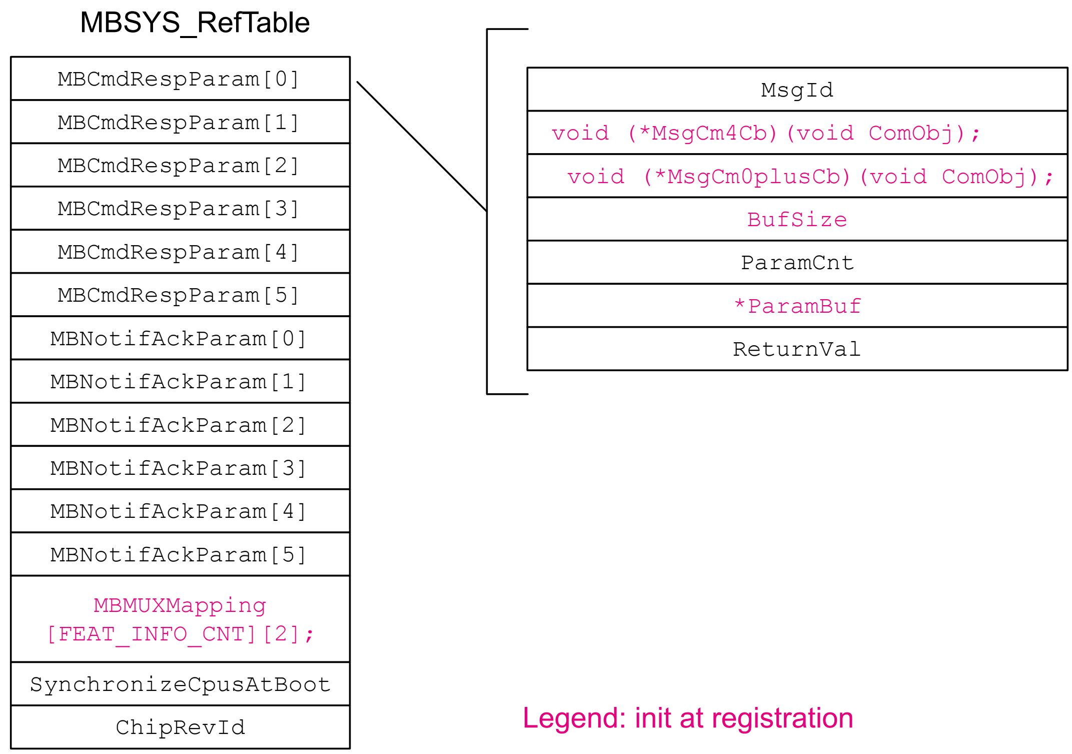


12.2.3 Mapping table

The *Mapping* table is a common structure in the MBMUX area of Figure 13. In RAM mapping, the memory mapping is referenced as MAPPING_TABLE.

The MBMUX communication table (MBSYS_RefTable) is described in the figure below.

Figure 14. MBMUX communication table



This MBSYS_RefTable includes:

- two communication parameter structures for both Command/Response and Notification/Acknowledge parameters for each of the sic IPCC channels.

  Each communication parameter, as shown in MBMUX *Mapping* table area of Figure 13, is composed of:
  - `MsgId`: message ID of `feature_func_X()`
  - `*MsgCm4Cb`: pointer to CPU1 callback `feature_func_X()`
  - `*MsgCm0plusCb`: pointer to CPU2 callback `feature_func_X()`
  - `BufSize`: buffer size
  - `ParamCnt`: message parameter number
  - `ParamBuf`: message pointer to parameters
  - `ReturnVal`: return value of `feature_func_X()`
- `MBMUXMapping`: chart used to map channels to features

  This chart is filled at the MBMUX initialization during the registration. For instance, if the radio feature is associated to `Cmd/Response channel number = 1`, then `MBMUXMapping` must associate `[FEAT_INFO_RADIO_ID][1]`.
- `SynchronizeCpusAtBoot`: flags used to synchronize CPU1 and CPU2 processing as shown in Figure 15 sequence chart.
- `ChipRevId`: stores the hardware revision ID.

MB_MEM1 is used to send `command/response set ()` parameter and to get the return values for the CPU1.

12.2.4 Option-byte warning

A trap is placed in the code to avoid erroneous option-byte loading (see section *Option-byte loading failure at high MSI system clock frequency* in the product errata sheet). The trap can be removed if the system clock is set below or equal to 16 MHz.

12.2.5 RAM mapping

The tables below detail the mapping of both CPU1 and CPU2 RAM areas and the intercore memory.

Table 48. **STM32WL5x RAM mapping**

Page index (1)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Allocation region/section	CPU1 RAM																CPU1 RAM2_Shared		CPU2 RAM2_Shared		CPU2 RAM2											

1. 2 Kbytes for each page.

Table 49. **STM32WL5x RAM allocation and shared buffer**

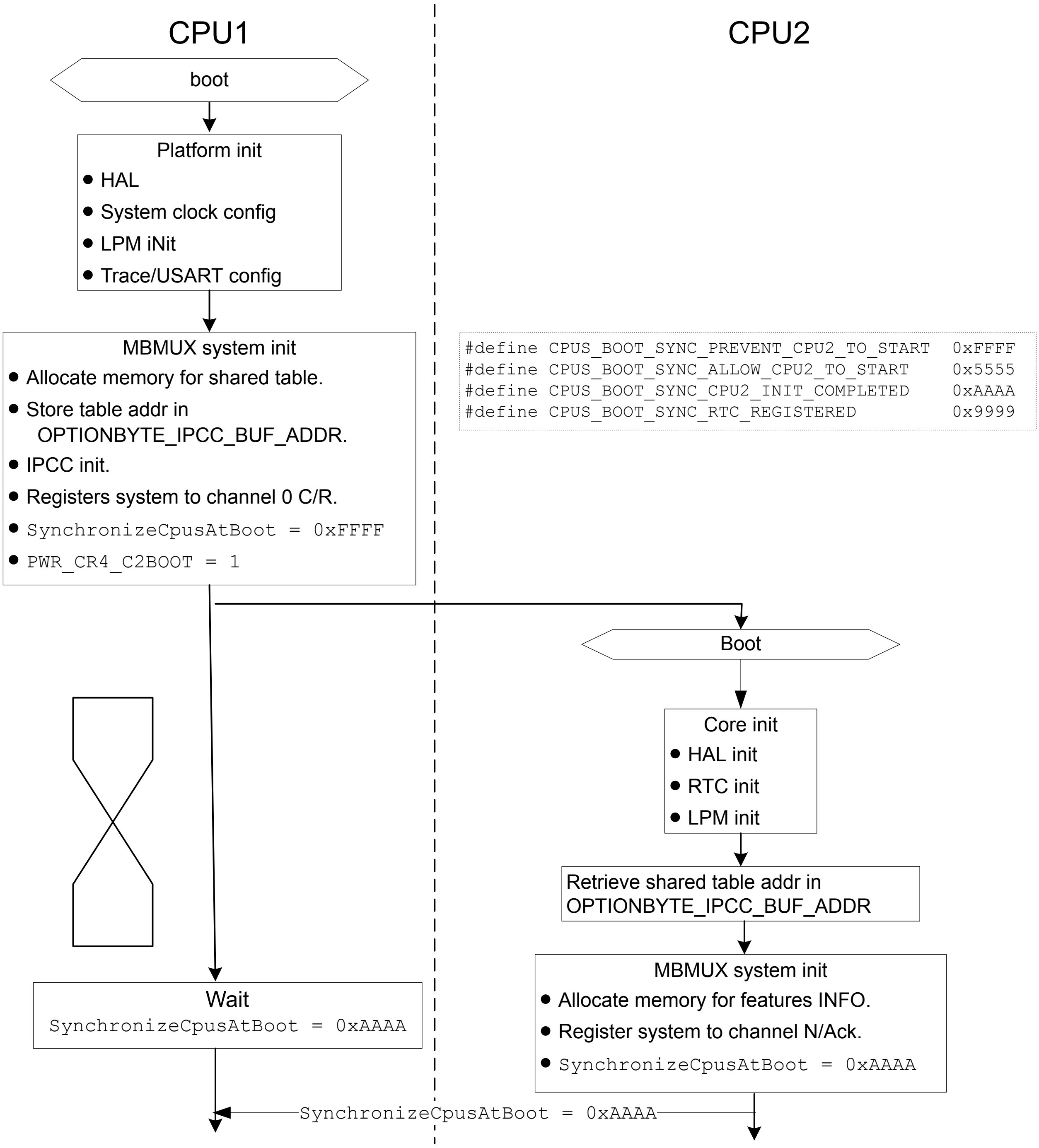
CPU region	Section	Module	Allocated symbol	Size (bytes)	Total (bytes)
CPU1 RAM	readwrite	-			
	CSTACK				
	HEAP				
CPU1 RAM2 _Shared	MAPPING _TABLE	MBMUX_SYSTEM	MBMUX_ComTable_t MBSYS_RefTable	316	316
	MB_MEM1	MBMUX_LORAWAN	uint32_t aLoraCmdRespBuff[]	60	524
			uint32_t aLoraNotifAckBuff[]	20	
		MBMUX_RADIO	uint32_t aRadioCmdRespBuff[]	60	
			uint32_t aRadioNotifAckBuff[]	16	
		MBMUX_TRACE	uint32_t aTraceNotifAckBuff[]	44	
		MBMUX_SYSTEM	uint32_t aSystemCmdRespBuff[]	28	
			uint32_t aSystemNotifAckBuff[]	20	
			uint32_t aSystemPrioACmdRespBuff[]	4	
			uint32_t aSystemPrioANotifAckBuff[]	4	
			uint32_t aSystemPrioBCmdRespBuff[]	4	
			uint32_t aSystemPrioBNotifAckBuff[]	4	

CPU region	Section	Module	Allocated symbol	Size (bytes)	Total (bytes)
CPU1 RAM2 _Shared	MB_MEM1	LMH_MBWRAPPER	uint8_t aLoraMbWrapShareBuffer[]	260	524
CPU2 RAM2 _Shared	MB_MEM2	MBMUX_TRACE	uint8_t ADV_TRACE_Buffer[]	1024	1664
		MBMUX_LORAWAN	LoraInfo_t loraInfo	16	
			FEAT_INFO_Param_t Feat_Info_Table	80	
			FEAT_INFO_List_t Feat_Info_List	8	
		LMH_MBWRAPPER	uint8_t aLoraMbWrapShare2Buffer[]	280	
		RADIO_MBWRAPPER	uint8_t aRadioMbWrapRxBuffer[]	256	
CPU2 RAM2	readwrite	-			
	CSTACK				
	HEAP				

12.3 Startup sequence

The startup sequence for CPU1 and CPU2 is detailed in the figure below.

Figure 15. Startup sequence



```
#define CPUS_BOOT_SYNC_PREVENT_CPU2_TO_START   0xFFFF
#define CPUS_BOOT_SYNC_ALLOW_CPU2_TO_START     0x5555
#define CPUS_BOOT_SYNC_CPU2_INIT_COMPLETED     0xAAAA
#define CPUS_BOOT_SYNC_RTC_REGISTERED          0x9999
```

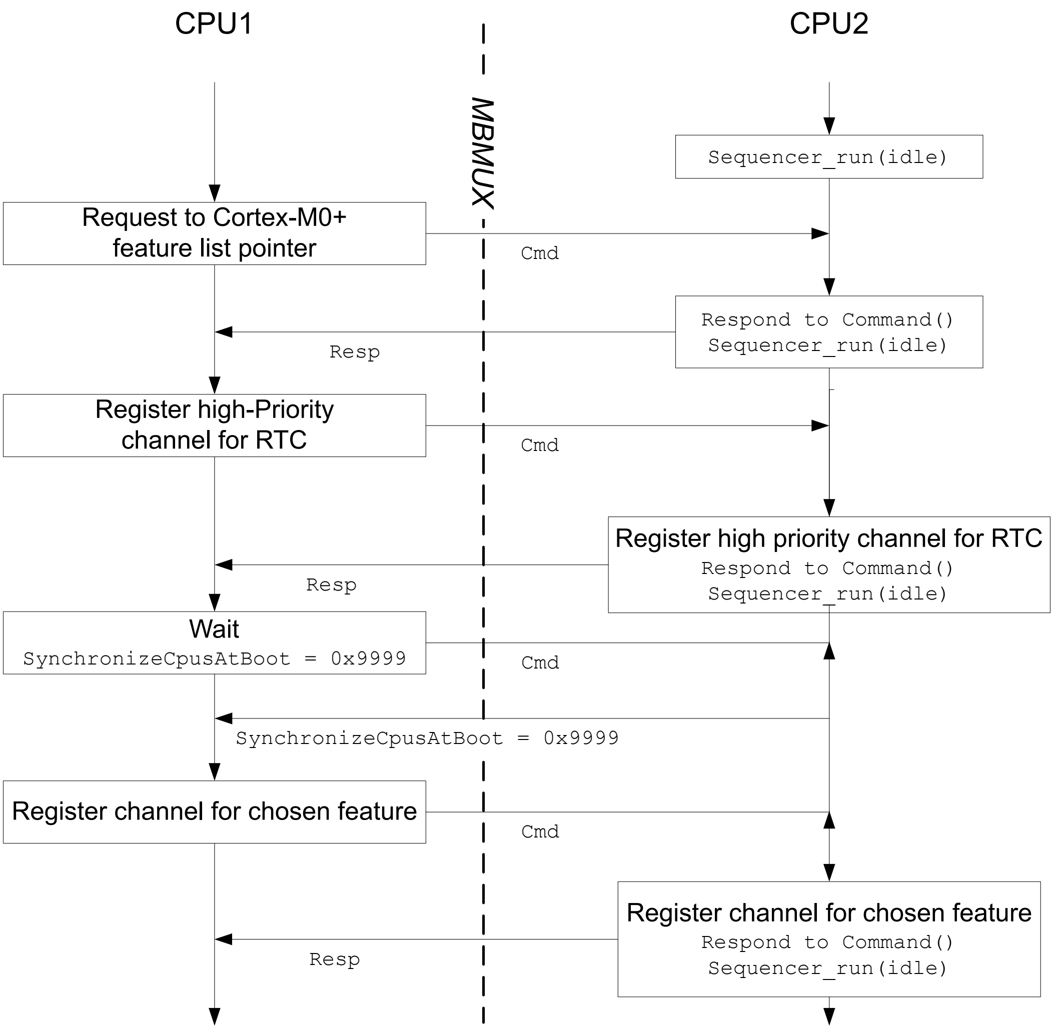
CPU1
CPU2
boot
Platform init
● HAL
● System clock config
● LPM iNit
● Trace/USART config
MBMUX system init
● Allocate memory for shared table.
● Store table addr in OPTIONBYTE_IPCC_BUF_ADDR.
● IPCC init.
● Registers system to channel 0 C/R.
● SynchronizeCpusAtBoot = 0xFFFF
● PWR_CR4_C2BOOT = 1
Boot
Core init
● HAL init
● RTC init
● LPM init
Retrieve shared table addr in OPTIONBYTE_IPCC_BUF_ADDR
MBMUX system init
● Allocate memory for features INFO.
● Register system to channel N/Ack.
● SynchronizeCpusAtBoot = 0xAAAA
Wait
SynchronizeCpusAtBoot = 0xAAAA
SynchronizeCpusAtBoot = 0xAAAA

The various steps are the following:

1. The CPU1, that is the master processor in this init sequence:
    a. executes the platform initialization.
    b. initializes the MBMUX system.
    c. sets the `PWR_CR4_C2BOOT` flag to 1, which starts the CPU2.
    d. waits that CPU2 sets the `SynchronizeCpusAtBoot` flag to 0xAAAA.
2. The CPU2 boots and:
    a. executes the core initialization.
    b. retrieves the shared table address.
    c. initializes the MBMUX system.
    d. sets the `SynchronizeCpusAtBoot` to 0xAAAA to inform the CPU1 that he has ended its init sequence and that he is ready.
3. The CPU1 acknowledges this CPU2 notification.

Then both cores are initialized, and the initialization goes on via MBMUX, as shown in the figure below.

Figure 16. MBMUX initialization

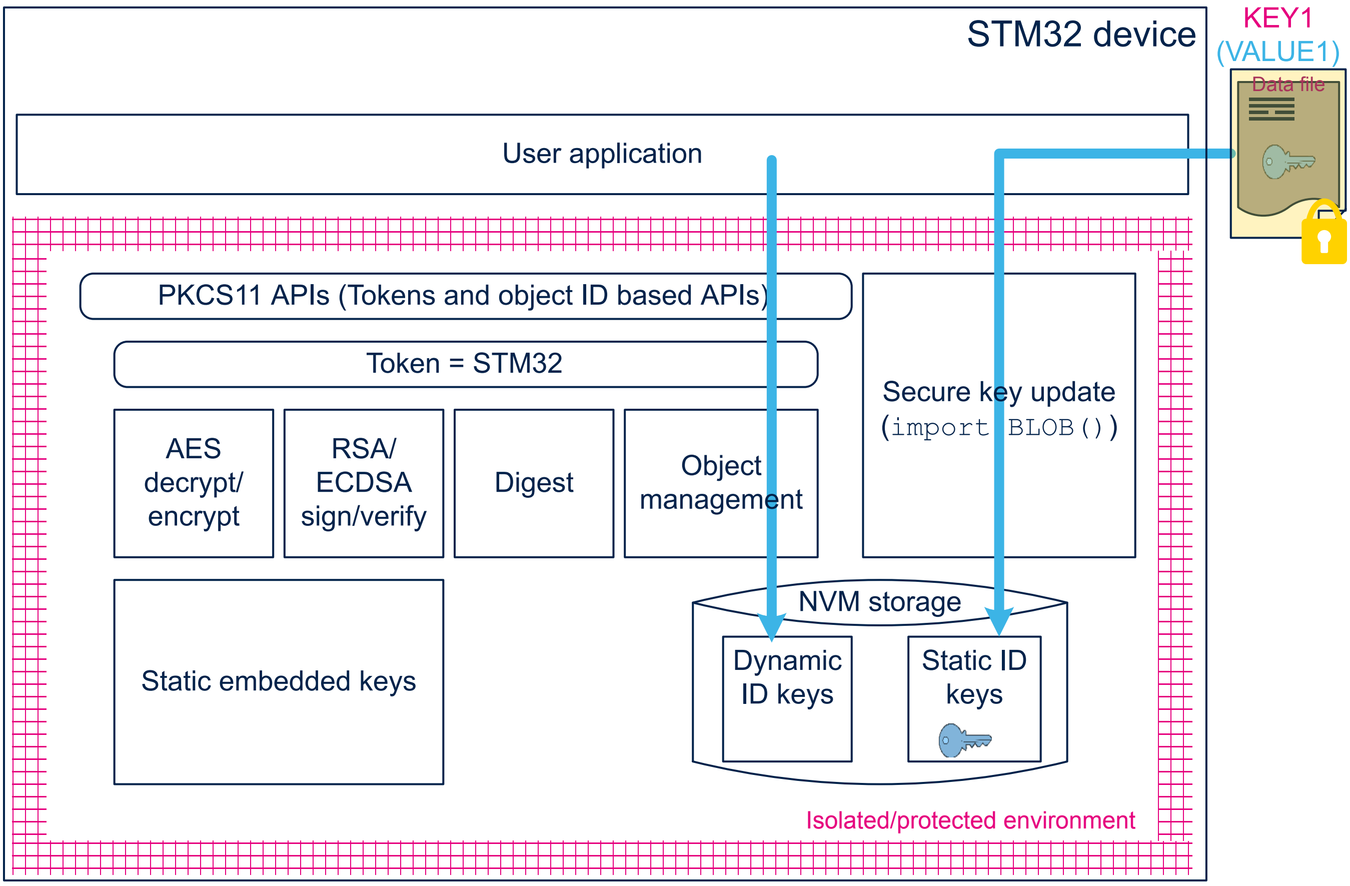

CPU1
CPU2
MBMUX
Sequencer_run(idle)
Request to Cortex-M0+ feature list pointer
Cmd
Respond to Command()
Sequencer_run(idle)
Resp
Register high-Priority channel for RTC
Cmd
Register high priority channel for RTC
Respond to Command()
Sequencer_run(idle)
Resp
Wait
SynchronizeCpusAtBoot = 0x9999
Cmd
SynchronizeCpusAtBoot = 0x9999
Register channel for chosen feature
Cmd
Register channel for chosen feature
Respond to Command()
Sequencer_run(idle)
Resp

13 Key management services (KMS)

Key management services (KMS) provide cryptographic services through the standard PKCS#11 APIs (developed by OASIS), are used to abstract the key value to the caller (using object ID and not directly the key value).

KMS can be executed inside a protected/isolated environment in order to ensure that key value cannot be accessed by an unauthorized code running outside the protected/isolated environment, as shown in the figure below.

Figure 17. KMS overall architecture



For more details, refer to KMS section in the user manual *Getting Started with the SBSFU of STM32CubeWL* (UM2767) .

To activate the KMS module, `KMS_ENABLE` must be set to 1 in C/C++ compiler project options.

KMS supports only PKCS #11 APIs listed below:

- Object management functions (creation/update/deletion)
- AES encryption/decryption functions (CBC, CCM, ECB, GCM, CMAC algorithms)
- Digesting functions
- RSA and ECDSA signing/verifying functions
- Key management functions (key generation/derivation)

13.1 KMS key types

KMS manages three key types but only the two following ones are used:

- Static embedded keys
  - predefined keys embedded within the code that cannot be modified
  - immutable keys
- NVM_DYNAMIC keys:
  - runtime keys
  - keys IDs that may be defined when keys are created using KMS (DeriveKey() or CreateObject())
  - keys that can be deleted or defined as mutable

13.2 KMS key definition

Static and dynamic keys used by stack occupies different sizes.

Static key

Each static key is composed of the two following elements:

- a blob header: five 4-byte fields (total = 20 bytes): version, configuration, blob_size, blob_count, and object_id.
- a blob buffer: some required and option blob elements, from the list of elements defined as follows:

Table 50. Global KMS blob elements

Attribute	Value	Required	Size (bytes)	Description
CKA_CLASS	CKO_SECRET_KEY	Yes	12	Type of blob element
CKA_KEY_TYPE	CKK_AES	Yes	12	Type of key
CKA_VALUE	"KEY_VALUE"	Yes	24	Key value (uint32_t format)
CKA_DERIVE	TRUE/FALSE	No	12	Optional parameters to enable/ disable capabilities by default. These fields are TRUE if not defined.
CKA_ENCRYPT	TRUE/FALSE	No	12	
CKA_DECRYPT	TRUE/FALSE	No	12	
CKA_COPYABLE	TRUE/FALSE	No	12	
CKA_EXTRACTABLE	TRUE/FALSE	No	12	
CKA_LABEL	"UNIQUE LABEL"	Yes	12 for static key 16 for dynamic key	Unique label

Example:

A static key composed of a blob header and eight blob elements (CKA_CLASS, CKA_KEY_TYPE, CKA_VALUE, CKA_LABEL, CKA_DERIVE, CKA_DECRYPT, CKA_COPYABLE, and CKA_EXTRACTABLE), uses a total size of 128 bytes (blob header = 20 bytes, and blob buffer = (12 × 7 + 24) = 108 bytes).

Dynamic key

Each dynamic key is composed of three elements, a data header, a blob header, and a blob buffer, with:

- a data header: eight 4-byte fields (total = 32 bytes): magic1, magic2, slot, instance, next, data_type, size, and checksum.

Example:

A dynamic key composed of a data header, a blob header and five blob elements (CKA_CLASS, CKA_KEY_TYPE, CKA_VALUE, CKA_LABEL, and CKA_EXTRACTABLE) uses a total size of 128 bytes (data header = 32 bytes, blob header = 20 bytes, and blob buffer = (12 + 12 + 24 + 12 + 16) = 76 bytes.

Note:
- *The NVM dynamic memory always starts with an initial data header element.*
- *At each dynamic key 'deletion' (such as an obsolete key replaced by a new key value), an additional data header is written in the memory to declare that the previous instance cannot longer be used.*

13.3 LoRaWAN keys

In STM32CubeWL applications, the KMS are used on CPU2 only on dual-core application. The root keys are chosen to be static embedded keys. All derived keys are NVM_DYNAMIC keys.

For LoRaWAN stack, the immutable root keys are detailed in the table below.

Table 51. LoRaWAN static keys with blob attributes

Attribute	Key				
	LoRaWAN_Zero_Key	LoRaWAN_APP_Key	LoRaWAN_NWK_Key	LoRaWAN_NWK_S_Key (ABP only)	LoRaWAN_APP_S_Key (ABP only)
CKA_CLASS	CKO_SECRET_KEY				
KA_KEY_TYPE	CKK_AES				
CKA_VALUE	"KEY_VALUE"				
CKA_DERIVE	FALSE	TRUE	TRUE	TRUE	TRUE
CKA_ENCRYPT	TRUE	FALSE	FALSE	TRUE	TRUE
CKA_DECRYPT	FALSE	FALSE	FALSE	TRUE	TRUE
CKA_COPYABLE	FALSE				
CKA_EXTRACTABLE	FALSE	TRUE/FALSE defined by #define KEY_EXTRACTABLE			
CKA_ LABEL	"UNIQUE LABEL"				

All other keys are mutable NVM_DYNAMIC generated keys, detailed in the table below.

Table 52. **LoRaWAN dynamic keys with blob attributes**

Attribute	Key							
	LoRaWAN_NWK_S_Key (OTAA only)	LoRaWAN_APP_S_Key (OTAA only)	MC_ROOT_Key	MC_KE_Key	MC_KEY_0	MC_APP_S_Key_0	MC_NWK_S_Key_0	SLOT_RAND_ZERO_Key
CKA_CLASS	CKO_SECRET_KEY							
KA_KEY_TYPE	CKK_AES							
CKA_VALUE	"KEY_VALUE"							
CKA_DERIVE	TRUE							
CKA_ENCRYPT	TRUE							
CKA_DECRYPT	TRUE							
CKA_COPYABLE	TRUE							
CKA_EXTRACTABLE	TRUE/FALSE defined by #define KEY_EXTRACTABLE							
CKA_ LABEL	"UNIQUE LABEL"							

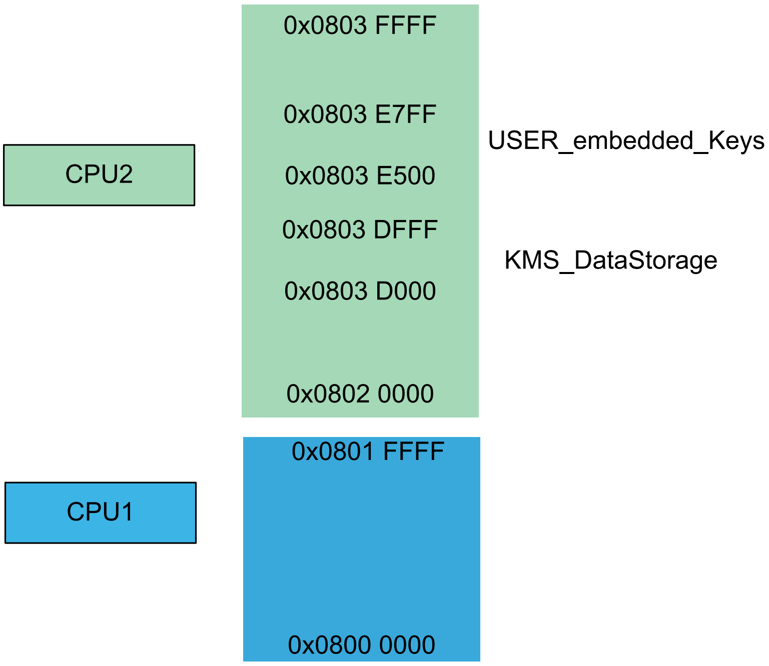
13.4 KMS key memory mapping for user applications

Static embedded keys correspond to USER_embedded_Keys (used for root keys). They are placed in a dedicated data storage in Flash memory/ROM. The linker files for user applications locate them from 0x0803 E500 to 0x0803 E7FF, as shown in the figure below.

NVM_DYNAMIC keys are placed in KMS key data storage area, KMS_DataStorage.

The total data storage area must be 4 Kbytes, as explained in Section 13.5 . They have been placed from: 0x0803 D000 to 0x0803 DFFF, as shown in the figure below. This size may be increased if more keys are necessary.

Figure 18. ROM memory mapping



13.5 How to size NVM for KMS data storage

The NVM is organized by 2-Kbyte pages. Due to the double buffering (flip/flop EEPROM emulation mechanism), each page needs a “twin”. So the minimum to be allocated for NVM is 4 Kbytes. The allocation size is defined in the linker file.

The linker files proposed by the applications use the minimum allowed size (2 × 2 Kbytes). The associated limitations/drawbacks are explained below. The user must size NVM depending on the application specific need.

The applications use the NVM only to store the dynamic keys. A LoRaWAN dynamic key with a data header, a blob header, and a blob buffer of five elements, occupies 108 bytes. An empty NVM is initialized with a global 32-byte data header and, for each obsolete key, an additional 32-byte data header is written to declare the previous instance as unusable.

Given the above values, it is possible to evaluate how many keys can be stored in 2 Kbytes:

(2048 - 32) / (108 + 32) = 14.4 ==> 14 dynamic keys can be stored into a 2-Kbyte memory page before the flop operation.

In user application configuration, only NVM_DYNAMIC is used. NVM_STATIC can be filled via blob, but is not covered by user applications.

NVM_DYNAMIC can host derived keys (via C_DeriveKey()) and root keys (via C_CreateObject()).

The LoRaWAN application generates:

- two derived keys each join in ABP mode
- four derived keys each join in OTAA mode

Up to ten derived keys simultaneously active can be generated in more complex scenarios (such as setting multicast). If a user wants to write one application that uses more than 14 keys, additional NVM pages must be allocated to the linker file.

Smaller is the NVM size, more the NVM is written and erased, shorter becomes its life expectation.

Destroy a key does not mean that this key is erased, but that this key is tagged as destroyed and is not copied at the next flip-flop switch. A destroy flag also occupies some NVM bytes: after destroying eight keys, the remaining place is less than four keys.

For a scenario where four keys are generated each join, and after having destroyed the previous join key, the life expectation is estimated as follows:

- At the 3rd join session, four new keys are derived but no place in page 1 for the last key. All four keys (being still active) are placed in page 2. Page 1 is erased at once as the NVM page can only be fully erased .
- At the 5th join also, page 2 is erased and keys are stored back on page 1. After 40.000 joins, the two NVM pages have been erased 10.000 times, that is the estimated lifetime of the Flash sector.
- If the user application is supposed to join excessively frequently (for example every 2 hours), the expected NVM live is 80.000 hours (around nine years). If the join process is done once a day, the lifetime is much greater than ten years.

Bigger are the amount of requested derived keys simultaneously active (not destroyed), less efficient is the flip-flop mechanism.

To conclude, for applications that need to preserve the NVM life-time duration, it is suggested to keep the NVM size rather bigger than the number of keys active simultaneously (not destroyed).

Note: *Obsolete keys must be destroyed otherwise, if page 1 is fully filled by active keys, the flip-flop switch cannot be done and an error is generated.*

13.6 KMS configuration files to build the application

The KMS are used in the LoRaWAN application by setting the code

```
#define LORAWAN_KMS  1
```

in `CM0PLUS/LoRaWAN/Target/lorawan_conf.h`

The following files must be filled with the information on SubGHz_Phy stack keys:

- embedded keys structures defined in `CM0PLUS/Core/Inc/kms_platf_objects_config.h`
- embedded object handles associated to SubGHz_Phy stack keys, use of KMS modules defined in `CM0PLUS/Core/Inc/kms_platf_objects_interface.h`

13.7 Embedded keys

The embedded keys of the SubGHz_Phy protocol stack must be stored in a ROM region in which a secure additional software (such as SBSFU, Secure Boot and Firmware Update) ensures data confidentiality and integrity. For more details on the SBSFU, refer to the application note *Integration guide of SBSFU on STM32CubeWL* (AN5544).

These embedded keys are positioned in the ROM as indicated in Figure 18. ROM memory mapping.

14 How to secure a LoRaWAN application

The document [7] describes how to secure a dual-core LoRaWAN application using the SBSFU framework.

15 System performances

15.1 Memory footprint

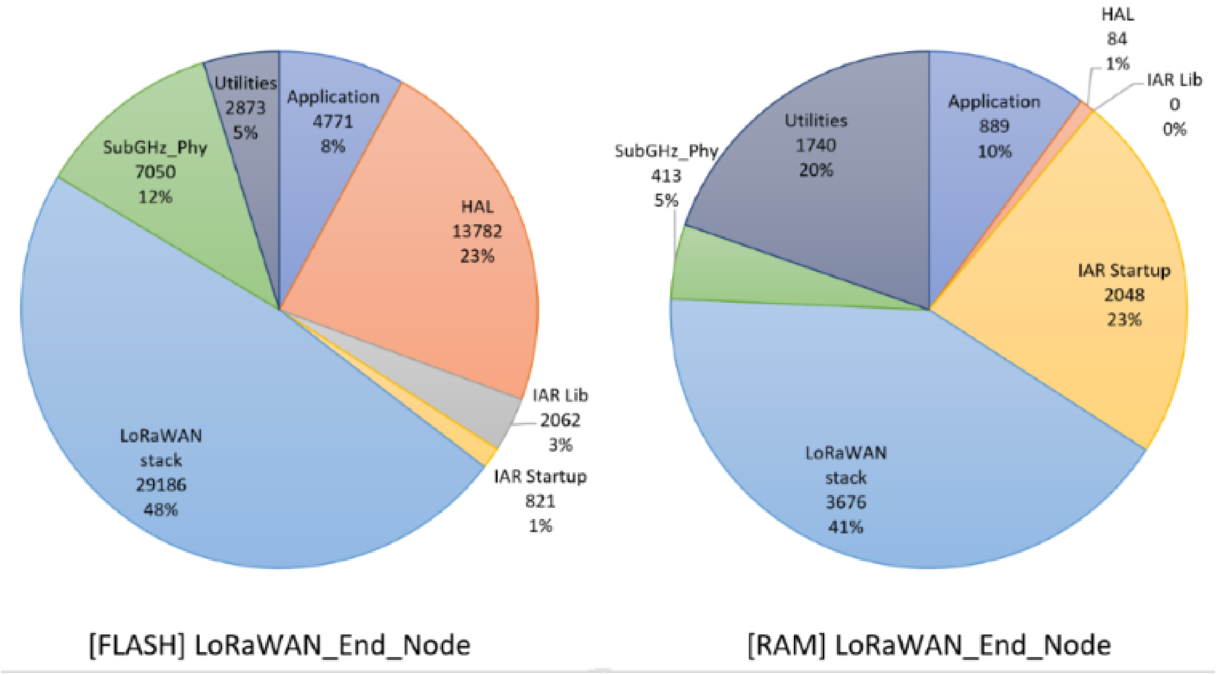
Values in the table below are measured in the following configuration of the IAR Embedded Workbench® compiler (EWARM version 8.30.1):

- optimization level 3 for size
- debug option off
- trace option: VLEVEL_M (debug traces enabled)
- target : NUCLEO-WL55JC1
- LoRaWAN_End_Node application
- LoRaMAC Class A
- LoRaMAC region EU868 and US915

Table 53. **Memory footprint values for LoRaWAN_End_Node application**

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	4771	889	Core, Application and Target components
LoRaWAN stack	29186	3676	Middleware Lmhandler interface, crypto, MAC and Region
HAL	13782	84	STM32WL HAL and LL drivers
Utilities	2873	1740	All STM32 services (sequencer, time server, low-power manager, trace, mem)
SubGHz_Phy	7050	413	Middleware radio interface
IAR lib	2062	0	Proprietary IAR libraries
IAR startup	821	2048	Int_vect, init routines, init table, CSTACK and HEAP
Total application	**60545**	**8850**	**Memory footprint for LoRaWAN_End_Node application**

Figure 19. **Flash memory and RAM footprint**

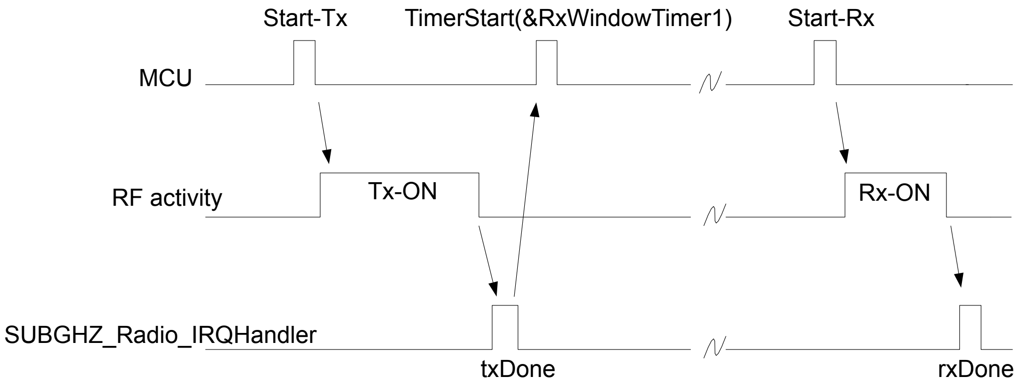


15.2 Real-time constraints

The LoRa RF asynchronous protocol implies to follow a strict Tx/Rx timing recommendation (see the figure below).

The STM32WL Nucleo board (NUCLEO-WL55JC) is optimized for user-transparent low-lock time and fast auto-calibrating operation. The BSP integrates the transmitter startup time and the receiver startup time constraints (refer to Section 4 BSP STM32WL Nucleo boards).

Figure 20. **Rx/Tx time diagram**



Rx window channel starts. The Rx1 window opens 1 second (±20 µs) after the txDone falling edge. The Rx2 window opens 1 second (±20 µs) after the txDone falling edge.

The JOIN_ACCEPT uses a 5 seconds (±20 µs) and 6 seconds delay after the end of the uplink modulation.

The current scheduling interrupt-level priority must be respected. In other words, all the new user-interrupts must have an interrupt priority higher than the Radio IRQ_interrupt in order to avoid stalling the received startup time.

15.3 Power consumption

The power-consumption measurement is done for the STM32WL Nucleo board NUCLEO-WL55JC1.

Measurements setup:

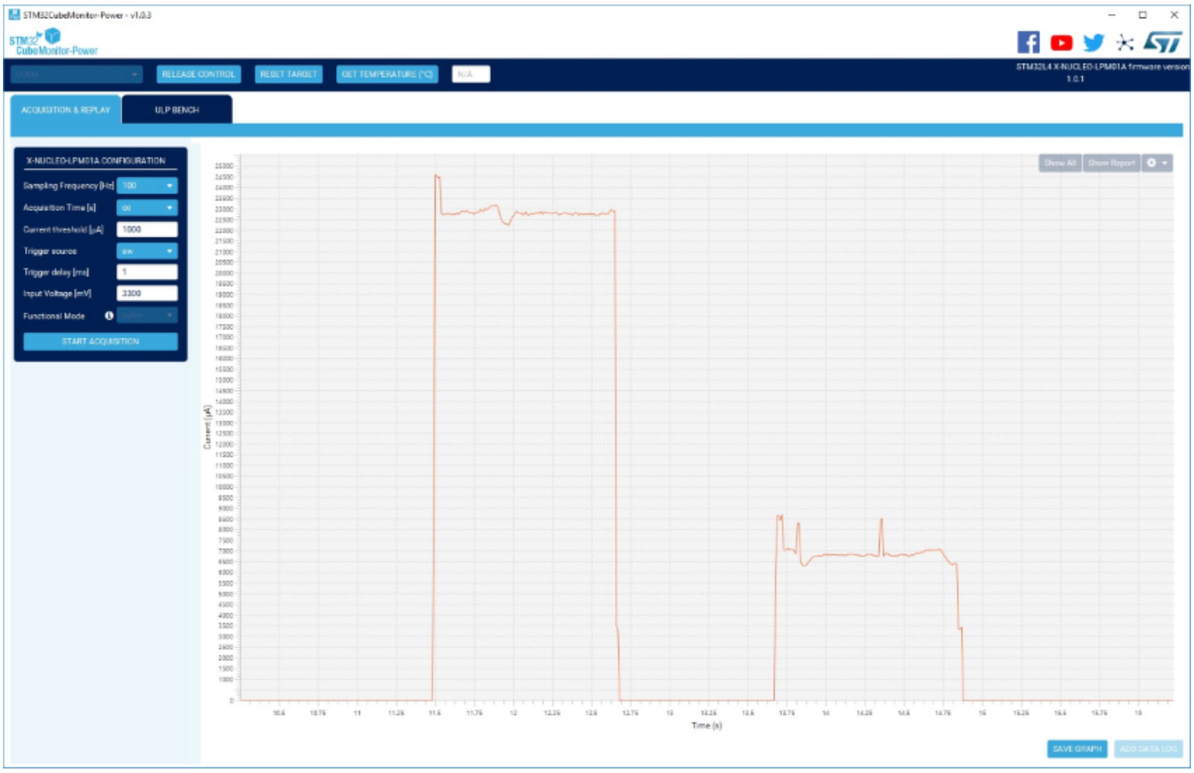
- no debug
- trace level VLEVEL_OFF (no trace)
- no SENSOR_ENABLED

Measurements results:

- Typical consumption in Stop 2 mode = 2 µA (see Figure 22).
- Typical consumption with TCXO in Tx = 23 mA (see Figure 21).
- Typical consumption with TCXO in Rx = 7 mA (see Figure 21).

Measurements figures: instantaneous consumption over 30 seconds.

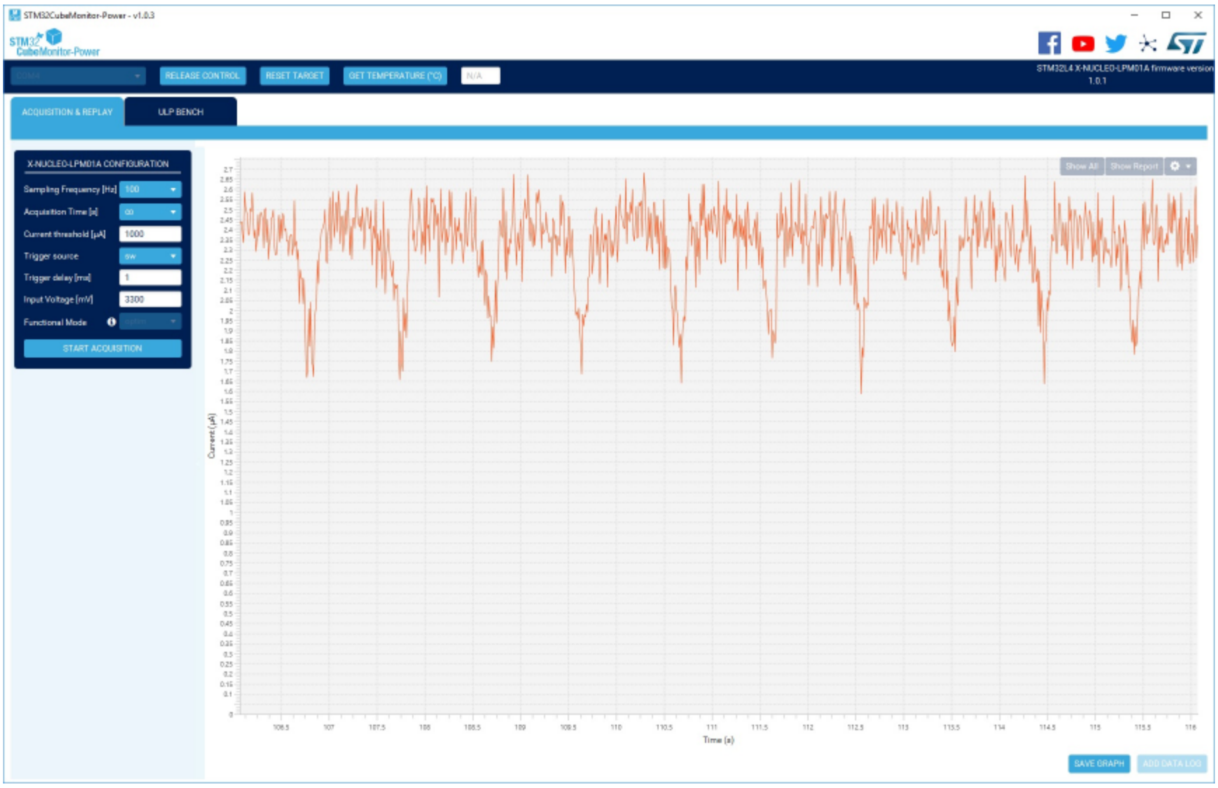
Figure 21. **NUCLEO-WL55JC1 current consumption versus time**



X-NUCLEO-LPM01A CONFIGURATION

Sampling Frequency [Hz]: 100
Acquisition Time [s]: ∞
Current threshold [µA]: 1000
Trigger source: sw
Trigger delay [ms]: 1
Input Voltage [mV]: 3300

Figure 22. **NUCLEO-WL55JC1 current consumption in Stop 2 mode**



X-NUCLEO-LPM01A CONFIGURATION

Sampling Frequency [Hz]: 100
Acquisition Time [s]: ∞
Current threshold [µA]: 1000
Trigger source: sw
Trigger delay [ms]: 1
Input Voltage [mV]: 3300

Revision history

Table 54. **Document revision history**

Date	Version	Changes
10-Dec-2019	1	Initial release.
27-Apr-2020	2	Global update of the document structure and content.
17-Nov-2020	3	Updated: • *Figure 1. Project file structure* • Note in *Section 4 BSP STM32WL Nucleo-73 boards* • Intro of *Section 6 LoRaWAN middleware description* • Title and intro of *Section 6.6 Middleware LmHandler application function* • *Table 22* and *Table 23* • *Section 8.1.1 Activation methods and keys* • *Table 38. Switch options for End_Node application configuration* • *Table 39. Switch options for AT_Slave application configuration* • *Table 40. Switch options for PingPong application configuration* • *Section 13.1 Memory footprint* Added: • *Table 26. LmHandler process* • *Section 11 Dual-core management* • *Section 12 Key management services (KMS)* Removed tables "Board unique ID" and "Board random seed" from *Section 6.7*.
18-Feb-2021	4	Updated: • `lora_app.c` renamed `lora_app.h` in Section 8.1.2, Section 8.1.4, Section 8.1.5, Section 8.1.6, Section 8.1.7, Section 8.1.8 and Section 8.1.9 • Table 38. Switch options for End_Node application configuration • Table 39. Switch options for AT_Slave application configuration
9-Jul-2021	5	Updated: • Overview section renamed Section 1 General information • Intro of Section 10 SubGhz_Phy_PingPong application • Figure 8. SubGhz_Phy_PingPong application setup • Intro of Section 12 Dual-core management • Section 12.1.3 MBMUX messages • Section 12.2.5 RAM mapping • Section 13.2 KMS key definition • Section 13.3 LoRaWAN keys • Figure 18. ROM memory mapping • Section 13.5 How to size NVM for KMS data storage • Section 13.6 KMS configuration files to build the application • Section 15.1 Memory footprint • Intro of Section 10 SubGhz_Phy_PingPong application • Figure 8. SubGhz_Phy_PingPong application setup • Intro of Section 12 Dual-core management • Section 12.1.3 MBMUX messages • Section 12.2.5 RAM mapping • Section 13.2 KMS key definition • Section 13.3 LoRaWAN keys • Figure 18. ROM memory mapping • Section 13.5 How to size NVM for KMS data storage

Date	Versi on	Changes
9-Jul-2021 (cont'd)	5	Updated: • Section 13.6 KMS configuration files to build the application • Section 15.1 Memory footprint Added: • Seven tables in Section 6.7 Application callbacks • Section 11 SubGhz_Phy_Per application • Section 14 How to secure a LoRaWAN application

Contents

List of tables

List of figures