LoRaWAN® firmware update over the air with STM32CubeWL

Introduction

This application note describes the FUOTA (firmware update over the air) application embedded in the STM32CubeWL MCU package, and explains how to use the overall FUOTA process to provide the components needed for a FUOTA campaign.

There are three environment projects:

- LoRaWAN® FUOTA SingleCore solution based on STM32WLE5xx, STM32WL55xx or STM32WL5MOCHxx microcontrollers series
- LoRaWAN® FUOTA DualCore solution based on STM32WL55xx or STM32WL5MOCHxx microcontrollers series
- LoRaWAN® FUOTA DualCore with External Flash (for Download slot) solution based on STM32WL5MOCHxx microcontrollers series

This document applies within the framework of a FUOTA project, and targets particularly the FUOTA project integrators, or those integrating FUOTA modules in a wider system implementing end-device functions.

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

1 General information

The STM32CubeWL runs on STM32WLSeries microcontrollers based on Arm® Cortex®-M processor.

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

arm

The FUOTA application in STM32CubeWL is compliant with the LoRa Alliance® specification protocol (LoRaWAN Link Layer v1.0.3 and v1.0.4, see document [1]).

The FUOTA feature implemented in the application layer is based on and compliant with the specific functionalities defined by the LoRa Alliance. These functionalities allow a multicast group (Remote Multicast Setup Spec V1.0.0, see document [4]) to be set up, to fragment and to send data packets (Fragmented Data Block Transport Specification v1.0.0, see document [3]), and finally to synchronize clocks (LoRaWAN Application Layer Clock Synchronization Specification v1.0.0, see document [2]), so that all devices agree on the start of a FUOTA session.

Note: Throughout this application note, the IAR Embedded Workbench® EWARM and Keil® MDK-ARM IDEs are used as an example to provide guidelines for project configuration.

The FUOTA application:

- supports a full firmware upgrade image (entire firmware image sent to the end device)
- is only applicable in Class-C mode
- runs on STM32WL55xx or STM32WL5MOCHxx targets for DualCore, or STM32WLE5xx, STM32WL55xx or STM32WL5MOCHxx target for SingleCore
- supports a third-party middleware, mbed-crypto for the cryptographic services

Table 1. Acronyms and terms

Acronym or term	Definition
ABP	Activation by personalization
APDU	Application protocol data unit
AS	Application server
BFU	Boot and Firmware Update
DAP	Direct access port
DMA	Direct memory access
End device	Device used as sensor or actuator in a networked system
FEC	Forward error correction
Firmware image	Binary image (executable) run by the end device
Firmware header	Meta-data describing the firmware image to be installed
FUOTA	Firmware update over the air
HAL	Hardware abstract layer
IDWG	Independent watchdog
KMS	Key management services
L2	Link layer
LoRa	Long-range radio technology
LoRaWAN	LoRa wide-area network
LDPC	Low-density parity code
MAC	Media access control
Mbed-crypto	Mbed cryptography library implementation of the cryptography interface of the Arm PSA (platform security architecture)

Acronym or term	Definition
MCPS	MAC common part sublayer
MCU	Microcontroller
MIB	MAC information base
MLME	MAC layer management entity
MPDU	MAC protocol data unit
MPU	Memory protection unit
MSC	Message sequence chart
NS	Network server
OTA	Over the air
OTAA	Over-the-air activation
PLME	Physical layer management entity
PPDU	Physical protocol data unit
RP	Regional parameters
SAP	Service access point
SE	Secure Engine
SBSFU	Secure Boot and Secure Firmware Update
`sfb` file	Binary file packing the firmware header and the firmware image
SFU	Secure Firmware Update
SKMS	Secure key management services

Table 2. **Document references**

Reference	Document
[1]	LoRa Alliance Specification Protocol (LoRaWAN version V1.0.3), March 2018
[2]	LoRa Alliance Application layer clock synchronization over LoRaWAN Specification v1.0.0, September 2018 - [TS-003]
[3]	LoRa Alliance Fragmented Data Block Transport over LoRaWAN Specification v1.0.0, September 2018 - [TS-004]
[4]	LoRa Alliance Remote Multicast Setup over LoRaWAN Specification v1.0.0, September 2018 - [TS-005]
[5]	*Integration guide of SBSFU on STM32CubeWL (including KMS)* (AN5544)
[6]	*Getting Started with the SBSFU of STM32CubeWL* (UM2767)
[7]	*How to build a LoRa application with STM32CubeWL* (AN5406)
[8]	*How to secure LoRaWAN and Sigfox with STM32CubeWL* (AN5682)

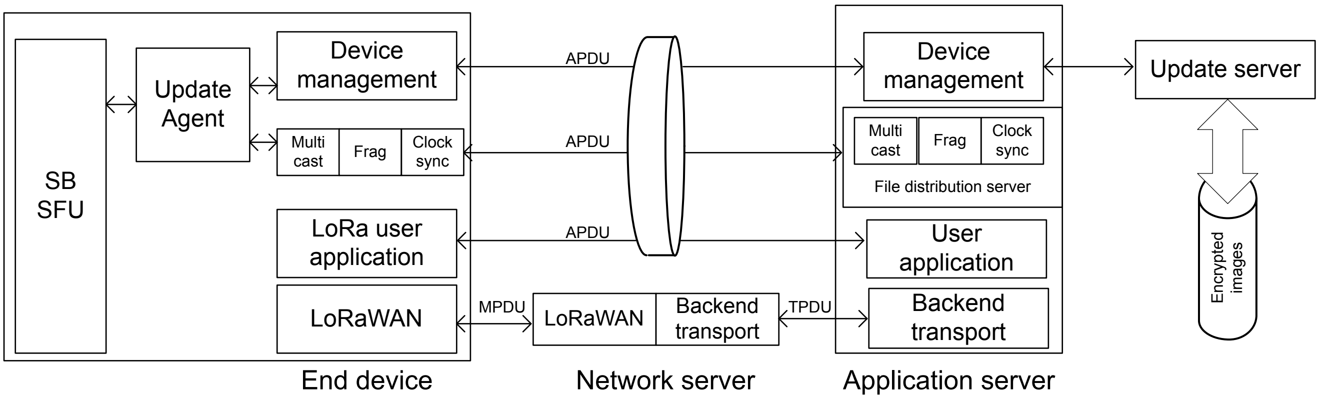
2 LoRaWAN standard and FUOTA application feature

This section provides a general overview of the LoRa and LoRaWAN recommendations. It deals with the LoRaWAN end-device and the FUOTA feature, that are the core subjects of this application note.

2.1 Network architecture

The figure below shows the components and their protocol relationships, allowing the implementation of the FUOTA feature.

Figure 1. **Network diagram**

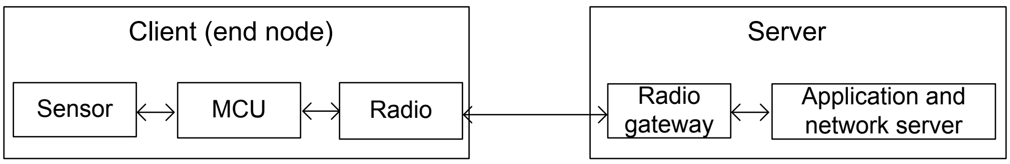


Note: The LoRa Alliance technical FUOTA working group works on the LoRaWAN Firmware Management Protocol Specification, that documents and defines this block. The proposed implementation in the FUOTA application, is a proof of concept.

2.1.1 Client/server architecture

In the figure below, the end device where the software or firmware must be updated, is referred to as the end node or client. The other part of the system is referred to as the cloud or server, and provides the new software or firmware.

Figure 2. **Client/server architecture example**



2.1.2 End-device architecture

The end device consists of a host MCU (microcontroller) that reads sensor data to transmit the sensor reading over the LoRaWAN network by means of the LoRa radio module.

Data is encrypted by the host MCU and the radio packet is received by the gateway, that forwards it to the network server. The network server then sends data to the application server, that has the right key to decrypt the application data.

2.2 End-device classes

The LoRaWAN protocol specification (see document [1]) has several end-device classes to address the various needs of a wide range of applications.

The FUOTA application described in this document is only 'Class-C enable'. In other words, the FUOTA application is validated for network infrastructure supporting at least Class-C mode.

Note: The end device supports Class-B mode. Nevertheless, it is only 'Class B capable'. To be 'Class B enable', a new integration and validation phase must be proceeded on a network infrastructure supporting Class-B mode for the FUOTA campaign.

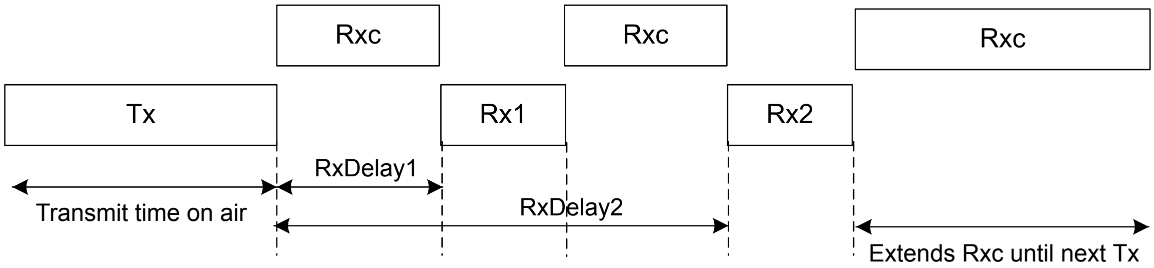
Class definition

The definition of the various classes is (see document [1]):

- Class A: bi-directional end devices (all devices)
- Class B: bi-directional end devices with scheduled receive slots (Beacon)
- Class C: bi-directional end devices with maximal receive slots (Continuous)

The Class-C mode is implemented to support FUOTA. Class-C end devices have almost continuously open receive windows (RxC where the data blocks are received), and are only closed when transmitting (Tx) and receiving (Rx1, Rx2) in Class-A mode (see the figure below).

Figure 3. **Tx/Rx timing diagram (Class C)**



2.3 FUOTA overview

The FUOTA update process transfers a new software image (data file) from the server to the client, and updates the current software image (version N) running on the client with the new received software image (version N+1). Obstacles to successful completion of the FUOTA update process are listed below:

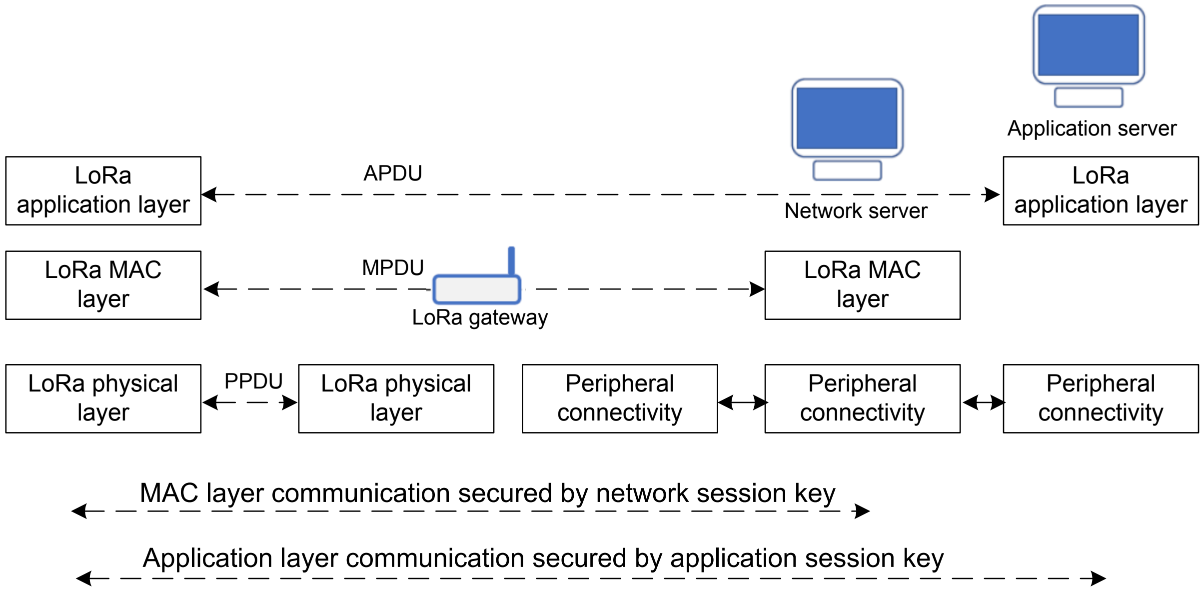
- Communication
  The new firmware image must be sent from the server to the client. This challenge is performed through the application-layer protocols running over LoRaWAN, that provide remote-multicast setup, fragmented data-block transport, and application-layer clock-synchronization services. The LoRaWAN MAC layer provides Class-C mode to transmit the data file in unicast or multicast mode.
- Firmware update
  The client must migrate from the current to the new firmware image. This task is performed by the Update Agent module. To succeed, the Update Agent module relies on the services provided by the SBSFU (Secure Boot and Secure Firmware Update) application, using the SE (Secure Engine), KMS (key management services) and mbed-crypto middleware.
- Memory
  The software architecture must be organized so that it can be executed when the update process completes. The solution must ensure the recovery of the new software version, if there are installation issues. This task is handled by the SBSFU application.
- Security
  When a new firmware image is sent wireless from server to client, several security services must be assured (such as authentication, confidentiality, and integrity). This must be done either through the LoRaWAN protocol or by means of the SBSFU application security services.

2.4 Network protocol architecture

The figure below describes the end-to-end network protocol architecture. The following protocol exchanges are used:

- MAC protocol data unit exchanges (MPDU)
- application protocol of an application data unit exchanges (APDU)
- LoRaWAN protocol physical protocol data unit layer (PPDU)

Figure 4. **LoRaWAN network protocols**

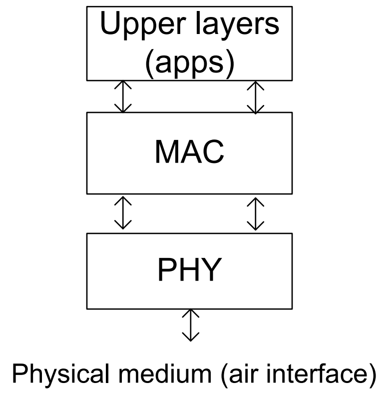


2.4.1 Network layer

The LoRaWAN architecture is defined in terms of blocks called layers. As shown in the figure below, each layer is responsible for one part of the standard and offers services to higher layers. An end device includes the following elements:

- PHY: embeds the radio frequency transceiver
- MAC sublayer: provides access to the physical channel
- application layers: provide access to the LoRaWAN services protocol

Figure 5. **LoRaWAN layers**



2.4.2 Physical layer (PHY)

The physical layer provides the following services:

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

2.4.3 MAC layer

The MAC layer provides the following services:

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

2.4.4 Application layer

The application layer provides several messaging packages running over the LoRaWAN protocol. FUOTA scopes the following ones:

- Remote multicast setup package (Port 200)
  - remotely creates a multicast group security context inside a group of end devices
  - reports the list of multicast contexts existing in the end device
  - remotely deletes a multicast security context
  - programs a Class-C multicast session
  - programs a Class-B multicast session
- Fragmented-data-block transport package (Port 201)
  - sets up, reports and deletes fragmentation transport sessions
  - may support several fragmentation sessions simultaneously for an end device
  - can be used either over multicast or unicast
  - reports the status of a fragmentation session
- Clock synchronization package (Port 202)
  - synchronizes the end-device real-time clock to the network GPS
  - makes all end devices of a multicast group to switch to Class C temporarily and synchronously
- Firmware management package (Port 203) - (proof-of-concept implementation only)
  - queries the firmware version running on an end device (including availability of the firmware update version)
  - queries the end-device hardware version
  - manages the end-device reboot at a given time
- Update agent module
  - interfaces a LoRaWAN stack block to an SBSFU block
  - gets, recombines and stores the complete file in the Download Image Slot, before the SFU execution of the SBSFU (called by NVIC_Reset action)
- User application
  - Sensor/actuator processing – application use cases
  - required to start a FUOTA session, with some user uplinks to open useful Rx windows for the packages described above

2.5 Network/end-device interworking

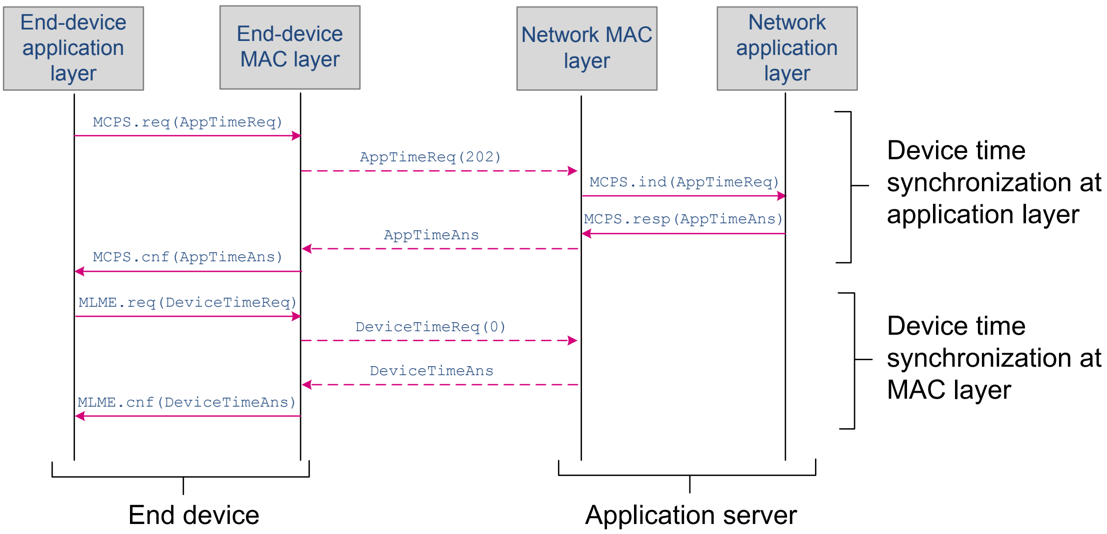
This section only shows the information flow between the end device and the application server, at the application-layer level, during a FUOTA campaign. For a complete view and description of the end device and network interactions, see document [7].

Multicast and fragmentation setup are detailed below.

2.5.1 Time synchronization

Before setting up a FUOTA session, the end device must have synchronized its timing with the network, using either `AppTimeReq` or `DeviceTimeReq` as shown in the figure below.

Figure 6. MSC for device timing



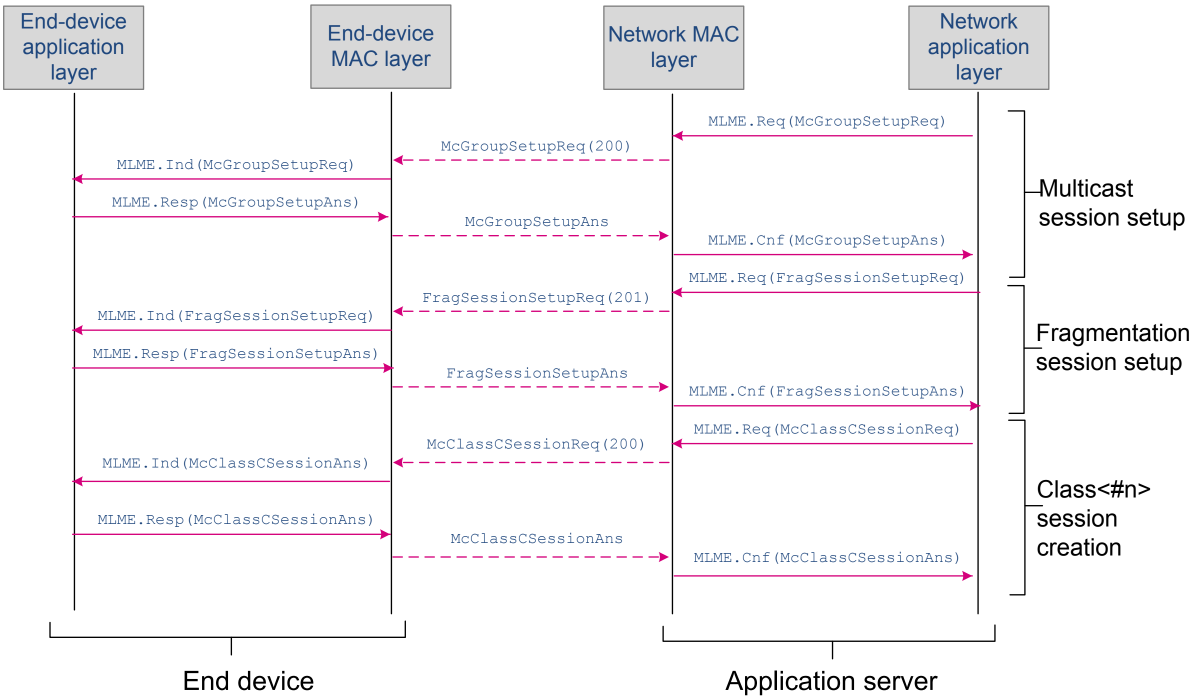
Note: For the purposes of this presentation, the `TimeReq` sent by the MSC is divided into `DeviceTimeReq` and `AppTimeReq` parts. The LoRaWAN specification allows a MAC command to be piggybacked in an application payload. In the current implementation, `DeviceTimeReq` is piggybacked in the `AppTimeReq` payload.

2.5.2 Multicast, fragmentation setup and session creation (Class C only)

To receive a data block at the application level, it is necessary to have some exchanges between the network application layer and the end-device application layer. These exchanges are mainly to define the following:

- a multicast group ID
- fragmentation parameters (frag number and frag size)
- multicast Class C session (start time and end time)

Figure 7. MSC for Class-C creation

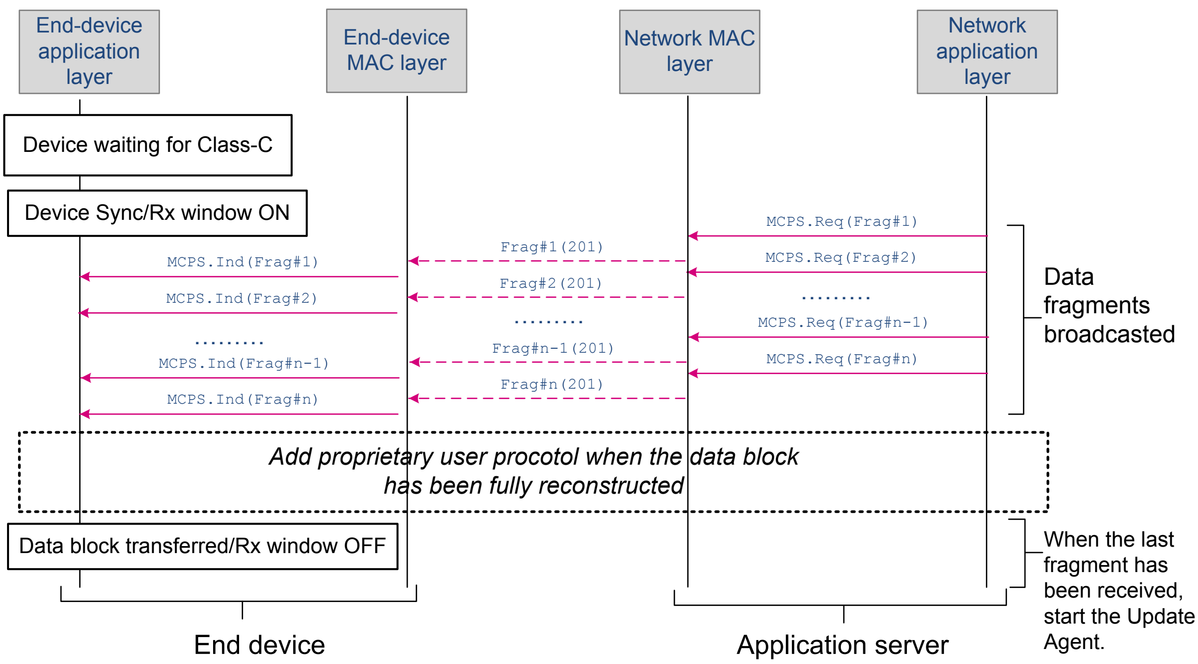


2.5.3 Fragment broadcasting and secure firmware update process

As soon as the end device is synchronized (Class C), it opens its Rx window to receive the data fragments (see document [3]). The end device stays in this state until all the data fragments are received.

When the complete data block (see the note below) is received, the end device closes its Rx windows, and, if everything is OK from the 'data-block transfer' point-of-view, the end device calls the Update Agent to start the SFU process.

Figure 8. MSC for data-block broadcasting



Note: *Additional user 'proprietary' protocol statement:*

The V1.0 package, and particularly Fragmented Data Transport specification [TS-004] (see document [3]), does not provide a way to inform the server that all data blocks have been properly received in order to rebuild the current download file. This is the case in the currently proposed implementation. The server always sends all the fragments (uncoded and coded), even if the current download file has been rebuilt before the end of the complete broadcast fragmentation transaction.

If needed, the user is responsible for implementing a 'proprietary' protocol to avoid such behavior. For instance, when all the required fragments have been received and the current download file rebuilt, a simple crc32 can be computed and sent back to the server. The server should decide to stop broadcasting the remaining fragments.

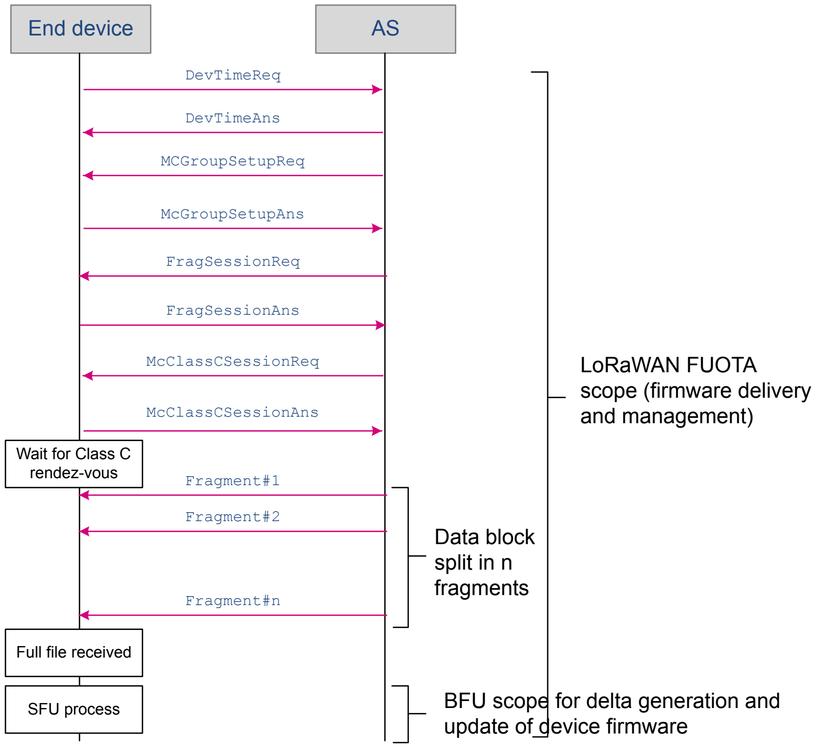
This approach requires cooperation between the device maker and the network operator to define the 'proprietary' part of the protocol.

3 FUOTA campaign

The MSC in the figure below presents the whole FUOTA communication exchange between the STM32WL end device and the application server (AS). The FUOTA campaign is divided into the following parts:

- FUOTA session setup between AS and end device
- fragment session exchange
  The file to be transferred is divided into N data fragments (called data block in the MSC).
- firmware update authentication check, firmware reboot and swap on new firmware

Figure 9. MSC for FUOTA campaign



The BFU allows the update of the device with a new firmware version, adding new features and correcting potential issues. The update process is performed in a secure way to prevent unauthorized updates.

3.1 How to create and manage a FUOTA campaign

This section does not show how to create a FUOTA campaign on an application server. These aspects of a FUOTA campaign depend on the services provided by the network operator. Only the salient points relating to FUOTA campaign support are outlined in this section.

The application server must:

- support the following packages:
  – Clock Synchro package (TS-003) (see document [2])
  – Fragmentation package (TS-004) (see document [3])
  – Multicast Setup package (TS-005) (see document [4])
- support Class-C mode (as defined in the LoRaWAN specifications V1.0.3 or V1.0.4, see document [1])
- be compliant with the 'Interop test' proposed by the FUOTA working group of the LoRa Alliance
  The 'Interop test' is the minimum test proving that the end device is able to receive a data block file from the server. This minimum test is shown in Section 2.5 .
- have the capability to manage the data block (firmware image) to be downloaded

3.2 Reconstruction of missing fragments

During the FUOTA campaign, some LoRaWAN frames may be lost to the end device (following reception problem, frame corruption, or server lag for example).

To face this problem, the AS sends redundancy fragments to the end device after the whole uncoded fragments have been sent. The end device continues its FUOTA session to retrieve these redundancy fragments and try to reconstruct the missing fragments thanks to the redundancy ones.

3.2.1 Generation of redundancy fragments

The AS runs an algorithm that generates as many redundancy fragments as fragments of firmware update file sent. A maximum number of redundancy fragments, equivalent to the number of uncoded fragments, can be sent at the end of each FUOTA campaign.

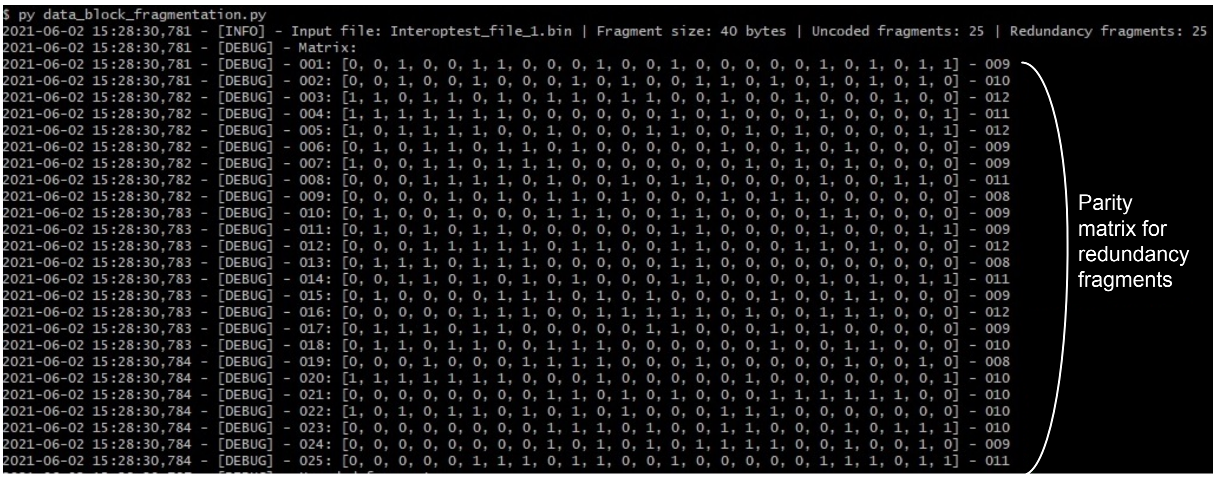
The limitation to a smaller amount of redundancy fragments sent is on the AS responsibility.

Section Appendix A details the Python script of the algorithm generating redundancy fragments.

To simplify, the use case below has been run on an interoperability file (1 Kbyte), that is composed of 25 fragments of 40 bytes. The algorithm performed by the Python script generates also 25 redundancy fragments of 40 bytes (100% of redundancy fragments).

Each redundancy fragment is displayed on each line, as shown in the figure below.

Figure 10. Redundancy fragments



The Python script displays the parity matrix (25 x 25 in this use case). Each redundancy fragment is composed of some of the sent fragments XOR between themselves.

Example:

Redundancy fragment 1 = frag3 ⊕ frag6 ⊕ frag7 ⊕ frag11 ⊕ frag14 ⊕ frag20 ⊕ frag22 ⊕ frag 24 ⊕ frag25

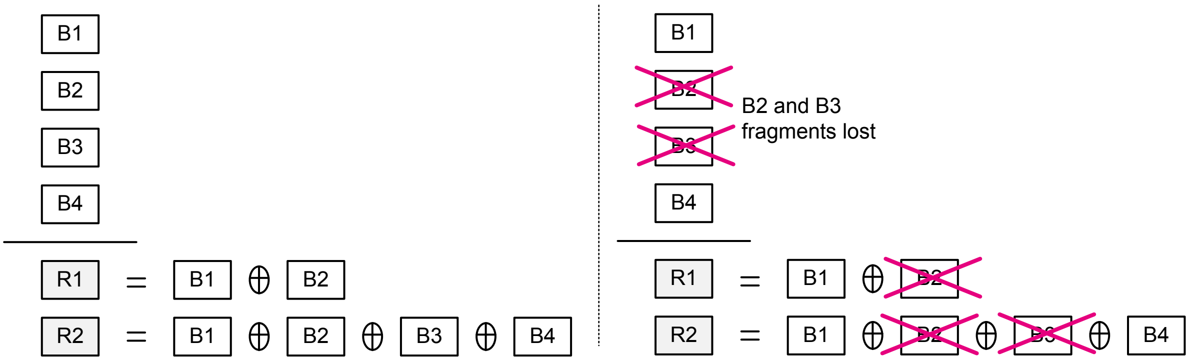
Warning: *Below a certain percentage of redundancy fragments (depending on the information sent), the decoder is not able to retrieve the missing fragments.*

3.2.2 Reconstruction algorithm

As an example of FEC algorithm, the LDPC has been implemented in the LoRaWAN middleware stack in `FragDecoderProcess()` function from `FragDecoder.c` file (see [3] for more details).

The LDPC is used to retrieve the missed fragments from the redundancy fragments sent after the whole firmware image to update.

Figure 11. **Reconstruction algorithm principle**



Example

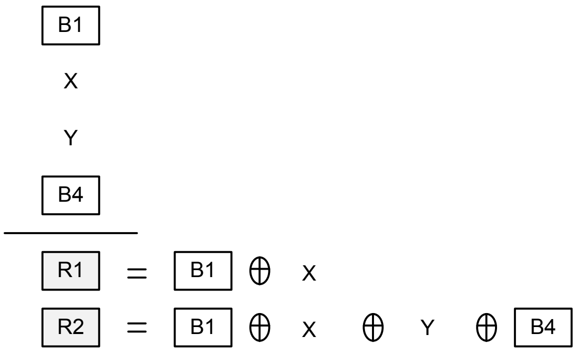
Four fragments (B1 to B4) and two redundancy fragments (R1 and R2) are sent, depicted in the figure below. In case B2 and B3 are missing fragments on end-device side (lost due to bad communication), the redundancy blocks contain the information of several blocks:

- R1 contains B1 and B2 .
- R2 contains B1, B2, B3, and B4 information.

Note: *The operation between B1 and B2 to obtain R1 is a simple XOR, very convenient and easy to compute.*

Considering X as B1 and Y as B2, the two fragments to be recovered, the problem can be expressed as in the figure below.

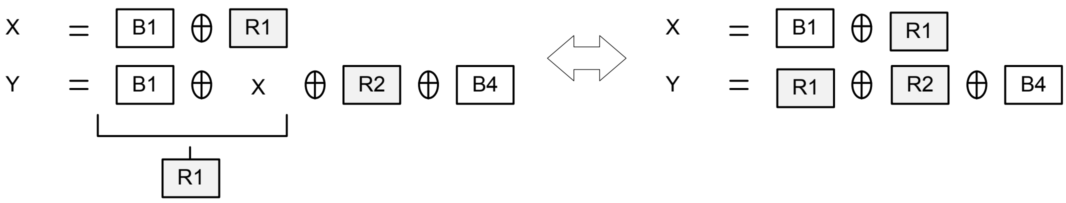
Figure 12. **FEC reconstruction example**



Due to XOR associated operations:

- X is recovered from a XOR between B1 and R1.
- Y is recovered from a XOR between R1, R2, and B4.

Figure 13. **FEC reconstruction of lost fragments**



Finally, this simple example shows that missing fragments B1 and B2 can be recovered by the combination of other fragments and redundancy fragments.

3.2.3 Simulation and validation of lost fragments

The LDPC algorithm can be tested with the following steps:

- Simulate a loss of frames by an incomplete send of all the uncoded and coded fragments at different positions.
- Check that, depending on the redundancy fragments received, the initial data block can be recovered.

More details are provided in Section Appendix B .

3.3 End-device design choices

3.3.1 Fragmentation decoder definitions

The fragmentation decoder process contains several defines that need to be sized depending on the application (details in `LoRaWAN_End_Node\LoRaWAN\Target\frag_decoder_if.h`):

- FRAG_MAX_SIZE: maximum fragment size that can be handled. It must be inferior or equal to the maximum MACPayload size length (M) of supported regions with maximum datarate (DR).
- FRAG_MAX_NB: maximum number of fragments that can be handled. It is obtained thanks to the formula:

```
FRAG_MAX_NB = SLOT_DWL_SIZE/ FRAG_MAX_SIZE
with SLOT_DWL_SIZE = SLOT_DWL_1_END -
SLOT_DWL_1_START
```

  SLOT_DWL is the download area allocated in flash memory to store the new firmware image retrieved with FUOTA download.
- FRAG_MIN_SIZE: minimum fragment size that can be handled. It must be superior or equal to the minimum MACPayload size length (M) of supported regions with minimum datarate (DR). FRAG_MIN_SIZE must be also inferior or equal to FRAG_MAX_SIZE.
- FRAG_MAX_REDUNDANCY: maximum number of redundancy fragments that can be handled. In the STM32CubeWL FUOTA applications, the memory is allocated for a maximum of 10% redundancy fragments (design choices on end-device side).

Note: In a FUOTA campaign, if the number of lost fragments is higher than the 10% redundancy fragments retrieved by the end-device, the decoder aborts the current fragmentation session in the end-device.

In STM32CubeWL firmware, FUOTA must be performed on flash memory. Due to conception constraint, any copy of flash memory blocks must be a multiple of eight, in order to avoid erasing a full flash memory sector at each data write. This applies to FRAG_MAX_SIZE and FRAG_MIN_SIZE.

Every definition corresponds to the worst-case value for the used region and multicast session data rate configurations.

3.3.2 Fragmentation decoder implementation

This section details how the four previous defines have been determined in STM32CubeWL FUOTA applications.

3.3.2.1 FRAG_MAX_SIZE computation

FRAG_MAX_SIZE worst-case (region and DR dependent) is equal to FRAG_MAX_SIZE = 240.

The critical criterion is the RAM use, which is defined at compilation, and used by fragmentation decoder process depending on these defines. The simplified fragmentation decoder RAM use equation is defined below:

$$FragRam = 2 \times N + S + ((R >> 3) + 1) \times (R + 4) + 18$$

where:

N = FRAG_MAX_NB

S = FRAG_MAX_SIZE

R = FRAG_MAX_REDUNDANCY

3.3.2.2 FRAG_MAX_REDUNDANCY computation

The number of redundancy fragments are defined as 10% of the maximum number of fragments for the FUOTA campaign, which means:

$$FRAG\_MAX\_REDUNDANCY = FRAG\_MAX\_NB \times 10\,\%$$

3.3.2.3 *FRAG_MAX_NB and FRAG_MIN_SIZE computation*

FragRam is composed of

$$FragDecoder\_t + matrix + dataTemp$$

where matrix is composed of

$$matrixRow + matrixDataTemp$$

and dataTemp is composed of

$$dataTempVector + dataTempVector2$$

For more details, refer to `Middlewares\Third_Party\LoRaWAN\LmHandler\Packages\FragDecoder.c`.

FUOTA single core fragmentation decoder implementation example

- In FUOTA single core, SLOT_DWL size is 86016 bytes.
- The best tradeoff is FRAG_MIN_SIZE = 40 and FRAG_MAX_NB = 2151.

Note: If FRAG_MIN_SIZE= 16 and FRAG_MAX_NB = 5376, the remaining RAM application is drastically reduced.

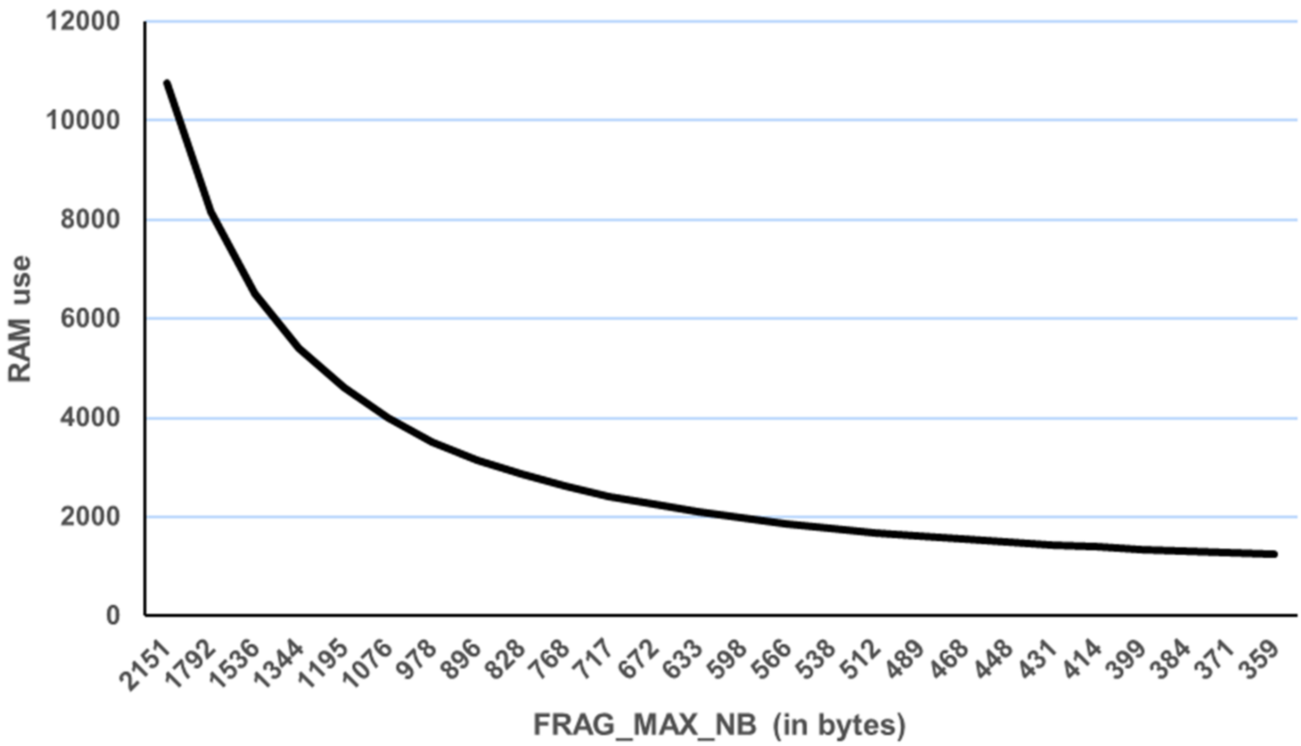
All these values are computed with FRAG_MAX_SIZE = 240 (see Section 3.3.2.1 FRAG_MAX_SIZE computation).

Table 3. FUOTA single core RAM memory requirements

-	Frag decoder configuration			RAM memory requirement			
	FRAG_MIN_SIZE	FRAG_MAX_NB	FRAG_MAX_REDUNDANCY	FragDecoder_t	matrix	dataTemp	FragRam
Not enough memory	8	10752	1076	167456	1353	271	169312
OK but very reduced memory left	16	5376	538	47557	689	137	48607
	24	3584	359	23701	473	92	24482
	32	2688	269	14743	369	70	15390

-	Frag decoder configuration			RAM memory requirement			
-	FRAG_MIN_SIZE	FRAG_MAX_NB	FRAG_MAX_REDUNDANCY	FragDecoder_t	matrix	dataTemp	FragRam
OK sufficient memory left	40	2151	216	10396	310	56	10962
	48	1792	180	7856	273	47	8368
	56	1536	154	6229	249	41	6703
	64	1344	135	5137	233	36	5582
	72	1195	120	4344	223	32	4767
	80	1076	108	3751	216	29	4156
	88	978	98	3286	212	27	3677
	96	896	90	2925	209	25	3303
	104	828	83	2630	209	23	2998
	112	768	77	2383	209	22	2742
	120	717	72	2182	211	20	2533
	128	672	68	2018	213	19	2362
	136	633	64	1869	217	18	2208
	144	598	60	1733	220	17	2066
	152	566	57	1622	224	17	1951
	160	538	54	1521	229	16	1846
	168	512	52	1440	233	15	1760
	176	489	49	1353	239	15	1671
	184	468	47	1284	244	14	1598
	192	448	45	1219	249	14	1530
	200	431	44	1173	255	13	1481
	208	414	42	1115	261	13	1421
	216	399	40	1062	267	12	1365
	224	384	39	1021	273	12	1322
	232	371	38	985	280	12	1285
	240	359	36	940	286	11	1237

Figure 14. **RAM use vs FRAG_MAX_NB (single core)**



To conclude for FUOTA single core, the following implementation is proposed:

```
#define FRAG_MAX_SIZE                  240
#define FRAG_MAX_NB                    2151
#define FRAG_MIN_SIZE                  40
#define FRAG_MAX_REDUNDANCY            216
```

Note: *All previous values are the consequence of the SLOT_DWL size and the region/DR configuration.*

For more information, refer to document [3].

FUOTA dual core fragmentation decoder example

- In FUOTA dual core, SLOT_DWL size is 65536 bytes.
- The best tradeoff is FRAG_MIN_SIZE = 48 and FRAG_MAX_NB = 1366.

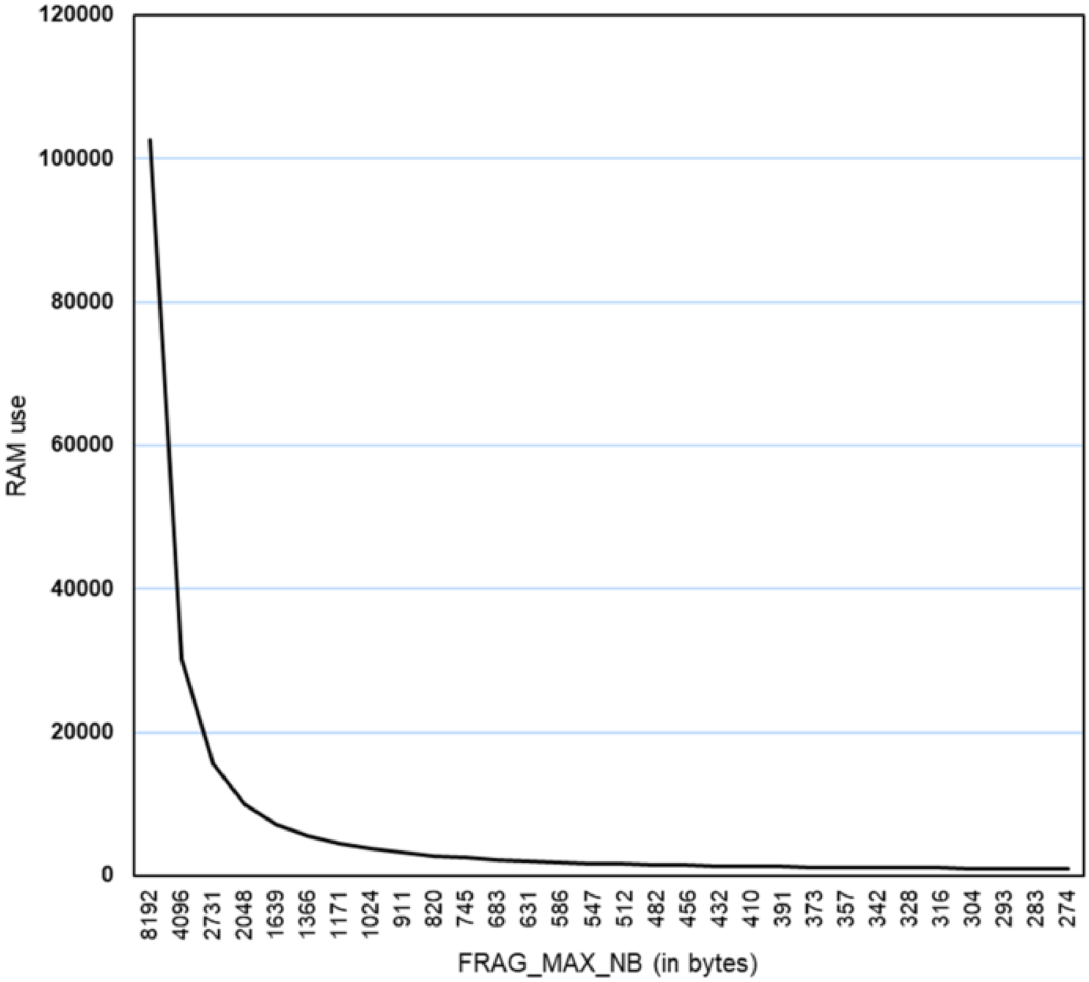
Warning: *STM32CubeIDE toolchain requires 1.2 Kbytes more (private SECore bin stack) which must be aligned to 2-Kbyte boundary due to TZIC security. Consequently the highest values are FRAG_MIN_SIZE = 96 and FRAG_MAX_NB = 683.*

For a harmonization purpose, this setting has been chosen for all toolchains. With these settings, APPLI_RAM and APPLI_ROM memory areas are almost full.

Table 4. FUOTA dual core RAM memory requirements

-	Frag decoder configuration			RAM memory requirement			
	FRAG_MIN_SIZE	FRAG_MAX_NB	FRAG_MAX_REDUNDANCY	FragDecoder_t	matrix	dataTemp	FragRam
Not enough memory	8	8192	820	101376	1033	207	102616
	16	4096	410	29685	529	105	30319
	24	2731	274	15174	367	71	15612
	32	2048	205	9599	289	54	9942
	40	1639	164	6844	246	43	7133
Not enough memory for STM32CubeIDE only	48	1366	137	5252	220	37	5509
	56	1171	118	4235	204	32	4471
	64	1024	103	3509	193	28	3730
	72	911	92	3003	187	25	3215
	80	820	82	2592	184	23	2799
	88	745	75	2297	183	21	2501
OK for all toolchains	96	683	69	2058	183	20	2261
	104	631	64	1865	184	18	2067
	112	586	59	1693	187	17	1897
	120	547	55	1553	190	16	1759
	128	512	52	1440	193	15	1648
	136	482	49	1339	198	15	1552
	144	456	46	1248	202	14	1464
	152	432	44	1175	207	13	1395
	160	410	41	1096	213	13	1322
	168	391	40	1046	218	12	1276
	176	373	38	989	224	12	1225
	184	357	36	936	230	11	1177
	192	342	35	896	236	11	1143
	200	328	33	849	242	11	1102
	208	316	32	815	249	10	1074
	216	304	31	782	255	10	1047
	224	293	30	752	262	10	1024
	232	283	29	723	269	10	1002
	240	274	28	697	276	9	982

Figure 15. **RAM use vs FRAG_MAX_NB (dual core)**



To conclude for FUOTA dual core, the following implementation is proposed:

```
#define FRAG_MAX_SIZE                   240
#define FRAG_MAX_NB                     683
#define FRAG_MIN_SIZE                   96
#define FRAG_MAX_REDUNDANCY             69
```

Note: *All previous values are the consequence of the SLOT_DWL size and the region/DR configuration.*
For more information, refer to document [3].

4 LoRaWAN_FUOTA project overview

4.1 Single-core project overview

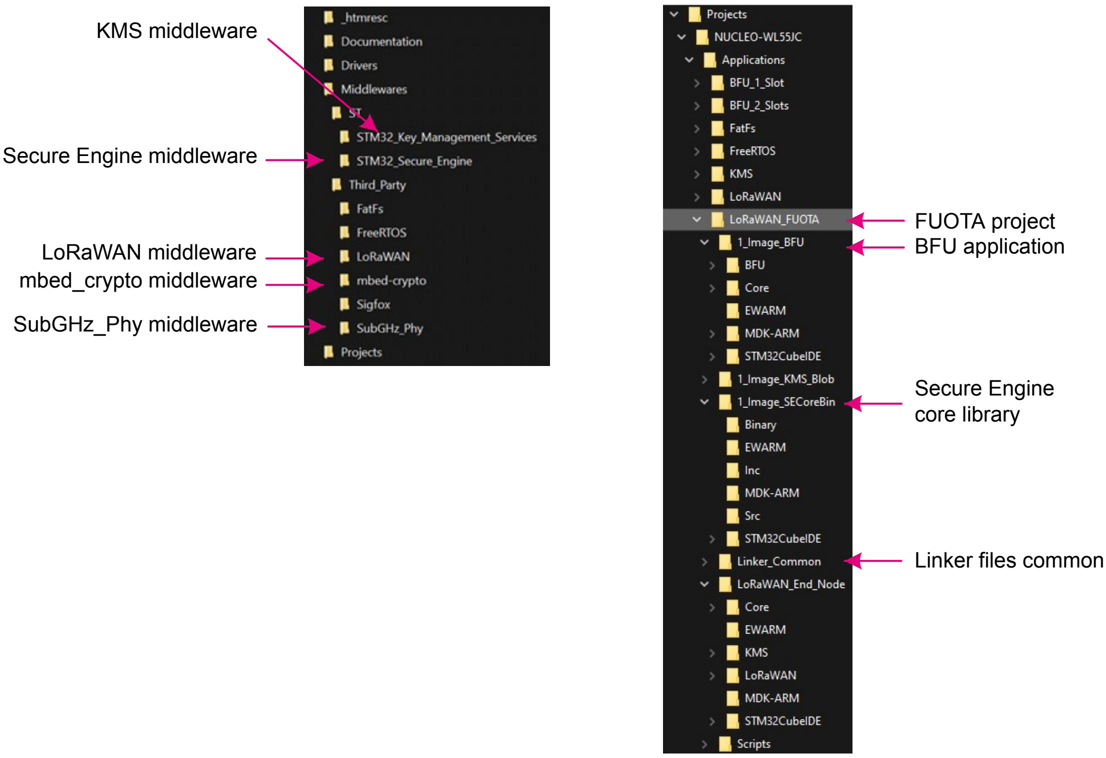
LoRaWAN_FUOTA is a single-core project based on BFU_2_Slots solution for the STM32WLE5xx, STM32WL55xx, or STM32WL5MOCHxx microcontrollers series.

This LoRaWAN_FUOTA project is split into three main subprojects:

- 1_Image_SECoreBin (generating the Secure Engine core binary file to be linked with the BFU application)
- 1_Image_BFU (updates of the MCU built-in program with new firmware versions)
- LoRaWAN_End_Node (End_Node application tailored for FUOTA)

The middleware is provided in source-code format and is compliant with the STM32CubeWL HAL driver.

Figure 16. Single-core project file structure



The other directories of the LoRaWAN_FUOTA project are more specific:

- `Linker_Common`: generates linker files shared between the three projects:
  - `mapping_fwimg.icf` contains firmware image definitions such as active slots, download slots, and swap area.
  - `mapping_sbsfu.icf` contains BFU definitions such as SE_Code_region, SE_Key_region, and SE_IF_region.
  - `mapping_export.h` exports the symbols from `mapping_sbsfu.icf` and `mapping_fwimg.icf` to the BFU applications.
- `Scripts`: automatic scripts to build all projects in a specific order and to program the final all-in-one binary file

1_Image_KMS_Blob generates KMS blob binary file to be downloaded with KMS through the `ImportBlob()` API. This feature is present but not used in this project.

4.2 Dual-core project overview

Two DualCore applications are available based on SBSFU_2_Slots_DualCore solution:

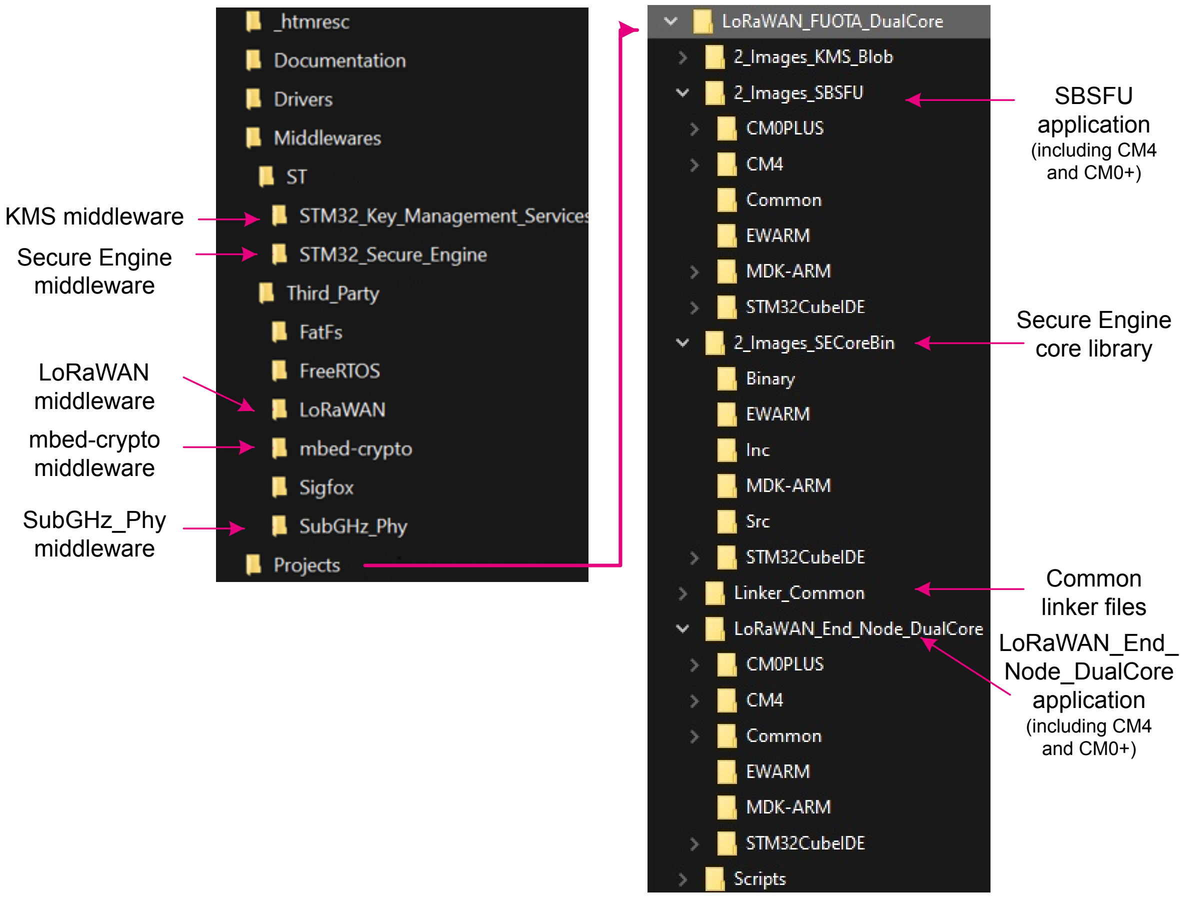
- LoRaWAN_FUOTA_DualCore for the STM32WL55xx MCUs
- LoRaWAN_FUOTA_DualCore_ExtFlash for the STM32WL5MOCHxx MCUs

Theses LoRaWAN_FUOTA_DualCore projects are split into five main subprojects:

- 2_Images_SECoreBin
- 2_Images_SBSFU [CM4 and CM0PLUS]
- LoRaWAN_EndNode_DualCore_[CM4 and CM0PLUS]

The middleware is provided in source-code format and is compliant with the STM32CubeWL HAL driver.

Figure 17. Project file structure



4.3 BFU and SBSFU features

4.3.1 BFU single-core configuration

This section is dedicated to single-core FUOTA and BFU features. For more information concerning SBSFU and dual-core security features, Secure Boot and SKMS features, refer to document [8].

The BFU allows the update of the STM32 microcontroller built-in program with new firmware versions, adding new features, and correcting potential issues. The update process is performed in a secure way to prevent unauthorized updates.

Boot (root-of-trust services)

- checks and activates the STM32 security mechanisms to protect critical operations and secret data from an attack

- checks the authenticity and integrity of the user application code before every execution, to ensure that an invalid or malicious code cannot be run

Firmware Update (FU)

- detects the new (encrypted) firmware version to install, pre-downloaded over-the-air via the user application (LoRaWAN)
- manages the firmware version by checking for unauthorized update/installation
- decrypts the firmware (if encryption activated)
- checks the firmware authentication and integrity
- installs the firmware
- recovers the firmware image if any error occurrence during the new image installation (rollback to the previous valid firmware version not supported)
- executes the installed firmware (once authenticated and integrity checked)

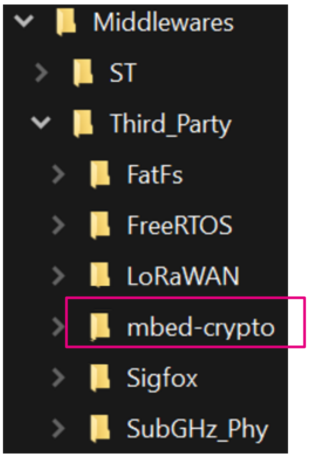
Key management services (KMS)

- provides cryptographic services to the user application, through the PKCS #11 APIs
- provides cryptographic services to the SFU to authenticate the user application with some protected keys

BFU cryptographic middleware

The BFU for STM32CubeWL supports the mbed-crypto (open-source code) cryptographic services for SHA256.

Figure 18. **Cryptographic library structure**



BFU cryptographic schemes

The BFU for STM32CubeWL is delivered with the following cryptographic schemes, using symmetric and asymmetric cryptographic operations:

- ECDSA asymmetric cryptography for firmware verification without firmware encryption
- ECDSA asymmetric cryptography for firmware verification, and AES-CBC symmetric cryptography for firmware decryption
- AES-GCM symmetric cryptography, for both firmware verification and decryption

By default, the LoRaWAN_FUOTA project is configured with asymmetric cryptography. The firmware authentication, integrity, and confidentiality (encryption) are ensured.

Figure 19. File structure of cryptographic scheme

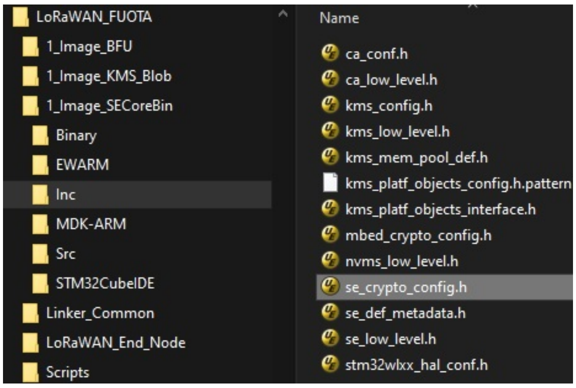


Figure 20. Default cryptographic scheme

```
#define SECBOOT_CRYPTO_SCHEME SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256

#define SECBOOT_ECCDSA_WITHOUT_ENCRYPT_SHA256
#define SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256
#define SECBOOT_AES128_GCM_AES128_GCM_AES128_GCM
```

BFU application features

Various configuration possibilities are offered through option compilation switches (due to the optimized memory mapping, several options are set by default):

- All the following algorithms are enabled:
  - AES CBC and AES GCM used to perform encryption and decryption
  - AES ECB used to perform encryption decryption and key derivation
  - AES CMAC used to perform signature and verification
  - ECDSA used to perform verification

Figure 21. File structure of cryptographic definition

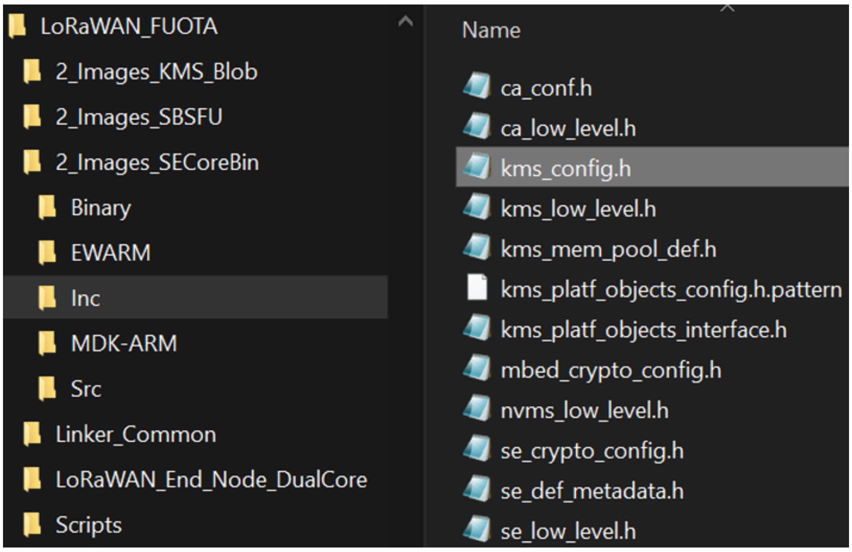


Table 5. Default features

Feature	Status
AES CBC algorithm support	Encryption and decryption
AES CCM algorithm support	No
AES ECB algorithm support	Encryption, decryption and key derivation
AES CGM algorithm support	Encryption and decryption
AES CMAC algorithm support	Signature and verification
RSA algorithm support	No
RSA algorithm	Not activated
RSA 1024-bit modulus length	No
RSA 2048-bit modulus length	No
ECDSA algorithm support	Verification
ECDSA algorithm	Activated and associated to an elliptic curve
Elliptic curve SECP-192	No
Elliptic curve SECP-256	Yes
Elliptic curve SECP-384	No
SHA1 digest algorithm	No
SHA256 digest algorithm	Digest

All security peripherals can be enabled in the `app_sfu.h` configuration file, where a general define can be used to enable/disable all security peripherals at once as shown below.

Figure 22. File structure of general security configuration

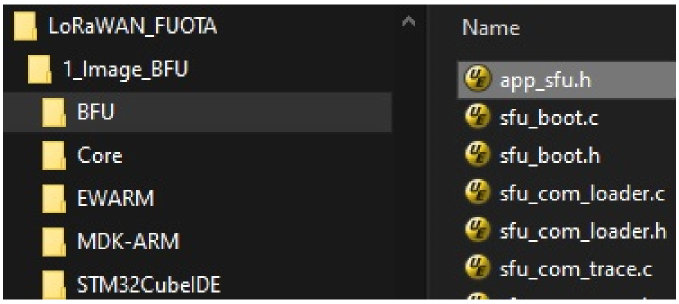


Figure 23. General security disable flag

```
/* The define below allows disabling all security IPs at once.
 *
 * Enabled: all security IPs (WRP, watchdog...) are disabled.
 * Disabled: the security IPs can be used (if their specific compiler switches are enabled too).
 *
 */

#define SECBOOT_DISABLE_SECURITY_IPS  /*!< Disable all security IPs at once when activated */
```

4.3.2 SBSFU dual-core specific configuration

The main difference between BFU and SBSFU is that the SBSFU embeds all available security features, whereas BFU allows limited security features.

The SBSFU allows the update of the STM32 microcontroller built-in program with new firmware versions, adding new features and correcting potential issues. The update process is performed in a secure way to prevent unauthorized updates and access to confidential on-device data (such as secret code and firmware encryption key):

- no local loader inside the SBSFU application (firmware update only possible through OTA)
- no debug mode (no more information displayed on the terminal during the SBSFU execution)

4.4 LoRaWAN features

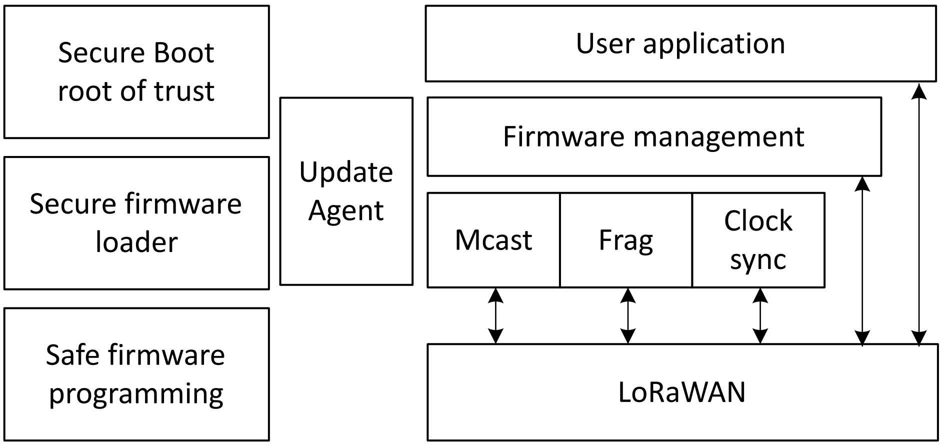
The main LoRaWAN features are listed below:

- LoRaWAN L2 (link layer) V1.0.3 or V1.0.4: Class A (baseline), Class C (continuous) and Class B (beacon)
- LoRaWAN RP (regional parameters) V1.0.3 or RP002-1.0.1
- LoRaWAN additional packages:
  - v1.0.0 packages include:
    - Application Layer Clock Synchronization v1.0.0
    - Remote Multicast Setup v1.0.0
    - Fragmented Data Block Transport v1.0.0
  - v2.0.0 packages include:
    - Application Layer Clock Synchronization v2.0.0
    - Remote Multicast Setup v2.0.0
    - Fragmented Data Block Transport v2.0.0
    - Firmware Management Protocol v1.0.0

4.5 Firmware architecture

The figure below summarizes the firmware design and the components involved in an end device supporting the FUOTA feature.

Figure 24. **Top-level firmware design**



5 LoRaWAN middleware programming guidelines

This section describes the LoRaMAC handler APIs.

5.1 LoRaWAN middleware initialization

`LmHandlerInit` initializes the LoRaMAC layer. This function initializes the callback system primitives of the MCPS and MLME services (see document [7]), and registers the following required packages:

- PACKAGE_ID_COMPLIANCE (mandatory)
- PACKAGE_ID_CLOCK_SYNC (*)
- PACKAGE_ID_REMOTE_MCAST_SETUP (*)
- PACKAGE_ID_FRAGMENTATION (*)
- PACKAGE_ID_FIRMWARE_MANAGEMENT (*)

Table 6. LmHandlerInit description

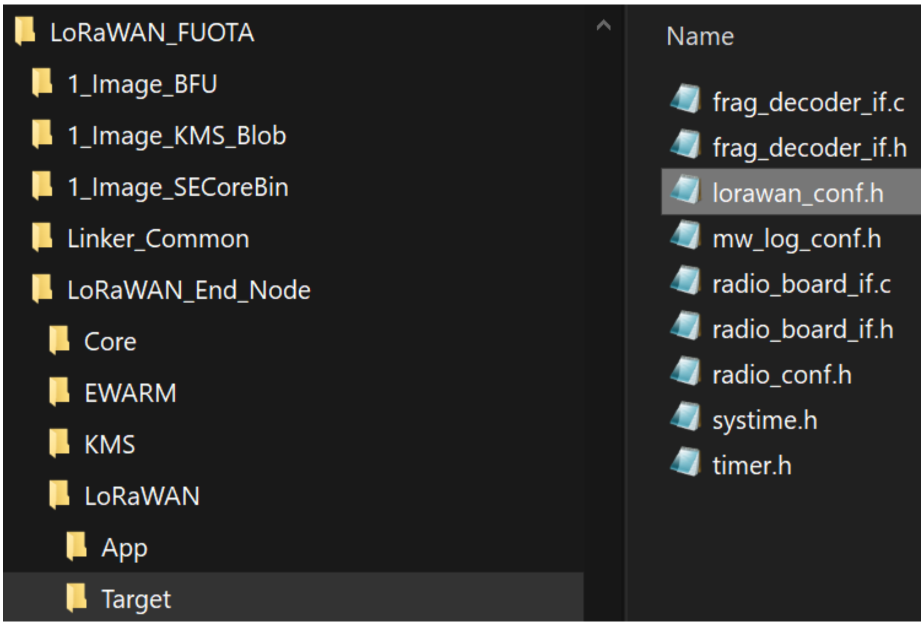
Function	Parameter	Description
`LmHandlerErrorStatus_t LmHandlerInit (…)`	`LmHandlerCallbacks_t *handlerCallbacks`	LoRaMAC handler initialization

Note: All packages (*) are hidden and disabled by default into the LoRaWAN middleware. It is necessary to add the following define to activate these features.

```
#define LORAWAN_DATA_DISTRIB_MGT 1
```

This constant is set into the `lorawan_conf.h` configuration file.

Figure 25. File structure of LoRaWAN configuration



5.2 LoRaWAN middleware configuration

`LmHandlerConfigure` configures the run-time LoRaMAC layers (such as active region or Tx parameters).

Table 7. LmHandlerConfigure description

Function	Parameter	Description
`LmHandlerErrorStatus_t LmHandlerConfigure (…)`	`LmHandlerParams_t *handlerParams`	LoRaMAC handler configuration

5.3 LoRaWAN middleware process

`LmHandlerProcess` processes the LoRaMAC and radio events.

Table 8. LmHandlerProcess description

Function	Parameter	Description
`void LmHandlerProcess (…)`	`void`	Processes the LoRaMAC and Radio events. Asks to go in low-power mode, when no pending operation.

5.4 LoRaWAN middleware start process

`LmHandlerJoin` runs the LoRaMAC layer with a `MLME JoinReq`, if OTAA mode is used. This run action requires to process periodically some uplink frames.

Table 9. LmHandlerJoin description

Function	Parameter	Description
`void LmHandlerJoin (…)`	`ActivationType_t mode`	Starts the LoRaMAC. For OTAA mode, performs a `JoinReq`. For ABP mode, this is a pass-through function.

5.5 LoRaWAN middleware stop process

`LmHandlerStop` stops the LoRaMAC layer execution and disables all the internal timers.

Table 10. LmHandlerStop description

Function	Parameter	Description
`LmHandlerErrorStatus_t LmHandlerStop (…)`	`void`	Stops a LoRa network connection.

5.6 LoRaWAN middleware send uplink frame

`LmHandlerSend` requests the LoRaMAC layer to send a Class A uplink frame.

Table 11. LmHandlerSend description

Function	Parameters	Description

`LmHandlerErrorStatus_t LmHandlerSend (…)`	`LmHandlerAppData_t *appData`
	`LmHandlerMsgTypes_t isTxConfirmed`
	`TimerTime_t *nextTxIn`
	`bool allowDelayedTx`
		Requests application data to be sent with an indication of whether the Tx is confirmed or unconfirmed.

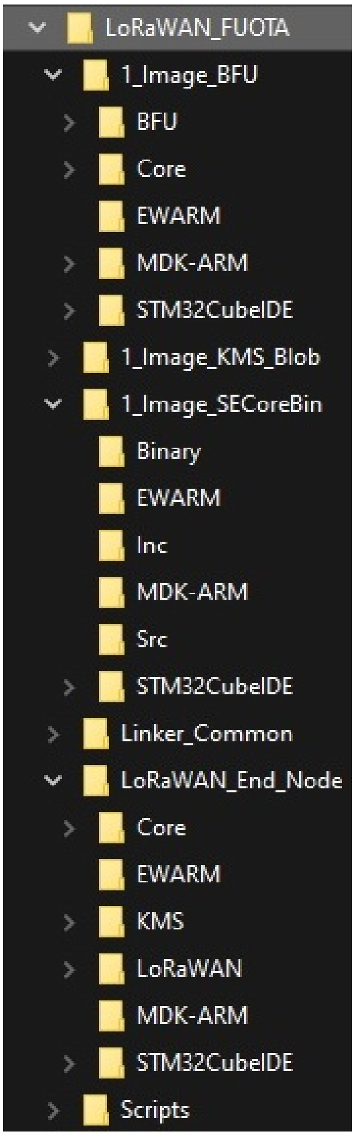
6 Getting started

This section is dedicated to single-core FUOTA firmware programming guide. For more information concerning the dual-core firmware programming guide, refer to document [8].

6.1 Single-core firmware programing guide

This section describes how to generate a FUOTA single-core application, and this flow must be followed step-by-step. The figure below shows a top-level view of the file structure.

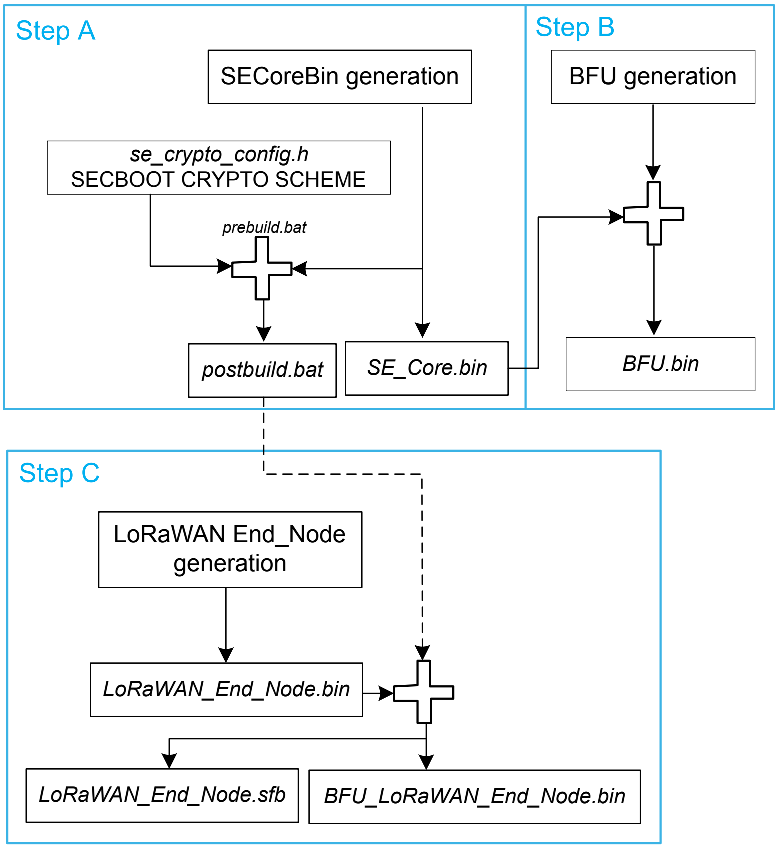
Figure 26. Project order structure



6.1.1 How to generate a FUOTA single-core application

The following steps must be followed to generate a FUOTA single-core application. For each step, open the associated subproject in the dedicated IDE folder, and regenerate the respective binary files.

Figure 27. Application generation steps



The following output binaries are generated in these steps (all of them in clear format, not encrypted):

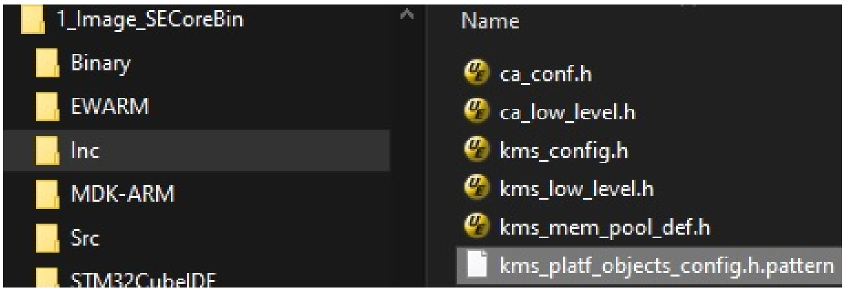
- `SE_Core.bin`
- `BFU.bin`
- `LoraWAN_End_node.bin`

In addition, the following output files are generated through the postbuild process:

- `LoraWAN_End_node.sfb` (`LoraWAN_End_node.bin` encrypted + header)
- `BFU_LoraWAN_End_node.bin` (three first binary files merged with the memory placement to produce the final memory image)

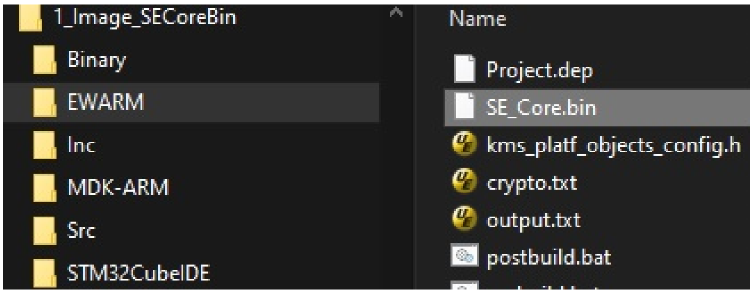
1. **1_Image_SECoreBin**
   This step is needed to create the SECoreBin binary that includes all the required “trusted” code and keys. The binary is linked to the BFU code in step B.
   The LoRaWAN keys are stored through the `kms_platf_objects_config.h.pattern` configuration file, using the `Commissioning.h` header file from LoRaWAN_End_Node project.

Figure 28. File structure of KMS user keys configuration



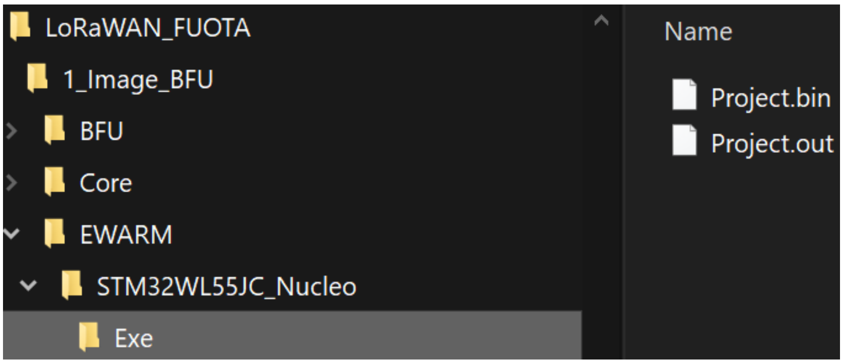
For more details about the KMS configuration, refer to the section 'Key Management Services' of the document [7].
The generated `SE_Core.bin` output file is located in the IDE folder.

Figure 29. File structure of SECoreBin output



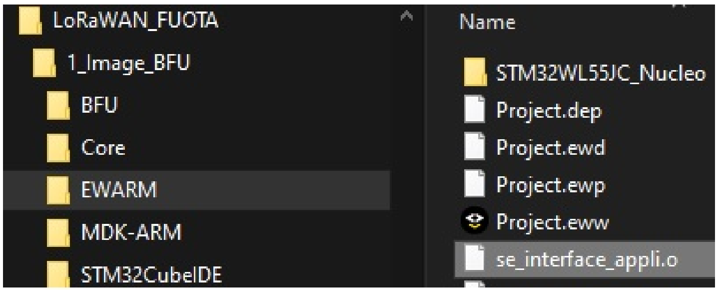
2. **1_Image_BFU**
This step compiles the BFU source code that implements the state machine protection configurations. This step links the code with the Secure Engine bin, including the “trusted” code.
The generated `Project.bin` output file is located in the IDE folder.

Figure 30. **File structure of BFU output**



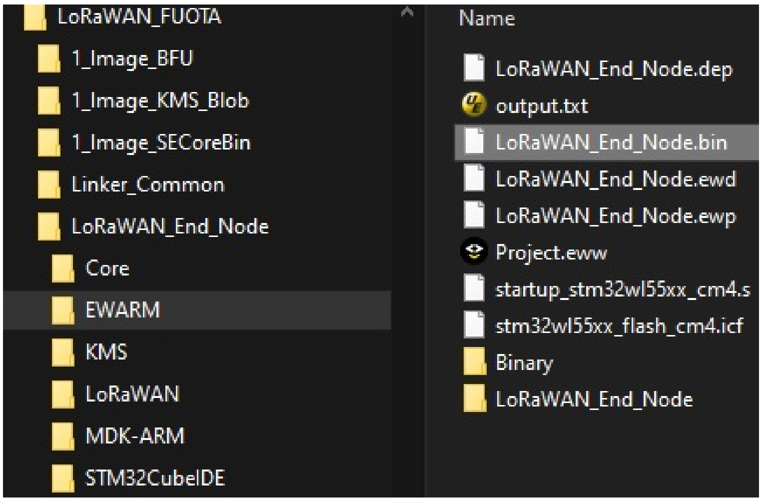
This step also generates a file that includes symbols used by the LoRaWAN_End_Node application to call the SE interface public functions.

Figure 31. **File structure of SE interface**



3. **LoRaWAN_End_Node**
This step compiles the LoRaWAN End_Node source code that implements the LoRaWAN middleware, the user application, and the sequence configuration.
The generated `LoRaWAN_End_Node.bin` output file is located in the IDE folder.

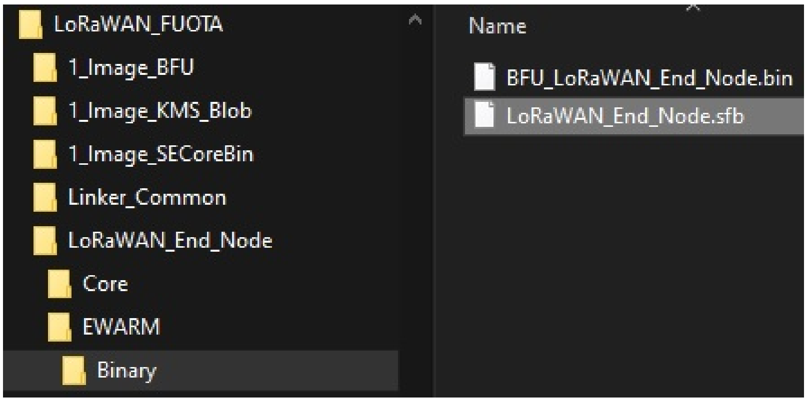
Figure 32. **File structure of LoRaWAN_End_Node output (1 / 2)**



This step also generates the following files:
- `LoRaWAN_End_Node.sfb` (user application binary in encrypted format including the SFU header.)
- `BFU_LoRaWAN_End_Node.bin` (final big binary that concatenates the BFU binaries and user application binaries in clear format)

`BFU_LoRaWAN_End_Node.bin` must be used to program the STM32WL55xx flash memory on the first use. `LoRaWAN_End_Node.sfb` must be used to generate a firmware update

Figure 33. **File structure of LoRaWAN_End_Node output (2 / 2)**



6.1.2 How to download the firmware to the end device (single-core configuration)

There are only three ways to download a firmware:

- through the STM32CubeProgrammer tool, using the final big binary `BFU_LoRaWAN_End_Node.bin`

> **Warning:** *This action requires to erase the full device flash memory, and to remove all security option bytes.*

- through local download via UART Virtual COM port
  LoRaWAN FUOTA single-core project offers the possibility to download the firmware update via Ymodem transfer procedure and TeraTerm tool. For more details refer to document [6].
- through the remote download, via FUOTA mechanisms proposed by the LoRaWAN protocol, using the `LoRaWAN_End_Node.sfb`.

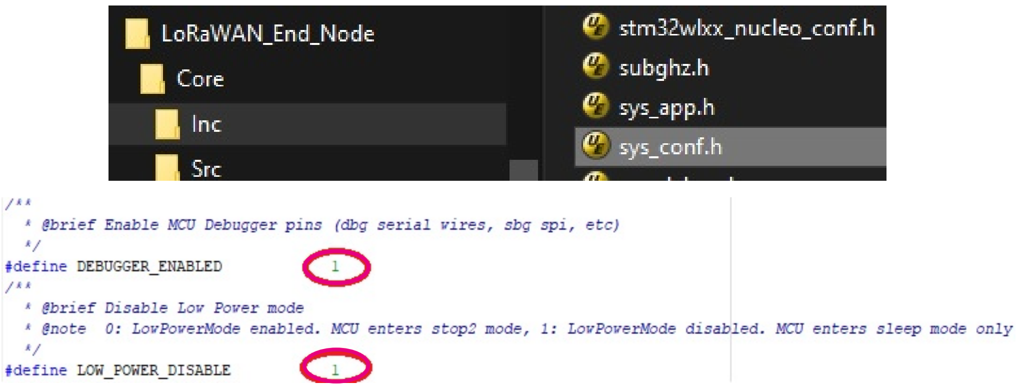
6.1.3 How to debug the End_Node application

The complete system consists of a Secure Boot and an End_Node application. When the target resets, the Secure Boot starts first. After a low-level initialization, the BFU starts and checks all required security steps. If the BFU does not detect any system error, the Secure Boot codes jump to the entry point of the application.

Unless general security has been disabled as explained in Section 4.3 , since the End_Node application is linked to the Secure Boot, the `BFU_LoRaWAN_End_Node.bin` cannot be downloaded directly with the debugger. To debug the End_Node application, the following steps must be respected:

1. Set the debugger and low-power defines on End_Node.

Figure 34. File structure of End_Node_DualCore debug configuration



2. Compile the End_Node projects as described in Section 6.1.1 .
3. Flash the target with the complete big binary `BFU_LoRaWAN_End_Node.bin`, using the STM32CubeProgrammer tool.
4. Once the target is flashed, the subproject can be attached to the running target in debug mode (with breakpoints, watch variables, and so on).

6.2 Dual-core specific programming guide

Only the dual-core programming specific to FUOTA application is described here. All other dual-core programming parts are detailed in document [8].

6.2.1 How to download the firmware to the end device (dual-core configuration)

Due to the memory optimization on the SBSFU project, there are only two ways to download a firmware:

- through the STM32CubeProgrammer tool, using the final big binary `SBSFU_LoRaWAN_End_Node_DualCore_CM4.bin`

Warning: *This action requires to erase the full device flash memory, and to remove all security option bytes.*

- through the remote download, via FUOTA mechanisms proposed by the LoRaWAN protocol, using `LoRaWAN_End_Node_DualCore_CM4.sfb` or `LoRaWAN_End_Node_DualCore_CM0Plus.sfb`.

As it is not possible to update the Cortex-M0+ and Cortex-M4 firmware in the same time, it is mandatory to create some modifications that do not lead to an execution error (like a missing function interface with a call between Cortex-M4 and Cortex-M0+).

6.2.2 How to automate the generate and load processes

Three scripts are available to automate the compilation of all SBSFU projects and the programming of the concatenate binary on the STM32WL55xx or STM32WL5MOCHxx flash memory: `build.bat`, `program.bat`, and `disable_security.bat`.

Figure 35. File structure of automated process scripts

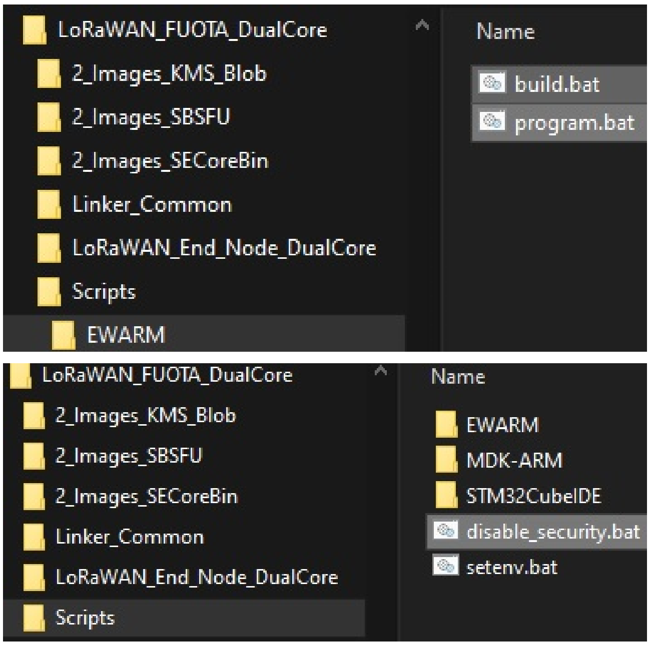


Table 12. Automated process scripts

Script	Description
Scripts\EWARM\build.bat	Compiles all project files with IAR Embedded Workbench, including prebuild.bat and postbuild.bat scripts, with the mandatory project order. The -app parameter is used to compile only the user application if the SBSFU projects are not modified.
Scripts\EWARM\program.bat	Runs the disable_security.bat script to remove the write-access protection. Programs the SBSFU_UserApp_M4.bin to the STM32WL55xx, using the STM32CubeProgrammer tool.
Scripts\disable_security.bat	Resets all option bytes to be compliant with a non-secure firmware (including a full erase memory).

Note: The path of the tools must be updated according to the versions and location of the user installations, by modifying the `Scripts\setenv.bat` *file content.*

6.3 Firmware configuration

6.3.1 Crypto switches

Table 13. Crypto switches

Location: LoRaWAN_FUOTA_DualCore\2_Images_SECoreBin\Inc\se_crypto_config.h

Symbols	Description
SECBOOT_ECCDSA_WITHOUT_ENCRYPT_SHA256	No firmware encryption Only authentication and integrity are ensured with asymmetric cryptography.
SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256	Authentication, integrity, and confidentiality are ensured with asymmetric cryptography.
SECBOOT_AES128_GCM_AES128_GCM_AES128_GCM	Authentication, integrity, and confidentiality are ensured with symmetric cryptography.

These switches are managed with an additional define given in the table below.

Table 14. Crypto default switch

Symbols	Description	Default state
SECBOOT_CRYPTO_SCHEME	Selected crypto scheme	SECBOOT_ECCDSA_WITH_AES128_CBC_SHA256

6.3.2 Security switches

The SBSFU instantiates the security item selected through SECBOOT_DISABLE_SECURITY_IPS. When this symbol is defined, the security protections (such as WRP, RDP, IWDG, DAP) are disabled for all peripherals (see the document [5]).

Single-core security switches

Table 15. Single-core security switches

Location: `LoRaWAN_FUOTA\1_Image_BFU\BFU\App\app_sfu.h`

Symbols	Description	Default state
SECBOOT_DISABLE_SECURITY_IPS	Disables all secure peripherals simultaneously when activated.	Enabled
SFU_WRP_PROTECT_ENABLE	Write-access protection to protect trusted code	Enabled
SFU_RDP_PROTECT_ENABLE	RDP protection	Enabled
SFU_DAP_PROTECT_ENABLE	DAP protection	Enabled
SFU_DMA_PROTECT_ENABLE	DMA access protection	Enabled
SFU_IWDG_PROTECT_ENABLE	IDWG protection	Disabled
SFU_MPU_PROTECT_ENABLE	MPU area protection	Enabled
SFU_MPU_USERAPP_ACTIVATION	User application MPU area protection	Enabled

Dual-core security switches

Table 16. Security common switches

Location: `LoRaWAN_FUOTA_DualCore\2_Images_SBSFU\Common\app_sfu_common.h`

Symbols	Description	Default state
SECBOOT_DISABLE_SECURITY_IPS	Disables all secure peripherals simultaneously when activated.	Disabled
SFU_WRP_PROTECT_ENABLE	Write-access protection to protect trusted code	Enabled
SFU_DAP_PROTECT_ENABLE	DAP protection	Enabled
SFU_DMA_PROTECT_ENABLE	DMA access protection	Enabled
SFU_IWDG_PROTECT_ENABLE	IDWG protection	Disabled
SFU_C2_DDS_PROTECT_ENABLE	Static CPU2 (Cortex-M0+)debug protection	Enabled
SFU_SECURE_USER_PROTECT_ENABLE	Secure user memory protection	Enabled
SFU_FINAL_SECURE_LOCK_ENABLE	Secure production protection	Disabled
SFU_HIDE_PROTECTION_CFG	Hide-protection area configuration	Enabled
OB_SECURE_SYSTEM_AND_FLASH	Flash memory and system secure area protection	Enabled
OB_SECURE_SRAM1	SRAM1 area protection	Disabled
OB_SECURE_SRAM2	SRAM2 area protection	Enabled

Table 17. Security Cortex-M4 switches

Location: `LoRaWAN_FUOTA_DualCore\2_Images_SBSFU\CM4\Inc\app_sfu.h`

Symbols	Description	Default state
SFU_MPU_PROTECT_ENABLE	MPU protection on Cortex-M4 regions	Enabled
SFU_MPU_USERAPP_ACTIVATION	User application memory protection during execution	Enabled

Table 18. Security Cortex-M0+ switches

Location: `LoRaWAN_FUOTA_DualCore\2_Images_SBSFU\CM0PLUS\SBSFU\App\app_sfu.h`

Symbols	Description	Default state
SFU_RDP_PROTECT_ENABLE	Read-access protection	Enabled
SFU_TAMPER_PROTECT_ENABLE	Tamper protection (hardware pin)	Disabled
SFU_MPU_PROTECT_ENABLE	MPU protection on Cortex-M0+ regions	Enabled
SFU_MPU_USERAPP_ACTIVATION	User application memory protection during execution	Enabled
SFU_GTZC_PROTECT_ENABLE	GTZC protection	Enabled
SFU_C2SWDBG_PROTECT_ENABLE	Dynamic CPU2 (Cortex-M0+) debug protection	Enabled

6.3.3 Debug switches

The End_Node_DualCore_Mx projects can enable some debug features through two defines on each core.

Table 19. Debug switches

Location for single core: `LoRaWAN_FUOTA\LoRaWAN_End_Node\Core\Inc\sys_conf.h`
Location for dual core:
`LoRaWAN_FUOTA_DualCore\LoRaWAN_End_Node_DualCore\CMxxx\Core\Inc\sys_conf.h`

Symbols	Description	Default state
DEBUGGER_ENABLE	Enables the debugger mode: • 1: debugger and four debug pins enabled • 0: debugger disabled	0
LOW_POWER_DISABLE	Disables the low-power mode: • 0: low-power mode enabled (MCU enters Stop 2 mode) • 1: low-power mode disabled (MCU enters Sleep mode only)	0

7 Memory mapping

7.1 LoRaWAN_FUOTA

The flash and RAM memory mapping of the device contains the following elements:

- SB (Secure Boot)
- BFU (Boot and Firmware Update)
- Active slots (including the active user application firmware)
- Firmware header (flash memory area where the not-contiguous firmware header is stored)
- Download slot (downloaded firmware header and encrypted firmware to be installed at next reboot)
- Swap area (flash memory area used to swap the content of active and download slots during the installation process)
- KMS Data Storage (non-volatile memory area to store session keys)

Figure 36. Mapping of flash memory and RAM (single-core)

Start address	End address	Flash memory region
0x0800 0000	0x0800 8FFF	Secure Engine
0x0800 9000	0x0801 2FFF	BFU
0x0801 3000	0x0801 4FFF	KMS Data Storage (8 Kbytes)
0x0801 5000	0x0801 5FFF	Swap area
0x0801 6000	0x0802 9FFF	Download image (80 Kbytes)
0x0802 A000	0x0802 A1FF	Active image #1 header
0x0802 A200	0x0803 DFFF	Active image #1 (80 Kbytes)
0x0803 F000	0x0803 FFFF	LoRaWAN NVM

Start address	End address	RAM region
0x2000 0000	0x2000 0BFF	Secure Engine stack
0x2000 0C00	0x2000 33FF	Secure Engine region
0x2000 3400	0x2000 7FFF	BFU boot region

DT69152V1

7.2 LoRaWAN_FUOTA_DualCore

The flash and RAM memory mapping of the device contains the following elements:

- SB CM4
- SBSFU CM0+
- SE CM0+
- Active slots (including the active user application firmware)
- Firmware header (flash memory area where the not-contiguous firmware header is stored)
- Download slot (downloaded firmware header and encrypted firmware to be installed at next reboot)
- Swap area (flash memory area used to swap the content of active and download slots during the installation process)
- KMS Data Storage (non-volatile memory area to store session keys)
- User/SE keys (LoRaWAN and Secure Engine static embedded keys)

Figure 37. Mapping of flash memory and RAM (dual-core)

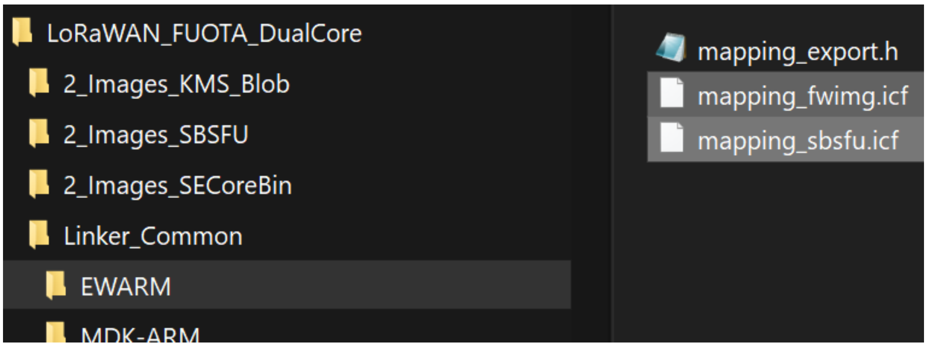
Start address	End address	Flash memory region
0x0800 0000	0x0800 1FFF	Secure Boot CM4
0x0800 2000	0x0800 2FFF	Swap area
0x0800 3000	0x0800 31FF	Download image header
0x0800 3200	0x0801 2FFF	Download image (64 Kbytes)
0x0801 3000	0x0801 31FF	Reserved
0x0801 3200	0x0801 DFFF	Active image #2 End_Node_DualCore_CM4 (44 Kbytes)
0x0801 E000	0x0801 EFFF	LoRaWAN NVM
0x0801 F000	0x0801 F1FF	Reserved
0x0801 F200	0x0802 EFFF	Active image #1 End_Node_DualCore_CM0+ (64 Kbytes)
0x0802 F000	0x0802 FFFF	KMS Data Storage (4 Kbytes)
0x0803 0000	0x0803 12FF	SE interface Cortex-M0+
0x0803 1300	0x0803 6FFF	SBSFU Cortex-M0+
0x0803 7000	0x0803 71FF	SBSFU CM0+ vector table
0x0803 7200	0x0803 E4FF	SE Cortex-M0+
0x0803 E500	0x0803 E7FF	User keys
0x0803 E800	0x0803 EFFF	SE keys
0x0803 F000	0x0803 F7FF	Active image #2 header
0x0803 F800	0x0803 FFFF	Active image #1 header

Start address	End address	RAM region
0x2000 0000	0x2000 0CDF	Secure Boot CM4
0x2000 0CE0	0x2000 0CFF	Cortex-M0+/M4 sync flag
0x2000 0D00	0x2000 6FFF	End_Node_DualCore_CM4
0x2000 7000	0x2000 73FF	Mapping table Mailbox MEM1 Cortex-M4
0x2000 7400	0x2000 7FFF	Mailbox MEM2 Cortex-M0+
0x2000 8000	0x2000 D3FF	SBSFU Cortex-M0+ End_Node_DualCore_CM0+
0x2000 D400	0x2000 FFFF	SE Cortex-M0+

DT68185V1

These elements are defined into two common linker script files in the `Linker_Common` folder.

Figure 38. Structure of common linker files



For more details about this configuration, refer to the document [5].

7.3 LoRaWAN_FUOTA_DualCore_ExtFlash

The flash and RAM memory mapping of the device contains the following elements:

- SB CM4
- SBSFU CM0+
- SE CM0+
- Active slots (including the active user application firmware)
- Firmware header (flash memory area where the not-contiguous firmware header is stored)
- KMS Data Storage (non-volatile memory area to store session keys)
- User/SE keys (LoRaWAN and Secure Engine static embedded keys)

The external flash memory available on the B-WL5M-SUBG1 board contains the following element:

- Download slot (downloaded firmware header and encrypted firmware to be installed at next reboot)

Figure 39. Mapping of flash memory, RAM, and external flash memory

Start address	End address	Flash memory region
0x0800 0000	0x0800 67FF	Secure Boot CM4
0x0800 6800	0x0800 6FFF	Download slot (2 Kbytes) for KMS blob
0x0800 7000	0x0800 71FF	Reserved
0x0800 7200	0x0801 3FFF	Active image #2 End_Node_DualCore_CM4 (52 Kbytes)
0x0801 4000	0x0801 4FFF	LoRaWAN NVM
0x0801 5000	0x0801 51FF	Reserved
0x0801 5200	0x0802 97FF	Active image #1 End_Node_DualCore_CM0+ (82 Kbytes)
0x0802 9800	0x0802 B7FF	KMS Data Storage (8 Kbytes)
0x0802 B800	0x0802 CBFF	SE interface Cortex-M0+
0x0802 CC00	0x0803 5FFF	SBSFU Cortex-M0+
0x0803 6000	0x0803 61FF	SBSFU CM0+ vector table
0x0803 6200	0x0803 E4FF	SE Cortex-M0+
0x0803 E500	0x0803 E7FF	User keys
0x0803 E800	0x0803 EFFF	SE keys
0x0803 F000	0x0803 F7FF	Active image #2 header
0x0803 F800	0x0803 FFFF	Active image #1 header

Start address	End address	External flash memory region
0x9000 0000	0x9000 01FF	Download image header
0x9000 0200	0x9001 4FFF	Download image (84 Kbytes)

Start address	End address	RAM region
0x2000 0000	0x2000 0CDF	Secure Boot CM4
0x2000 0CE0	0x2000 0CFF	Cortex-M0+/M4 sync flag
0x2000 0D00	0x2000 6FFF	End_Node_DualCore_CM4
0x2000 7000	0x2000 73FF	Mapping table Mailbox MEM1 Cortex-M4
0x2000 7400	0x2000 7FFF	Mailbox MEM2 Cortex-M0+
0x2000 8000	0x2000 C7FF	SBSFU Cortex-M0+ End_Node_DualCore_CM0+
0x2000 C800	0x2000 FFEF	SE Cortex-M0+
0x2000 FFF0	0x2000 FFFF	KMS DataStorage key (encrypt/decrypt blob)

DT71516V1

8 Memory footprint

8.1 LoRaWAN application memory footprint

8.1.1 LoRaWAN_End_Node of LoRaWAN_FUOTA SingleCore solution

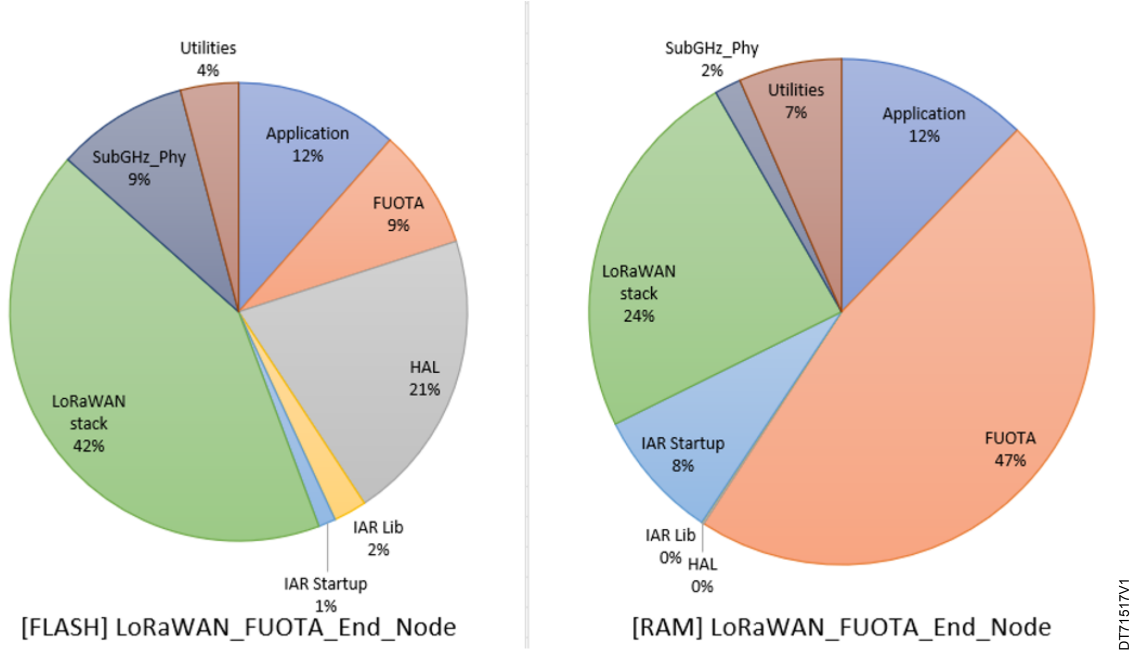
Values in the table below are measured for the following configuration of the IAR Embedded Workbench compiler (EWARM version 9.20.1):

- Optimization level 3 for size
- Debug option off
- Trace option VLEVEL_MEDIUM
- Target: STM32WL55
- End_Node application
- LoRaMAC Class A+C
- LoRaMAC region EU868 only

Table 20. **Memory footprint values for LoRaWAN_End_Node application**

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	8240	3005	Core, application, and target components
FUOTA	6084	11524	Firmware update packages modules
HAL	14826	36	STM32WL HAL and LL drivers
IAR Lib	1674	0	Proprietary IAR libraries
IAR Startup	869	2048	Int_vect, init routines, init table, CSTACK, and HEAP
LoRaWAN stack	30264	5894	Middleware LmHandler interface, crypto, MAC, and region
SubGHz_Phy	6676	417	Middleware radio interface
Utilities	2931	1620	All STM32 services (sequencer, time server, low-power mgr, trace, mem)
Total application	71564	24544	Memory footprint for LoRaWAN_End_Node application

Figure 40. **Flash memory and RAM footprint for LoRaWAN_End_Node**



8.1.2 LoRaWAN_End_Node_DualCore of LoRaWAN_FUOTA_DualCore solution

Values in the tables below are measured for the following configuration of the IAR Embedded Workbench compiler (EWARM version 9.20.1):

- Optimization level 3 for size
- Debug option off
- Trace option VLEVEL_LOW (minimal traces)
- Target: STM32WL55xx
- End_Node_DualCore application
- LoRaMAC Class A+C
- LoRaMAC region EU868 only

Table 21. Memory footprint values for LoRaWAN_End_Node_CM0+ application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	4001	176	Core, application, and target components
FUOTA	6369	2928	Firmware update packages modules
HAL	5330	0	STM32WL HAL and LL drivers
IAR Lib	1038	0	Proprietary IAR libraries
IAR Startup	583	4096	Int_vect, init routines, init table, CSTACK, and HEAP
LoRaWAN stack	28654	5846	Middleware LmHandler interface, crypto, MAC, and region
MBMux	2556	1156	Mailbox multiplexer wrappers and services
SubGHz_Phy	6491	417	Middleware radio interface
Utilities	3169	1648	All STM32 services (sequencer, time server, low-power mgr, trace, mem)
Total application	58191	16267	Memory footprint for LoRaWAN_End_Node_DualCore_CM0+ application

Figure 41. Flash memory and RAM footprint for LoRaWAN_End_Node_CM0+

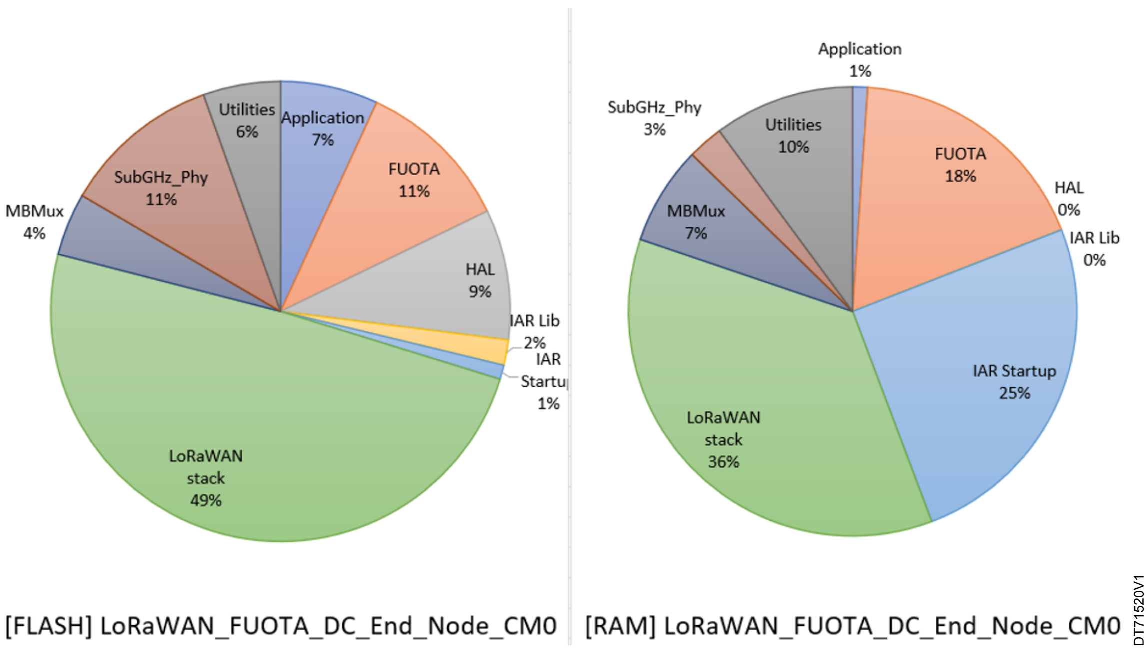
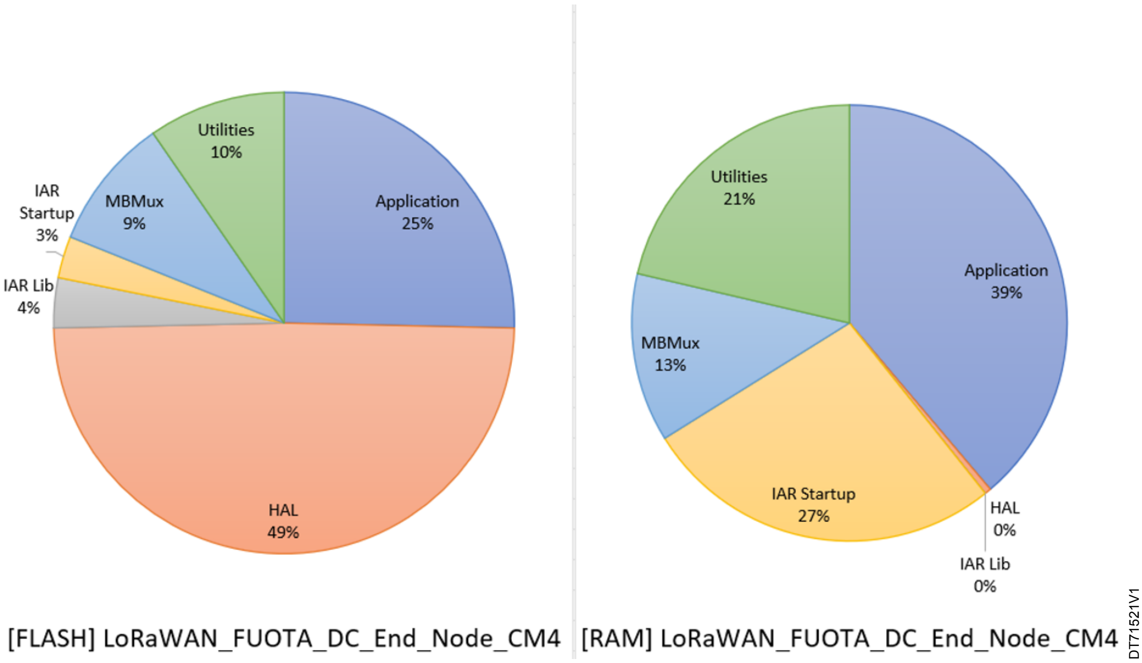


Table 22. Memory footprint values for LoRaWAN_End_Node_CM4 application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	7331	2958	Core, application, and target components
HAL	14262	36	STM32WL HAL and LL drivers
IAR Lib	1012	0	Proprietary IAR libraries
IAR Startup	867	2048	Int_vect, init routines, init table, CSTACK, and HEAP
MBMux	2686	954	Mailbox multiplexer wrappers and services
Utilities	2784	1628	All STM32 services (sequencer, time server, low-power mgr, trace, mem)
Total application	28922	7624	Memory footprint for LoRaWAN_End_Node_DualCore_CM4 application

Figure 42. Flash memory and RAM footprint for LoRaWAN_End_Node_CM4



8.1.3 LoRaWAN_End_Node_DualCore of LoRaWAN_FUOTA_DualCore_ExtFlash solution

Values in the tables below are measured for the following configuration of the IAR Embedded Workbench compiler (EWARM version 9.20.1):

- Optimization level 3 for size
- Debug option off
- Trace option VLEVEL_LOW (minimal traces)
- Target: STM32WL5MOCHxx
- End_Node_DualCore application
- LoRaMAC Class A+C
- LoRaMAC region EU868 only

Table 23. Memory footprint values for LoRaWAN_End_Node_ExtFlash_CM0+ application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	3897	176	Core, application, and target components
FUOTA	6371	2928	Firmware update packages modules
HAL	5290	0	STM32WL HAL and LL drivers
IAR Lib	1038	0	Proprietary IAR libraries
IAR Startup	582	4096	Int_vect, init routines, init table, CSTACK, and HEAP
LoRaWAN stack	28653	5846	Middleware LmHandler interface, crypto, MAC, and region
MBMux	2556	1156	Mailbox multiplexer wrappers and services
SubGHz_Phy	6491	417	Middleware radio interface
Utilities	3169	1648	All STM32 services (sequencer, time server, low-power mgr, trace, mem)
Total application	58047	16267	Memory footprint for LoRaWAN_End_Node_DualCore_CM0+ application

Figure 43. Flash memory and RAM footprint for LoRaWAN_End_Node_ExtFlash_CM0+

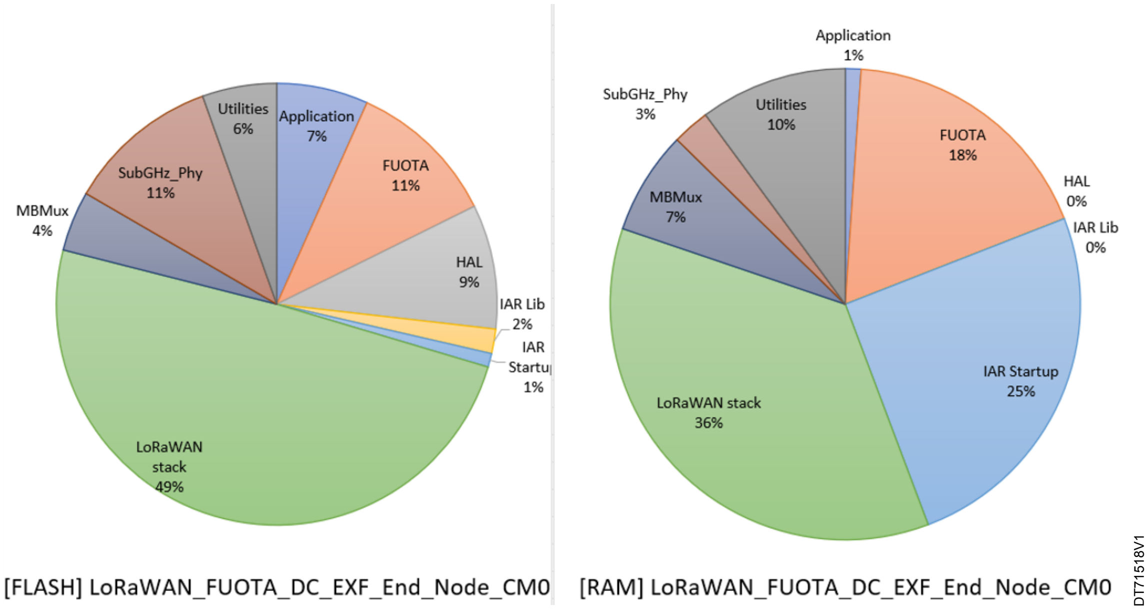
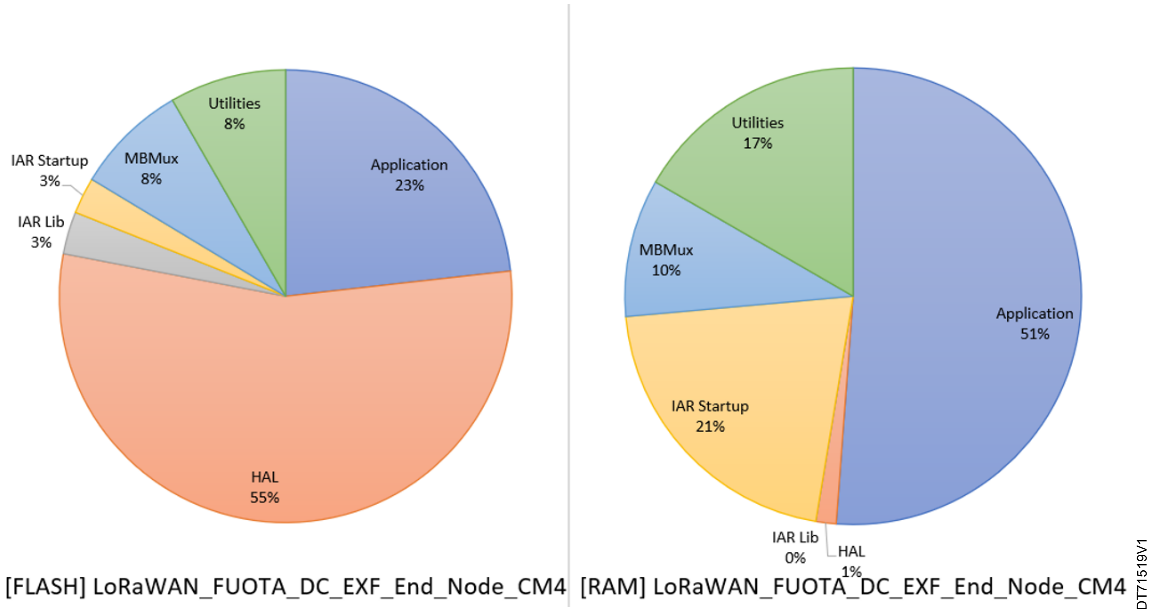


Table 24. Memory footprint values for LoRaWAN_End_Node_ExtFlash_CM4 application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	7761	5002	Core, application, and target components
HAL	18312	140	STM32WL HAL and LL drivers
IAR Lib	1012	0	Proprietary IAR libraries
IAR Startup	867	2048	Int_vect, init routines, init table, CSTACK, and HEAP
MBMux	2684	954	Mailbox multiplexer wrappers and services
Utilities	2784	1628	All STM32 services (sequencer, time server, low-power mgr, trace, mem)
Total application	33420	9772	Memory footprint for LoRaWAN_End_Node_DualCore_CM4 application

Figure 44. Flash memory and RAM footprint for LoRaWAN_End_Node_ExtFlash_CM4



8.2 BFU and SBSFU application memory footprint

8.2.1 SECoreBin and BFU of LoRaWAN_FUOTA SingleCore solution

Values in the tables below are measured for the following configuration of the IAR Embedded Workbench compiler (EWARM version 9.20.1):

- Optimization level 3 for size
- Debug option off
- Trace option off
- Target: STM32WL55xx

Table 25. **Memory footprint values for SECoreBin single-core application**

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	666	4	Core, application, and target components
HAL	4052	24	STM32WL HAL and LL drivers
IAR Lib	134	0	Proprietary IAR libraries
IAR Startup	188	0	Int_vect, init routines, init table, CSTACK, and HEAP
KMS	20452	9276	Middleware Key Management Services
SE	1088	16	Middleware Secure Engine
Total application	26580	9320	Memory footprint for SECoreBin application

Figure 45. **Flash memory and RAM footprint for SECoreBin single-core**

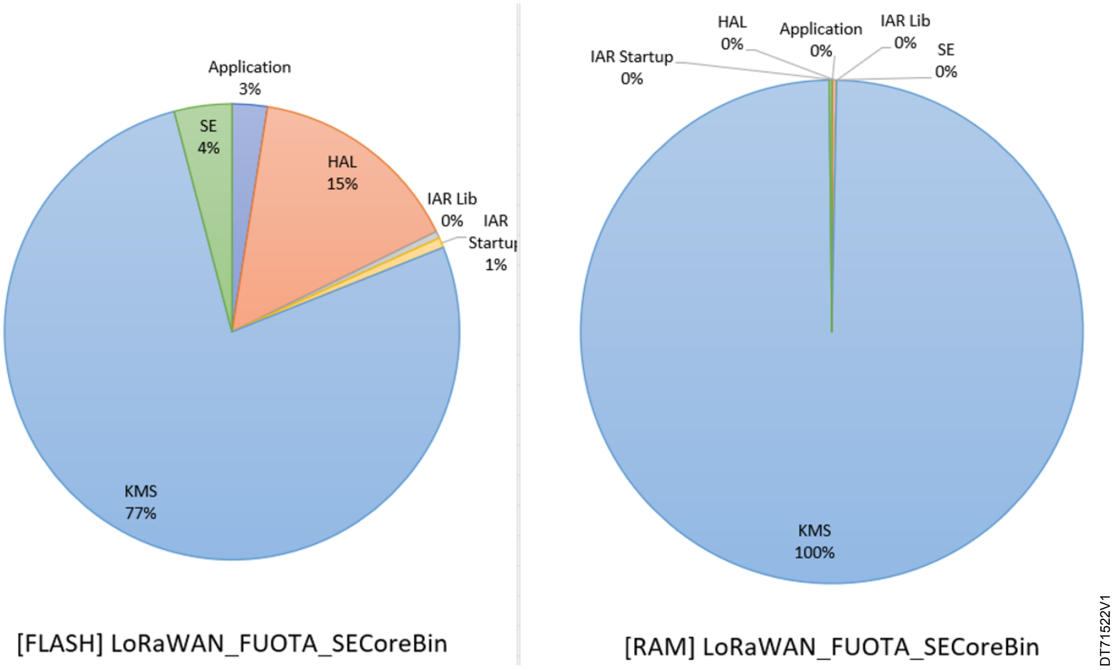
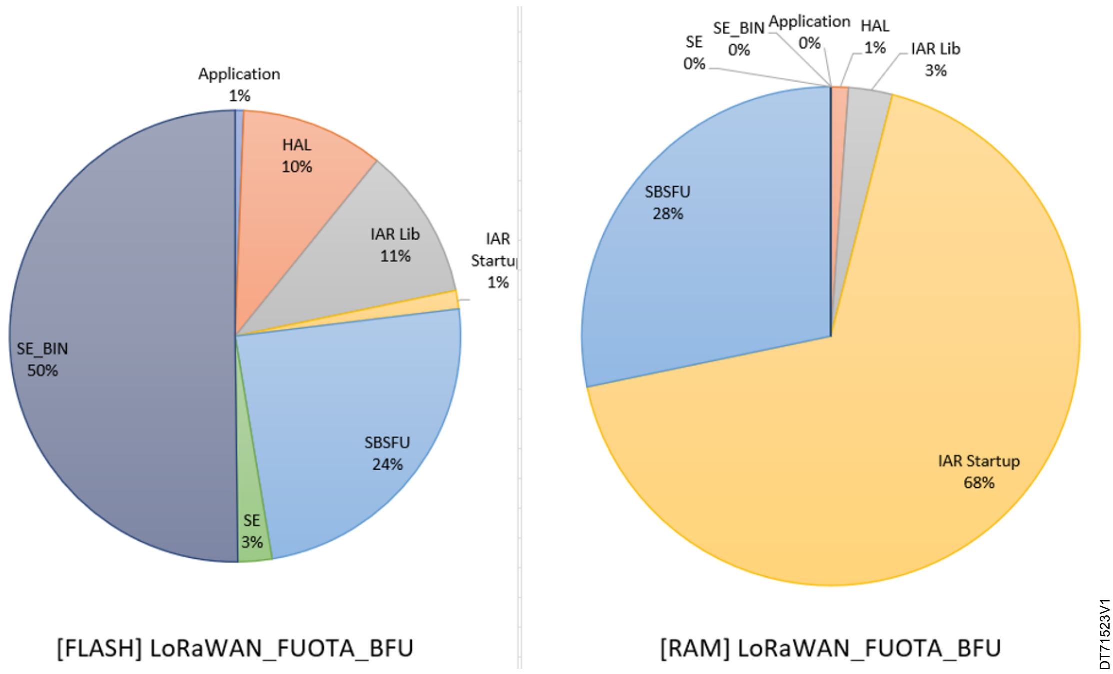


Table 26. Memory footprint values for BFU single-core application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	417	4	Core, application, and target components
HAL	6952	108	STM32WL HAL and LL drivers
IAR Lib	7424	280	Proprietary IAR libraries
IAR Startup	897	6656	Int_vect, init routines, init table, CSTACK, and HEAP
SBSFU	16530	2776	Secure Firmware Update and Secure boot
SE	1654	1	Middleware Secure Engine
SE_BIN	34132	0	Secure Engine compiled library
Total application	68006	9825	Memory footprint for BFU application

Figure 46. Flash memory and RAM footprint for BFU



8.2.2 SECoreBin and SBSFU of LoRaWAN_FUOTA_DualCore solution

Values in the tables below are measured for the following configuration of the IAR Embedded Workbench compiler (EWARM version 9.20.1):

- Optimization level 3 for size
- Debug option off
- Trace option off
- Target: STM32WL55xx

Table 27. Memory footprint values for SECoreBin dual-core application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	838	4	Core, application, and target components
HAL	4664	76	STM32WL HAL and LL drivers
IAR Lib	180	0	Proprietary IAR libraries
IAR Startup	220	0	Int_vect, init routines, init table, CSTACK, and HEAP
KMS	22050	9284	Middleware Key Management Services
SE	1380	16	Middleware Secure Engine
Total application	29332	9380	Memory footprint for SECoreBin application

Figure 47. Flash memory and RAM footprint for SECoreBin dual-core

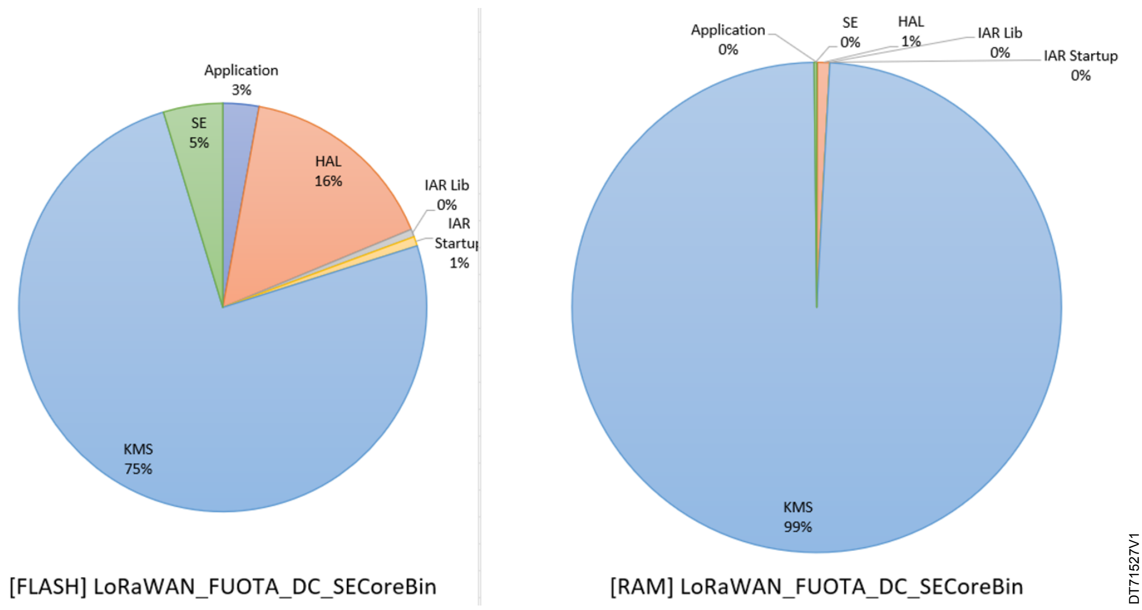


Table 28. Memory footprint values for SBSFU CM0+ application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	182	4	Core, application, and target components
HAL	3525	100	STM32WL HAL and LL drivers
IAR Lib	268	0	Proprietary IAR libraries
IAR Startup	514	6656	Int_vect, init routines, init table, CSTACK, and HEAP
SBSFU	12140	1260	Secure Firmware Update and Secure boot
SE	4768	1	Middleware Secure Engine
SE_BIN	30972	0	Secure Engine compiled library
Total application	52369	8021	Memory footprint for SBSFU CM0+ application

Figure 48. Flash memory and RAM footprint for SBSFU CM0+

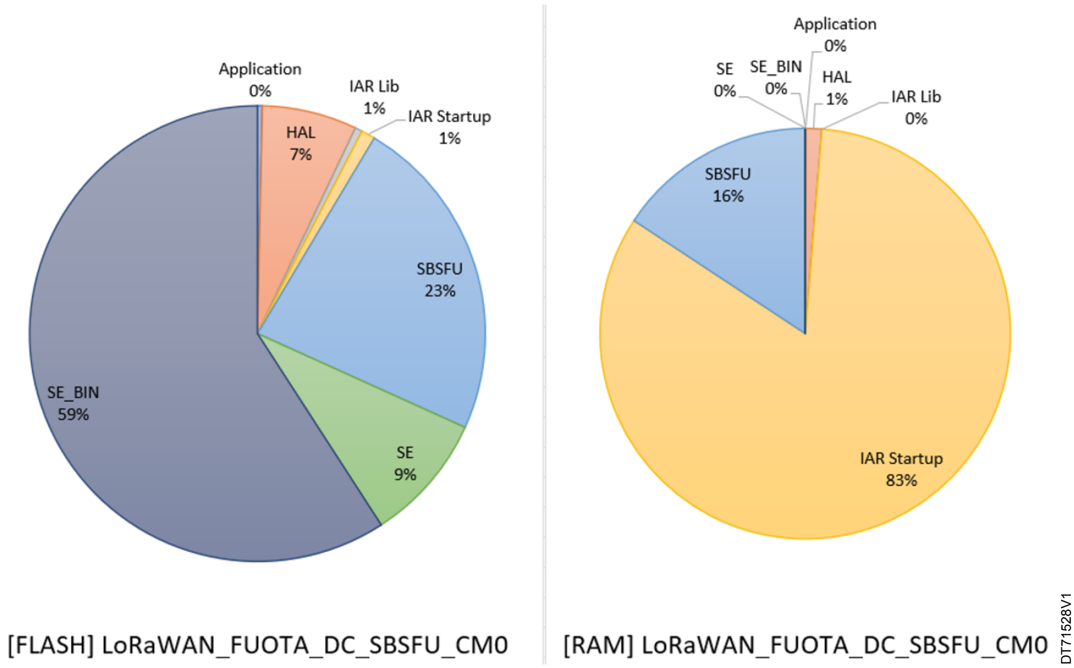
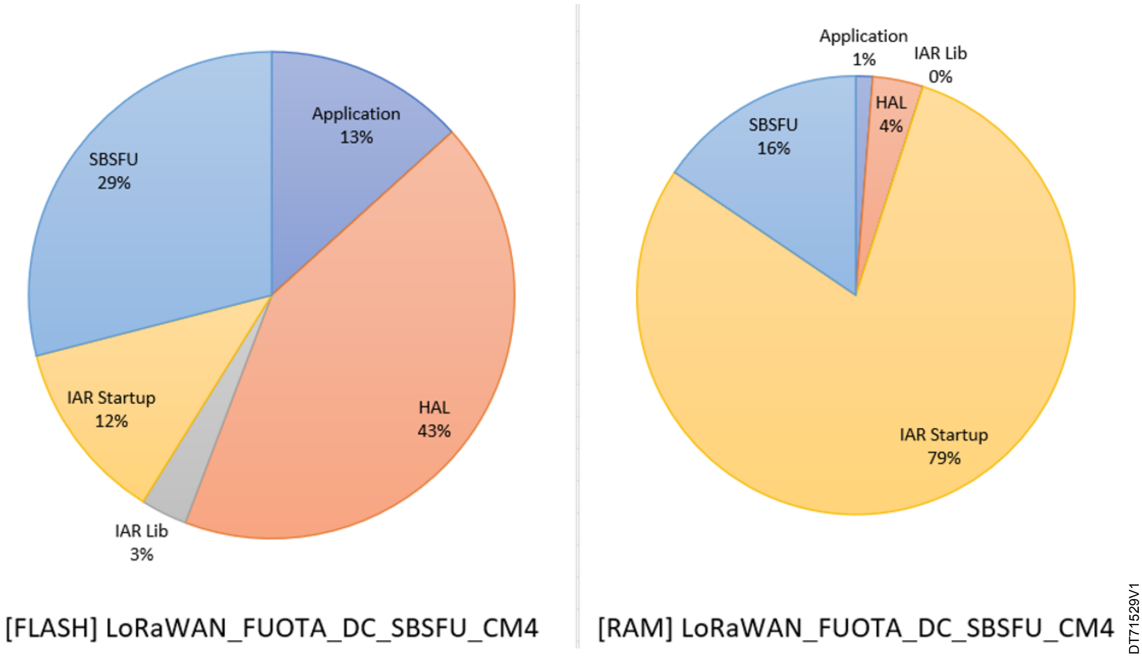


Table 29. Memory footprint values for SBSFU CM4 application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	800	8	Core, application, and target components
HAL	2554	24	STM32WL HAL and LL drivers
IAR Lib	186	0	Proprietary IAR libraries
IAR Startup	728	512	Int_vect, init routines, init table, CSTACK, and HEAP
SBSFU	1744	100	Secure Firmware Update and Secure boot
Total application	6012	644	Memory footprint for SBSFU CM4 application

Figure 49. **Flash memory and RAM footprint for SBSFU CM4**



8.2.3 SECoreBin and SBSFU of LoRaWAN_FUOTA_DualCore_ExtFlash solution

Values in the tables below are measured for the following configuration of the IAR Embedded Workbench compiler (EWARM version 9.20.1):

- Optimization level 3 for size
- Debug option off
- Trace option off
- Target: STM32WL5MOCHxx

Table 30. **Memory footprint values for SECoreBin_ExtFlash dual-core application**

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	890	4	Core, application, and target components
HAL	5778	76	STM32WL HAL and LL drivers
IAR Lib	180	0	Proprietary IAR libraries
IAR Startup	220	0	Int_vect, init routines, init table, CSTACK, and HEAP
KMS	23964	10380	Middleware Key Management Services
SE	1380	16	Middleware Secure Engine
Total application	32412	10476	Memory footprint for SECoreBin application

Figure 50. Flash memory and RAM footprint for SECoreBin_ExtFlash

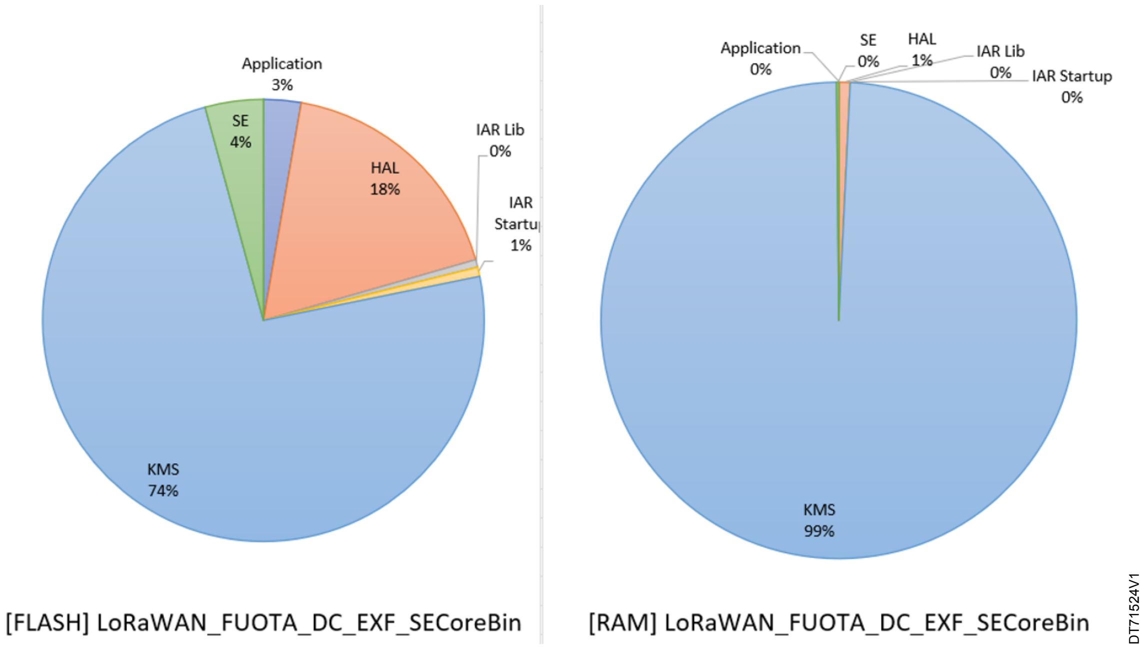


Table 31. Memory footprint values for SBSFU_ExtFlash_CM0+ application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	222	4	Core, application, and target components
HAL	10001	212	STM32WL HAL and LL drivers
IAR Lib	6794	284	Proprietary IAR libraries
IAR Startup	576	6656	Int_vect, init routines, init table, CSTACK, and HEAP
SBSFU	16163	1528	Secure Firmware Update and Secure boot
SE	4768	1	Middleware Secure Engine
SE_BIN	35116	0	Secure Engine compiled library
Total application	73640	8685	Memory footprint for SBSFU CM0+ application

Figure 51. Flash memory and RAM footprint for SBSFU_ExtFlash_CM0+

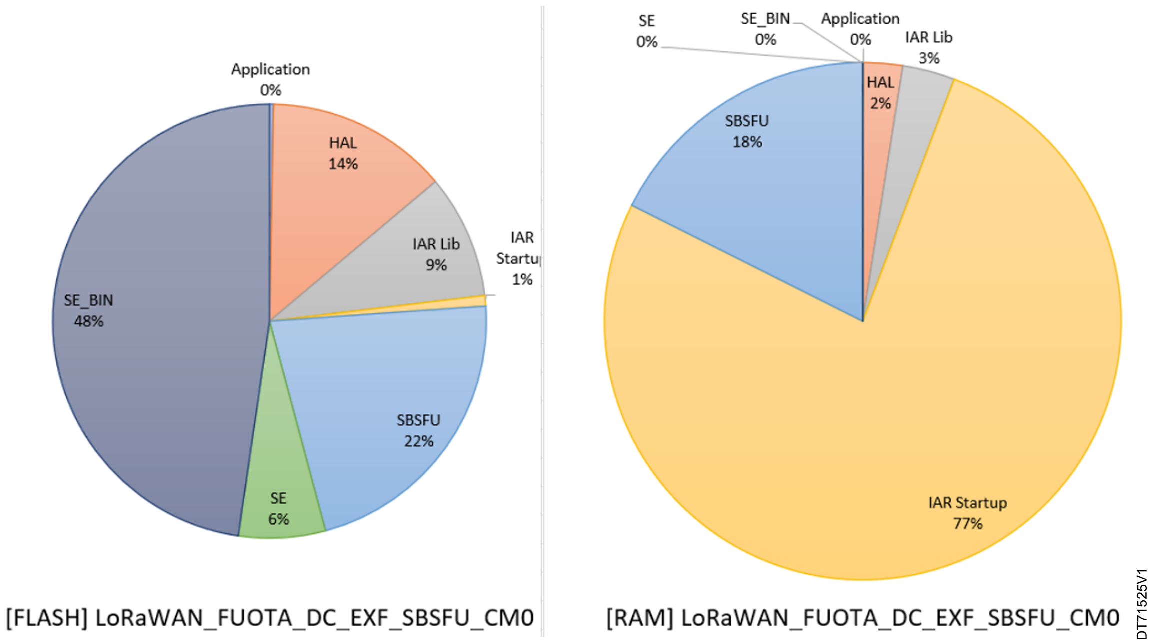
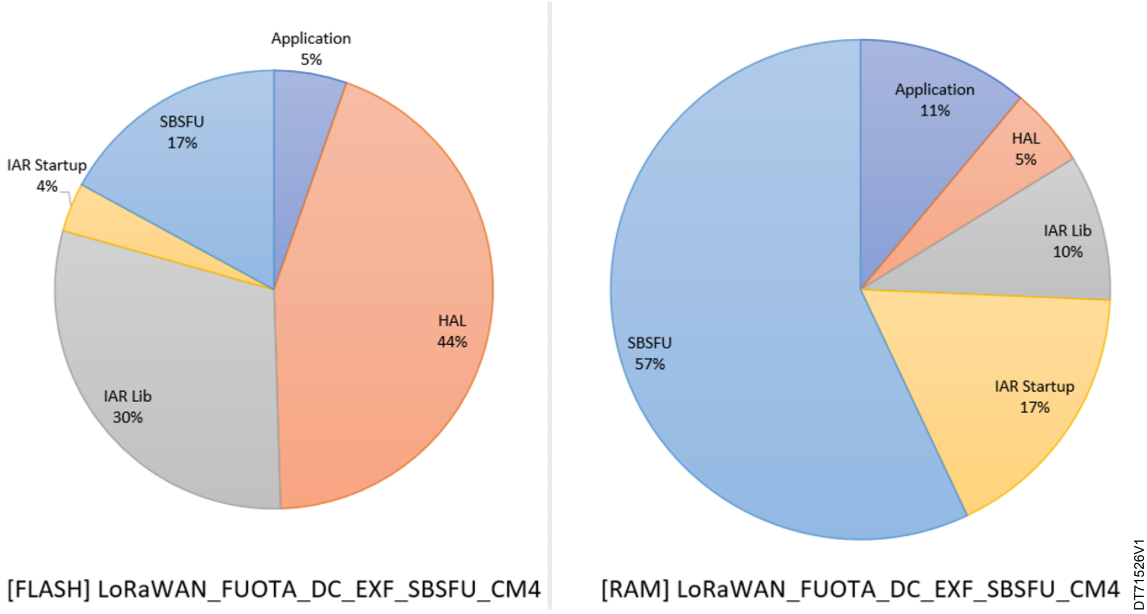


Table 32. Memory footprint values for SBSFU_ExtFlash_CM4 application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	1329	328	Core, application, and target components
HAL	10913	152	STM32WL HAL and LL drivers
IAR Lib	7373	280	Proprietary IAR libraries
IAR Startup	891	512	Int_vect, init routines, init table, CSTACK, and HEAP
SBSFU	4215	1688	Secure Firmware Update and Secure boot
Total application	24721	2960	Memory footprint for SBSFU CM4 application

Figure 52. Flash memory and RAM footprint for SBSFU_ExtFlash_CM4



9 Implementation validation over network operators

Actility	FUOTA campaign has been performed in front of Actility network in several regions. For more details, refer to the related blog article: https://blog.st.com/thingpark/.
Senet	FUOTA campaign has been performed in front of Senet network in US region
AWS IoT Core for LoRaWAN	FUOTA campaign has been performed in front of AWS Network server. The hardware setup requires an AWS certified Gateway. For more details, refer to: https://aws.amazon.com/iot-core/lorawan/.

Appendix A

The Python script that includes the algorithm generating redundancy fragments is detailed below.

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

# =====================================================
# Imports
# =====================================================
import os
import sys
import logging
import socket
import binascii
import math
from datetime import datetime
from time import sleep

# =====================================================
# Global variables
# =====================================================
logger = logging.getLogger('__name__')

test_matlab = False
test_interop = False
fec_algo_version = 1

if test_interop:
    input_file = 'Interoptest_file_1.bin'
    fragment_size = 40
    redundancy = 25
else:
    input_file = 'LoRaWAN_End_Node.sfb'
    fragment_size = 120
    redundancy = 72


# =====================================================
# =====================================================
def prbs23(start):
    '''The prbs23() function implements a PRBS generator with 2^23 period.
       standard implementation of a 23bit prbs generator'''
    x = start
    b0 = x & 1
    b1 = int((x & 32) / 32)
    x = (x >> 1) | ((b0 ^ b1) << 22)
    return x


# =====================================================
# =====================================================
def matrix_line(N, M):
    '''the matrix_line function generating a parity check vector:
       this function returns line N of the MxM parity matrix'''
    nb_coeff = 0
    line = [0]*M

    # if M is a power of 2
    if (M & (M - 1) == 0) and M != 0:
        pow2 = 1
    else:
        pow2 = 0
    # initialize the seed differently for each line
    x = 1 + (1001 * (N + 1))

    # will generate a line with M / 2 bits set to 1 (50 % )
    while (((fec_algo_version == 2) and (line.count(1) < math.floor(M / 2))) or
           ((fec_algo_version == 1) and (nb_coeff < math.floor(M / 2)))):
```

```
        r = math.pow(2, 16)
        # this can happen if m=1, in that case just try again with a different random number
        while r >= M:
            x = prbs23(x)
            # bit number r of the current line will be switched to 1
            r = x % (M + pow2)

        # set to 1 the column which was randomly selected
        line[r] = 1
        nb_coeff += 1
    return line


# =====================================================
# =====================================================
if __name__ == '__main__':
    logging.basicConfig(format='%(asctime)s - [%(levelname)s] - %(message)s',
level=logging.DEBUG)

    output_file = os.path.splitext(input_file)[0] + '_coded' + os.path.splitext(input_file)
[1]

    uncoded_frag = []
    # nb of bytes per fragment

    if test_matlab:
        fragment_size = 10
        nb_fragment = 32
        for i in range(nb_fragment):
            buffer = ''
            for j in range(fragment_size):
                buffer += '{:02X}'.format((i*fragment_size+j) % 256)
            logger.debug(buffer)
            uncoded_frag.append(buffer)

    else:
        # Read the binary file and convert into fragments list of size <fragment_size>
        try:
            with open(input_file, "rb") as f:
                while bytes_str := f.read(fragment_size):
                    uncoded_frag.extend([binascii.hexlify(bytes_str).decode()])
        except FileNotFoundError as e:
            logger.error(e)
            exit(1)

        # Get the number of fragments into the binary file
        nb_fragment = len(uncoded_frag)

        # 0-Padding of the last fragments
        uncoded_frag[-1] += '0'*(fragment_size*2-len(uncoded_frag[-1]))

        logger.info(('Input file: {} | Fragment size: {} bytes | Uncoded fragments: {} | '
                     'Redundancy fragments: {}').format(input_file, fragment_size,
nb_fragment, redundancy))

    # generate a coded array based on uncoded content
    coded_frag = []
    logger.debug('Matrix:')
    for y in range(nb_fragment):
        s = '0'*(2 * fragment_size)
        # line y of M x M matrix
        A = matrix_line(y, nb_fragment)
        logger.debug('{:03}: {} - {:03}'.format(y + 1, A, A.count(1)))

        for x in range(nb_fragment):
            # if bit x is set to 1 then xor the corresponding fragment
            if A[x] == 1:
                s = '{:X}'.format(int(s, 16) ^ int(uncoded_frag[x], 16))
        # prevent Odd-length string
        s = '0'*((fragment_size * 2) - len(s)) + s
```

```
        # save coded fragment
        coded_frag.extend([s])

    # write the ouput file containing uncoded + coded fragments
    with open(output_file, "wb") as f:
        logger.debug('Uncoded fragments:')
        for num, frag in enumerate(uncoded_frag, start=1):
            logger.debug('{:03}: {}'.format(num, frag.upper()))
            f.write(binascii.unhexlify(frag.encode()))
        logger.debug('Coded fragments:')
        for num, frag in enumerate(coded_frag, start=1):
            logger.debug('{:03}: {}'.format(num, frag.upper()))
            f.write(binascii.unhexlify(frag.encode()))

    logger.info('Output file: {} | File size: {}'.format(output_file,
os.path.getsize(output_file)))
```

Appendix B

This appendix details simulations needed to validate the reconstruction of the initial firmware update image in the flash memory area (full `.sfb` image).

B.1 Context

The Python script defined in Section Appendix A sends via a LoRaWAN tester `LoRaWAN_End_Node.sfb` via a fragmentation session on the embedded firmware.

The serial is recovered in `Serial.log` file and logs trace values. The Python script simulates the loss of several fragments.

In FUOTA single-core case, redundancy (`FRAG_MAX_REDUNDANCY`) is allocated to 10% of total fragment number (`FRAG_MAX_NB`) with the code below:

```
#define FRAG_MAX_NB 716
#define FRAG_MAX_REDUNDANCY 72
```

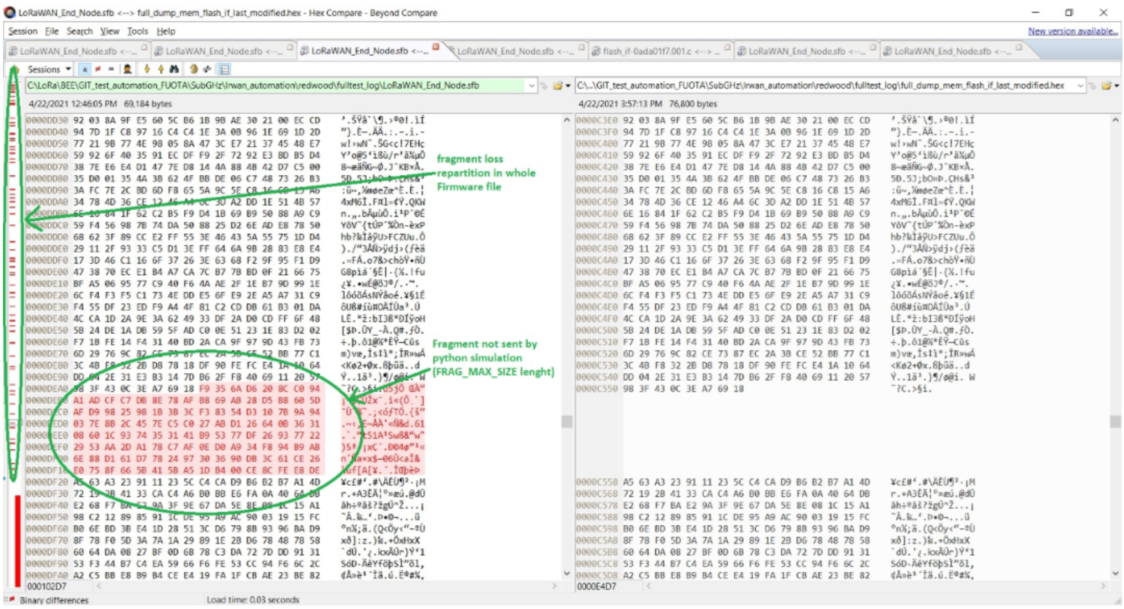
Due to the flash memory limitation, a fragment size must be a multiple of eight (8 × 8 bytes = 64 bits):

```
#define FRAG_MAX_SIZE 120
```

This simulation environment is used to validate FUOTA firmware with a loss up to 72 fragments. If the communication is more degraded and leads to more than 72 fragments loss, the reconstruction is discarded.

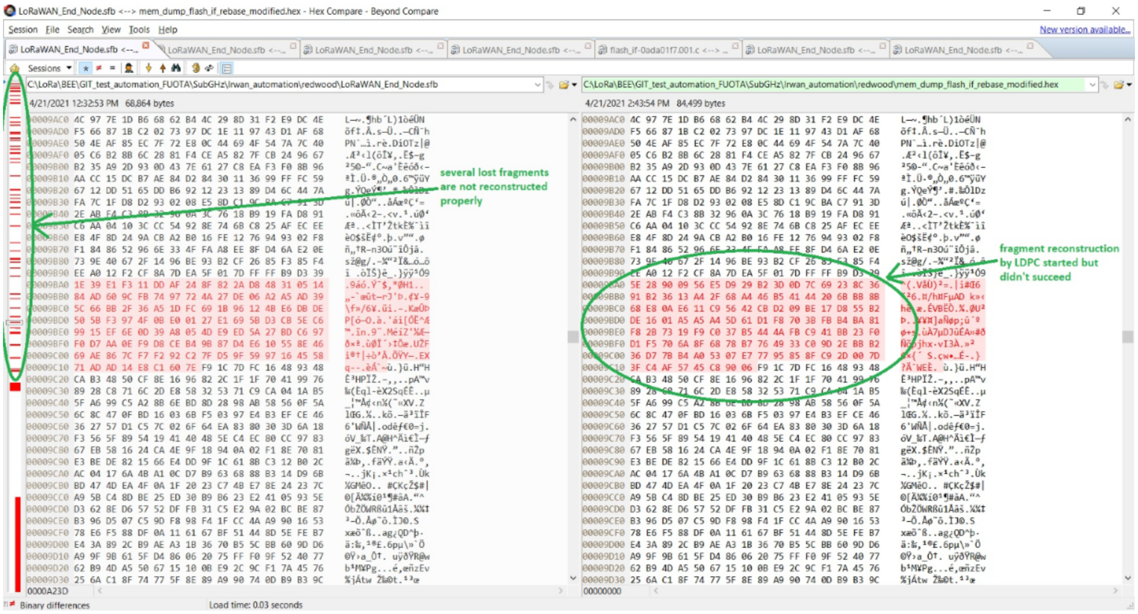
The figure below shows the difference between the full firmware to download (on the left), and data sent by Python simulation (on the right).

Figure 53. Example of datablock sent by FUOTA Python simulation on a LoRaWAN tester



After the loss of several fragments, the Frag decoder calls the LDPC algorithm to reconstruct these missing fragments. The figure below shows a memory dump in case of a failed memory reconstruction.

Figure 54. Example of failed datablock reconstruction by LDPC

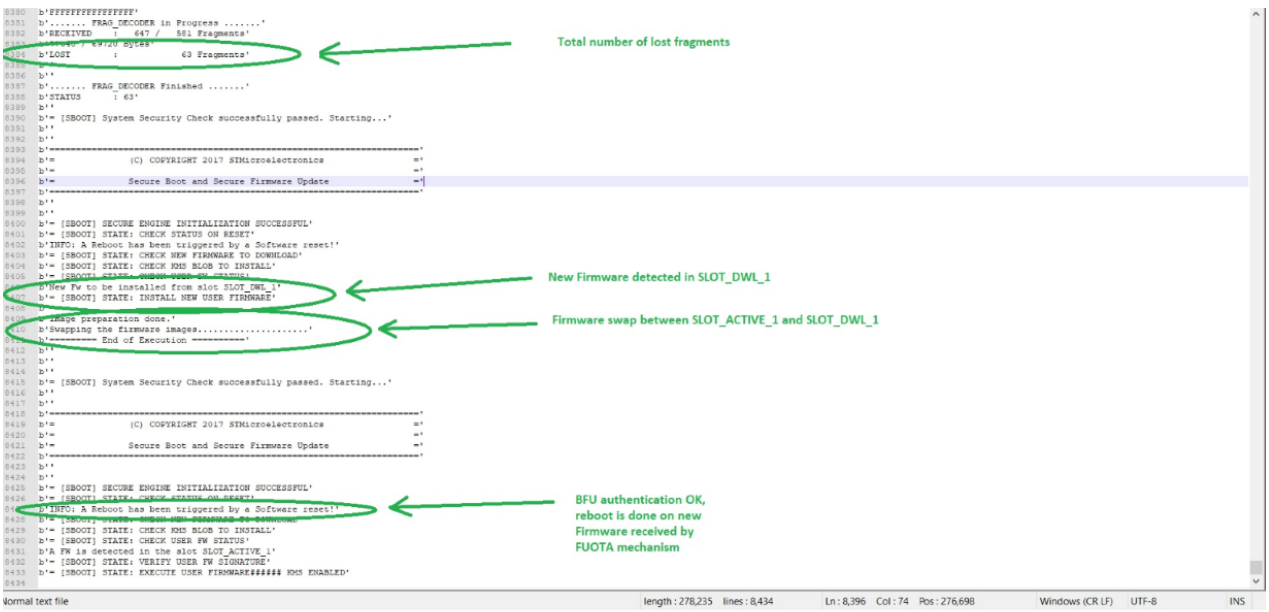


B.2 Debug manipulations

To check the FUOTA session integrity, the execution must be broken at the end of the FUOTA session, before the BFU takes the hand to authenticate the firmware obtained in `SLOT_DWL_1`. A dump of the whole `SLOT_DWL_1` area is required to compare it to `<myAppli>.sfb` sent by the network over FUOTA.

Log traces below display the end of the FUOTA mechanism: BFU process to check integrity and reboot on new firmware received by the firmware update mechanism.

Figure 55. End of FUOTA campaign/ datablock check, authentication by BFU/ reboot on updated firmware traces



Revision history

Table 33. **Document revision history**

Date	Version	Changes
26-Nov-2020	1	Initial release.
9-Jul-2021	2	Updated: • Figure 7. MSC for Class-C creation • Figure 8. MSC for data-block broadcasting • Section 3: SBSFU/end-device manager relationship replaced by Section 3 FUOTA campaign • Section 4.1 Single-core project overview • Section 4.2 Dual-core project overview • Section 4.3 BFU and SBSFU features • Figure 23. File structure of LoRaWAN configuration • Section 6 Getting started • Section 7 Memory mapping • Section 8 Memory footprint Added: • Section Appendix A • Section Appendix B
22-Feb-2022	3	Updated: • Section 3.3 End-device design choices • Table 20. Memory footprint values for LoRaWAN_End_Node application • Table 21. Memory footprint values for LoRaWAN_End_Node_CM0+ application • Table 22. Memory footprint values for LoRaWAN_End_Node_CM4 application • Table 24. Memory footprint values for BFU single-core application • Table 26. Memory footprint values for SBSFU CM0+ application Added: • Section 3.3.1 Fragmentation decoder definitions • Section 3.3.2 Fragmentation decoder implementation • Section 3.3.2.1 FRAG_MAX_SIZE computation • Section 3.3.2.2 FRAG_MAX_REDUNDANCY computation • Section 3.3.2.3 FRAG_MAX_NB and FRAG_MIN_SIZE computation • Section 9 Implementation validation over network operators
17-Nov-2022	4	Updated: • Section Introduction • Section 1 General information • Section 3.1 How to create and manage a FUOTA campaign • Section 4.1 Single-core project overview • Section 4.2 Dual-core project overview • Section 4.4 LoRaWAN features • Section 4.5 Firmware architecture • Section 6.2.2 How to automate the generate and load processes • Figure 36. Mapping of flash memory and RAM (single-core) • Figure 37. Mapping of flash memory and RAM (dual-core) • Section 8.1.1 LoRaWAN_End_Node of LoRaWAN_FUOTA SingleCore solution • Section 8.1.2 LoRaWAN_End_Node_DualCore of LoRaWAN_FUOTA_DualCore solution • Section 8.2.1 SECoreBin and BFU of LoRaWAN_FUOTA SingleCore solution • Section 8.2.2 SECoreBin and SBSFU of LoRaWAN_FUOTA_DualCore solution Added: • Section 7.3 LoRaWAN_FUOTA_DualCore_ExtFlash • Section 8.1.3 LoRaWAN_End_Node_DualCore of LoRaWAN_FUOTA_DualCore_ExtFlash solution • Section 8.2.3 SECoreBin and SBSFU of LoRaWAN_FUOTA_DualCore_ExtFlash solution
10-Jan-2023	5	Updated: • Section 3.2.1 Generation of redundancy fragments • Section Appendix A • Section B.1 Context

Contents

List of tables

List of figures