UM2073
User manual

STM32 LoRa® Expansion Package for STM32Cube

Introduction

This user manual describes the I-CUBE-LRWAN LoRa® Expansion Package implementation on the STM32L0, STM32L1 and STM32L4 Series. 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 the interoperability with the LoRa network. The LoRa 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 (see *Section 2*).

This user manual provides customer application 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
- 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 (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

Contents

1 **Overview** 7

1.1 Acronyms and abbreviations 7

1.2 References 8

2 **LoRa standard overview** 9

2.1 Overview 9

2.2 Network architecture 10

2.2.1 End-device architecture 10

2.2.2 End-device classes 10

2.2.3 End-device activation (joining) 12

2.2.4 Regional spectrum allocation 12

2.3 Network layer 13

2.3.1 Physical layer (PHY) 13

2.3.2 MAC sublayer 14

2.4 Message flow 14

2.4.1 End-device activation details (joining) 14

2.4.2 End-device data communication (Class A) 14

2.4.3 End-device class B mode establishment 17

2.5 Data flow 18

3 **I-CUBE-LRWAN middleware description** 19

3.1 Overview 19

3.2 Features 21

3.3 Architecture 22

3.4 Hardware related components 23

3.4.1 Radio reset 23

3.4.2 SPI 23

3.4.3 RTC 23

3.4.4 Input lines 24

4 **I-CUBE-LRWAN middleware programming guidelines** 25

4.1 Middleware initialization 25

4.2 Middleware MAC layer functions 25

4.2.1 MCPS services . . . 25
4.2.2 MLME services . . . 26
4.2.3 MIB services . . . 26
4.3 Middleware MAC layer callbacks . . . 26
4.3.1 MCPS . . . 26
4.3.2 MLME . . . 27
4.3.3 MIB . . . 27
4.3.4 Battery level . . . 27
4.4 Middleware MAC layer timers . . . 27
4.4.1 Delay Rx window . . . 27
4.4.2 Delay for Tx frame transmission . . . 27
4.4.3 Delay for Rx frame . . . 28
4.5 Middleware utility functions . . . 28
4.5.1 Timer server APIs description . . . 28
4.5.2 Low-power functions . . . 28
4.5.3 System time functions . . . 29
4.5.4 Trace functions . . . 30
4.5.5 Queuing functions . . . 31
4.6 Emulated secure-element . . . 32
4.7 Middleware End_Node application function . . . 33
4.7.1 LoRa End_Node initialization . . . 37
4.7.2 LoRa End_Node Join request entry point . . . 38
4.7.3 LoRa End-Node start Tx . . . 38
4.7.4 Request End-Node Join Status . . . 38
4.7.5 Send an uplink frame . . . 38
4.7.6 Request the current network time . . . 38
4.7.7 Request the next beacon timing . . . 39
4.7.8 Switch class request . . . 39
4.7.9 Get End-device current class . . . 39
4.7.10 Request beacon acquisition . . . 39
4.7.11 Send unicast ping slot info periodicity . . . 39
4.8 LIB End_Node application callbacks . . . 40
4.8.1 Current battery level . . . 40
4.8.2 Current temperature level . . . 40
4.8.3 Board unique ID . . . 40
4.8.4 Board random seed . . . 40

4.8.5 Make Rx frame . . . 40
4.8.6 Request class mode switching . . . 40
4.8.7 End_Node class mode change confirmation . . . 41
4.8.8 Send a dummy uplink frame . . . 41

5 Example description . . . 42
5.1 Single MCU end-device hardware description . . . 42
5.2 Split end-device hardware description (two-MCUs solution) . . . 43
5.3 Package description . . . 45
5.4 End_Node application . . . 46
5.4.1 Activation methods and keys . . . 46
5.4.2 Debug switch . . . 46
5.4.3 Sensor switch . . . 47
5.5 PingPong application description . . . 47
5.6 AT_Slave application description . . . 48
5.7 AT_Master application description . . . 48
5.8 FUOTA application description . . . 49

6 System performances . . . 50
6.1 Memory footprints . . . 50
6.2 Real-time constraints . . . 50
6.3 Power consumption . . . 51

7 Revision history . . . 53

List of tables

Table 1. List of acronyms and abbreviations . . . 7
Table 2. LoRa classes intended usage . . . 9
Table 3. LoRaWAN regional spectrum allocation . . . 12
Table 4. Middleware initialization function . . . 25
Table 5. MCPS services function . . . 25
Table 6. MLME services function . . . 26
Table 7. MIB services functions . . . 26
Table 8. MCPS primitives . . . 26
Table 9. MLME primitive . . . 27
Table 10. Battery level function . . . 27
Table 11. Delay Rx functions . . . 27
Table 12. Delay for Tx frame transmission function . . . 27
Table 13. Delay for Rx frame function . . . 28
Table 14. Timer server functions . . . 28
Table 15. Low-power functions . . . 28
Table 16. System time functions . . . 29
Table 17. Trace functions . . . 30
Table 18. Middleware queuing functions . . . 31
Table 19. Secure-element functions . . . 32
Table 20. LoRa class A initialization function . . . 37
Table 21. LoRa End_Node Join request entry point . . . 38
Table 22. LoRa End-Node start Tx . . . 38
Table 23. End-Node Join status . . . 38
Table 24. Send an uplink frame . . . 38
Table 25. Current network time . . . 38
Table 26. Next beacon timing . . . 39
Table 27. Switch class request . . . 39
Table 28. Get End-Device current class . . . 39
Table 29. Request beacon acquisition . . . 39
Table 30. Unicast ping slot periodicity . . . 39
Table 31. Current battery level function . . . 40
Table 32. Current Temperature function . . . 40
Table 33. Board unique ID function . . . 40
Table 34. Board random seed function . . . 40
Table 35. Make Rx frame . . . 40
Table 36. LoRa HasJoined function . . . 40
Table 37. End_Node class mode change confirmation function . . . 41
Table 38. Send a dummy uplink frame . . . 41
Table 39. Nucleo-based supported hardware . . . 42
Table 40. LoRa radio expansion board characteristics . . . 42
Table 41. STM32L0xx IRQ priorities . . . 43
Table 42. Switch options for the application's configuration . . . 47
Table 43. BSP programming guidelines . . . 49
Table 44. Memory footprint values for End_Node application . . . 50
Table 45. Document revision history . . . 53

List of figures

Figure 1. Network diagram . . . 10
Figure 2. TX/Rx time diagram (Class A) . . . 11
Figure 3. Tx/Rx time diagram (Class B) . . . 11
Figure 4. Tx/Rx time diagram (Class C) . . . 12
Figure 5. LoRaWAN layers . . . 13
Figure 6. Message sequence chart for joining (MLME primitives) . . . 14
Figure 7. Message sequence chart for confirmed-data (MCPS primitives) . . . 15
Figure 8. Message sequence chart for unconfirmed-data (MCPS primitives) . . . 16
Figure 9. MSC MCPS class B primitives . . . 17
Figure 10. Data flow . . . 18
Figure 11. Project files structure . . . 20
Figure 12. Main design of the firmware . . . 22
Figure 13. LoRaMacCrypto module design . . . 32
Figure 14. Operation model . . . 34
Figure 15. LoRa state behavior . . . 35
Figure 16. LoRa class B system state behavior . . . 37
Figure 17. Concept for split end-device solution . . . 44
Figure 18. I-CUBE-LRWAN structure . . . 45
Figure 19. PingPong setup . . . 48
Figure 20. Rx/Tx time diagram . . . 50
Figure 21. STM32L0 current consumption against time . . . 52

1 Overview

The I-CUBE-LRWAN Expansion Package for STM32Cube runs on STM32 32-bit microcontrollers based on the Arm®(a) Cortex®-M processor.

1.1 Acronyms and abbreviations

Table 1. List of acronyms and abbreviations

Term	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-this-air activation
PLME	Physical sublayer management entity
PPDU	Physical protocol data unit
SAP	Service access point
SBSFU	Secure boot, secure firmware update

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

1.2 References

- LoRa Alliance specification protocol named LoRaWAN version V1.0.3 - 2018, March - Final - Released
- IEEE Std 802.15.4TM - 2011. Low-Rate Wireless Personal Area Networks (LR-WPANs)
- LoRaWAN version 1.1 Regional Parameters - RevB - 2018, January - Released
- LoRa Alliance Fragmented Data Block Transport over LoRaWAN Specification v1.0.0 – 2018, September – [TS-004]
- LoRa Alliance Remote Multicast Setup over LoRaWAN Specification v1.0.0 -2018, September – [TS-005]
- LoRa Alliance Application layer clock synchronization over LoRaWAN Specification v1.0.0 –2018, September – [TS-003]
- Application note *Integration Guide for the X-CUBE-SBSFU STM32Cube Expansion Package* – AN5056
- Application note *FUOTA I-CUBE-LRWAN* – AN5411

2 LoRa standard overview

2.1 Overview

This section provides a general overview of the LoRa and LoRaWAN recommendations, focusing in particular 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 LoRa Expansion Package is compliant with the LoRa Alliance specification protocol named LoRaWAN.

The table below shows the LoRa class usage definition. Refer to *Section 2.2.2* for further details on these classes.

Table 2. LoRa classes intended usage

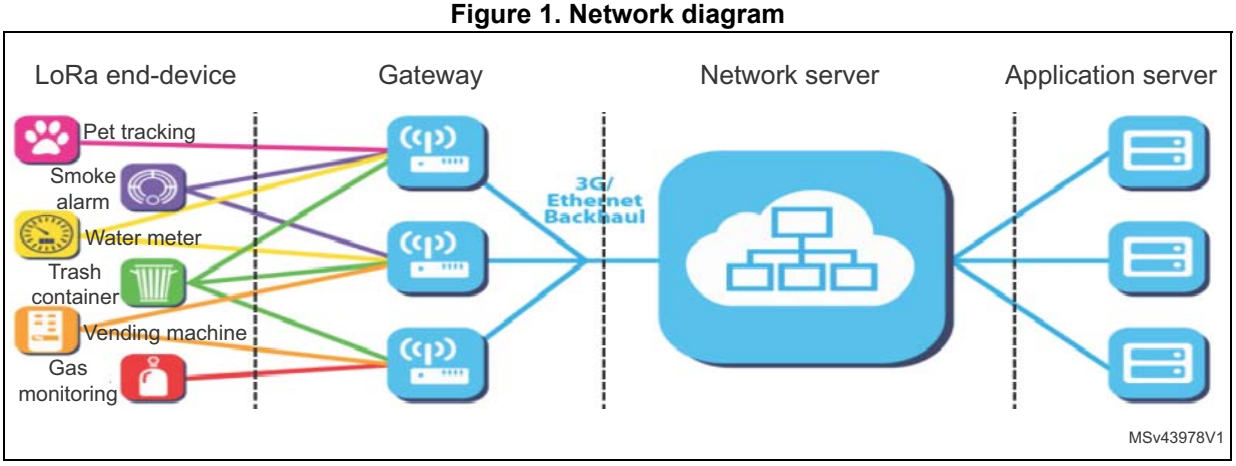
Class name	Intended usage
A - All	– Battery-powered sensors or actuators with no latency constraint – Most energy-efficient communication class – Must be supported by all devices
B - Beacon	– Battery-powered actuators – Energy-efficient communication class for latency controlled downlink – Based on slotted 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 LoRa link to one gateway as shown in the figure below.

Figure 1. Network diagram



2.2.1 End-device architecture

The end-device is made 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.

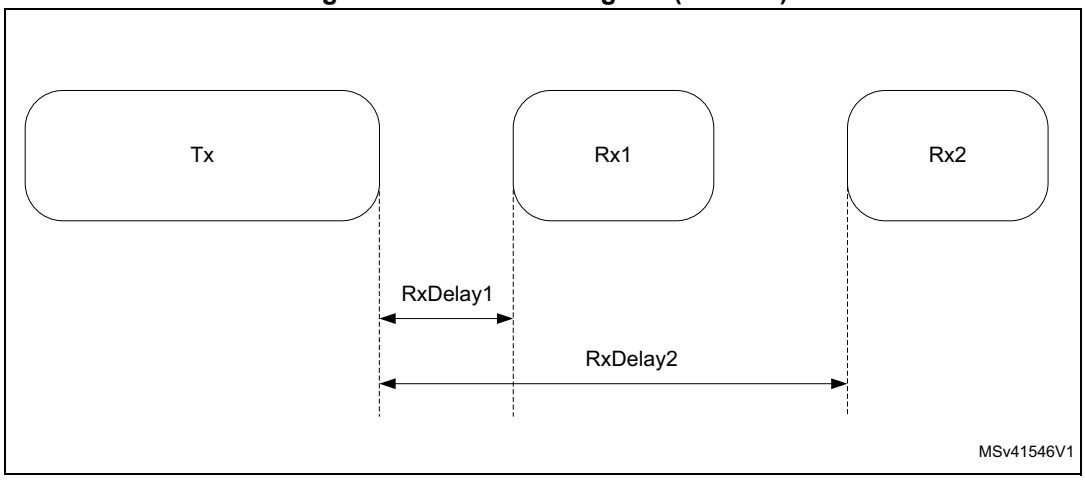
2.2.2 End-device classes

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

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

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

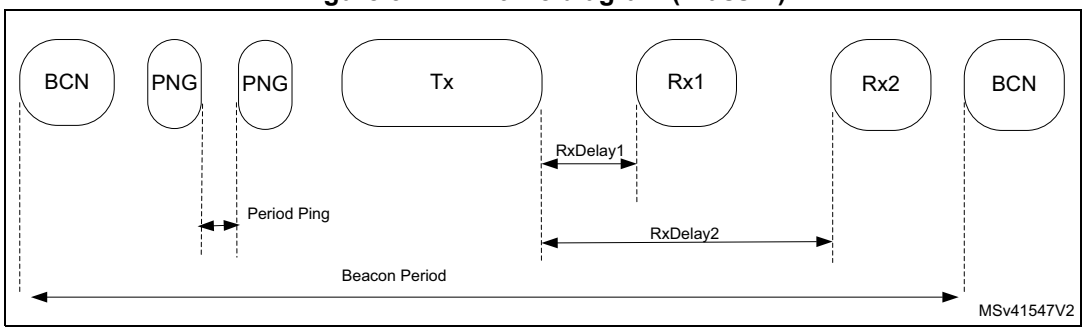
Figure 2. TX/Rx time diagram (Class A)



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

- Mid power consumption
- Class B devices open extra receive windows at scheduled times (see *Figure 3*).
- In order for the end-device to open the receive window at the scheduled time, the end-device receives a time-synchronized beacon from the gateway.

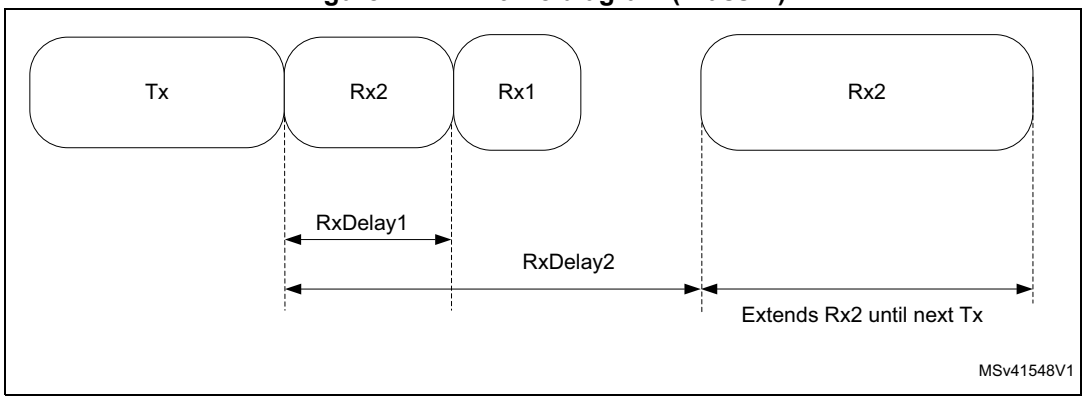
Figure 3. Tx/Rx time diagram (Class B)



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

- Large power consumption
- End-devices of class C have nearly continuously open receive windows, only closed when transmitting (see *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 LoRa end-device to participate in a LoRa network. Both the LoRa end-device and the application server share the same secret key known as AppKey. During a joining procedure, the LoRa 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 LoRa end-device that sends the data directly to the LoRa network.

2.2.4 Regional spectrum allocation

The LoRaWAN specification varies slightly from region to region. The European, North American and Asian markets have different spectrum allocations and regulatory requirements (see the table below for more details).

Table 3. LoRaWAN regional spectrum allocation

Region	Supported	Band (MHz)	Duty cycle (%)	Output power (dBm)
EU	Y	868	< 1	+14
EU	Y	433	< 1	+10
US	Y	915	< 2 (BW < 250 kHz) or < 4 (BW ≥ 250 kHz) Transmission slot < 0.4 s	+20

Table 3. LoRaWAN regional spectrum allocation (continued)

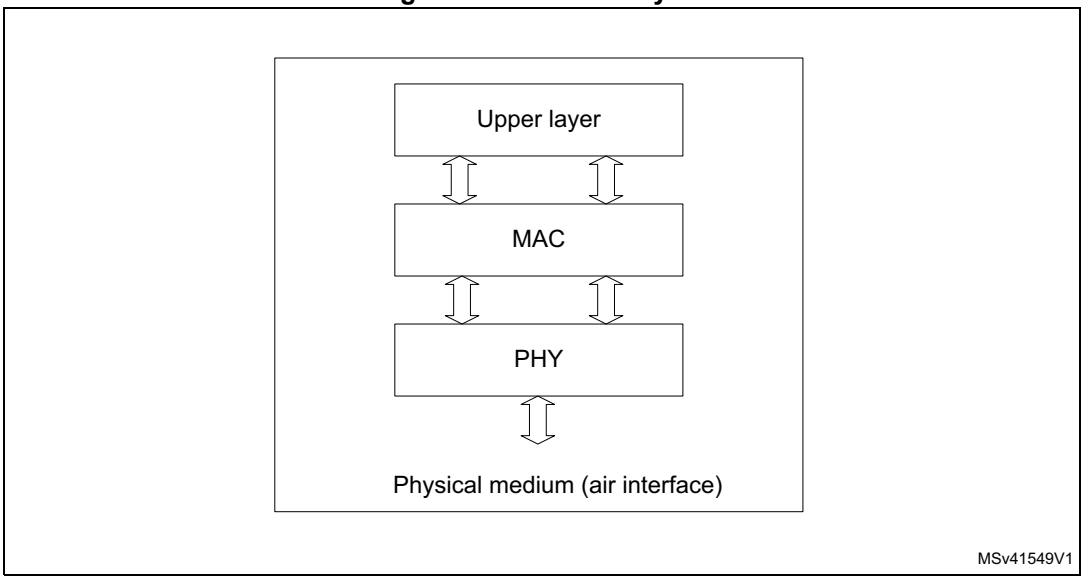
Region	Supported	Band (MHz)	Duty cycle (%)	Output power (dBm)
CN	N	779	< 0.1	+10
AS	Y	923	< 1	+16
IN	Y	865	No	+20
KR	Y	920	No	+10
RU	Y	868	< 1	+16

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 (see the figure below).

Figure 5. LoRaWAN layers



2.3.1 Physical layer (PHY)

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

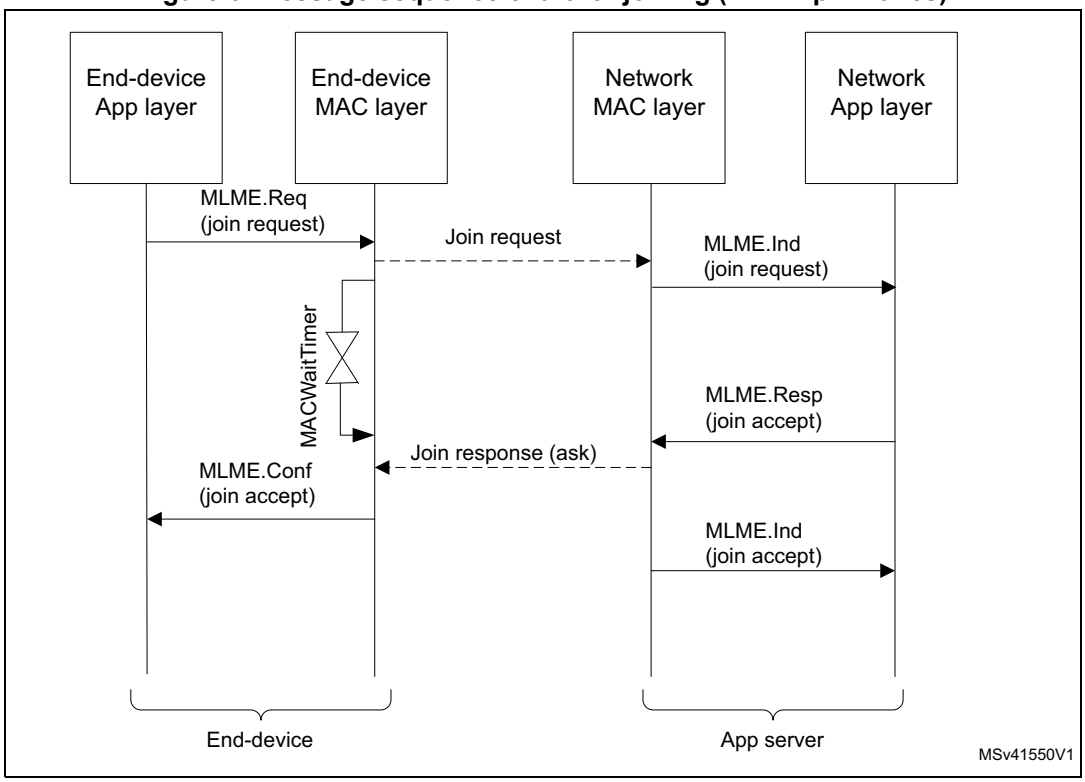
This section describes the information flow between the N-user and the N-layer. The request for service is done through a service primitive.

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 *Section 2.2.3*.

The message sequence chart (MSC) in the figure below shows the OTAA activation method.

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



2.4.2 End-device data communication (Class A)

The end-device transmits data by one of the following methods: through a confirmed-data message method (see *Figure 7*) or through an unconfirmed-data message (see *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 should wait during an acknowledgment duration (‘AckWaitDuration’) to receive the acknowledgment frame (Refer to *Section 4.3.1: MCPS*).

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)

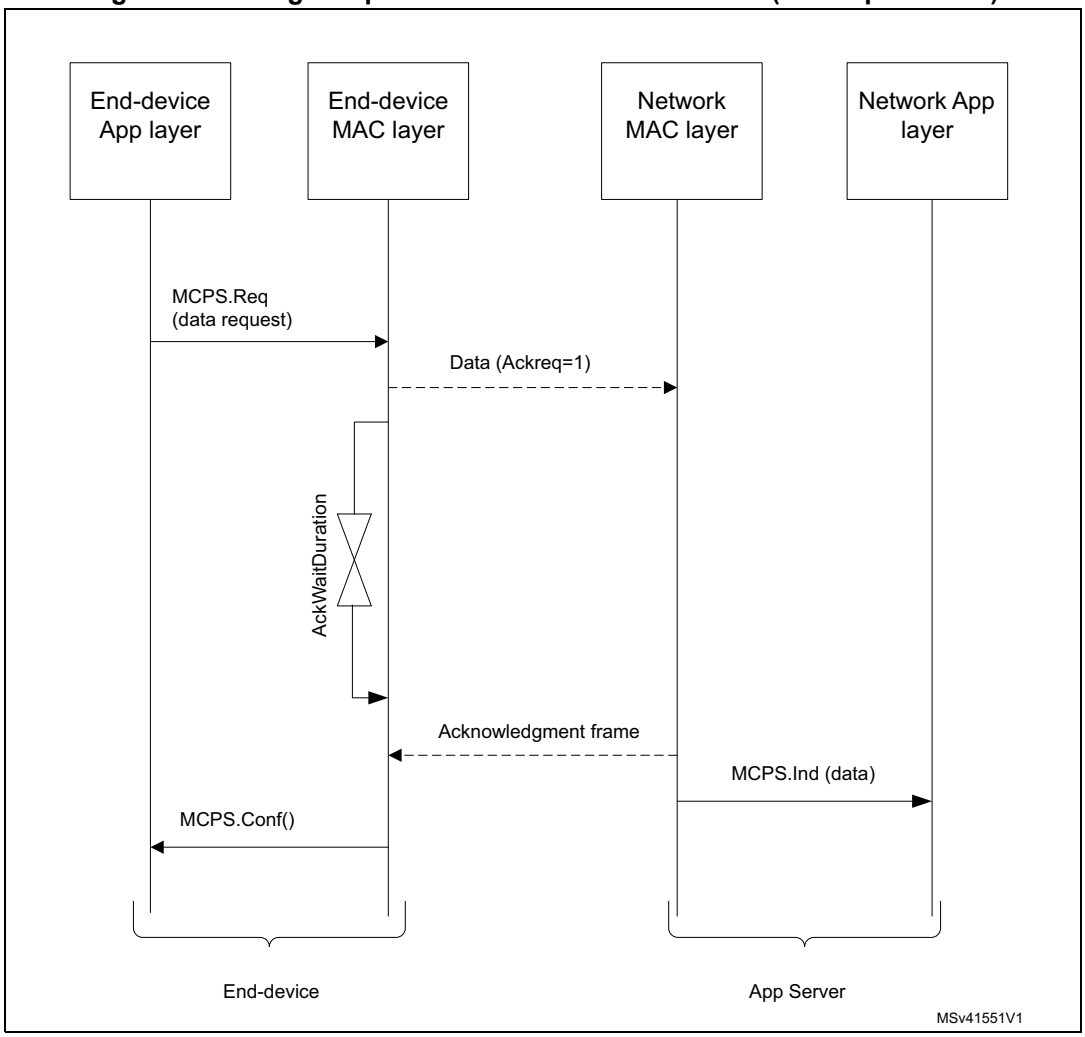
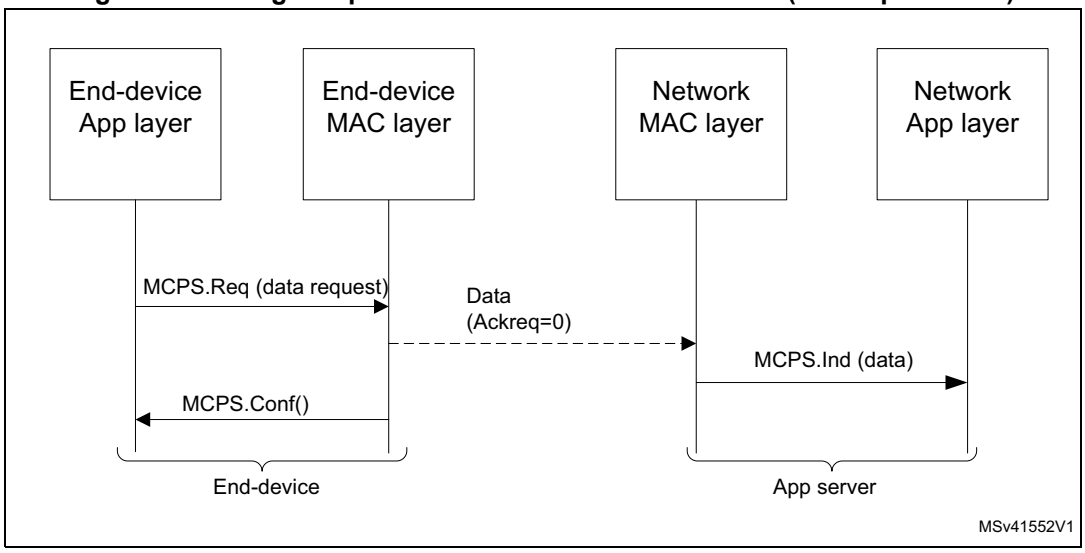


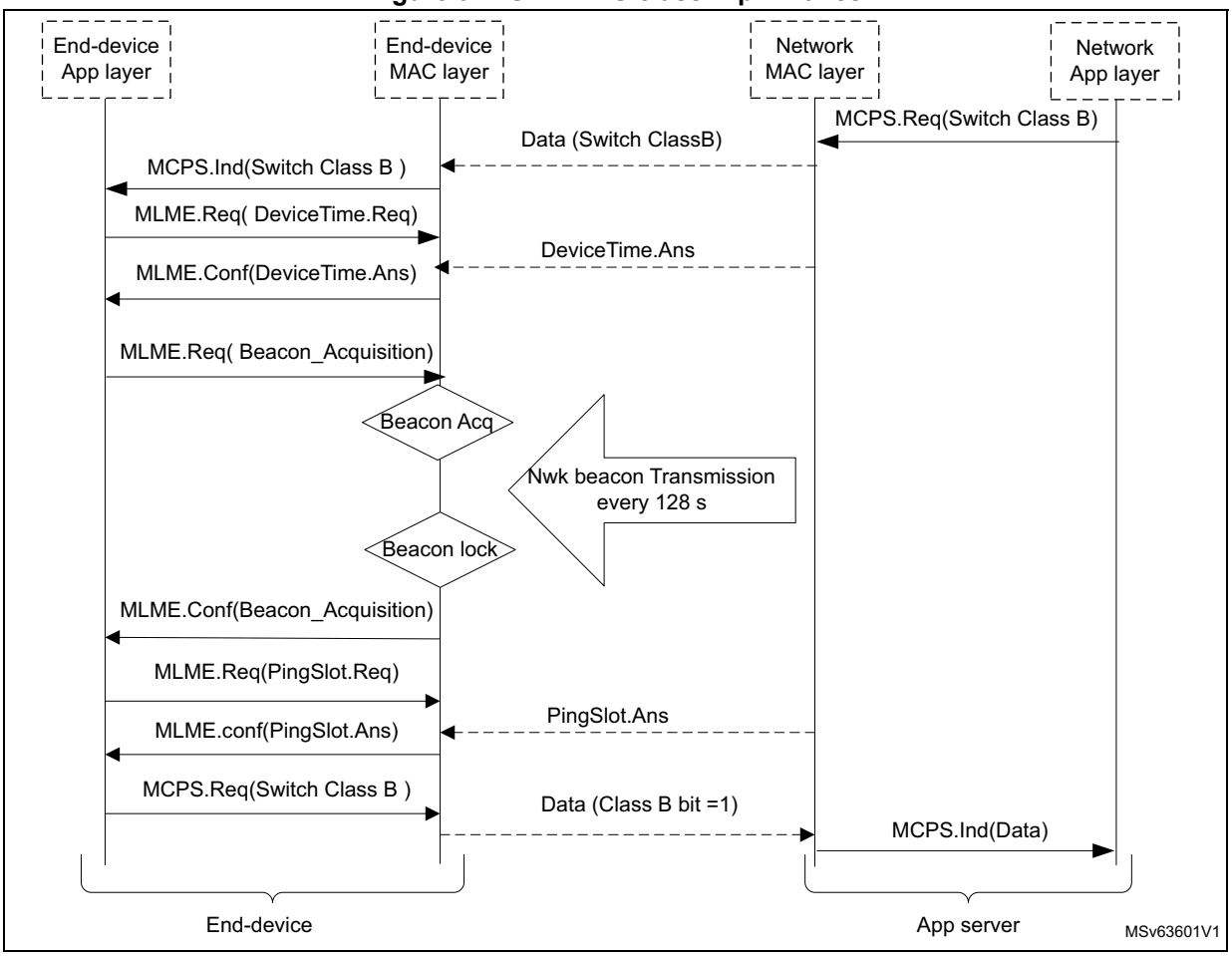
Figure 8. Message sequence chart for unconfirmed-data (MCPS primitives)



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 GW sending a beacon on a regular basis (128 s) to synchronize all the end-devices in the network so that the end-device can open a short Rx window called ‘ping slot’. The decision to switch from class A to class B always comes from the application layer.

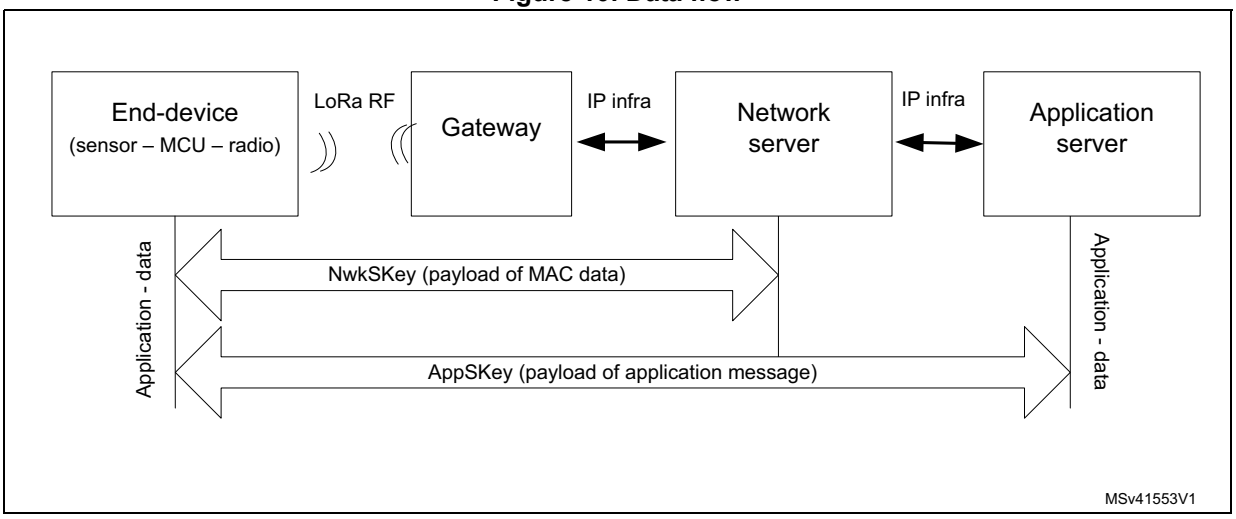
Figure 9. MSC MCPS class B primitives



2.5 Data flow

The data integrity is ensured by the network session key (NwkSKey) and the application session key (AppSKey). The NwkSKey is used to encrypt and decrypt the MAC payload data and the AppSKey is used to encrypt and decrypt the application payload data. See the figure below for the data flow representation.

Figure 10. Data flow



The NwkSKey is shared between the end-device and the network server. The NwkSKey provides message integrity for the communication and provides security for the end-device towards the network server communication.

The AppSKey is shared between the end-device and the application server. The AppSKey is used to encrypt/decrypt the application data. In other words, the AppSKey provides security for the application's payload. In this way, the application data sent by an end-device can not be interpreted by the network server.

3 I-CUBE-LRWAN middleware description

3.1 Overview

This I-CUBE-LRWAN package offers a LoRa stack middleware for STM32 microcontrollers. This middleware is split into several modules:

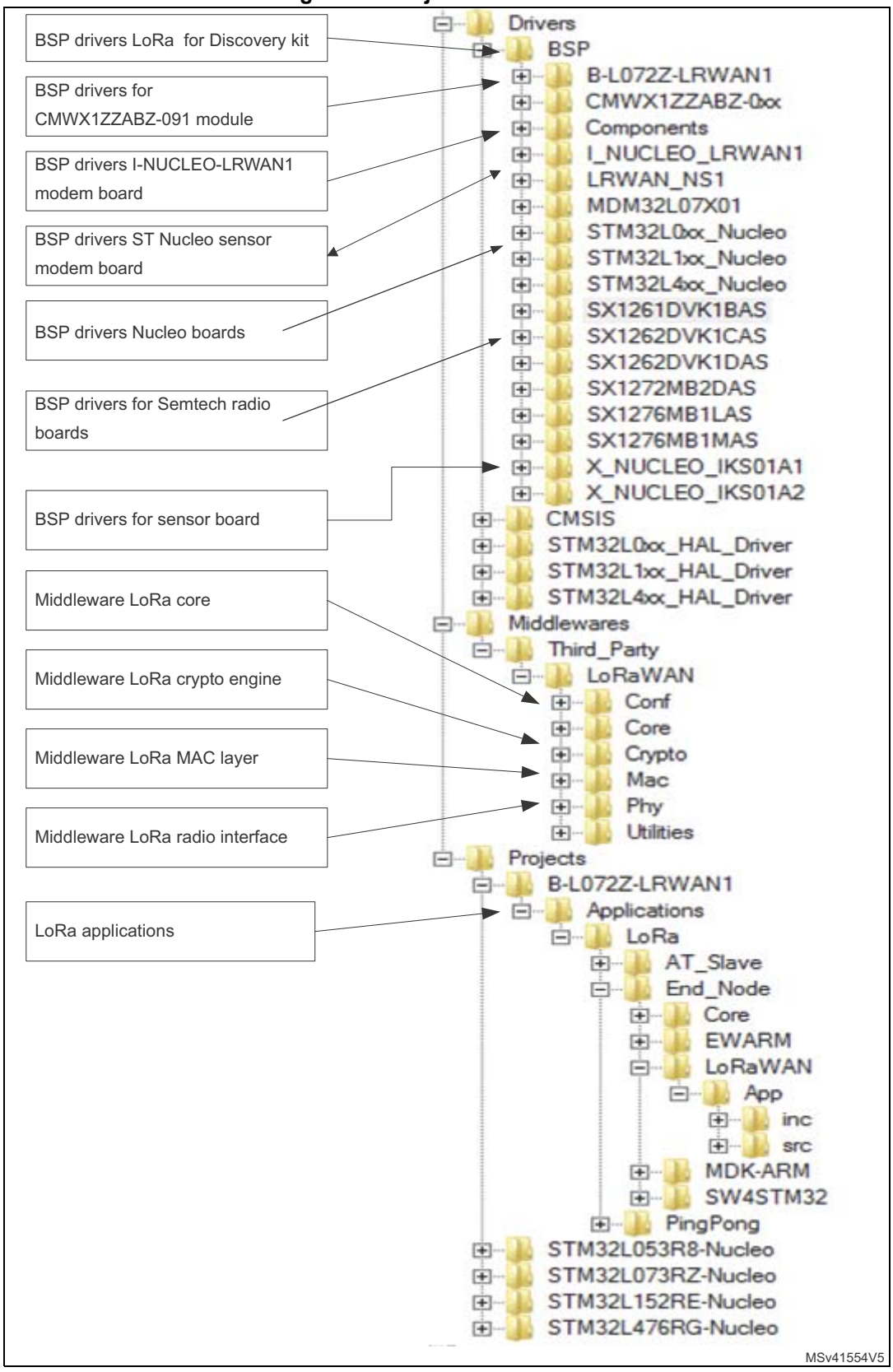
- LoRaMac layer module
- LoRa utility module
- LoRa crypto module
- LoRa core module

The LoRa core module implements a LoRa state machine coming on top of the LoRaMac layer. The LoRa 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 structure of the project files.

Figure 11. Project files structure



The I-CUBE-LRWAN package includes:

- The LoRa stack middleware:
  - LoRaWAN layer
  - LoRa utilities such as power, queue, system time, time server, and trace managements
  - LoRa software crypto engine
  - LoRa state machine
- Board support package:
  - Radio Semtech drivers
  - Sensor ST drivers
- STM32L0 HAL drivers
- LoRa main application example

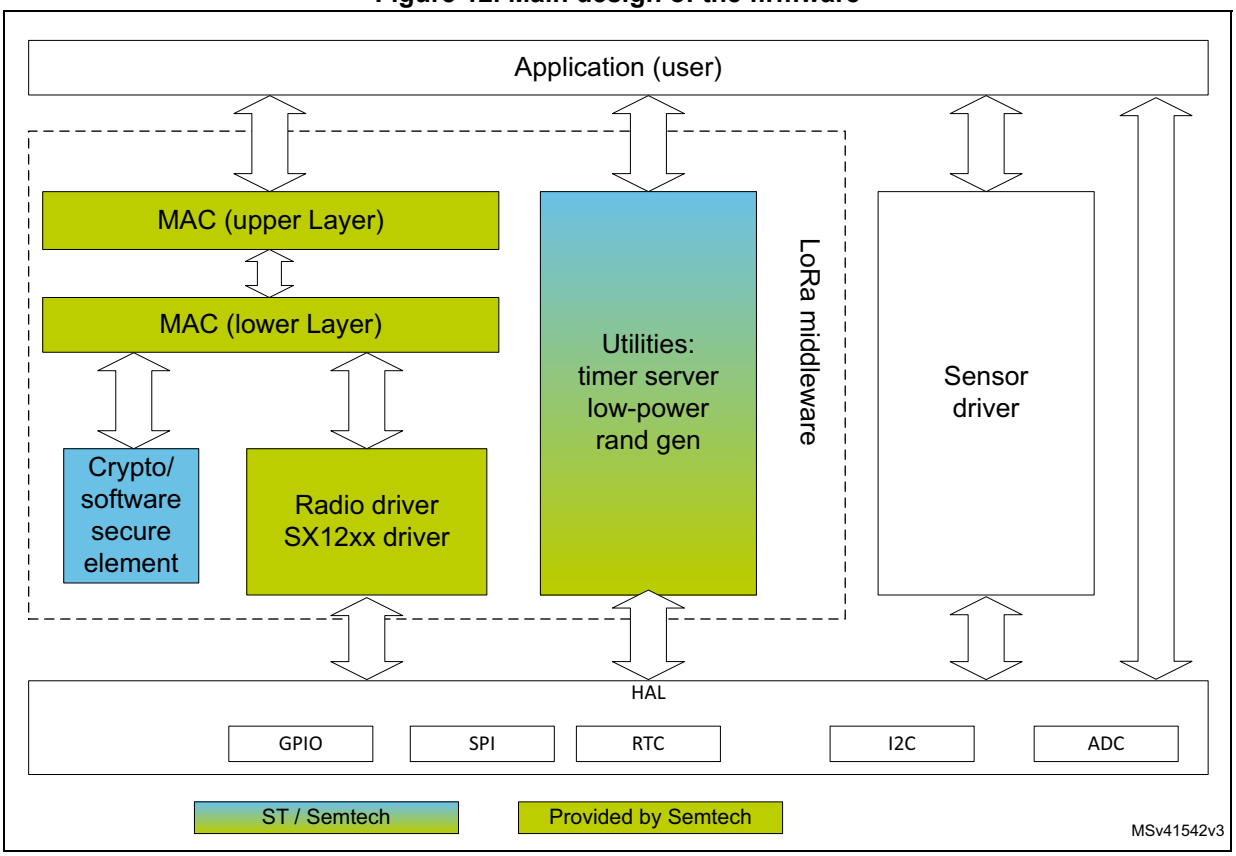
3.2 Features

- Compliant with the specification for the LoRa Alliance protocol named LoRaWAN
- On-board LoRaWAN class A, class B, and class C protocol stack
- EU 868MHz ISM band ETSI compliant
- EU 433MHz ISM band ETSI compliant
- US 915MHz ISM band FCC compliant
- KR 920Mhz ISM band defined by the Korean government
- RU 864Mhz ISM band defined by Russian regulation
- 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

The figure below describes the main design of the firmware for the I-CUBE-LRWAN application.

Figure 12. Main design of the firmware



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 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 (see *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 in 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 to 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, that controls the LoRa class A, is sensitive, an intermediate level of software is inserted (lora.c) between the MAC and the application (Refer to MAC's "upper layer" on *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 LoRa class A if any task must be done. This application also implements the 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 40: LoRa radio expansion board characteristics* and to *Section 5.1: Single MCU end-device hardware description*).

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 40* and to *Section 5.1*).

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 is around a 48 days period.

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 40* and to *Section 5.1*).

The DIO0 is used to signal that the LoRa radio successfully completed 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 Kbit/s.

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 can not 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 gives a description of the LoRaMac layer APIs. The proprietary PHY layer (see *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 (see the table below).

Table 4. Middleware initialization function

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

4.2 Middleware MAC layer functions

The provided APIs follow the definition of "primitive" defined in IEEE802.15.4-2011 (see *Section 1.2: References*).

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/indication are implemented using callbacks.

The LoRaMAC layer provides MCPS services, MLME services, and MIB services.

4.2.1 MCPS services

In general, the LoRaMAC layer uses the MCPS services for data transmissions and data receptions (see the table below).

Table 5. MCPS services function

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

4.2.2 MLME services

The LoRaMAC layer uses the MLME services to manage the LoRaWAN network (see the table below).

Table 6. 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, MIB_NET_ID) and holds the configuration of the LoRaMAC layer (for example the MIB_ADR, MIB_APP_KEY). The provided APIs are presented in the table below.

Table 7. MIB services functions

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 LoRaMAC layer uses the MCPS services for data transmission and data reception (see the table below).

Table 8. MCPS primitives

Function	Description
`void (*MacMcpsConfirm) (McpsConfirm_t *McpsConfirm)`	Event function primitive for the called callback to be implemented by the application. Response to a McpsRequest
`Void (*MacMcpsIndication) (McpsIndication_t *McpsIndication)`	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 LoRaMAC layer uses the MLME services to manage the LoRaWAN network (see the table below).

Table 9. 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 functions.

4.3.4 Battery level

The LoRaMAC layer needs a battery-level measuring service (see the table below).

Table 10. Battery level function

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

4.4 Middleware MAC layer timers

4.4.1 Delay Rx window

Refer to *Section 2.2.2: End-device classes*. See the table below for the delay Rx functions.

Table 11. Delay Rx 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 12. Delay for Tx frame transmission function

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

4.4.3 Delay for Rx frame

Table 13. Delay for Rx frame function

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

4.5 Middleware utility functions

4.5.1 Timer server APIs description

A timer server is provided so that the user can 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 LoRa stack. The user can request as many timers as the application requires.

Four APIs are provided as shown in the table below.

Table 14. Timer server functions

Function	Description
`void TimerInit (TimerEvent_t *obj, void (*callback) (void))`	Initialize the timer and associate a callback function when timer elapses
`void TimerSetValue (TimerEvent_t *obj, uint32_t value)`	Set the timer a timeout value on milliseconds
`void TimerStart (TimerEvent_t *obj)`	Start the timer
`void TimerStop (TimerEvent_t *obj)`	Stop the timer

The timer server is located in *Middlewares\Third_Party\Lora\Utilities*.

4.5.2 Low-power functions

The APIs presented in the table below, are used to manage the low-power modes of the core MCU.

Table 15. Low-power functions

Function	Description
`void LPM_EnterLowPower (void)`	To enter the system in low-power mode
`void LPM_EnterSleepMode (void)`	Allow the application to implement a dedicated code before entering Sleep mode
`void LPM_ExitSleepMode (void)`	Allow the application to implement a dedicated code before exiting Sleep mode
`void LPM_EnterStopMode (void)`	Enter low-power Stop mode
`void LPM_ExitStopMode (void)`	Exit low-power Stop mode
`void LPM_EnterOffMode (void)`	Allow the application to implement a dedicated code before entering Off mode

Table 15. Low-power functions (continued)

Function	Description
`void LPM_ExitOffMode(void)`	Allow the application to implement a dedicated code before exiting Off mode
`LPM_GetMode_t LPM_GetMode (void)`	Return the selected low-power mode
`void LPM_SetStopMode (LPM_Id_t id, LPM_SetMode_t mode)`	Used to enable or disable the Stop mode in order to require Sleep mode
`void LPM_SetOffMode (LPM_Id_t id, LPM_SetMode_t mode)`	Used to enable Stop mode or to enable Off mode

4.5.3 System time functions

MCU time is referenced to MCU reset. SysTime 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.

Table 16. 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 BACK_UP register (retained even in Standby mode). The system time reference is the Unix epoch starting January 1st, 1970.
`SysTime_t SysTimeGet (void)`	Get the current system time. The system time reference is UNIX epoch starting January 1st 1970.
`uint32_t SysTimeMkTime (const struct tm* localtime)`	Convert local time into Epoch time (see the note below)
`void SysTimeLocalTime (const uint32_t timestamp, struct tm *localtime)`	Convert Epoch time into local time (see the note below)

Note: 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 is a two-hour difference with Greenwich time. Assuming time is set, a local time can be printed on terminal:

```
{
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);
}
```

4.5.4 Trace functions

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.

Table 17. Trace functions

Function	Description
`void TraceInit (void)`	TraceInit must be called at the application initialization. It initializes the com or vcom hardware in DMA mode and registers the call back to be processed at DMA transmission completion.
`int32_t TraceSend (const char *strFormat,...)`	Convert string format into a buffer and buffer length and records it into the circular queue if sufficient space is left. Returns 0 when queue if sufficient space is left. Returns -1 when not enough room is left.

The `TraceSend(..)` function can be used in polling mode when no real-time constraints apply, typically during application initialization:

```
#define PPRINTF(...) do{} while (0!= TraceSend (__VA_ARGS__))
//Polling Mode.
```

The `TraceSend(..)` function can be used in real-time mode. In this case, when there is not space left in the circular queue, the string is not added and is not printed out in com port

```
#define PRINTF(...)     do {TraceSend (__VA_ARGS__);} while(0)
```

The `TraceSend(..)` function can be used in real-time mode by adding a timestamp:

```
#define PRINTNOW(a)      do{ \
SysTime_t stime  =SysTimeGetMcuTime();   \
TraceSend("%3ds%03d:%d ",stime.Seconds, stime.SubSeconds,a); \
}while(0)
```

A verbose level (VERBOSE_LEVEL_1,VERBOSE_LEVEL_2 in utilities_cnf.h) can be applied to the system trace.

```
#define TVL1(X)    do{ if(VERBOSE_LEVEL>=VERBOSE_LEVEL_1) { X }
}while(0);
#define TVL2(X)    do{ if(VERBOSE_LEVEL>=VERBOSE_LEVEL_2) { X }
}while(0);
```

The buffer length can be increased in case it is saturated in utilities_conf.h

```
#define DBG_TRACE_MSG_QUEUE_SIZE 256
```

4.5.5 Queuing functions

The queue module provides a set of services managing a buffer as a circular queue.

The APIs presented in the table below are used to manage a circular queuing buffer.

Table 18. Middleware queuing functions

Function	Description
`int CircularQueue_Init (queue_t *q, uint8_t* queueBuffer, uint32_t queueSize, uint16_t elementSize, uint8_t optionFlags)`	Initialize the circular buffer.
`uint8_t* CircularQueue_Add (queue_t *q, uint8_t* x, uint16_t elementSize, uint32_t nbElements)`	Add an element to the circular buffer.
`uint8_t* CircularQueue_Remove (queue_t *q, uint16_t* elementSize)`	Remove an element from the circular buffer.
`uint8_t* CircularQueue_Sense (queue_t *q, uint16_t* elementSize)`	Sense if the circular buffer is not empty. If not empty, it returns the address of the buffer and its length through element size.

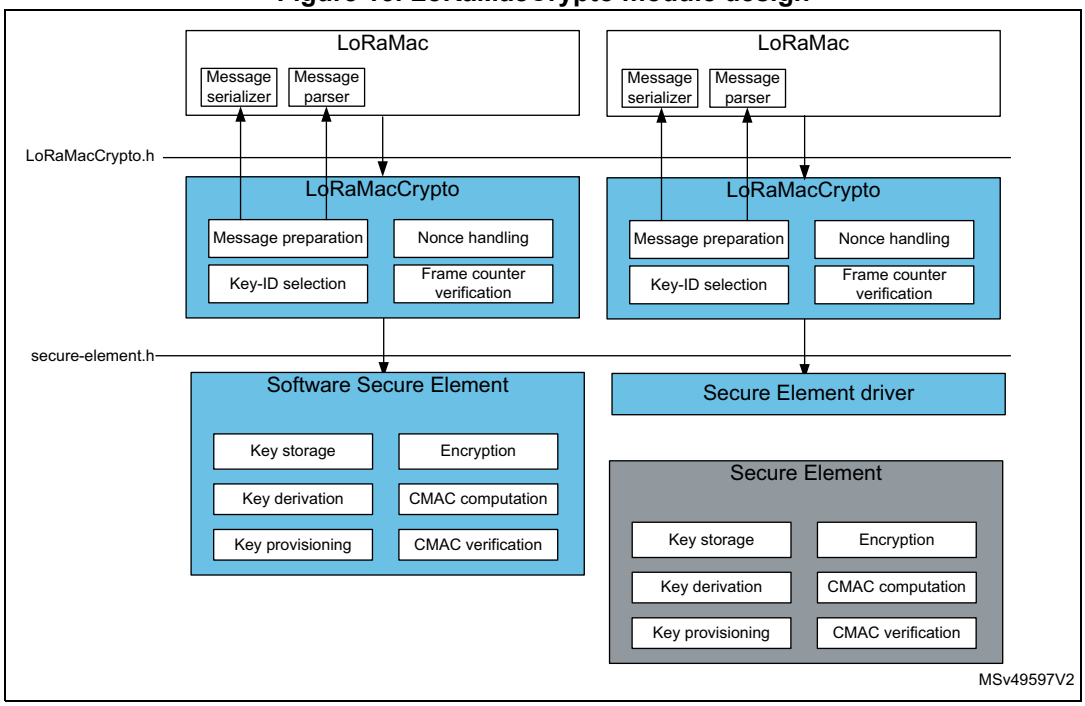
Note: The queue is filled with elements. Each element is composed of buffer length field (2 bytes) and the buffer. When an element is too large to fit at the end of the queue, it is fragmented into two elements.

4.6 Emulated secure-element

By default, the proposal hardware platforms do not integrate a secure-element device. Therefore this secure-element device is emulated by software.

The figure below describes the main design of the LoRaMacCrypto module.

Figure 13. LoRaMacCrypto module design



The APIs presented in the table below are used to manage the emulated secure-element.

Table 19. Secure-element functions

Function	Description
`SecureElementStatus_t SecureElementInit (EventNvmCtxChanged seNvmCtxChanged)`	Initialization of the secure-element driver The Callback function that 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 KeyID determines the AES key to use.

Table 19. Secure-element functions (continued)

Function	Description
`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 of the LoRaWAN versionKeyID, rootKeyID are used to identify the root key to perform the derivation.

4.7 Middleware End_Node application function

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

Standard mode

In standard mode, an interface file (see MAC upper layer in *Figure 12*) is provided to let the user start without worrying about the LoRa state machine. The interface file is located in Middlewares\Third_Party\Lora\Core\lora.c.

The interface file implements:

- a set of APIs allowing to access to the LoRaMAC services
- the LoRa certification test cases that are not visible to the application layer

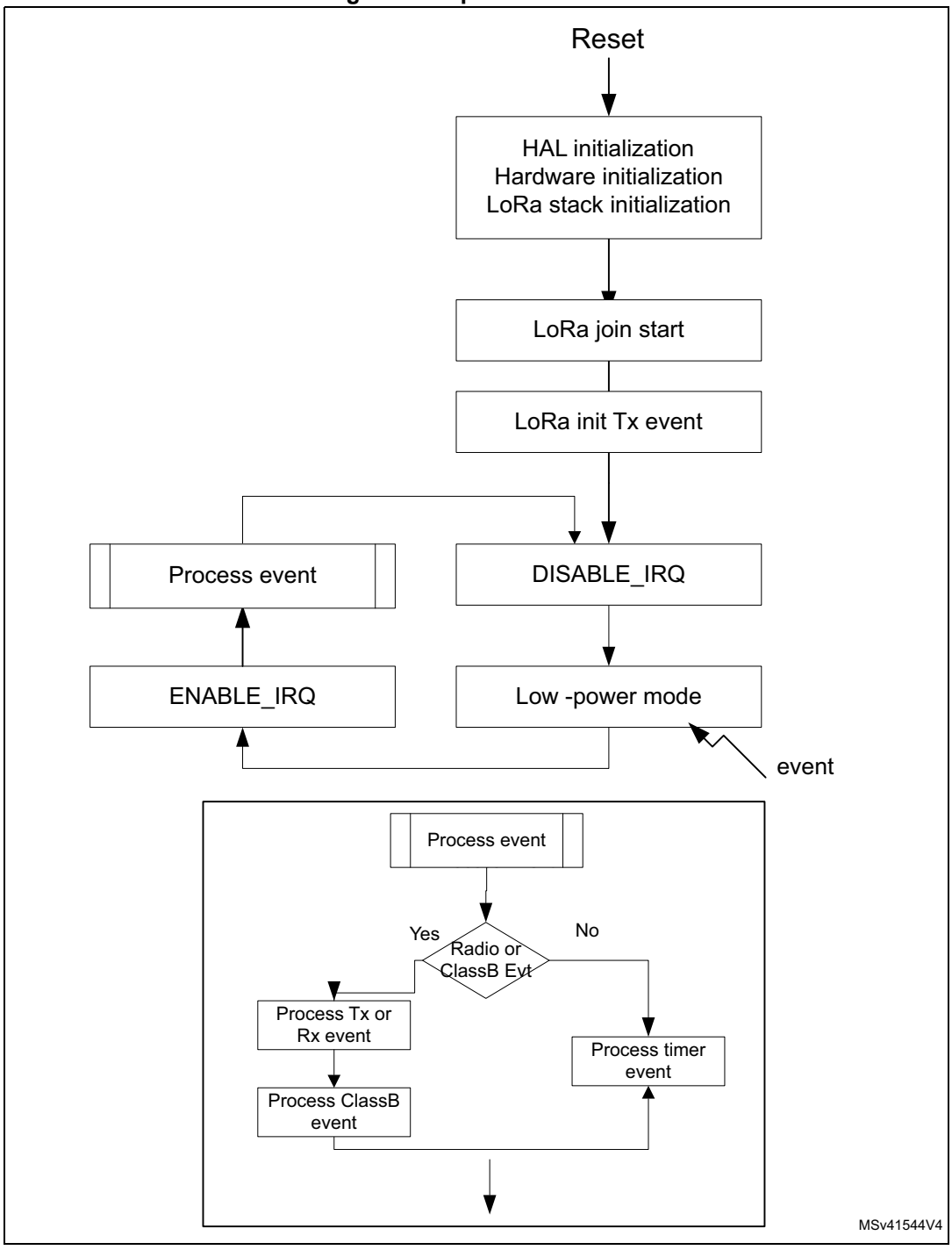
Advanced mode

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

Operation model

The operation model proposed for this LoRa End_Node (see *Figure 14*) is based on 'event-driven' paradigms including 'time-driven'. The behavior of the system LoRa is triggered either by a timer event or by a radio event plus a guard transition.

Figure 14. Operation model



LoRa system state behavior

Figure 15 describes the LoRa End_Node system state behavior.

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

The LoRa 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**'.

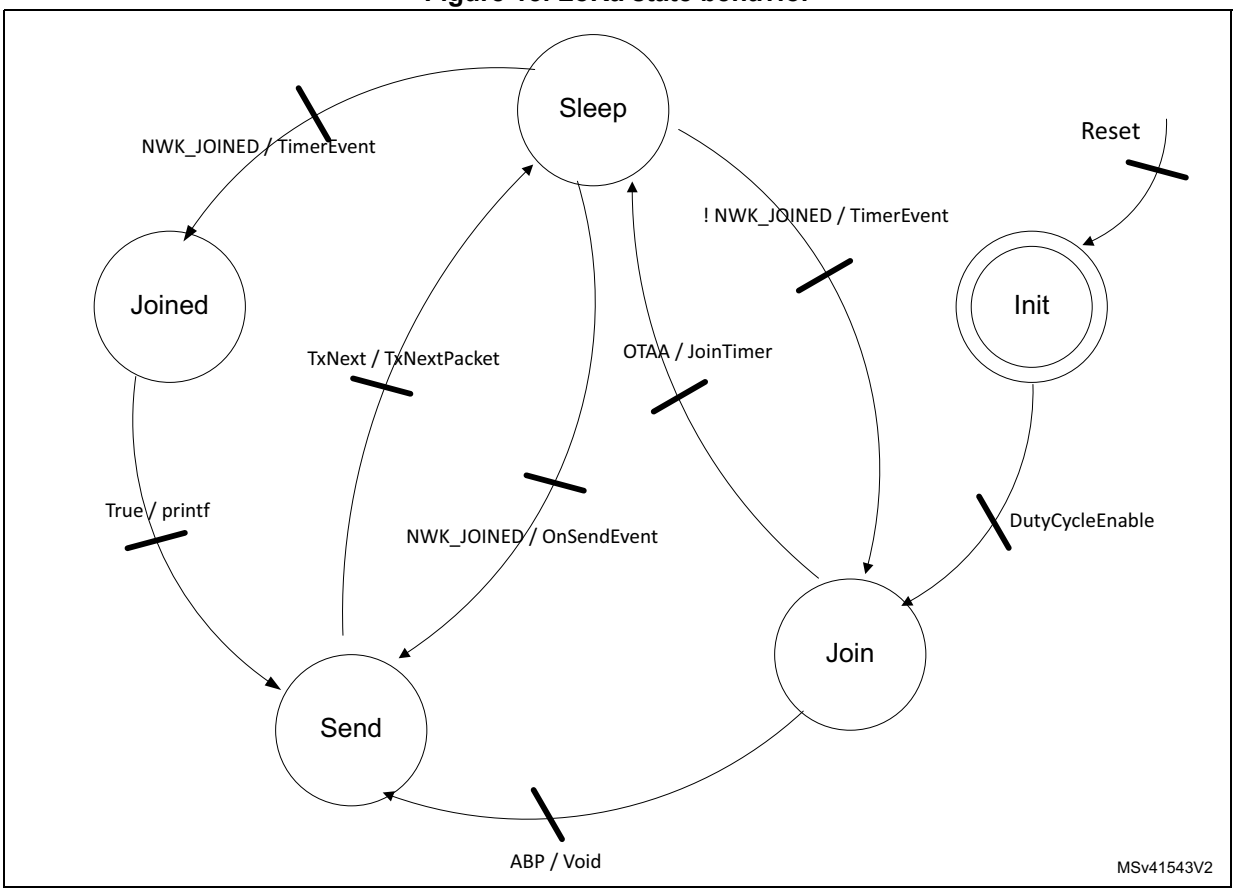
When using the "activation by personalization (ABP)", the network is already joined and therefore the LoRa End_Node system jumps directly to a state defined as '**Send**'.

From the state defined as '**Sleep**', if the end-device joined the network when a "TimerEvent" occurred, the LoRa 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 LoRa End_Node system goes into the state defined as '**Send**'.

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

Figure 15. LoRa state behavior



LoRa class B system state behavior

Figure 16 describes the LoRa 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 (see *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 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 LoRa class B system state goes in BEACON_STATE_ACQUISITION.

The LoRa end-device starts the beacon acquisition. When the MAC layer received a beacon in function RxBeacon successfully, the next state is BEACON_STATE_LOCKED.

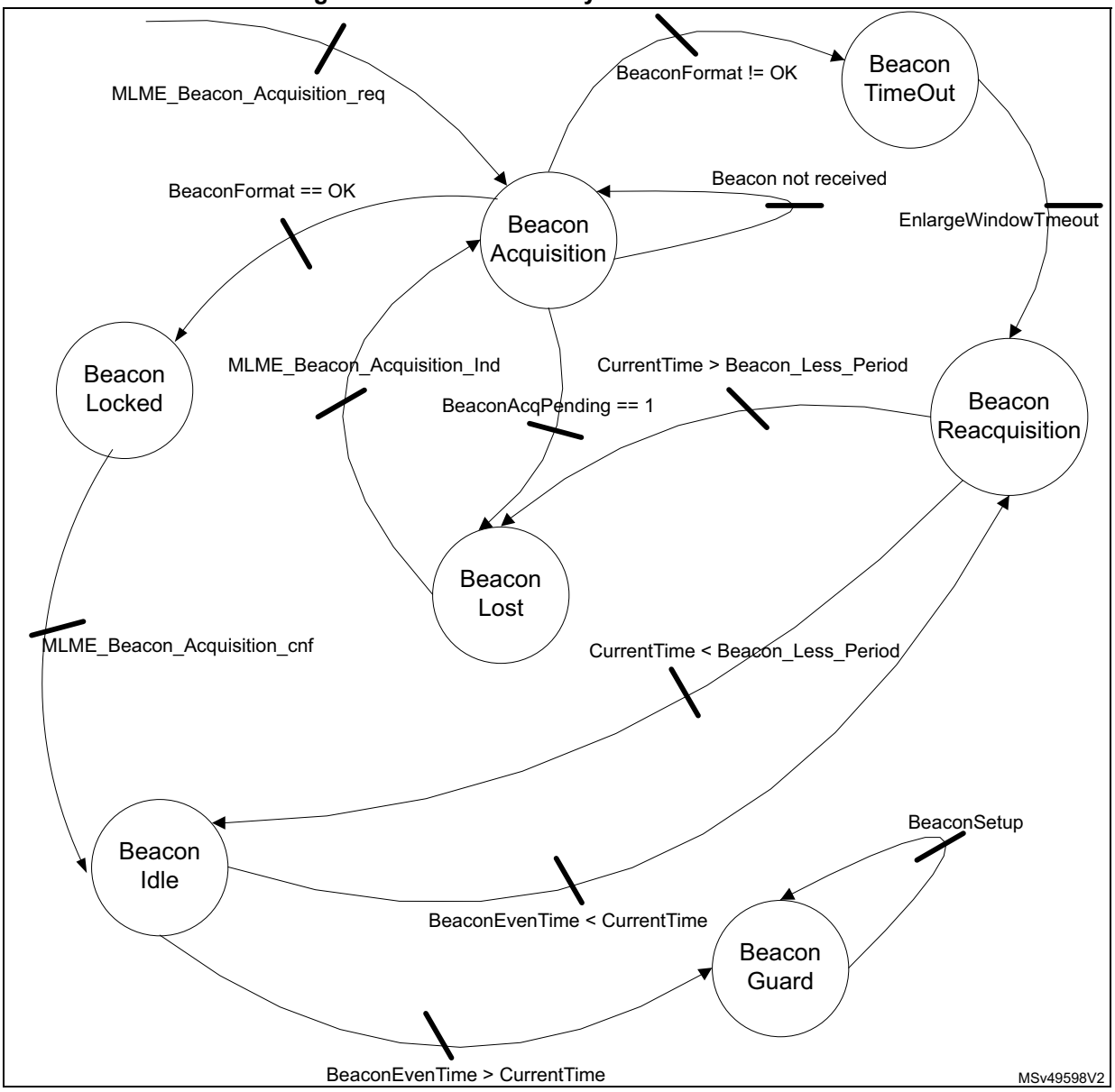
When the LoRa end-device receives a beacon, the acquisition is no longer pending: the MAC layer goes in BEACON_STATE_IDLE.

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. LoRa class B system state behavior



4.7.1 LoRa End_Node initialization

Table 20. LoRa class A initialization function

Function	Description
`void lora_Init (LoRaMainCallback_t *callbacks, LoRaParam_t* LoRaParamInit)`	Initialization of the LoRa class A finite state machine

4.7.2 LoRa End_Node Join request entry point

Table 21. LoRa End_Node Join request entry point

Function	Description
`void lora_Join (void)`	Join request to a network either in OTAA mode or ABP mode (The Join mode must be defined at compile time).

4.7.3 LoRa End-Node start Tx

Table 22. LoRa End-Node start Tx

Function	Description
`void loraStartTx (TxEventType_t EventType)`	Start the OnTxTimerEvent occurrence if EventType parameter is equal to TX_ON_TIMER. The user is free to implement its own code here.

4.7.4 Request End-Node Join Status

Table 23. End-Node Join status

Function	Description
`LoraFlagStatus LORA_JoinStatus (void)`	Check the End-Node activation type: ACTIVATION_TYPE_NONE, ACTIVATION_TYPE_ABP, ACTIVATION_TYPE_OTAA.

4.7.5 Send an uplink frame

Table 24. Send an uplink frame

Function	Description
`bool LORA_send (lora_AppData_t* AppData,LoraConfirm_t IsTxConfirmed)`	Send an uplink frame. This frame can be either an unconfirmed empty frame or an unconfirmed/confirmed payload frame.

4.7.6 Request the current network time

Table 25. Current network time

Function	Description
`LoraErrorStatus LORA_DeviceTimeReq (void)`	The end-device requests from the network the current network time (useful to accelerate the beacon discovery in class B mode) Note: To be used in place of BeaconTimeReq in LoRaWAN version ≥ 1.0.3

4.7.7 Request the next beacon timing

Table 26. Next beacon timing

Function	Description
`LoraErrorStatus LORA_BeaconTimeReq (void)`	The end-device requests from the network the next beacon timing (useful to accelerate the beacon discovery in class B mode) Note: command deprecated in the LoRaWAN V1.0.3

4.7.8 Switch class request

Table 27. Switch class request

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

4.7.9 Get End-device current class

Table 28. Get End-Device current class

Function	Description
`void LORA_GetCurrentClass(DeviceClass_t *currentClass)`	Request the current running class A, B or C.

4.7.10 Request beacon acquisition

Table 29. Request beacon acquisition

Function	Description
`LoraErrorStatus LORA_BeaconReq (void)`	Request the beacon slot acquisition.

4.7.11 Send unicast ping slot info periodicity

Table 30. Unicast ping slot periodicity

Function	Description
`LoraErrorStatus LORA_PingSlotReq (void)`	Transmit to the server the unicast ping slot info periodicity.

4.8 LIB End_Node application callbacks

4.8.1 Current battery level

Table 31. Current battery level function

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

4.8.2 Current temperature level

Table 32. Current Temperature function

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

4.8.3 Board unique ID

Table 33. Board unique ID function

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

4.8.4 Board random seed

Table 34. Board random seed function

Function	Description
`uint32_t HW_GetRandomSeed (void)`	Get a random seed value.

4.8.5 Make Rx frame

Table 35. Make Rx frame

Function	Description
`void LoraRxData (lora_AppData_t *AppData)`	To process the incoming frame application. The user is free to implement his own code here.

4.8.6 Request class mode switching

Table 36. LoRa HasJoined function

Function	Description
`void LORA_HasJoined (void)`	Notify the application that the End-Node joined.

4.8.7 End_Node class mode change confirmation

Table 37. End_Node class mode change confirmation function

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

4.8.8 Send a dummy uplink frame

Table 38. Send a dummy uplink frame

Function	Description
`void LORA_TxNeeded (void)`	Request the application to send a frame.

5 Example description

5.1 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 (see *Section 5.4*).

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 (no LoRa expansion board required)

Optionally, an ST X-NUCLEO-IKS01A1 sensor expansion board can be added on Nucleo boards and Discovery kits. The Nucleo-based supported hardware is presented in the table below.

Table 39. Nucleo-based supported hardware

Nucleo board	LoRa radio expansion board		
	SX1276MB1MAS	SX1276MB1LAS	SX1272MB2DAS
NUCLEO-L053R8	Supported	Supported	Supported
NUCLEO-L073RZ	Supported	Supported	Supported (P-NUCLEO-LRWAN1(1))
NUCLEO-L152RE	Supported	Supported	Supported
NUCLEO-L476RG	Supported	Supported	Supported

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

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 the table below.

Table 40. 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, see *Section 3.4.4*.
- 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 at *Projects\<platform>\Applications\LoRa\<App_Type>\Core\inc*.

The <platform> can be STM32L053R8-Nucleo, STM32L073RZ-Nucleo, STM32L152RE-Nucleo, STM32L476RG-Nucleo or B-L072Z-LRWAN1 (Murata modem device).

The <Target> can be STML0xx and the <App_Type> can be AT_Master, End_Node, PingPong or AT_Slave.

Interrupts

The table below shows the interrupt priorities level applicable for the Cortex system processor exception and for the STM32L0 Series LoRa application-specific interrupt (IRQ).

Table 41. STM32L0xx IRQ priorities

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

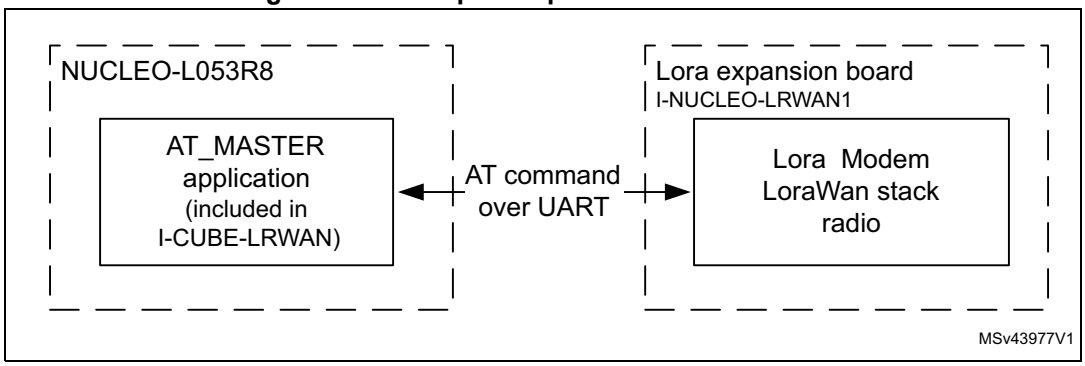
5.2 Split end-device hardware description (two-MCUs solution)

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

The AT_Master application implementing the LoRa host on a NUCLEO board is compatible with the AT_Slave application (see *Section 5.6*). The AT_Slave application demonstrates a modem on the CMWX1ZZABZ-091 LoRa module (Murata). The 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 (see *Section 5.7*).

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

Figure 17. Concept for split end-device solution

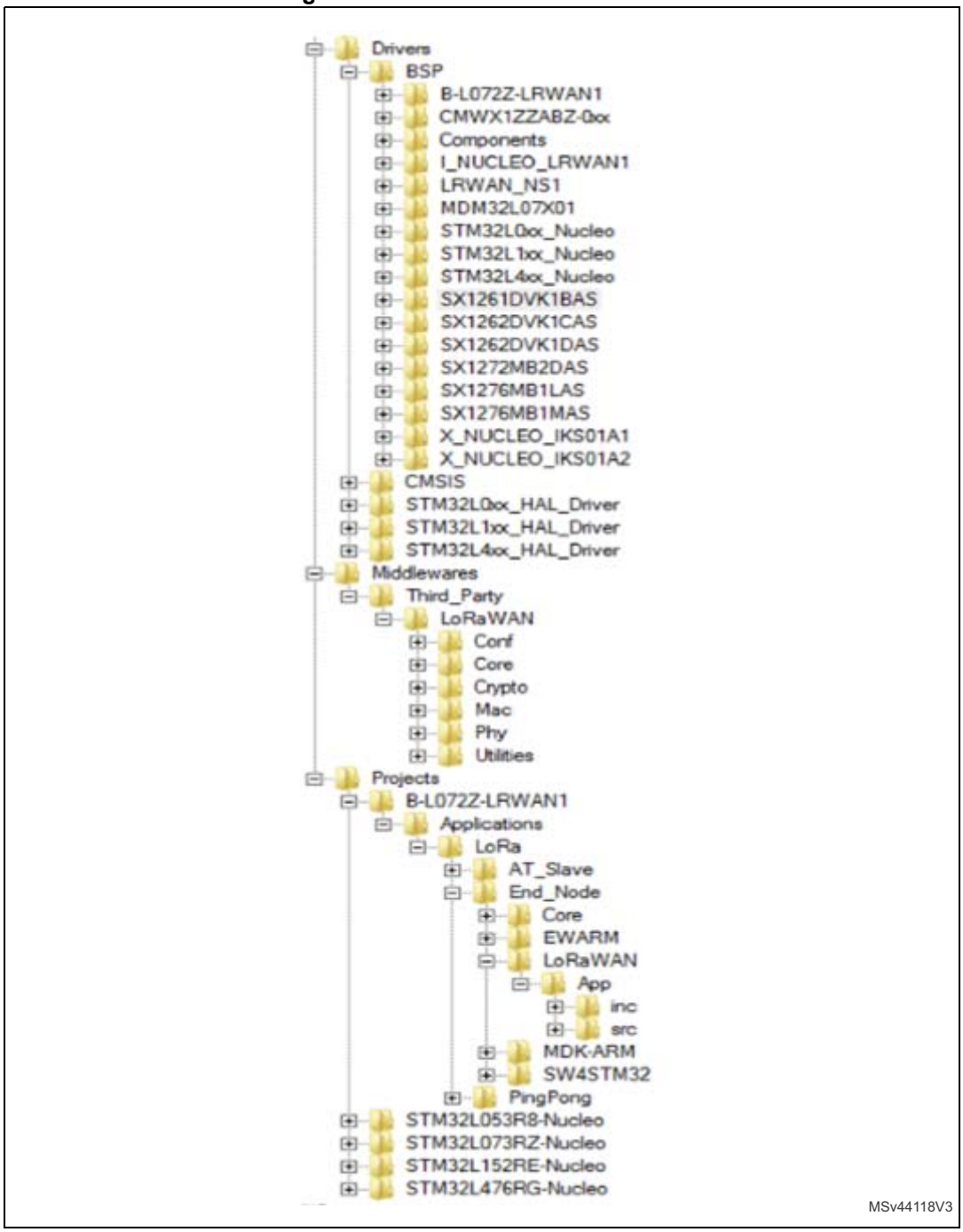


The interface between the LoRa modem and the LoRa host is a UART running AT commands.

5.3 Package description

When the user unzips the I-CUBE-LRWAN, the package presents the structure shown in the figure below.

Figure 18. I-CUBE-LRWAN structure



The I-CUBE-LRWAN package contains four applications: End_Node, PingPong, AT_Slave, and AT_Master. For each application, three toolchains are available: MDK-ARM, IAR, and SW4STM32.

5.4 End_Node application

This application reads the temperature, humidity and atmospheric pressure from the sensors through the I2C. The MCU measures the supplied voltage through V_{REFLNT} in order to calculate the battery level. These four data (temperature, humidity, atmospheric pressure, and battery level) are sent periodically to the LoRa network using the LoRa radio in class A at 868 MHz.

In order to launch the LoRa End_Node project, the user must go to *\Projects\<target>\Applications\LoRa\End_Node* and choose his favorite toolchain folder (in the IDE environment). The user selects then the LoRa project from the proper target board.

5.4.1 Activation methods and keys

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

\Projects\<target>\Applications\LoRa\End_Node\LoRaWAN\App\inc\Commissioning.h file gathers all the data related to the device activation. The chosen method, along with the commissioning data, is printed on the virtual port and visible on a terminal.

5.4.2 Debug switch

The user must go to *\Projects\Multi\Applications\LoRa\End_Node\inc\hw_conf.h* to enable the debug mode or/and the trace mode by commenting out
`#define DEBUG`

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:

- VERBOSE_LEVEL_0: traces disabled
- VERBOSE_LEVEL_1: enabled for functional traces
- VERBOSE_LEVEL_2: enabled for Debug traces

The user must go to *\Projects\<platform>\Applications\LoRa\<App>\LoRaWAN\App\inc\utilities_conf.h* to set the select trace level.

Note: In order to enable a true low-power, "`#define DEBUG`" mentioned above must be commented out.

5.4.3 Sensor switch

When no sensor expansion board is plugged on the set-up, `#define SENSOR_ENBALED` must be commented out on the *\Projects\<target>\Applications\LoRa\End_Node\LoRaWAN\App\inc*.

The table below provides a summary of the main options for the application configuration.

Table 42. Switch options for the application's configuration

Project	Switch option	Definition	Location
LoRa stack	OVER_THE_AIR_ACTIVATION	Application uses over-the-air activation procedure.	Commissioning.h
	STATIC_DEVICE_EUI	Static or dynamic end- device identification	Commissioning.h
	LORAMAC_CLASSB_ENABLED	Compile the relevant code for class B mode.	Compiler option setting
	USE_DEVICE_TIMING	Includes either "LORA_DeviceTimeReq ()" or "LORA_BeaconTimeReq (void)	lora.c
	STATIC_DEVICE_ADDRESS	Static or dynamic end- device address	Commissioning.h
	REGION_EU868	Enable the band selection	Compiler option setting
	REGION_EU433		
	REGION_US915		
	REGION_AS923		
	REGION_AU915		
	REGION_CN470		
	REGION_CN779		
	REGION_IN865		
	REGION_RU864		
	REGION_KR920		
Sensor	DEBUG	Enable ‘Led on/off’	hw_conf.h
	VERBOSE_LEVEL	Enable the trace level	utilities_conf.h
	SENSOR_ENABLED	Enable the call to the sensor board	hw_conf.h

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

5.5 PingPong application description

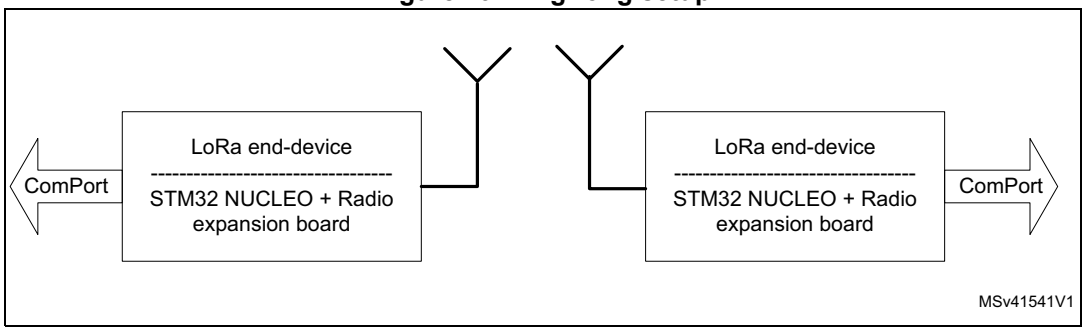
This application is a simple Rx/Tx RF link between two LoRa end-devices. By default, each LoRa end-device starts as a master and transmits a ‘Ping’ message and wait for an answer. The first LoRa end-device receiving a ‘Ping’ message becomes a slave and answers the master with a ‘Pong’ message. The PingPong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

In order to launch the PingPong project, the user must go to the *\Projects\<platform>\Applications\LoRa\PingPong* folder and follow the same procedure as for the LoRa End_Node project to launch the preferred toolchain.

Hardware and software set-up environment

To set up the STM32LXxx-NUCLEO, connect the NUCLEO (or the B-L072Z-LRWAN1) board to the computer with a USB cable type A to mini B to the ST-LINK connector (CN1). Ensure that the CN2 ST-LINK connector jumpers are fitted. See the figure below for a representation of the PingPong setup.

Figure 19. PingPong setup



5.6 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 LoRa 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.

In order to launch the AT_Slave project, the user must go to the folder *Projects\B-L072Z-LRWAN1\Applications\LoRa\AT_Slave* and follow the same procedure as for the LoRa End_Node project to launch the preferred toolchain.

The application note *Examples of AT commands on I-CUBE-LRWAN* (AN4967) gives the list of AT commands and their description.

5.7 AT_Master application description

This application reads sensor data and sends them to a LoRa network through an external LoRa modem. The AT_Master application implements a complete set of AT commands to drive the LoRa 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) or the LRWAN-NS1 expansion board featuring the RiSiNGHF modem available in P-NUCLEO-LRWAN3[(a)] and P-NUCLEO-LRWAN3[(b)].

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

BSP programming guidelines

The table below gives a description of the BSP driver APIs to interface with the external LoRa module.

Table 43. BSP programming guidelines

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 IO commands

Note: The NUCLEO board communicates with the expansion board via the UART (PA2, PA3). The following modifications must be applied (see section 5.8 of the user manual STM32 Nucleo-64 boards (UM1724)):

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

5.8 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.

This application is made up of secure boot and secure firmware update (SBSFU), LoRaWAN protocol stack and User Application.

This application only targets the SMT32L476 microcontroller

The FUOTA application Note on I-CUBE-LRWAN (AN5411) gives all the needed information to make use of the FUOTA I-CUBE-LRWAN part.

a. Refer to the user manual *Getting started with the P-NUCLEO-LRWAN2* (UM2587).

b. Refer to the user manual *Getting started with the P-NUCLEO-LRWAN3* (UM2612)

6 System performances

6.1 Memory footprints

The values in the table below are measured for the following configuration of the Keil compiler (Arm compiler 5.05):

- Optimization: optimized for size level 3
- Debug option: off
- Trace option: off
- Target: P-NUCLEO-LRWAN1 (STM32L073+ SX1272MB2DAS)

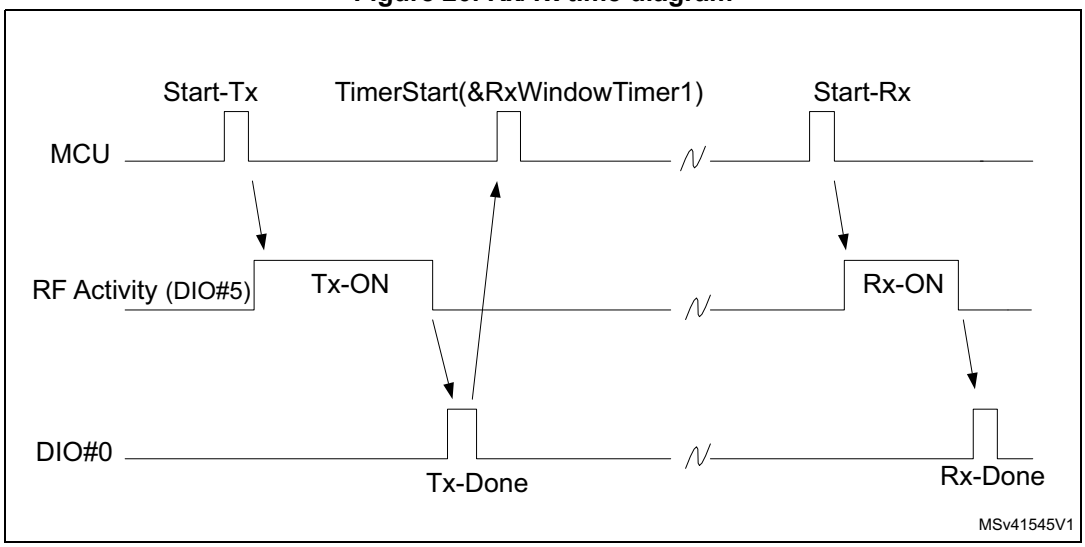
Table 44. Memory footprint values for End_Node application

Project	Flash (bytes)	RAM (bytes)	Description
Application layer	4336	456	Includes all microlib.
LoRa stack	29926	3486	Includes MAC + RF driver.
HAL	10362	1536	-
Utilities	2474	464	Includes services like system, timeserver, vcom and queue.
Total application	52468	6362	Memory footprint for the overall application

6.2 Real-time constraints

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

Figure 20. Rx/Tx time diagram



Rx window channel start

The Rx window opens the RECEIVE_DELAY1 for 1second (± 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 > DI0#n interrupt (see *Table 41*) in order to avoid stalling the received startup time.

6.3 Power consumption

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

Measurements setup

- No DEBUG
- No TRACE
- No SENSOR_ENABLED

Measurements results

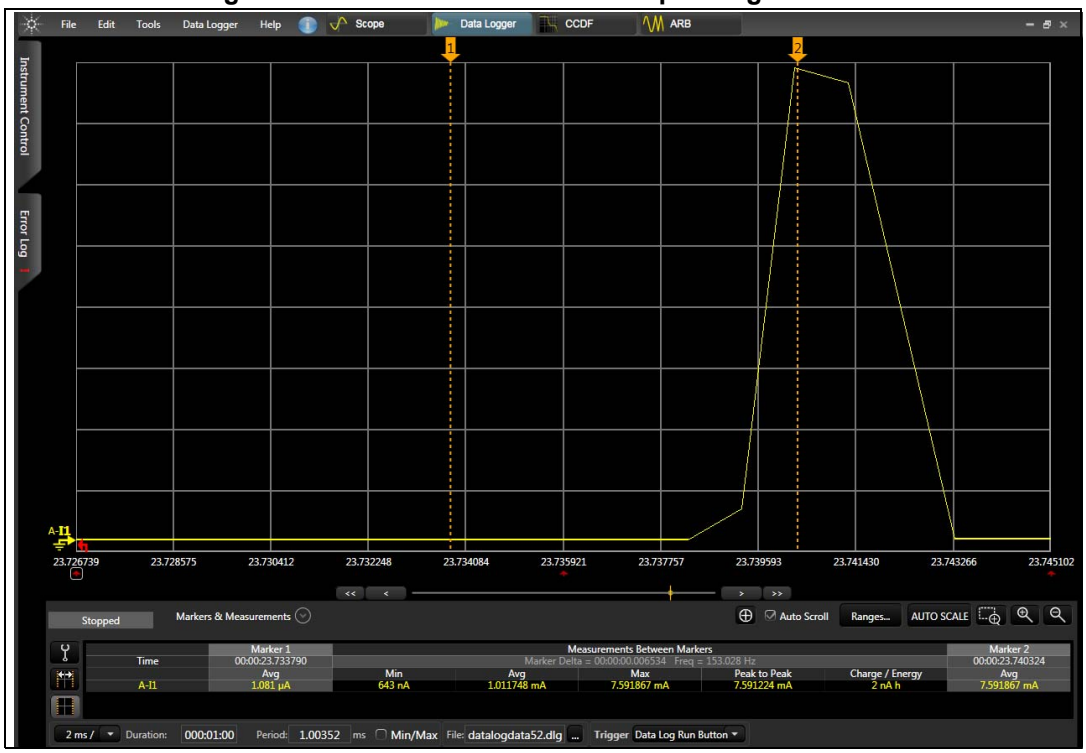
- Typical consumption in stop mode: 1.3 µA
- Typical consumption in run mode: 8.0 mA

Measurements figures

- Instantaneous consumption over 30 s

Figure 21 shows an example of the current consumption against time on a microcontroller of the STM32L0 Series.

Figure 21. STM32L0 current consumption against time



7 Revision history

Table 45. Document revision history

Date	Revision	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* with reference to 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: – *Section 5.5: 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

Table 45. Document revision history (continued)

Date	Revision	Changes
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 *Figure 17*, *Table 4*, *Table 10* to *Table 45*
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 *Table 1* – LoRa Alliance and application notes references in *Section 1.2* – New *Section 5.8: FUOTA application description*