UM2073

User manual

STM32 LoRaWAN® Expansion Package for STM32Cube

Introduction

This user manual describes the I-CUBE-LRWAN LoRaWAN® Expansion Package implementation on the STM32L0 Series, STM32L1 Series, and STM32L4 Series microcontrollers. This document also explains how to interface with the LoRaWAN® to manage the LoRa® wireless link.

LoRa® is a type of wireless telecommunication network designed to allow long-range communications at a very low bit-rate and enabling long-life battery-operated sensors. LoRaWAN® defines the communication and security protocol that ensures interoperability with the LoRa® network. The LoRaWAN® Expansion Package is compliant with the LoRa Alliance® specification protocol named LoRaWAN®.

The I-CUBE-LRWAN main features are the following:

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

The I-CUBE-LRWAN Expansion Package is based on the STM32Cube HAL drivers (Refer to LoRa standard overview).

This user manual provides customer examples on NUCLEO-L053R8, NUCLEO-L073RZ, NUCLEO-L152RE, and NUCLEO-L476RG using Semtech expansion boards SX1276MB1MAS, SX1276MB1LAS, SX1272MB2DAS, SX1262DVK1DAS, SX1262DVK1CAS, and SX1262DVK1BAS.

This document targets the following tools:

- P-NUCLEO-LRWAN1, STM32 Nucleo pack for LoRa® technology (Legacy only)
- P-NUCLEO-LRWAN2, STM32 Nucleo starter pack (USI®) for LoRa® technology
- P-NUCLEO-LRWAN3, STM32 Nucleo starter pack (RisingHF) for LoRa® technology
- B-L072Z-LRWAN1, STM32 Discovery kit embedding the CMWX1ZZABZ-091 LoRa® module from Murata
- I-NUCLEO-LRWAN1, LoRa® expansion board for STM32 Nucleo, based on the WM-SG-SM-42 LPWAN module (USI®) available in P-NUCLEO-LRWAN2
- LRWAN-NS1, expansion board featuring the RisingHF modem RHF0M003 available in P-NUCLEO-LRWAN3

1 General information

The I-CUBE-LRWAN Expansion Package runs on STM32 32-bit 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.

1.1 Terms and definitions

Table 1 presents the definitions of the acronyms that are relevant for a better contextual understanding of this document.

Table 1. List of acronyms

Acronym	Definition
ABP	Activation by personalization
App	Application
API	Application programming interface
BSP	Board support package
FSM	Finite-state machine
FUOTA	Firmware update over the air
HAL	Hardware abstraction layer
IoT	Internet of things
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
MPDU	MAC protocol data unit
OTAA	Over-the-air activation
PLME	Physical sublayer management entity
PPDU	Physical protocol data unit
SAP	Service access point
SBSFU	Secure Boot and Secure Firmware Update

1.2 Overview of available documents and references

Table 2 lists the complementary references for using I-CUBE-LRWAN.

Table 2. References

ID	Description
[1]	*LoRa Alliance specification protocol named LoRaWAN version V1.0.3* July 2018 final release
[2]	*Low-Rate Wireless Personal Area Networks (LRWPANs)* IEEE Std 802.15.4TM, 2011
[3]	*LoRaWAN® Regional Parameters v1.0.3revA*, July 2018 release
[4]	*LoRa Alliance Fragmented Data Block Transport over LoRaWAN Specification v1.0.0* September 2018 [TS-004]
[5]	*LoRa Alliance Remote Multicast Setup over LoRaWAN Specification v1.0.0* September 2018 [TS-005]
[6]	*LoRa Alliance Application layer clock synchronization over LoRaWAN Specification v1.0.0* September 2018 [TS-003]
[7]	Application note *Integration guide for the X-CUBE-SBSFU STM32Cube Expansion Package* (AN5056)
[8]	Application note *I-CUBE-LRWAN embedding FUOTA, application implementation* (AN5411)
[9]	Application note *Examples of AT commands on I-CUBE-LRWAN* (AN4967)
[10]	Application note *How to build a LoRa® application with STM32CubeWL* (AN5406)
[11]	User manual *Getting started with the P-NUCLEO-LRWAN2 and P-NUCLEO-LRWAN3 starter packs* (UM2587)
[12]	User manual *STM32 Nucleo-64 boards (MB1136)* (UM1724)
[13]	User manual *STM32WL Nucleo-64 board (MB1389)* (UM2592)
[14]	*WM-SG-SM-42 AT Command Reference Manual* located under USI_I-NUCLEO-LRWAN1(1)
[15]	*RHF-PS01709 LoRaWAN Class ABC AT-Command Specification* available from RiSiNGHF home page(1)

1. *This URL belongs to a third party. It is active at document publication, however, STMicroelectronics shall not be liable for any change, move, or inactivation of the URL or the referenced material.*

2 LoRa® standard overview

2.1 Overview

This section provides a general overview of the LoRa® and LoRaWAN® recommendations, particularly focusing on the LoRa® end device that is the core subject of this user manual.

LoRa® is a type of wireless telecommunication network designed to allow long-range communication at a very low bit rate and enabling long-life battery-operated sensors. LoRaWAN® defines the communication and security protocol ensuring interoperability with the LoRa® network.

The LoRaWAN® Expansion Package is compliant with the LoRa Alliance® specification protocol named LoRaWAN®.

Table 3 shows the LoRaWAN® class usage definition. Refer to Section 2.2.2 for further details on these classes.

Table 3. **LoRaWAN® classes intended usage**

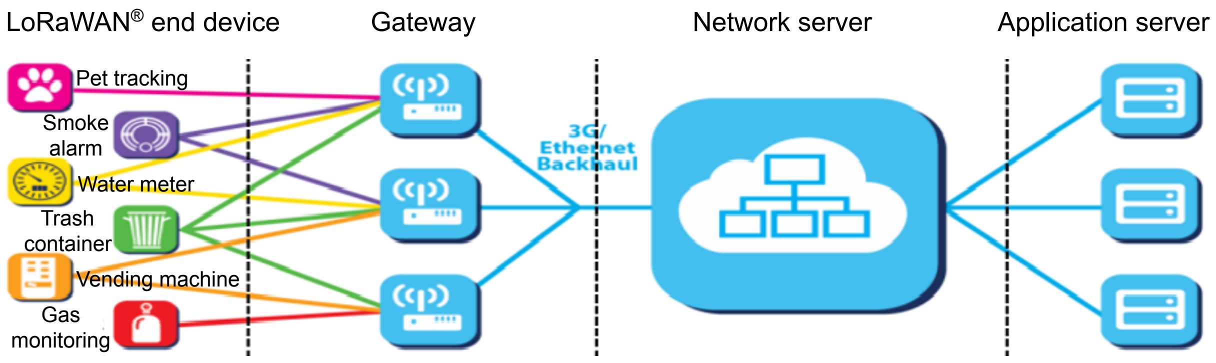
Class name	Intended usage
A - All	• Battery-powered sensors or actuators with no latency constraint • Most energy-efficient communication class • Must be supported by all devices
B - Beacon	• Battery-powered actuators • Energy-efficient communication class for latency controlled downlink • Based on slotted communication synchronized with a network beacon
C - Continuous	• Main powered actuators • Devices that can afford to listen continuously • No latency for downlink communication

Note: *While the physical layer of LoRa® is proprietary, the rest of the protocol stack (LoRaWAN®) is kept open and its development is carried out by the LoRa Alliance®.*

2.2 Network architecture

The LoRaWAN® network is structured in a star of stars topology, where the end devices are connected via a single LoRaWAN® link to one gateway as shown in Figure 1.

Figure 1. **Network diagram**



2.2.1 End-device architecture

The end device is composed of an RF transceiver (also known as radio) and a host STM32 MCU. The RF transceiver is composed of a modem and an RF up-converter. The MCU implements the radio driver, the LoRaWAN® stack and optionally the sensor drivers.

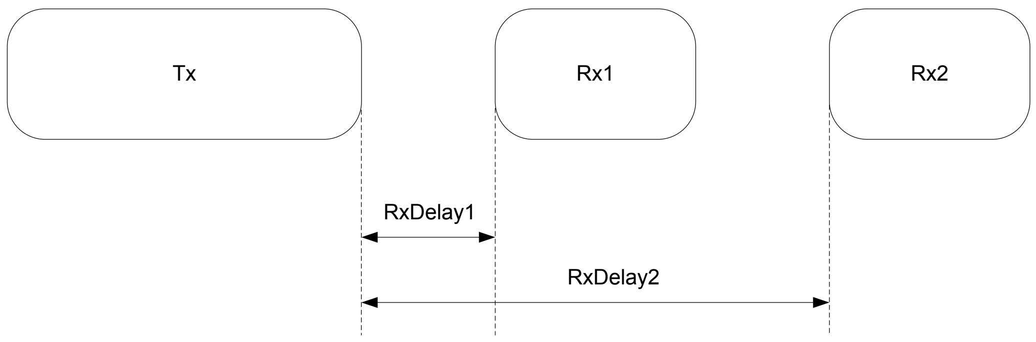
2.2.2 End-device classes

The LoRaWAN® has several different classes of end-point devices, addressing the different needs reflected in the wide range of applications.

Bi-directional Class-A end devices (all devices)

- Class-A operation is the lowest power end-device system.
- Each end-device uplink transmission is followed by two short downlinks receive windows.
- Downlink communication from the server shortly after the end-device has sent an uplink transmission (Refer to Figure 2).
- Transmission slot is based on the own communication needs of the end device (ALOHA-type protocol).

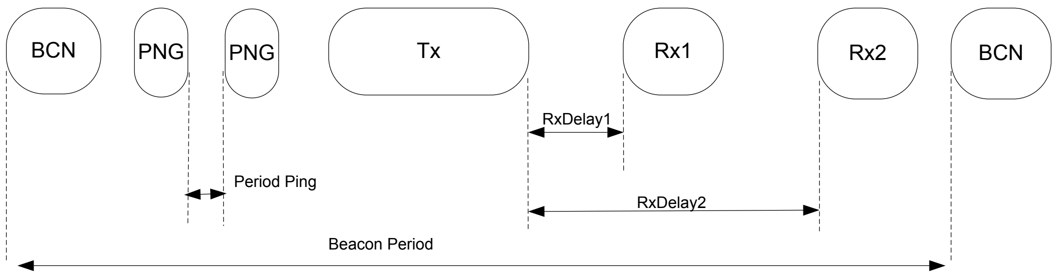
Figure 2. Tx/Rx time diagram (Class-A)



Bi-directional end-devices with scheduled receive slots - Class-B - (beacon)

- Mid power consumption
- Class-B devices open extra receive windows at scheduled times (Refer to Figure 3).
- For the end device to open the receive window at the scheduled time, the end device receives a time-synchronized beacon from the gateway.
- As Class-A has priority, the device replaces the periodic ping slots with an uplink (Tx) sequence followed by Rx1 or Rx2 received windows when required by the device.

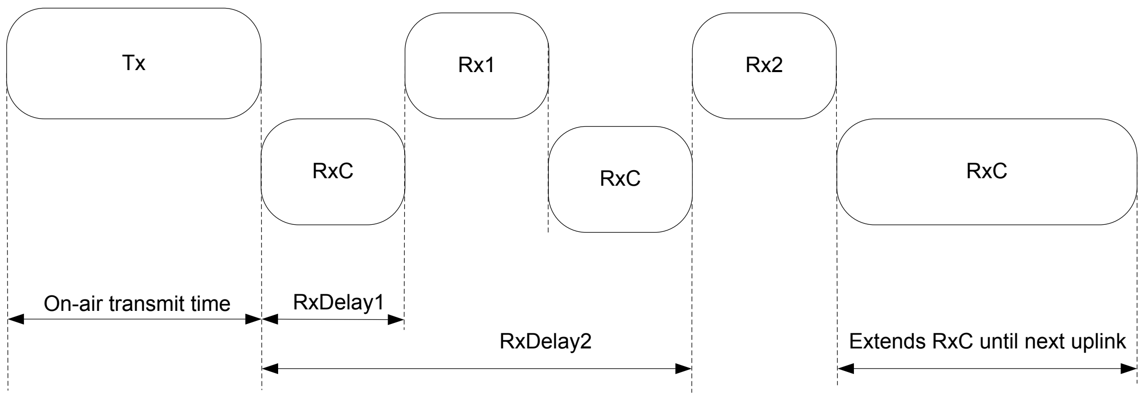
Figure 3. Tx/Rx time diagram (Class-B)



Bi-directional Class-C end devices with maximal receive slots (continuous)

- Large power consumption
- Class-C end devices have nearly continuously open receive windows, only closed when transmitting (Refer to Figure 4).

Figure 4. **Tx/Rx time diagram (Class-C)**



2.2.3 End-device activation (joining)

Over-the-air activation (OTAA)

The OTAA is a joining procedure for the LoRaWAN® end device to participate in a LoRaWAN® network. Both the LoRaWAN® end device and the application server share the same secret key known as AppKey. During a joining procedure, the LoRaWAN® end device and the application server exchange inputs to generate two session keys:

- A network session key (NwkSKey) for MAC commands encryption
- An application session key (AppSKey) for application data encryption

Activation by personalization (ABP)

In the case of ABP, the `NwkSkey` and `AppSkey` are already stored in the LoRaWAN® end device that sends the data directly to the LoRaWAN® network.

2.2.4 Regional spectrum allocation

The LoRaWAN® specification varies slightly from region to region. The European, North American, and Asian regions have different spectrum allocations and regulatory requirements (Refer to Table 4 for more details).

Table 4. LoRaWAN® regional spectrum allocation

Region	Supported	Band (MHz)	Duty cycle (%)	Output power (dBm)(1)
EU	Y	868	< 1	+13
EU	Y	433	< 1	+10
US	Y	915	No	+27
CN	N	779	< 0.1	+10
AS	Y	923	No	+13
IN	Y	865	No	+27
KR	Y	920	No	+11
RU	Y	864	< 1	+13
AU	Y	915	No	+28
CN	Y	470	No	+17

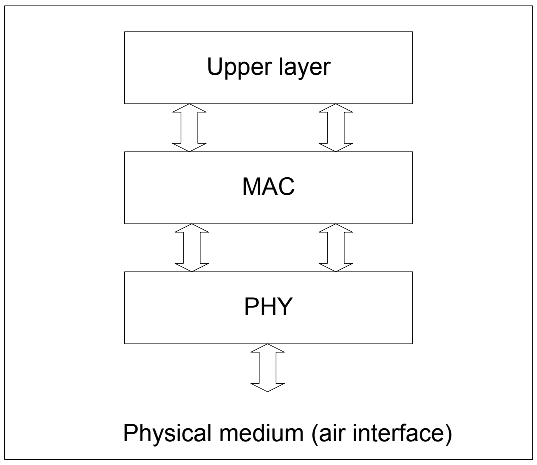
1. *The output power values are defined with the default maximal EIRP (Refer to the region associated section in [3]) and the default antenna gain (2.15 by default): Default_Power = floor (Default_Max_EIRP - Default_Antenna_Gain)*

2.3 Network layer

The LoRaWAN® architecture is defined in terms of blocks, also called “layers”. Each layer is responsible for one part of the standard and offers services to higher layers.

The end device is at least made of one physical layer (PHY), that embeds the radio frequency transceiver, a MAC sublayer providing access to the physical channel, and an application layer, as shown in Figure 5.

Figure 5. LoRaWAN® layers



2.3.1 Physical layer

The physical layer provides two services:

- The PHY data service, that enables the Tx/Rx of physical protocol data units (PPDUs)
- The PHY management service, that enables the personal area network information base (PIB) management

2.3.2 MAC sublayer

The MAC sublayer provides two services:

- The MAC data service, that enables the transmission and reception of MAC protocol data units (MPDU) across the physical layer
- The MAC management service, that enables the PIB management

2.4 Message flow

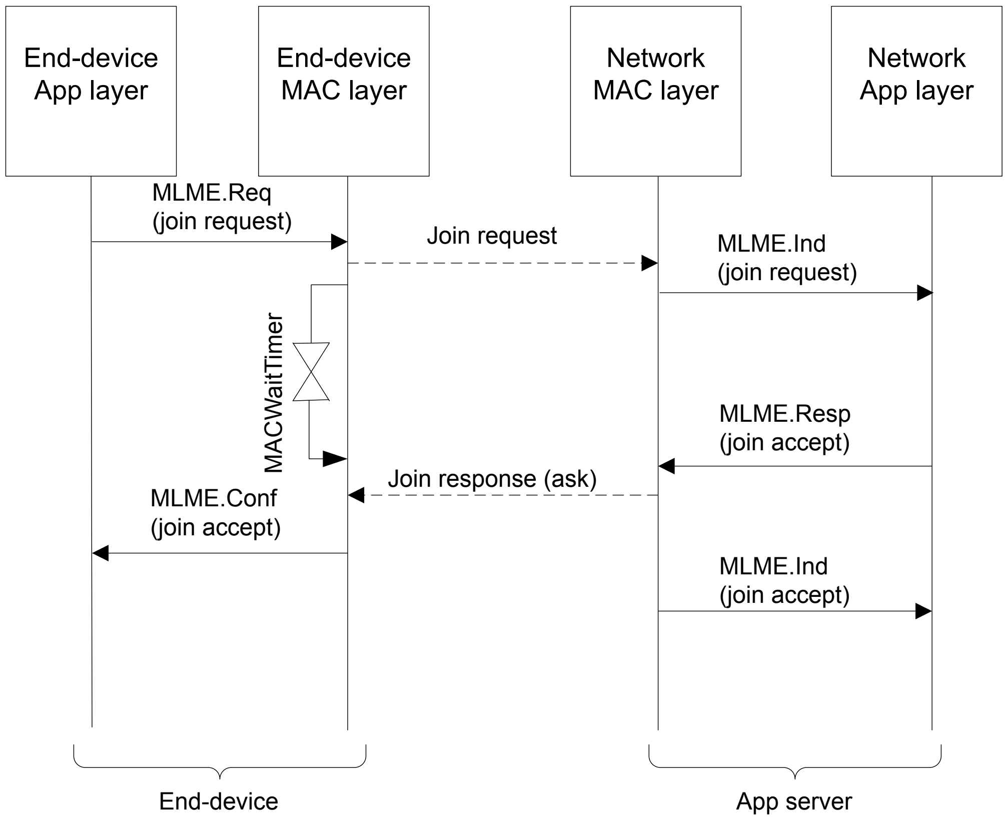
This section describes the information flow between the N-user and the N-layer. The service request is performed through a service primitive.

2.4.1 End-device activation details (joining)

Before communicating on the LoRaWAN® network, the end device must be associated or activated following one of the two activation methods described in End-device activation (joining).

The message sequence chart (MSC) in Figure 6 shows the OTAA activation method.

Figure 6. **Message sequence chart for joining (MLME primitives)**



2.4.2 End-device class-A data communication

The end device transmits data by one of the following methods: through a confirmed-data message method (Refer to Figure 7) or through an unconfirmed-data message (Refer to Figure 8).

In the first method, the end device requires an `Ack` (acknowledgment) to be done by the receiver while in the second method, the `Ack` is not required.

When an end device sends data with an `Ackreq` (acknowledgment request), the end device must wait during an acknowledgment duration `AckWaitDuration` to receive the acknowledgment frame (Refer to Section 4.3.1).

If the acknowledgment frame is received, then the transmission is successful, else the transmission failed.

Figure 7. Message sequence chart for confirmed data (MCPS primitives)

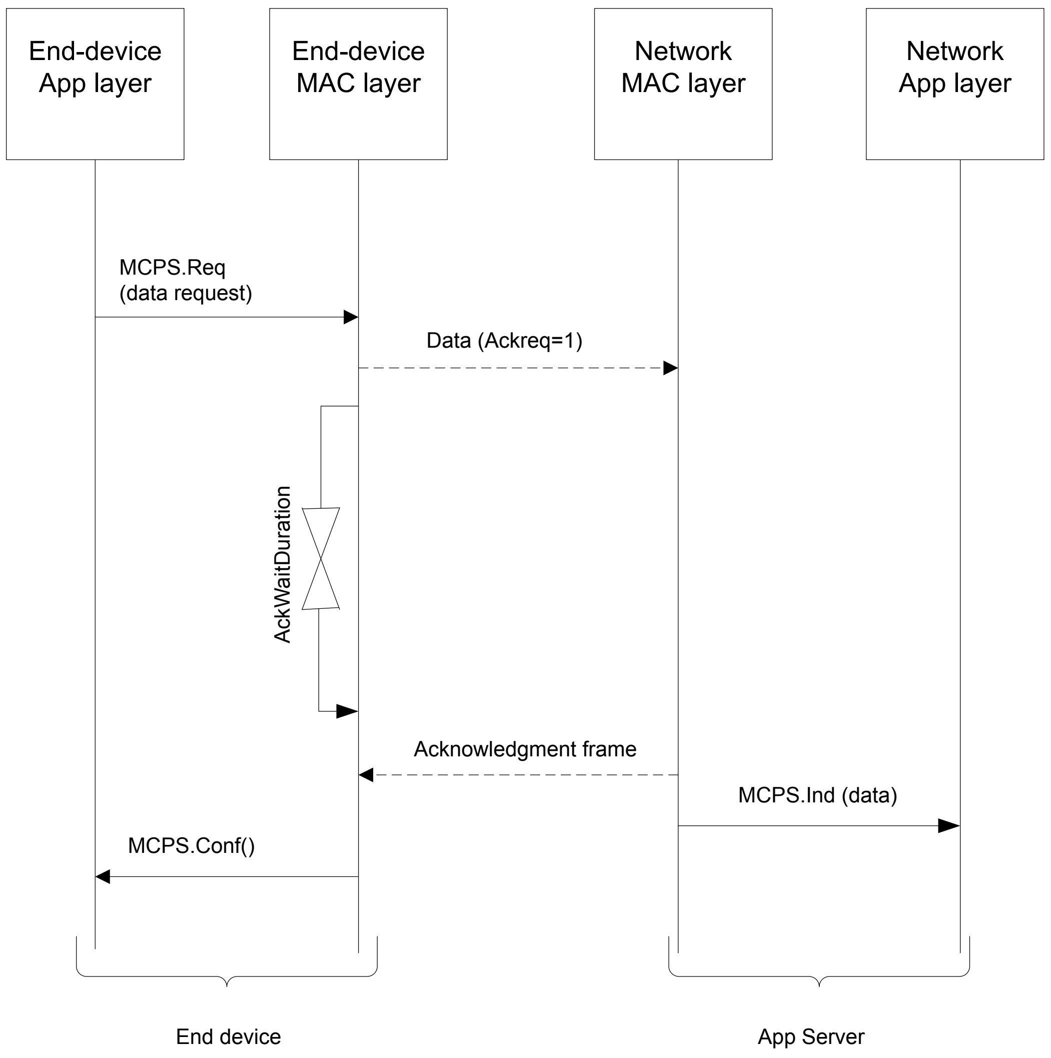
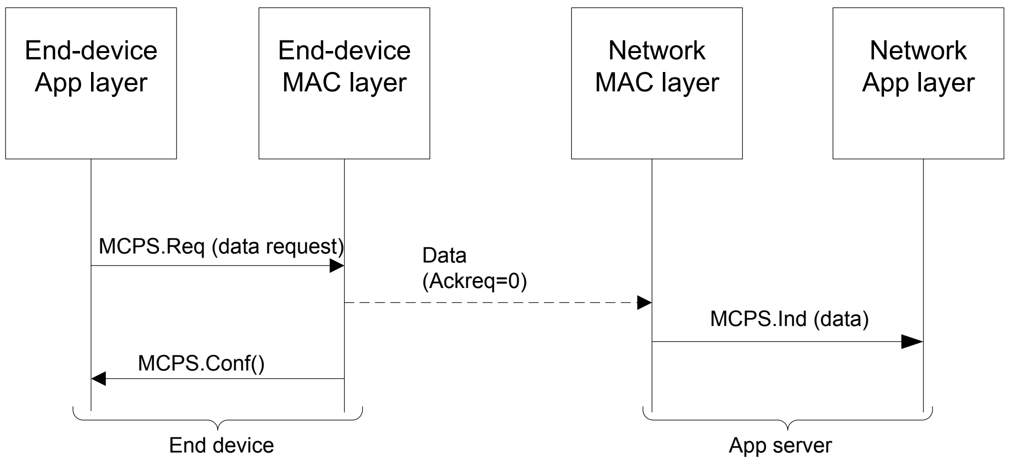


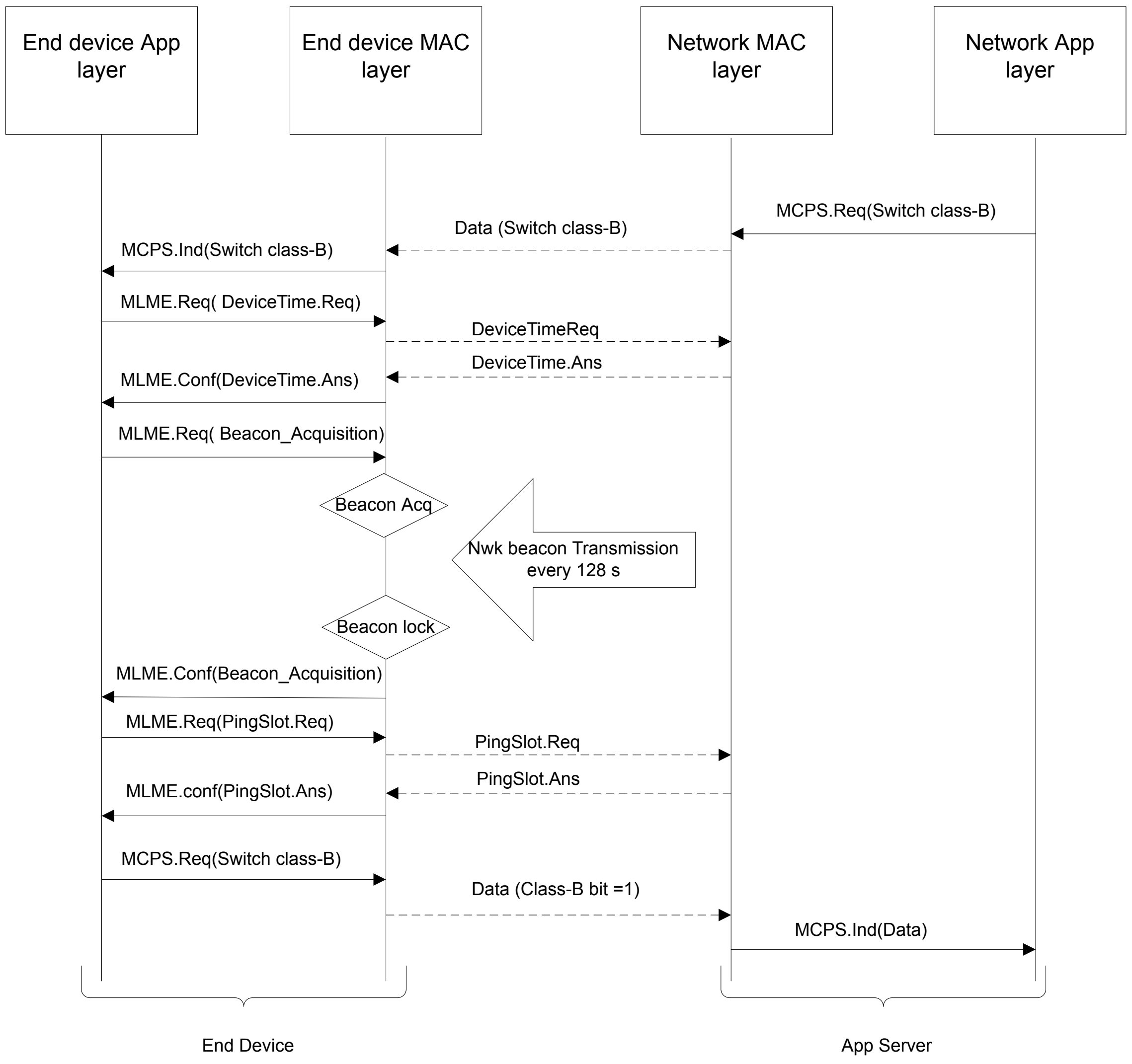
Figure 8. Message sequence chart for unconfirmed data (MCPS primitives)



2.4.3 End-device class-B mode establishment

This section describes the LoRaWAN® class-B mode establishment. Class-B is achieved by having the gateway sending a beacon on a 128 s regular basis to synchronize all the end devices in the network so that the end device can open a short Rx window called a `ping slot`. The decision to switch from class-A to class-B always comes from the application layer.

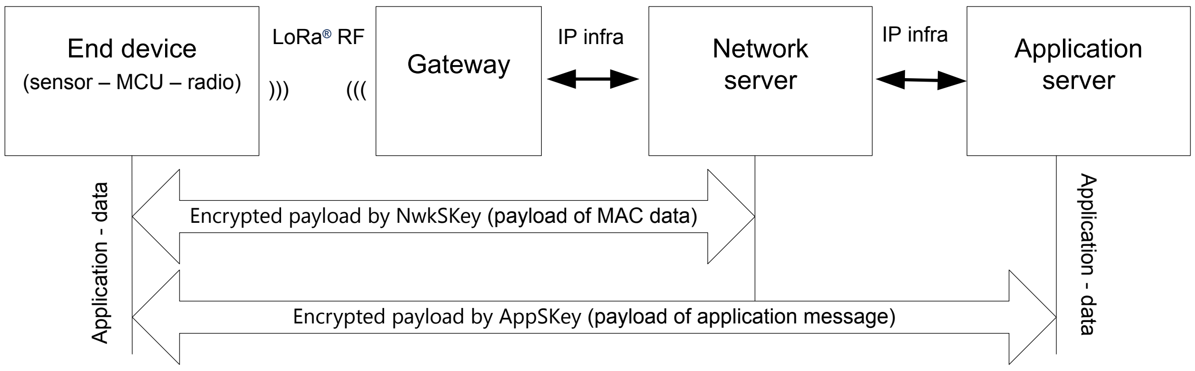
Figure 9. MSC MCPS class-B primitives



2.5 Data flow

The data integrity is ensured by the network session key `NwkSKey` and the application session key `AppSKey`. `NwkSKey` is used to encrypt and decrypt the MAC payload data and `AppSKey` is used to encrypt and decrypt the application payload data. Refer to Figure 10 for the data flow representation.

Figure 10. Data flow



`NwkSKey` is a private key that is derived from a root key and unique session identifier for each end device. `NwkSKey` provides message integrity for the communication and provides security for the end device towards the network server communication.

`AppSKey` is a private key that is derived from a root key and unique session identifier for each end device. `AppSKey` is used to encrypt/decrypt the application data. In other words, `AppSKey` provides security for the application payload. In this way, the application data sent by an end device cannot be interpreted by the network server.

3 I-CUBE-LRWAN middleware description

3.1 Overview

This I-CUBE-LRWAN Expansion Package offers a LoRaWAN® stack middleware for STM32 microcontrollers. This middleware is split into several modules:

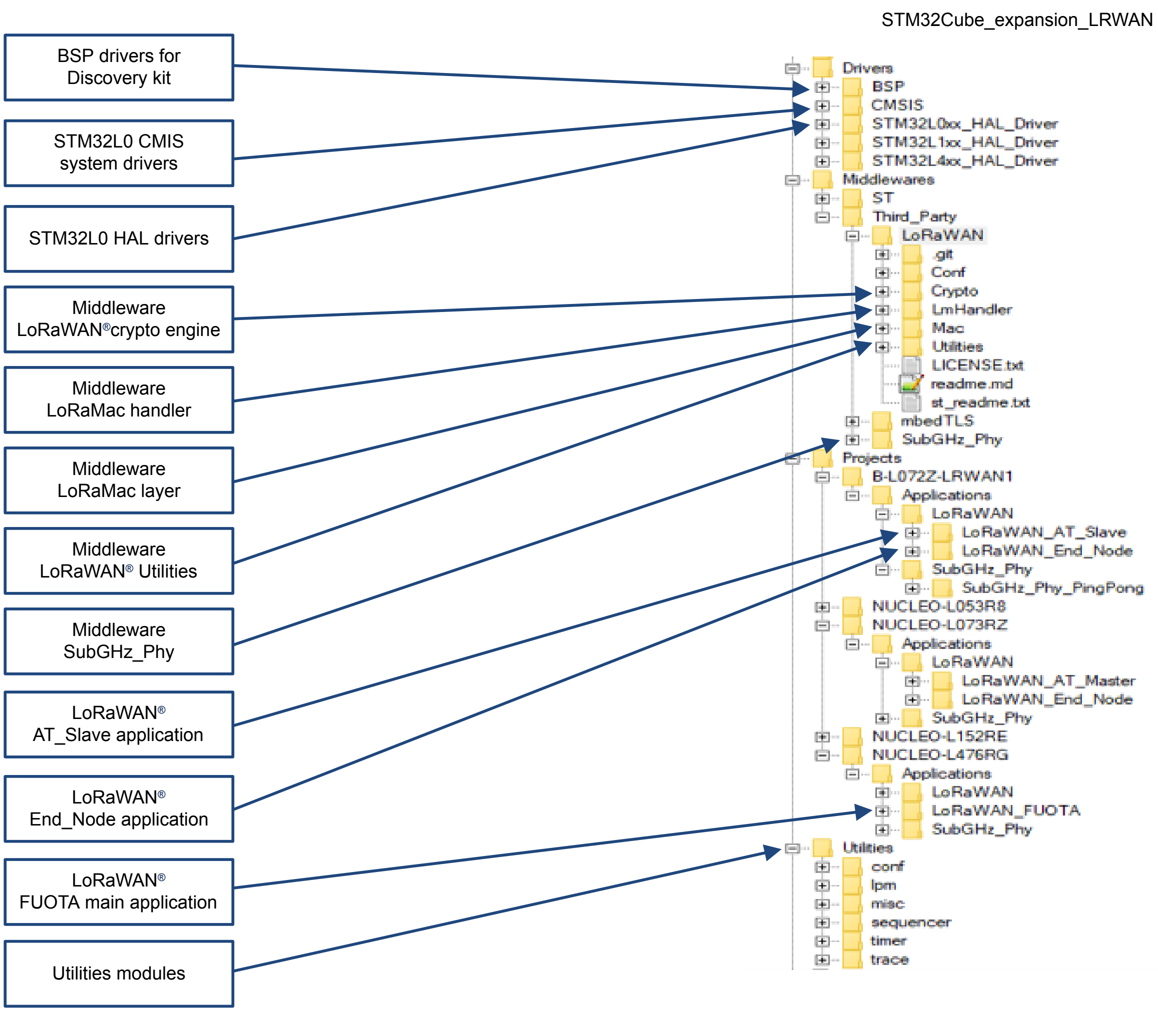
- LoRaMac layer module
- LoRaWAN® utility module
- LoRaWAN® crypto module
- LoRaMac handler

The LoRaMac handler module implements a LoRaWAN® state machine coming on top of the LoRaMac layer. The LoRaWAN® stack module interfaces with the BSP Semtech radio driver module.

This middleware is provided in a source-code format and is compliant with the STM32Cube HAL driver.

Refer to Figure 11 for the project file structure.

Figure 11. **Program file structure**



The I-CUBE-LRWAN Expansion Package includes:

- The LoRaWAN® stack middleware:
  - LoRaWAN® layer
  - LoRaWAN® utilities
  - LoRaWAN® software crypto engine
  - LoRaMac handler state machine
- Board support package:
  - Radio Semtech drivers
  - ST sensors drivers
- STM32L0xx HAL drivers
  STM32L1xx HAL drivers
  STM32L4xx HAL drivers
- Utilities:
  - Tool sequencer provides services to manage tasks.
  - Timer server provides timers service to the application.
  - Low-power management provides power management service to the application.
  - Trace provides trace capabilities to the application.
- LoRaWAN® applications:
  - LoRaWAN_AT_Slave
  - LoRaWAN_End_Node
  - LoRaWAN_AT_Master
  - LoraWAN_FUOTA
- SubGHz_Phy application:
  - SubGig_Phy_PingPong

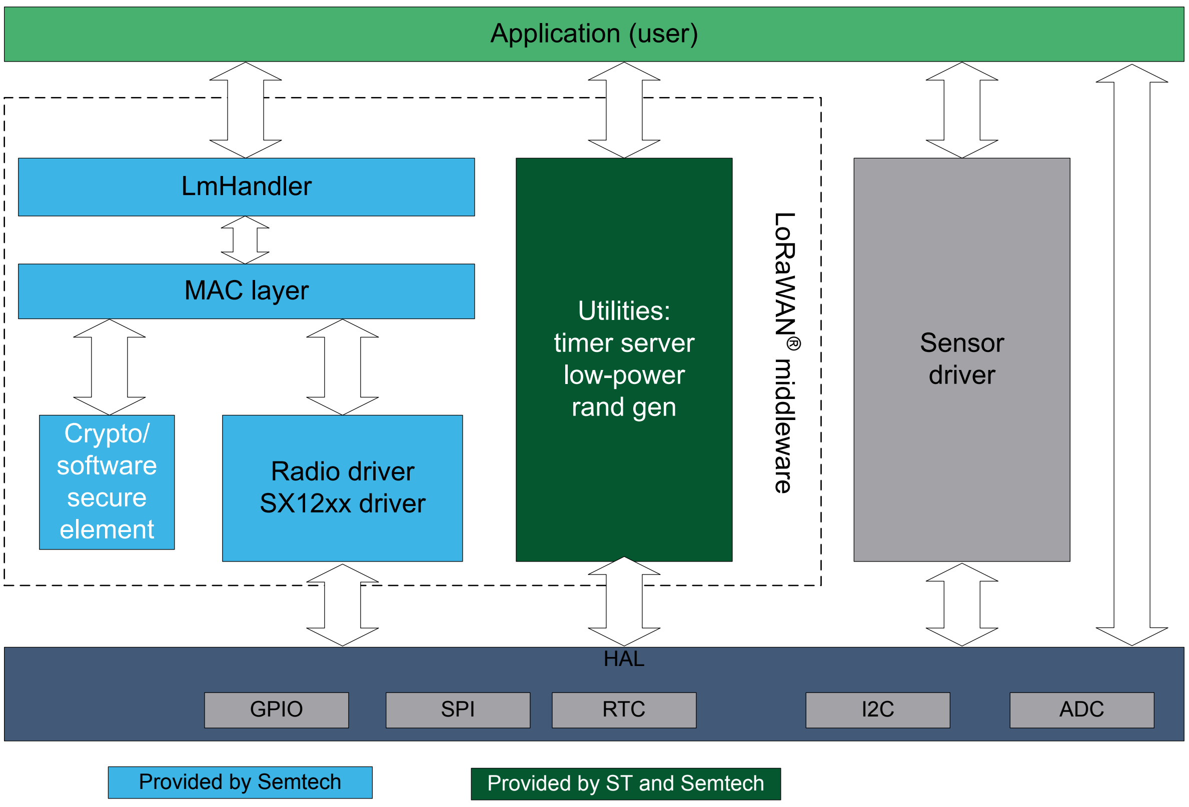
3.2 Features

- 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 the South Korean government
- RU 864 Mhz ISM band defined by Russian regulation
- AS 923 Mhz ISM band defined by Asian governments
- AU 915 Mhz ISM bands defined by the Australian government
- IN 865 Mhz ISM bands defined by the Indian government
- CN 470 Mhz ISM band defined by the People's Republic of China government
- CN 779 Mhz ISM band defined by the People's Republic of China government
- End-device activation either through over-the-air activation (OTAA) or through activation-by-personalization (ABP)
- Adaptive data rate support
- LoRaWAN® test application for certification tests included
- Low-power optimized

3.3 Architecture

Figure 12 describes the main design of the firmware for the I-CUBE-LRWAN application.

Figure 12. Main design of the firmware


Application (user)
LmHandler
MAC layer
Crypto/ software secure element
Radio driver SX12xx driver
Utilities: timer server low-power rand gen
LoRaWAN® middleware
Sensor driver
HAL
GPIO
SPI
RTC
I2C
ADC
Provided by Semtech
Provided by ST and Semtech

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

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

The radio driver uses the SPI and the GPIO hardware to control the radio (Refer to Figure 12). The radio driver also provides a set of APIs to be used by higher-level software.

The LoRa® radio is provided by Semtech, though the APIs are slightly modified to interface with the STM32Cube HAL.

The radio driver is split into two parts:

- The sx1276.c, sx1272.c and sx126x.c contain all functions that are radio dependent only.
- The sx1276mb1mas.c, sx1276mb1las, sx1272mb2das, sx1262dvk1das, sx1262dvk1cas and sx1262dvk1bas contain all the radio board dependent functions.

The MAC controls the PHY using the 802.15.4 model. The MAC interfaces with the PHY driver and uses the timer server to add or remove timed tasks and take care of the Tx time on-air. This action ensures that the duty-cycle limitation mandated by the ETSI is respected and also carries out the AES encryption/decryption algorithm to cipher the MAC header and the payload.

Since the state machine which controls the LoRaWAN® class-A is sensitive, an intermediate level of software is inserted (`LmHandler.c`) between the MAC and the application (Refer to MAC's upper layer in Figure 12). With a set of APIs limited as of now, the user is free to implement the class-A state machine at the application level.

The application, built around an infinite loop, manages the low-power, runs the interrupt handlers (alarm or GPIO) and calls the LoRaWAN® class-A if any task must be done. All the running tasks are managed by the sequencer. This application also implements sensor read access.

3.4 Hardware related components

3.4.1 Radio reset

One GPIO from the MCU is used to reset the radio. This action is done once at the initialization of the hardware (Refer to Table 42 and Section 6.1).

3.4.2 SPI

The sx127x or sx126x radio commands and registers are accessed through the SPI bus at 1 Mbit/s (Refer to Table 42 and Single MCU end-device hardware description).

3.4.3 RTC

The RTC calendar is used as a timer engine running in all power modes from the 32 kHz external oscillator. By default, the RTC is programmed to provide 1024 ticks (sub-seconds) per second. The RTC is programmed once at the initialization of the hardware when the MCU starts for the first time. The RTC output is limited to a 32-bit timer that can last 48 days.

If the user needs to change the tick duration, note that the tick duration must remain below 1 ms.

3.4.4 Input lines

3.4.4.1 sx127x interrupt lines

Four sx127x interrupt lines are dedicated to receiving the interrupts from the radio (Refer to Table 42 and Section 6.1).

The DIO0 is used to signal that the LoRa® radio completes a requested task (TxDone or RxDone).

The DIO1 is used to signal that the radio failed to complete a requested task (RxTimeout).

In FSK mode, a FIFO-level interrupt signals that the FIFO-level reached a predefined threshold and needs to be flushed.

The DIO2 is used in FSK mode and signals that the radio successfully detected a preamble.

The DIO3 is reserved for future use.

Note: The FSK mode in LoRaWAN® has the fastest data rate at 50 Kbps.

3.4.4.2 sx126x input lines

The sx126x interface is simplified compared to sx127x. One busy signal informs the MCU that the radio is busy and cannot treat any commands. The MCU must poll that the ready signal is deasserted before any new command can be sent.

DIO1 is used as a single-line interrupt.

4 I-CUBE-LRWAN middleware programming guidelines

This section describes the LoRaMac layer APIs. The proprietary PHY layer (Refer to Section 2.1 Overview) is out of the scope of this user manual and must be viewed as a black box.

4.1 Middleware initialization

The initialization of the LoRaMac layer is done through the `LoraMacinitialization` function. This function does the preamble run time initialization of the LoRaMac layer and initializes the callback primitives of the MCPS and MLME services (Refer to Table 5).

Table 5. **Middleware initialization function**

Function	Description
`LoRaMacStatus_t LoRaMacInitialization (LoRaMacPrimitives_t *primitives, LoRaMacCallback_t *callback, LoRaMacRegion_t region)`	Do initialization of the LoRaMac layer module (Refer to Section 4.3 Middleware MAC layer callbacks)

4.2 Middleware MAC layer functions

The provided APIs follow the definition of `primitive` defined in [2].

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 LoRaWAN® MAC layer confirms with a confirm primitive. Conversely, the LoRaWAN® MAC 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 confirmations and indications are implemented using callbacks.

The LoRaWAN® MAC layer provides MCPS services, MLME services, and MIB services.

4.2.1 MCPS services

The initialization of the LoRaMac layer is done through the `LoraMacinitialization` function. This function does the preamble run time initialization of the LoRaMac layer and initializes the callback primitives of the MCPS and MLME services (Refer to Table 6).

Table 6. **MCPS services function**

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

4.2.2 MLME services

The LoRaWAN® MAC layer uses the MLME services to manage the LoRaWAN® network (Refer to Table 7).

Table 7. **MLME services function**

Function	Description
`LoRaMacStatus_t LoRaMacMlmeRequest (MlmeReq_t *mlmeRequest)`	Used to generate a join request or request for a link check

4.2.3 MIB services

The MIB stores important runtime information, such as MIB_NETWORK_ACTIVATION, or MIB_NET_ID, and holds the configuration of the LoRaWAN® MAC layer, for example, MIB_ADR or MIB_APP_KEY. The provided APIs are presented in Table 8.

Table 8. **MLME services function**

Function	Description
`LoRaMacStatus_t LoRaMacMibSetRequestConfirm (MibRequestConfirm_t *mibSet)`	To set attributes of the LoRaMac layer
`LoRaMacStatus_t LoRaMacMibGetRequestConfirm (MibRequestConfirm_t *mibGet)`	To get attributes of the LoRaMac layer

4.3 Middleware MAC layer callbacks

Refer to Section 4.1 Middleware initialization for the description of the LoRaMac user event functions primitives and the callback functions.

4.3.1 MCPS

In general, the LoRaWAN® MAC layer uses the MCPS services for data transmission and data reception (Refer to Table 9).

Table 9. **MCPS primitives**

Function	Description
`void (*MacMcpsConfirm) (McpsConfirm_t *McpsConfirm) *McpsIndication)`	Event function primitive for the called callback to be implemented by the application. Response to a McpsRequest
`Void (*MacMcpsIndication) (McpsIndication_t`	Event function primitive for the called callback to be implemented by the application. Notifies application that a received packet is available

4.3.2 MLME

The LoRaWAN® MAC layer uses the MLME services to manage the LoRaWAN® network (Refer to Table 10).

Table 10. **MLME primitive**

Function	Description
`void (*MacMlmeConfirm) (MlmeConfirm_t *MlmeConfirm)`	Event function primitive so-called callback to be implemented by the application

4.3.3 MIB

No available function.

4.3.4 Battery level

The LoRaWAN® MAC layer needs a battery-level measuring service (Refer to Table 11).

Table 11. **Battery level function**

Function	Description
`uint8_t GetBatteryLevel (void)`	Get the measured battery level

4.4 Middleware MAC layer timers

4.4.1 Rx-delay window

Refer to Section 2.2.2 End-device classes. Refer to Table 12 for the Rx-delay functions.

Table 12. **Rx-delay functions**

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

4.4.2 Delay for Tx frame transmission

Table 13. **Delay for Tx frame transmission**

Function	Description
`void OnTxDelayedTimerEvent (void)`	Set timer for Tx frame transmission

4.4.3 Delay for Rx frame

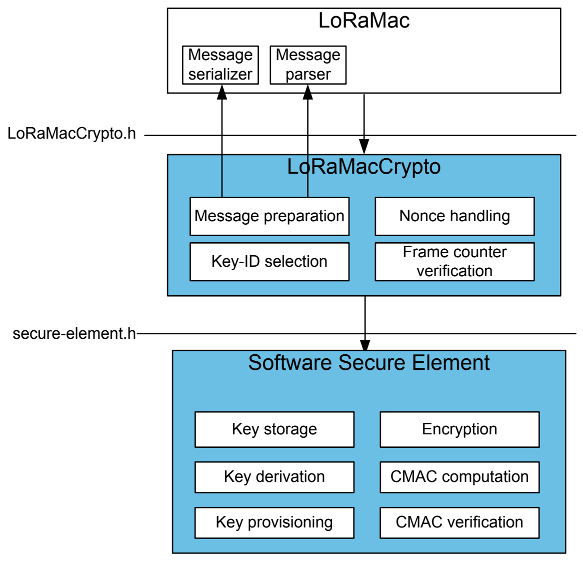
Table 14. **Delay for Rx frame function**

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

4.5 Emulated secure element

The proposed hardware platforms do not integrate a secure-element device. Therefore this secure-element device is emulated by software. Figure 13 describes the main design of the `LoRaMacCrypto` module.

Figure 13. **LoRaMacCrypto module design**



The APIs presented in Table 15 are used to manage the emulated secure element.

Table 15. **Secure-element functions**

Function	Description
`SecureElementStatus_t SecureElementInit (EventNvmCtxChanged seNvmCtxChanged)`	Initialization of the secure-element driver The Callback function is called when the non-volatile context must be stored.
`SecureElementStatus_t SecureElementRestoreNvmCtx (void* seNvmCtx)`	Restore the internal nvm context from passed pointer to non-volatile module context to be restored.
`void* SecureElementGetNvmCtx (size_t* seNvmCtxSize)`	Request address where the non-volatile context is stored.
`SecureElementStatus_t SecureElementSetKey (KeyIdentifier_t keyID, uint8_t* key)`	Set a key.
`SecureElementStatus_t SecureElementComputeAesCmac (uint8_t* buffer, uint16_t size, KeyIdentifier_t keyID, uint32_t* cmac)`	Compute a CMAC. The Key-ID determines the AES key to use.
`SecureElementStatus_t SecureElementVerifyAesCmac (uint8_t* buffer, uint16_t size, uint32_t expectedCmac, KeyIdentifier_t keyID)`	Compute cmac and compare with expected cmac. The KeyID determines the AES key to use.
`SecureElementStatus_t SecureElementAesEncrypt (uint8_t* buffer, uint16_t size, KeyIdentifier_t keyID, uint8_t* encBuffer)`	Encrypt a buffer. The KeyID determines the AES key to use.
`SecureElementStatus_t SecureElementDeriveAndStoreKey (Version_t version, uint8_t* input, KeyIdentifier_t rootKeyID, KeyIdentifier_t targetKeyID)`	Derive and store a key. The key derivation depends on the LoRaWAN® versionKeyID, rootKeyID are used to identify the root key to perform the derivation.

4.6 Middleware LmHandler application function

The interface to the MAC is done through the MAC interface file `LoRaMac.h`.

Standard mode

In standard mode, an interface file (Refer to *LmHandler* in Figure 12) is provided to let the user start without worrying about the LoRaWAN® state machine. The interface file is located in `Middlewares\Third_Party\LoRaWAN\LmHandler\LmHandler.c`.

The interface file implements:

- A set of APIs allowing access to the LoRaWAN® MAC services
- The LoRaWAN® certification test cases that are not visible to the application layer

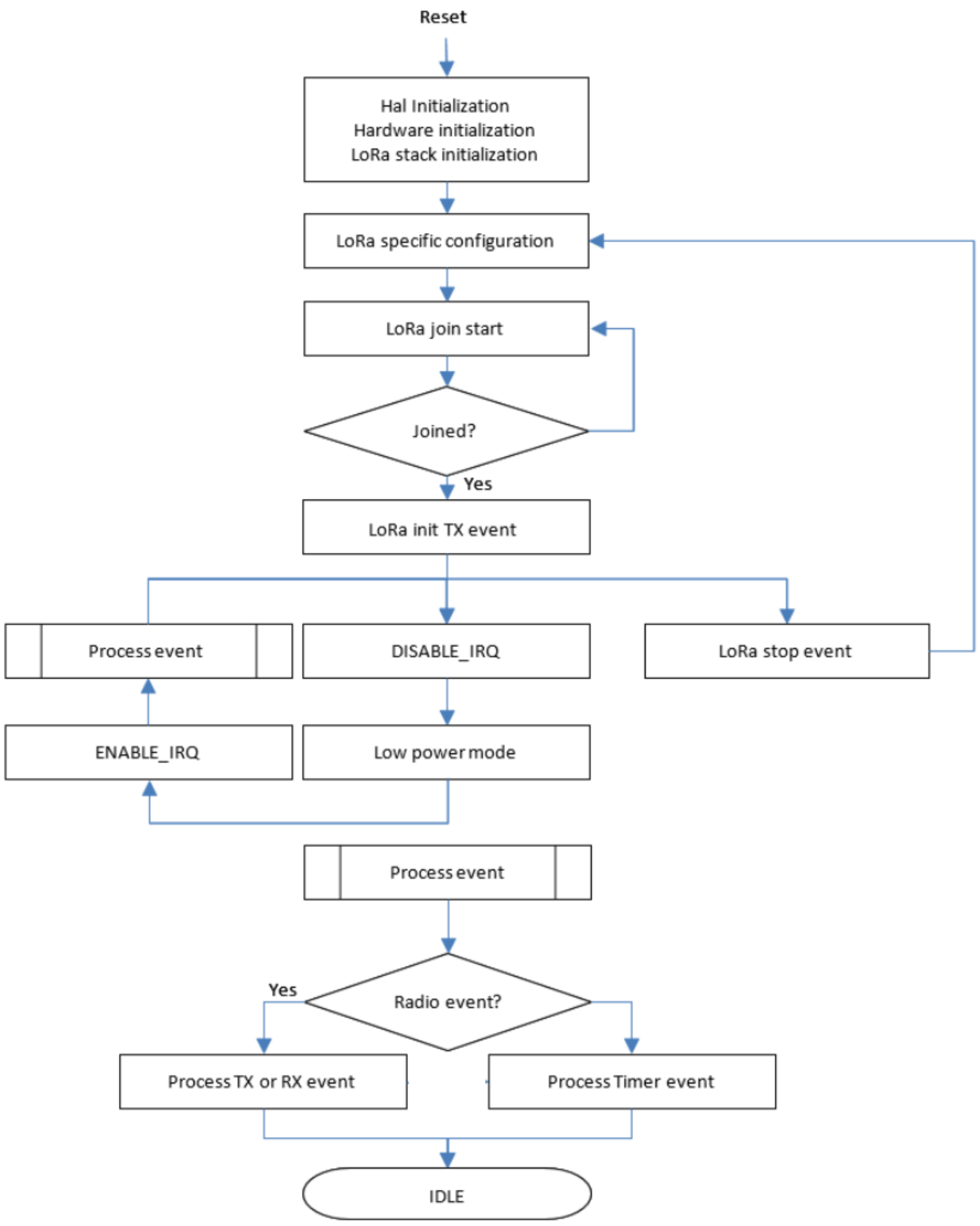
Advanced mode

In this mode, the user accesses directly the MAC layer by including the MAC in the user file.

Operational model

The operation model proposed for this LoRaWAN® End_Node (Refer to Figure 14) is based on event-driven paradigms including time-driven ones. The behavior of the LoRaWAN® system is triggered either by a timer event or by a radio event plus a guard transition.

Figure 14. Operation model



LoRaWAN® system state behavior

Figure 15 describes the LoRaWAN® End_Node system state behavior.

On reset, after the system initialization is done, the LoRaWAN® End_Node system goes into a Start state defined as *Init*.

The LoRaWAN® End_Node system sends a join network request when using the *over_the_air_activation (OTAA)* method and goes into a state defined as *Sleep*.

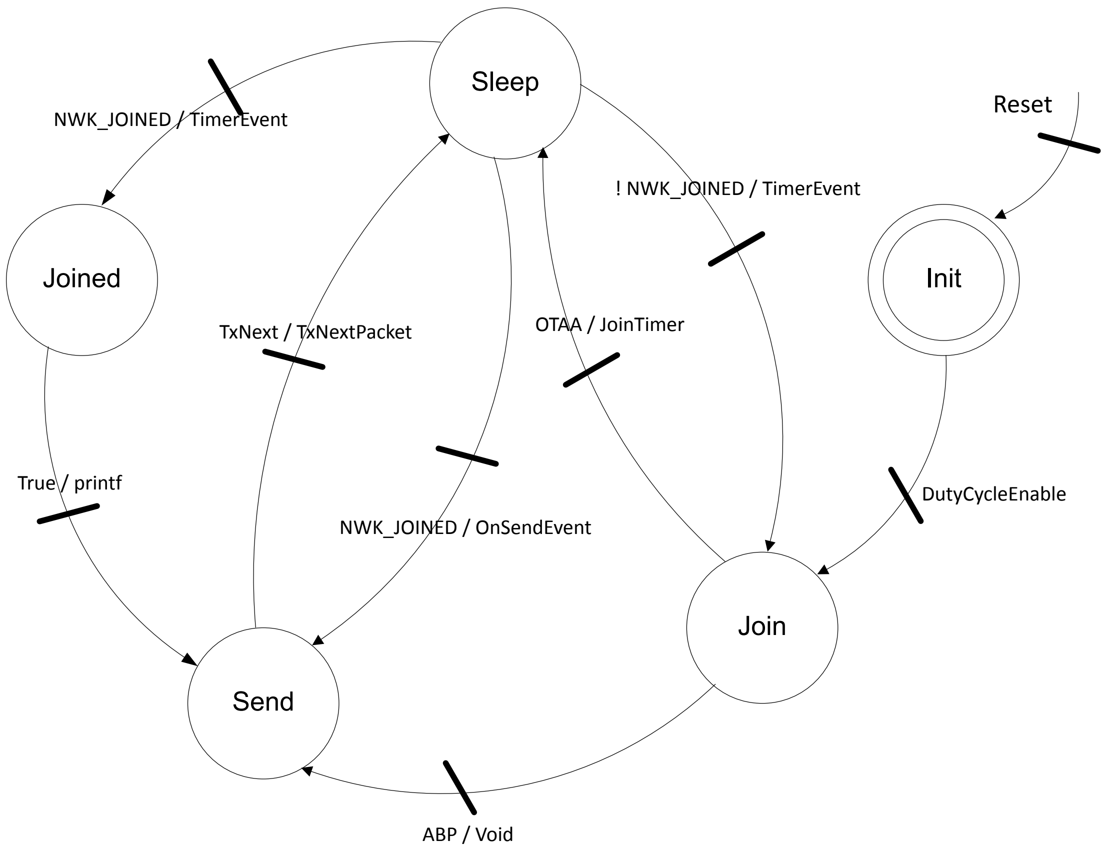
When using the *activation by personalization (ABP)*, the network is already joined, and therefore the LoRaWAN® End_Node system jumps directly to a state defined as *Send*.

From the state defined as *Sleep*, if the end device joined the network when a *TimerEvent* occurred, the LoRaWAN® End_Node system goes into a temporary state defined as *Joined* before going into the state defined as *Send*.

From the state defined as *Sleep*, if the end device joined the network when an *OnSendEvent* occurred, the LoRaWAN® End_Node system goes into the state defined as *Send*.

From the state defined as *Send*, the LoRaWAN® End_Node system goes back to the state defined as *Sleep* to wait for the *onSendEvent* corresponding to the next scheduled packet to be sent.

Figure 15. LoRaWAN® state behavior



LoRaWAN® class-B system state behavior

Figure 16 describes the LoRaWAN® class-B mode End-Node system state behavior.

Before doing a request to switch to class-B mode, an end device must be first in a Join state (Refer to Figure 14).

The decision to switch from class-A to class-B mode always comes from the application layer of the end device. If the decision comes from the network side, the application server must use the class-A uplink of the end device to send back a downlink frame to the application layer.

On MLME Beacon_Acquisition_req, the end-device LoRaWAN® class-B system state goes in BEACON_STATE_ACQUISITION.

The LoRaWAN® end device starts the beacon acquisition. When the MAC layer successfully receives a beacon in the *RxBeacon* function, the next state is BEACON_STATE_LOCKED.

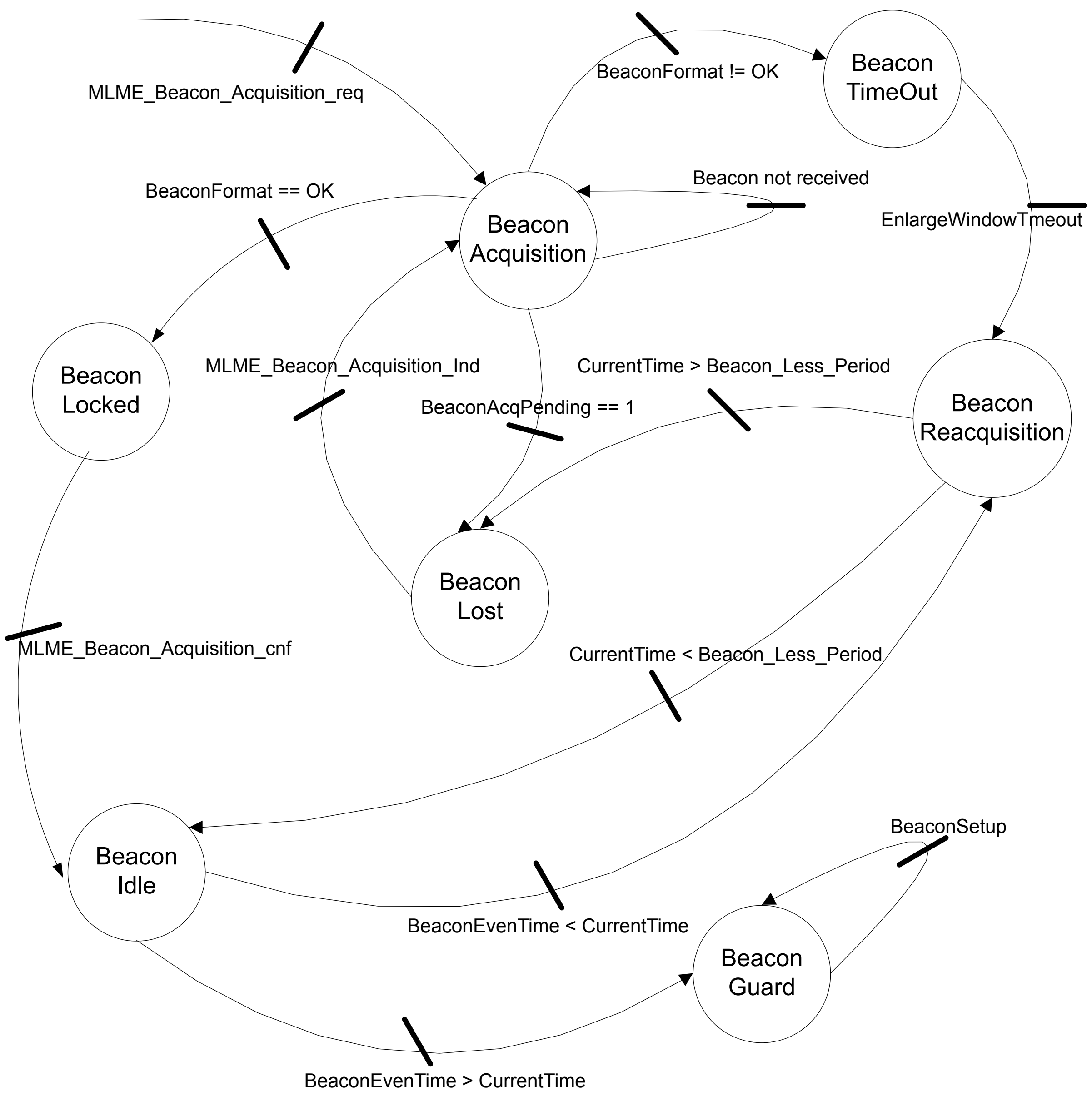
When the LoRaWAN® end device receives a beacon, the acquisition is no longer pending: the MAC layer goes in BEACON_STATE_IDLE.

In BEACON_STATE_IDLE, the MAC layer compares the *BeaconEventTime* with the current end-device time. If the beaconEventTime is less than the current end-device time, the MAC layer goes in BEACON_STATE_REACQUISITION. Otherwise, the MAC layer goes in BEACON_STATE_GUARD and performs a new beacon acquisition.

If the MAC layer does not find a beacon, the state machine stays in BEACON_STATE_ACQUISITION. This state detects that an acquisition was previously pending and changes the next state to BEACON_STATE_LOST.

When the MAC layer receives a bad beacon format, it must go in BEACON_STATE_TIMEOUT. It enlarges window timeouts to increase the chance to receive the next beacon and goes in BEACON_STATE_REACQUISITION.

Figure 16. LoRaWAN® class-B system state behavior



4.6.1 LoRa® initialization

Table 16. LoRa® initialization function

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

4.6.2 LoRa® join request entry point

Table 17. LoRa® join request entry point

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

4.6.3 LoRa® configuration

Table 18. LoRa® configuration

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

4.6.4 Request join status

Table 19. Request join status

Function	Description
`LmHandlerFlagStatus_t LmHandlerJoinStatus(void)`	Check the End-Node activation type: ACTIVATION_TYPE_NONE, ACTIVATION_TYPE_ABP, or ACTIVATION_TYPE_OTAA

4.6.5 Send an uplink frame

Table 20. Send an uplink frame

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

4.6.6 Request the current network time

Table 21. Current network time

Function	Description
`LmHandlerErrorStatus_t LmHandlerDeviceTimeReq(void)` [(1)]	The end device requests the current network time from the network. This is useful to accelerate the beacon discovery in class-B mode.

1. *To be used in place of BeaconTimeReq in LoRaWAN® version 1.0.3 or higher.*

4.6.7 Switch class request

Table 22. Switch class request

Function	Description
`LmHandlerErrorStatus LmHandlerRequestClass (DeviceClass_t newClass)`	Request the end device to switch from current to new class A, B, or C.

4.6.8 Get end-device current class

Table 23. Get end-device current class

Function	Description
`int32_t LmHandlerGetCurrentClass(DeviceClass_t *deviceClass)`	Request the currently running class-A, class-B, or class-C.

4.6.9 Request beacon acquisition

Table 24. Request beacon acquisition

Function	Description
`LmHandlerErrorStatus_t LmHandlerBeaconReq(void)`	Request the beacon slot acquisition.

4.6.10 Send unicast ping slot info periodicity

Table 25. Send unicast ping slot info periodicity

Function	Description
`LmHandlerErrorStatus_t LmHandlerPingSlotReq(uint8_t periodicity)`	Transmit to the server the unicast ping slot info periodicity.

4.6.11 Get current Tx data rate

Table 26. Get current Tx data rate

Function	Description
`int32_t LmHandlerGetTxDatarate( int8_t *txDatarate)`	Gets the current Tx data rate.

4.6.12 Set Tx data rate

Table 27. Set Tx data rate

Function	Description
`int32_t LmHandlerSetTxDatarate( int8_t txDatarate)`	Set the Tx data rate, if adaptive DR is disabled.

4.6.13 Get current Tx duty-cycle state

Table 28. Get current Tx duty-cycle state

Function	Description
`int32_t LmHandlerGetDutyCycleEnable( bool *dutyCycleEnable)`	Get the current Tx duty-cycle state.

4.6.14 Set Tx duty-cycle state

Table 29. Set Tx duty-cycle state

Function	Description
`int32_t LmHandlerSetDutyCycleEnable( bool dutyCycleEnable)`	Set the Tx duty-cycle state.

4.7 Library application callbacks

4.7.1 Current battery level

Table 30. Current battery level function

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

4.7.2 Current temperature level

Table 31. Current temperature level function

Function	Description
`uint16_t GetTemperatureLevel (void)`	Get the current temperature (degree Celsius) of the chipset in q7.8 format.

4.7.3 Board unique ID

Table 32. Board unique ID function

Function	Description
`void GetUniqueId (uint8_t *id)`	Get a unique identifier.

4.7.4 End_Node class mode change confirmation

Table 33. End_Node class mode change confirmation function

Function	Description
`void DisplayClassUpdate (DeviceClass_t Class)`	Notify the application that the End-Node class is changed.

4.8 Extended application functions

These functions are proposed to enhance when needed, application use cases.

Table 34. **Extended application functions**

Function	Description
`int32_t LmHandlerGetDevEUI( uint8_t *devEUI)`	Gets the LoRaWAN® device EUI.
`int32_t LmHandlerSetDevEUI( uint8_t *devEUI)`	Sets the LoRaWAN® device EUI.
`int32_t LmHandlerGetAppEUI( uint8_t *appEUI)`	Gets the LoRaWAN® application EUI.
`int32_t LmHandlerSetAppEUI( uint8_t *appEUI)`	Sets the LoRaWAN® application EUI.
`int32_t LmHandlerGetAppKey( uint8_t *appKey)`	Gets the LoRaWAN® application key.
`int32_t LmHandlerSetAppKey( uint8_t *appKey)`	Sets the LoRaWAN® application key.
`int32_t LmHandlerGetNetworkID( uint32_t *networkId)`	Gets the LoRaWAN® network ID.
`int32_t LmHandlerSetNetworkID( uint32_t networkId)`	Sets the LoRaWAN® network ID.
`int32_t LmHandlerGetDevAddr( uint32_t *devAddr)`	Gets the LoRaWAN® device.
`int32_t LmHandlerSetDevAddr( uint32_t devAddr)`	Sets the LoRaWAN® device.
`int32_t LmHandlerGetActiveRegion( LoRaMacRegion_t *region)`	Gets the active region.
`int32_t LmHandlerSetActiveRegion( LoRaMacRegion_t region)`	Sets the active region.
`int32_t LmHandlerGetAdrEnable( bool *adrEnable)`	Gets the adaptive data rate state.
`int32_t LmHandlerSetAdrEnable( bool adrEnable)`	Sets the adaptive data rate state.
`int32_t LmHandlerGetRX2Params( RxChannelParams_t *rxParams)`	Gets the current Rx2 data rate and frequency conf.
`int32_t LmHandlerSetRX2Params( RxChannelParams_t *rxParams)`	Sets the Rx2 data rate and frequency conf.
`int32_t LmHandlerGetTxPower( int8_t *txPower)`	Gets the current Tx power value.
`int32_t LmHandlerSetTxPower( int8_t txPower)`	Sets the Tx power value.
`int32_t LmHandlerGetRx1Delay( uint32_t *rxDelay)`	Gets the current Rx1 delay (after the Tx window).
`int32_t LmHandlerSetRx1Delay( uint32_t rxDelay)`	Sets the Rx1 delay (after the Tx window).
`int32_t LmHandlerGetRx2Delay( uint32_t *rxDelay)`	Gets the current Rx2 delay (after the Tx window).
`int32_t LmHandlerSetRx2Delay( uint32_t rxDelay)`	Sets the Rx2 delay (after the Tx window).
`int32_t LmHandlerGetJoinRx1Delay( uint32_t *rxDelay)`	Gets the current Join Rx1 delay (after the Tx window).
`int32_t LmHandlerSetJoinRx1Delay( uint32_t rxDelay)`	Sets the Join Rx1 delay (after the Tx window).
`int32_t LmHandlerGetJoinRx2Delay( uint32_t *rxDelay)`	Get the current Join Rx2 delay (after the Tx window).
`int32_t LmHandlerSetJoinRx2Delay( uint32_t rxDelay)`	Sets the Join Rx2 delay (after the Tx window).
`int32_t LmHandlerGetPingPeriodicity( uint8_t *pingPeriodicity)`	Gets the current Rx Ping Slot periodicity (If LORAMAC_CLASSB_ENABLED)
`int32_t LmHandlerSetPingPeriodicity( uint8_t pingPeriodicity)`	Sets the Rx Ping Slot periodicity (If LORAMAC_CLASSB_ENABLED)
`int32_t LmHandlerGetBeaconState( BeaconState_t *beaconState)`	Gets the beacon state (If LORAMAC_CLASSB_ENABLED)

5 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.

5.1 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 cannot switch to another task like an RTOS can do on the 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
- Wait for an 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.

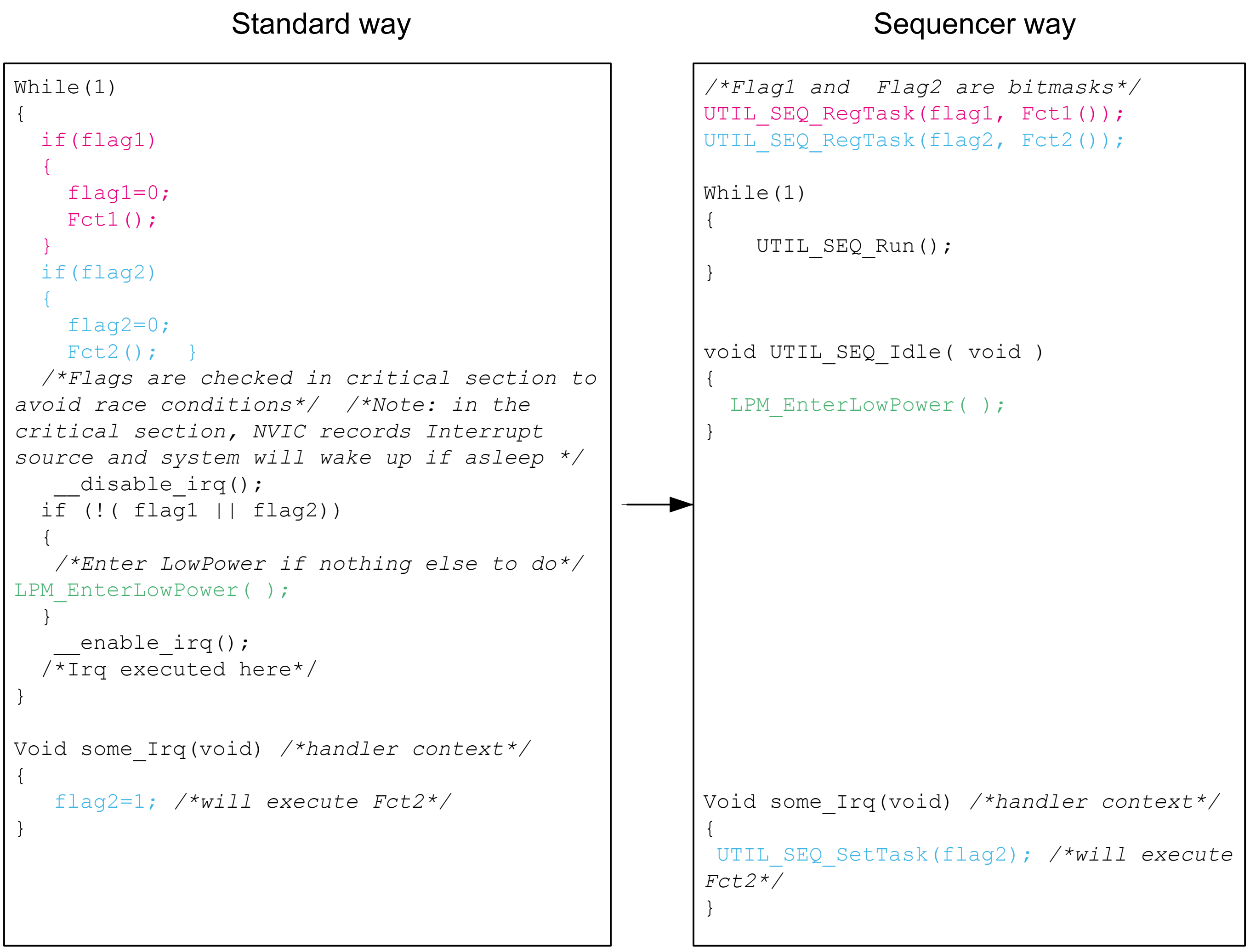
The `sequencer` utility is located in `Utilities\sequencer\stm32_seq.c`.

Table 35. **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_WaitEvt( 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 17. While-loop standard vs. sequencer implementation



```
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*/
}
```

```
/*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*/
}
```

5.2 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 36. **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.

5.3 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 used 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 37. **Low-power APIs**

Function	Description
`void UTIL_LPM_EnterLowPower( void )`	Enters the selected low-power mode. Called by the 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.

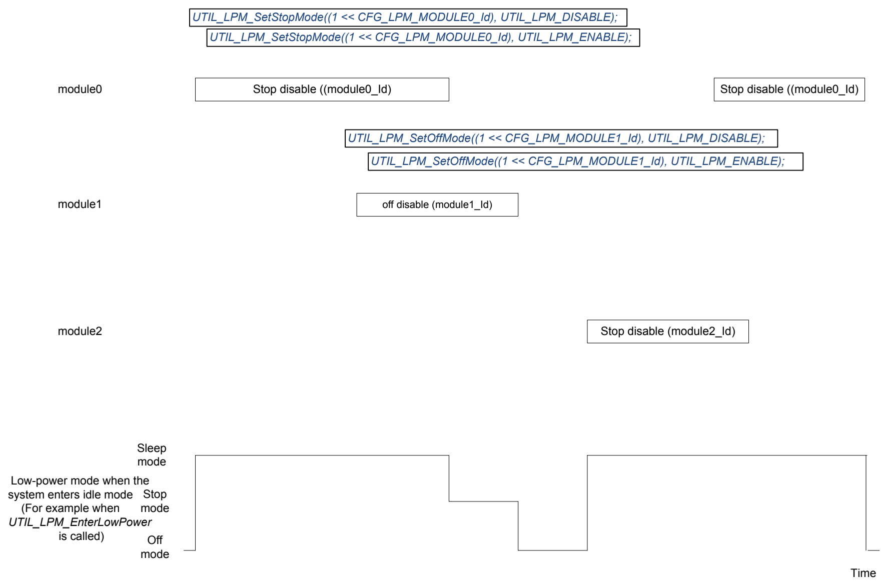
1. *Bitmaps for which the shift values are defined in `utilities_def.h`.*

The default low-power mode is Off mode, which may be Standby or Shutdown mode (defined in `void PWR_EnterOffMode (void)` from Table 38):

- 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.

Figure 18 depicts the behavior with three different firmware modules setting dependently their low-power requirements and low-power mode, selected when the system enters a low-power mode.

Figure 18. 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 the 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 38. 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 peripheral clock 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 of the rest of the system. See the product reference manual for more details.*

5.4 System time

The MCU time is referenced to the MCU reset. The system time can 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 39. 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 the UNIX epoch starting January 1st, 1970.*
2. `SysTimeMkTime` *and* `SysTimeLocalTime` *are also provided 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 summertime, there is two hours difference from Greenwich time. Assuming time is set, a local time can be printed on a 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);
}
```

5.5 Trace

The trace module enables printing 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 40. **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 the 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 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 );}
  ```

6 Example description

6.1 Single MCU end-device hardware description

The application layer, the Mac layer, and the PHY driver are implemented on one MCU. The End_Node application is implementing this hardware solution (Refer to End_Node application).

The I-CUBE-LRWAN runs on several platforms such as:

- STM32 Nucleo platform stacked with a LoRa$^{®}$ radio expansion board.
- B-L072Z-LRWAN1 Discovery kit, where LoRa$^{®}$ expansion board is not required.

Optionally, an ST X-NUCLEO-IKS01A2 sensor expansion board can be added on Nucleo boards and Discovery kits. The Nucleo-based supported hardware is presented in Table 41.

Table 41. Nucleo-based supported hardware

Nucleo board	LoRa$^{®}$ radio expansion board					
	SX1276MB1MAS	SX1276MB1LAS	SX1272MB2DAS	SX1261DVK1BAS	SX1261DVK1CAS	SX1261DVK1DAS
NUCLEO-L053R8	Supported					
NUCLEO-L073RZ	Supported		Supported (P-NUCLEO-LRWAN1[1])	Supported		
NUCLEO-L152RE	Supported					
NUCLEO-L476RG	Supported					

1. *This particular configuration is commercially available as a P-NUCLEO-LRWAN1 kit.*

The I-CUBE-LRWAN Expansion Package can easily be tailored to any other supported device and development board.

The main characteristics of the LoRa$^{®}$ radio expansion board are described in Table 42.

Table 42. LoRa$^{®}$ radio expansion board characteristics

Board	Characteristics
SX1276MB1MAS	868 MHz (HF) at 14 dBm and 433 MHz (LF) at 14 dBm
SX1276MB1LAS	915 MHz (HF) at 20 dBm and 433 MHz (LF) at 14 dBm
SX1272MB2DAS	915 MHz and 868 MHz at 14 dBm
SX1261DVK1BAS	E406V03A sx1261, 14 dBm, 868 MHz, XTAL
SX1262DVK1CAS	E428V03A sx1262, 22 dBm, 915 MHz, XTAL
SX1262DVK1DAS	E449V01A sx1262, 22 dBm, 860-930 MHz, TCXO

The radio interface is described below:

- The radio registers are accessed through the SPI.
- The DIO mapping is radio dependent, refer to Input lines.
- One GPIO from the MCU is used to reset the radio.
- One MCU pin is used to control the antenna switch to set it either in Rx mode or in Tx mode.

The hardware mapping is described in the hardware configuration files in `Projects\<target>\Applications\LoRaWAN\<App_Type>\Core\inc` folder, where:

- The `<target>` can be `STM32L053R8-Nucleo`, `STM32L073RZ-Nucleo`, `STM32L152RE-Nucleo`, `STM32L476RG-Nucleo`, or `B-L072Z-LRWAN1` (Murata modem device).
- The `<App_Type>` can be `LoRaWAN_AT_Master`, `LoRaWAN_End_Node`, `LoRaWAN_AT_Slave`, or `SubGHz_Phy_PingPong`.

Interrupts

Table 43 shows the interrupt priorities level applicable for the Cortex system processor exception and the STM32L0 Series LoRa® application-specific interrupt (IRQ).

Table 43. **STM32L0xx IRQ priorities**

Interrupt name	Preempt priority	Sub-priority
RTC	0	NA
EXTI2_3	0	NA
EXTI4_15	0	NA

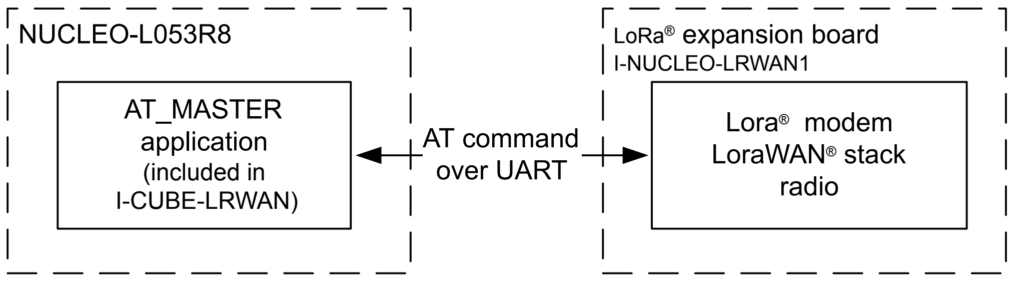
6.2 Split end-device hardware description (Two-MCU solution)

The application layer, the Mac layer, and the PHY driver are separated. The LoRaWAN® End_Node is composed of a LoRa® modem and a host controller. The LoRa® modem runs the LoRaWAN® stack (Mac and PHY layers) and is controlled by a LoRa® host implementing the application layer.

The `LoRaWAN_AT_Master` application implementing the LoRa® host on a Nucleo board is compatible with the `LoRaWAN_AT_Slave` application (Refer to Section 6.6). The `LoRaWAN_AT_Slave` application demonstrates a modem on the CMWX1ZZABZ-091 LoRa® module from Murata. The `LoRaWAN_AT_Master` application is also compatible with the I-NUCLEO-LRWAN1 expansion board featuring the WM-SG-SM-42 LPWAN module from USI and with the LRWAN_NS1 expansion board featuring the RiSiNGHF modem RHF0M003 available in P-NUCLEO-LRWAN3 (Refer to [11]).

This split solution is used to design the application layer without any constraint linked to the real-time requirement of the LoRaWAN® stack.

Figure 19. **Split end-device solution concept**

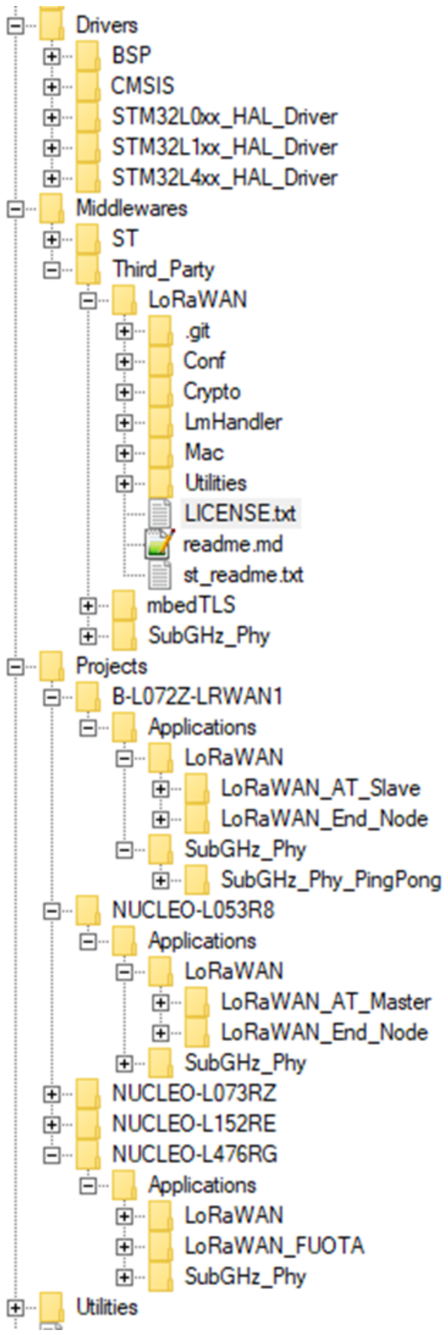


The interface between the LoRa® modem and the LoRa® host is a UART running AT commands.

6.3 Package description

When the user unzips the I-CUBE-LRWAN, the package presents the structure shown in Figure 20.

Figure 20. I-CUBE-LRWAN structure



```
Drivers
    BSP
    CMSIS
    STM32L0xx_HAL_Driver
    STM32L1xx_HAL_Driver
    STM32L4xx_HAL_Driver
Middlewares
    ST
    Third_Party
        LoRaWAN
            .git
            Conf
            Crypto
            LmHandler
            Mac
            Utilities
            LICENSE.txt
            readme.md
            st_readme.txt
        mbedTLS
        SubGHz_Phy
Projects
    B-L072Z-LRWAN1
        Applications
            LoRaWAN
                LoRaWAN_AT_Slave
                LoRaWAN_End_Node
            SubGHz_Phy
                SubGHz_Phy_PingPong
    NUCLEO-L053R8
        Applications
            LoRaWAN
                LoRaWAN_AT_Master
                LoRaWAN_End_Node
            SubGHz_Phy
    NUCLEO-L073RZ
    NUCLEO-L152RE
    NUCLEO-L476RG
        Applications
            LoRaWAN
            LoRaWAN_FUOTA
            SubGHz_Phy
Utilities
```

The I-CUBE-LRWAN Expansion Package contains five applications: `LoRaWAN_AT_Master`, `LoRaWAN_End_Node`, `LoRaWAN_AT_Slave`, `SubGHz_Phy_PingPong`, and `LoRaWAN_FUOTA` (Only supported on NUCLEO-L476RG). For each application, three toolchains are available: IAR Systems® IAR Embedded Workbench®, Keil® MDK-ARM, and STMicroelectronics STM32CubeIDE.

6.4 End_Node application

This application reads the temperature, humidity, and atmospheric pressure from the sensors through the I^2C. The MCU measures the supplied voltage through V_{REFLNT} to calculate the battery level. These four data (temperature, humidity, atmospheric pressure, and battery level) are sent periodically to the LoRaWAN® network using the LoRa® radio in class-A at 868 MHz.

To launch the LoRaWAN® End_Node project, the user must go to `\Projects\<target>\Applications` and choose his favorite toolchain folder in the IDE environment. The user selects then the LoRaWAN® project from the proper target board.

6.4.1 Activation methods and keys

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

The `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\lora_app.h` file gathers all the data related to the device activation. The chosen method, along with the commissioning data, located in the `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\LoRaWAN\App\se-identity.h` file, is printed on the Virtual COM port and visible on a terminal.

6.4.2 Debug switch

The user must edit the `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\Core\Inc\sys_conf.h` file to activate the debug or trace mode:

```
#define DEBUGGER_ENABLED 1
```

The debug mode enables the `DBG_GPIO_SET` and the `DBG_GPIO_RST` macros as well as the debugger mode, even when the MCU goes in low-power. For trace mode, three levels of tracing are proposed:

```
#define VERBOSE_LEVEL     VLEVEL_M
```

- VLEVEL_L 1: traces disabled
- VLEVEL_M 2: enabled for functional traces
- VLEVEL_H 3: enabled for Debug traces

The user must edit `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\Core\Inc\utilities_conf.h` to select the trace level.

Note: To enable a true low-power, `#define DEBUGGER_ENABLED` mentioned above must be set to 0.

6.4.3 Sensor switch

When no sensor expansion board is plugged on the set-up, `#define SENSOR_ENABLED` must be set to 0 in `\Projects\<target>\Applications\LoRaWAN\LoRaWAN_End_Node\Core\Inc\utilities_conf.h`.

Table 44 provides a summary of the main options for the application configuration.

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 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.	
	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.	

Note: The maximum allowed payload length depends on both the region and the selected data-rate, so the payload format must be carefully designed according to these parameters.

6.5 PingPong application description

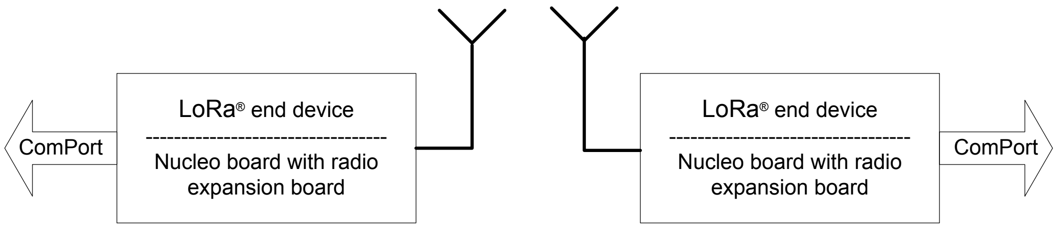
This application is a simple Rx/Tx RF link between two LoRa® end devices. By default, each LoRa® end device starts as a master, transmits a `Ping` message, and waits for an answer. The first LoRa® end device receiving a `Ping` message becomes a slave and answers the master with a *Pong* message. The *PingPong* is then started.

To launch the *PingPong* project, the user must go to the `Projects\NUCLEO-L053R8\Applications\SubGHz_Phy\SubGHz_Phy_PingPong` folder and follow the same procedure as for the LoRaWAN® End_Node project to launch the preferred toolchain.

Hardware and software set-up environment

To set up the Nucleo board, connect it or the B-L072Z-LRWAN1 board to the computer with a Type-A to Mini-B USB cable to the CN1 ST-LINK connector. Ensure that the CN2 ST-LINK connector jumpers are ON. Refer to Figure 21 for a representation of the *PingPong* setup.

Figure 21. PingPong setup


ComPort
LoRa® end device
Nucleo board with radio expansion board
LoRa® end device
Nucleo board with radio expansion board
ComPort

6.6 AT_Slave application description

The purpose of this example is to implement a LoRa® modem controlled through 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 B-L072Z-LRWAN1 Discovery kit embedding the CMWX1ZZABZ-091 LoRa® module. This application uses the STM32Cube low-layer drivers APIs targeting the STM32L072CZ to optimize the code size.

The 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 mode unless it processes an AT command from the external host.

To launch the AT_Slave project, the user must go to the folder `\Projects\B-L072Z-LRWAN1\Applications\LoRaWAN\LoRaWAN_AT_Slave` and follow the same procedure as for the `LoRaWAN_End_Node` project to launch the preferred toolchain.

The application note [9] gives the list of AT commands and their description.

6.7 AT_Master application description

This application reads sensor data and sends them to a LoRaWAN® network through an external LoRa® modem. The AT_Master application implements a complete set of AT commands to drive the LoRaWAN® stack that is embedded in the external LoRa® modem.

The external LoRa® modem targets the B-L072Z-LRWAN1 Discovery kit, the I-NUCLEO-LRWAN1 board (based on the WM-SG-SM-42 USI module [14]) or the LRWAN-NS1 expansion board featuring the RiSiNGHF modem [15] available in P-NUCLEO-LRWAN2, P-NUCLEO-LRWAN3, and NUCLEO-WL55JCx (High band x=1 / low band x=2 [13]).

This application uses the STM32Cube HAL drivers APIs targeting the STM32L0 Series.

BSP programming guidelines

Table 45 describes the BSP driver APIs to interface with the external LoRa® module.

Table 45. **System-time functions**

Function	Description
`ATEerror_t Modem_IO_Init (void)`	Modem initialization
`void Modem_IO_DeInit (void)`	Modem deinitialization
`ATEerror_t Modem_AT_Cmd (ATGroup_t, at_group, ATCmd_t Cmd, void *pdata)`	Modem I/O commands

Note: The Nucleo board communicates with the expansion board via the UART (PA2, PA3). The following modifications must be applied (Refer to section 6.8 of [12]):

- *SB62 and SB63 must be ON.*
- *SB13 and SB14 must be OFF to disconnect the UART from ST-LINK.*

6.8 FUOTA application description

The purpose of this application is to implement the firmware update over-the-air (FUOTA) feature. It provides a way to manage the firmware update over the LoRaWAN® protocol.

This application is based on the LoRaWAN® recommendations version V1.0.3 and the three application packages specification V1.0, Clock Synchronization, Fragmented Data Block Transport, and Remote Multicast Setup [1].

This application is made up of Secure Boot and Secure Firmware Update (SBSFU), LoRaWAN® protocol stack, and User Application [2].

This application only targets the SMT32L476RG microcontroller.

The application note [8] gives all the needed information to make use of the FUOTA I-CUBE-LRWAN part.

7 System performances

7.1 Memory footprints

The values in Table 46 are measured for the following IAR Embedded Workbench® EWARM 8.32 compiler configuration:

- Optimization: Optimized for the high size level
- Debug option: OFF
- Trace option: OFF
- Target STM32L073 with SX1272MB2DAS

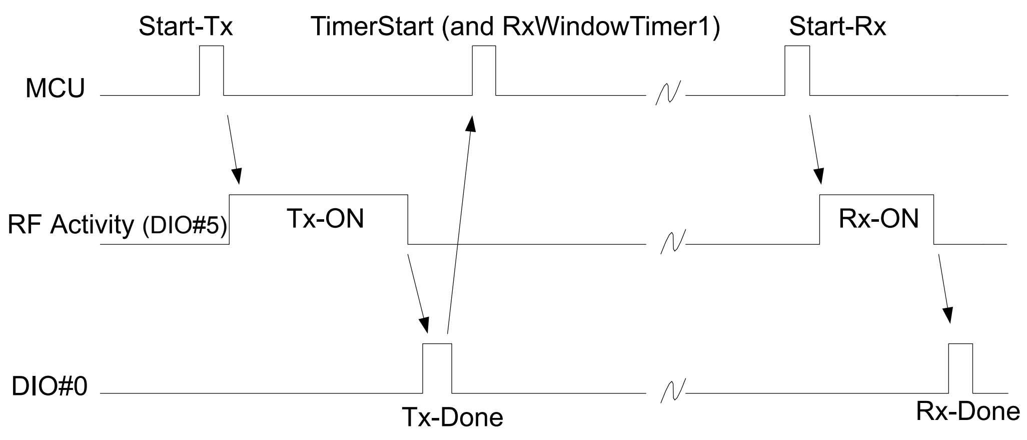
Table 46. **Memory footprint values for End_Node application**

Project	Flash (bytes)	RAM (bytes)	Description
Application layer	5363	874	Includes all microlib.
LoRaWAN® stack	37267	4582	Includes MAC + RF driver.
HAL	11908	0	-
Utilities	3058	1732	Includes services like system, timeserver, sequencer, trace, and low power.
Total application	57550	7188	Memory footprint for the overall application

7.2 Real-time constraints

The LoRa® RF asynchronous protocol implies following a strict Tx/Rx timing recommendation (Refer to Figure 22. Tx/Rx time diagram for a Tx/Rx diagram example). The SX1276MB1MAS expansion board is optimized for user-transparent low-lock time and fast auto-calibrating operation. The LoRaWAN® Expansion Package design integrates the transmitter startup-time and the receiver startup-time constraints.

Figure 22. **Tx/Rx time diagram**



Rx window channel start

The Rx window opens the RECEIVE_DELAY1 for 1 s (± 20 µs) or the JOIN_ACCEPT_DELAY1 for 5 s (± 20 µs) 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 DI0#n interrupt (Refer to Table 43) to avoid stalling the received startup time.

7.3 Power consumption

The power-consumption measurement is done for the Nucleo boards associated with the SX1276MB1MAS shield.

Measurements setup

- No DEBUGGER_ENABLED
- No TRACE
- No SENSOR_ENABLED

Measurement results

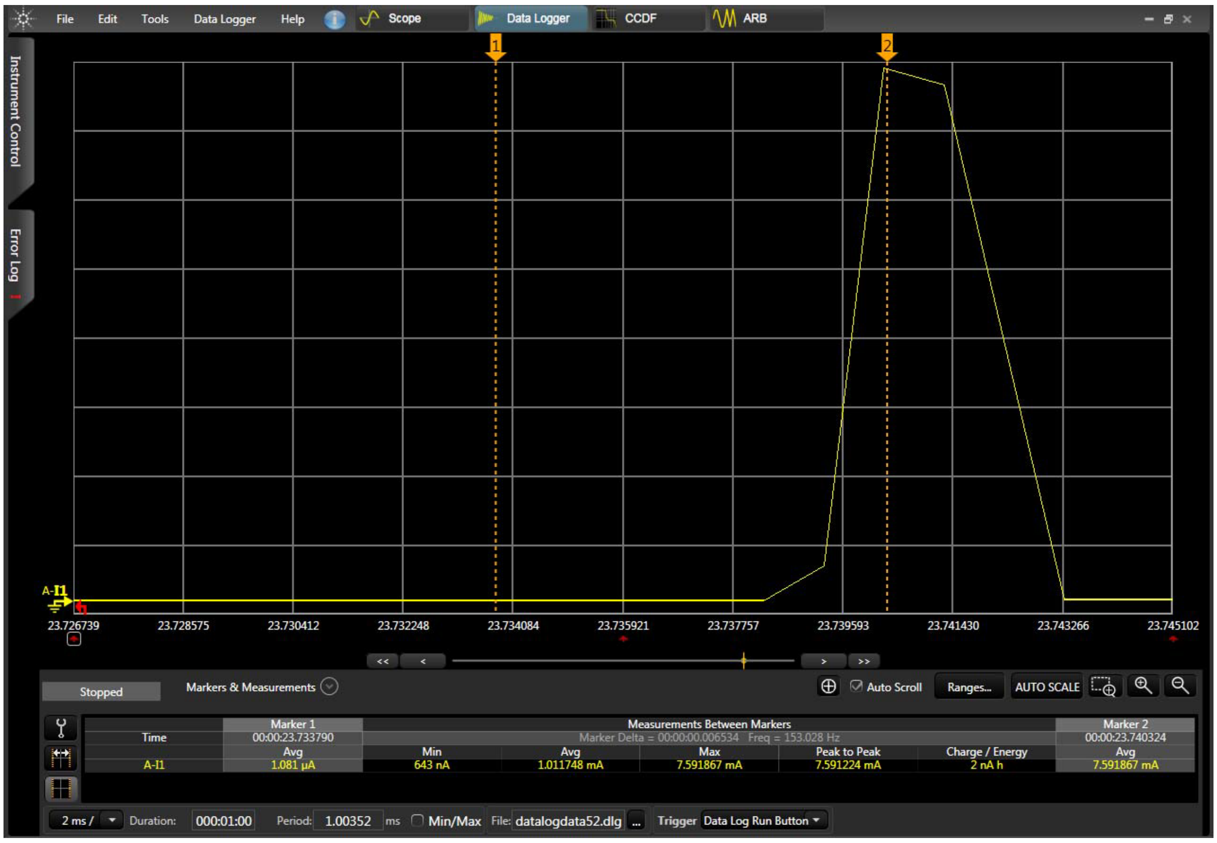
- Typical consumption in stop mode: 1.3 µA
- Typical consumption in run mode: 8.0 mA

Measurements figures

- Instantaneous consumption over 30 s

Figure 23 shows an example of the current consumption against time on an STM32L0 Series microcontroller.

Figure 23. STM32L0 current consumption against time



	Marker 1	Measurements Between Markers					Marker 2
Time	00:00:23.733790	Marker Delta = 00:00:00.006534 Freq = 153.028 Hz					00:00:23.740324
	Avg	Min	Avg	Max	Peak to Peak	Charge / Energy	Avg
A-I1	1.081 µA	643 nA	1.011748 mA	7.591867 mA	7.591224 mA	2 nA h	7.591867 mA

2 ms / Duration: 000:01:00 Period: 1.00352 ms Min/Max File: datalogdata52.dlg Trigger Data Log Run Button

Revision history

Table 47. **Document revision history**

Date	Version	Changes
27-Jun-2016	1	Initial release.
10-Nov-2016	2	Updated: • *–Introduction* • *Section 2.1: Overview* • *Section 3.2: Features* • *Section 5: Example description* • *Section 6: System performances*
4-Jan-2017	3	Updated: • *Introduction* concerning the CMWX1ZZABZ-xxx LoRa® module (Murata). • *Section 5.1: Hardware description*: 3rd hardware configuration file added. • *Section 5.2: Package description*: AT_Slave application added. Added: • *Section5.5: AT_Slave application description*
21-Feb-2017	4	Updated: • *Introduction* with I-NUCLEO-LRWAN1 LoRa® expansion board • *Figure 10: Project files structure* • *Section 5.1: Single MCU end-device hardware description* • *Figure 15: I-CUBE-LRWAN structure* • *Section 5.4: End_Node application* • *Section Table 27.: Switch options for the application's configuration* • *Section 5.5: PingPong application description* • *Section 5.6: AT_Slave application description* • *Table 29: Memory footprint values for End_Node application* Added: • *–Section 5.2: Split end-device hardware description (two-MCUs solution)* • *Section 5.7Section 5.7: AT_Master application description*
18-Jul-2017	5	Added: • Note to *Section 5.4:End_Node application* on maximum payload length allowed • Note to *Section 5.7:AT_Master application description* on the Nucleo board communication with expansion board via UART
14-Dec-2017	6	Added: • New modem reference: expansion board featuring the RiSiNGHF®modem RHF0M003 Updated: • New architecture design (LoRa® FSM removed) • *Figure 10: Project files structure* • *Figure 13: Operation model*
4-Jul-2018	7	Added: • New expansion boards • Introduction of LoRaWAN® class-B mode Updated: • *Figure 10* to *Figure17*, *Table 4*, and *Table 10* to *Table 45*

Date	Version	Changes
13-Dec-2018	8	Removed: • Class B restriction regarding AT commands in *Section 5.6: AT_Slave application description*
9-Jul-2019	9	Updated: • P-NUCLEO-LPWAN2/3 in *Introduction* and *Section 5.7:AT_Master application description* • Added *Section 2.4.3:End-device class B mode establishment*
4-Nov-2019	10	Added: • FUOTA and SBSFU acronyms in *Table1* • LoRa Alliance® and application notes references in *Section 1.2* • New *Section 5.8: FUOTA application description*
19-Feb-2021	11	Updated: • Title • *Table 2*, *Table 46*, and *Table 48* • *Figure 4*, *Figure 11*, *Figure 13*, and *Figure 14* • Package content in *Section 3.1* • Regions added to *Section 3.2* • Functions in *Section 4.6* and *Section 4.7* Added: • *Section 4.8 Extended application functions* • *Section 5 Utilities description* Removed: • *Middleware utility functions*
30-Sep-2021	12	Updated: • Section 5 Utilities description and Table 44 both aligned with corresponding parts in the application note *How to build a LoRa® application with STM32CubeWL* (AN5406) • Table 46. Memory footprint values for End_Node application

Contents

List of tables

List of figures