AN5480

Application note

How to build a Sigfox™ application with STM32CubeWL

Introduction

This application note explains how to build specific Sigfox™ applications based on STM32WL5x/Ex microcontrollers. This document groups together the most important information and lists the aspects to be addressed.

Sigfox™ is a type of wireless telecommunication network designed to allow long-range communication at very low bit rates, and to enable the use of long-life battery-operated sensors. The Sigfox Stack™ library manages the channel access and security protocol that ensures interoperability with the Sigfox™ network.

The application based on the NUCLEO_WL55JC Nucleo development board with STM32WL55JC (order codes NUCLEO-WL55JC1 for high-frequency band) and B-WL5M-SUB1 connectivity expansion board with STM32WL5M, and firmware in the STM32CubeWL MCU Package is Sigfox Verified™.

Sigfox™ application main features are:

- Application integration ready
- RC1, RC2, RC3c, RC4, RC5, RC6 and RC7 Sigfox Verified™
- Sigfox™ Monarch (STMicroelectronics algorithm patented)
- Extremely low CPU load
- No latency requirements
- Small STM32 memory footprint
- Utilities services provided

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

1 General information

The STM32CubeWL runs on STM32WL5x/Ex microcontrollers based on the Arm® Cortex®-M processor.

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

Table 1. **Acronyms and terms**

Acronym	Definition
BSP	Board support package
CS	Carrier sense
DC/DC	Direct current to direct current converter
FH	Frequency hopping
IoT	Internet of things
LBT	Listen before talk
PAC	Porting authorization code
POI	Point of interest
PA	Power amplifier
PRBS	Pseudo-random bit sequence
RC	Region configuration
RSA	Radio Sigfox analyzer
RSSI	Receive signal strength indicator
Rx	Reception
SDR	Software-defined radio
TCXO	Temperature compensated crystal oscillator
Tx	Transmission

2 Sigfox standard

This section provides a general Sigfox overview, focusing, in particular, the Sigfox end device.

Sigfox is a wireless telecommunication network operator designed to allow long range communication at a low bitrate enabling long-life battery operated sensors. The firmware of the STM32CubeWL MCU Package includes the Sigfox Stack library.

Sigfox limits the use of its network to 144 messages per day and per device. Each message can be from 1 bit up to 12 bytes.

2.1 End-device hardware architecture

The end device is the STM32WL55JC microcontroller mounted on NUCLEO-WL55JC board or STM32WL5MOCH microcontroller mounted on B-WL5M-SUBG1 board.

This MCU, with integrated sub-GHZ radio operating in the150 - 960 MHz ISM band, belongs to the STM32WL5x/Ex devices that include microcontrollers with different memory sizes, packages and peripherals.

2.2 Regional radio resource

The European, North American and Asian markets have different spectrum allocations and regulatory requirements. Sigfox has split requirements in various RCs (region configurations) listed in the table below.

Table 2. **Region configurations**

RC	Countries
RC1	Europe, Oman, Lebanon, South Africa, Kenya
RC2	USA, Canada, Mexico
RC3c	Japan
RC4	Brazil, Colombia, Peru, New–Zealand, Australia and Singapore
RC5	South Korea
RC6	India
RC7	Russia

The table below provides an overview of the regulatory requirements for the region configurations.

Table 3. **RF parameters for region configurations**

RF parameter	RC1	RC2	RC3c	RC4	RC5	RC6	RC7
Frequency band downlink (MHz)	869.525	905.2	922.2	922.3	922.3	866.3	869.1
Frequency band uplink (MHz)	868.130	902.2	923.2	920,8	923.3	865.2	868.8
Uplink modulation	DBPSK						
Downlink modulation	GFSK						
Uplink data-rate	100	600	100	600	100	100	100
Down-link data-rate	600						
Max output power (dBm)	14	22	13	22	13	13	14
Medium access	Duty cycle 1%	Frequency hopping Max on time 400 ms/20 s	Carrier sense	Frequency hopping Max on time 400 ms/20 s	Carrier sense	Duty cycle 1%	
CS center frequency (MHz)	NA		923.2	NA	923.3	NA	
CS bandwidth (kHz)			200	NA	200		
CS threshold (dBm)			-80	NA	-65		

2.3 Rx/Tx radio time diagram

The end device transmits data to the network in an asynchronous manner. This is due to the fact that transmission data is only sent per device-report event. The figures below depict the timing sequences with and without a downlink.

Figure 1. **Timing diagram for uplink only**

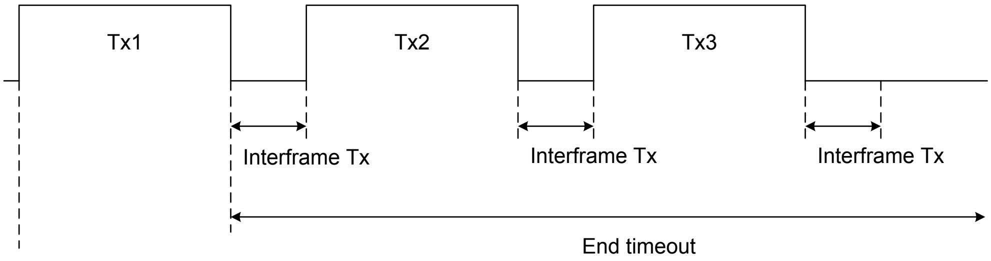
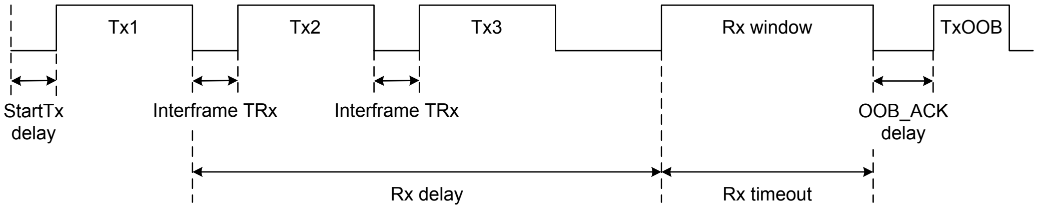


Figure 2. **Timing diagram for uplink with downlink**



The three transmissions Tx1, Tx2 and Tx3 contain the same payload information. These consecutive transmissions maximize the probability of a correct reception by the network. When the device observes good link quality to the network, it may decide to send only Tx1 to save power consumption (if downlink frame is requested). The preferred scheme of the API to select is described in Section 7.1.2: Send frames/bits.

The timings shown in the previous figures are detailed in the table below for the various regional configurations.

Table 4. Timings

RC	StartTx delay	Interframe Tx/TRx	Rx delay	Rx timeout	OOB_ACK delay	End timeout
RC1	0 s	500 ms	20 s	25 s	1.4 s	NA
RC2						10 s
RC3c	100 ms max (start LBT)	500 ms + LBT	19 s	34 s		NA
RC4	10 s	500 ms	20 s	25 s		
RC5	100 ms max (LBT)	500 ms + LBT	19 s	34 s		
RC6	0 s	500 ms	20 s	25 s		
RC7						

The Tx periods depend on the number of bytes sent and on the RC zone:

- It takes 10 ms to send a bit in RC1 and RC3c.
- It takes 1.66 ms to send a bit in RC2 and RC4.

A message can be 26-byte long maximum (including sync word, header, and payload data). For RC1, a Tx period can be maximum 26 x 8 x 10 ms = 2.08 s.

2.4 Listen before talk (LBT)

In RC3c and RC5, LBT is mandatory before any transmission.

In RC3c, the device must listen and check if the channel is free. The channel is considered as free if the power within a 200 kHz bandwidth stays below -80 dBm (CS threshold) for 5 ms.

When the channel is free, the device starts a transmission. The transmission is not started otherwise.

2.5 Monarch

Monarch is a Sigfox beacon placed at a point of interest (POI). The signal of the Sigfox beacon is emitted at a frequency allowed by the region the POI belongs to. The beacon contains region configuration (RC) information that a Monarch-capable device can demodulate.

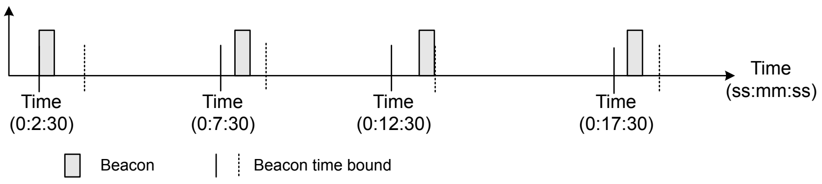
Upon reception of this information, the Monarch-capable device is able to switch automatically to the right RC and send information to the network.

The Monarch feature allows a Sigfox IoT device to roam seamlessly across the world.

2.5.1 Monarch signal description

The Monarch signal is sent at POI every 5 minutes plus a random back-off period of 10 seconds. The beacon frequency is region specific. The beacon lasts in total 400 ms. If a device clock is set, a scan window can be opened only when the Monarch signal is present to reduce the end-device current consumption.

Figure 3. Monarch beacon



The signal is OOK modulated, meaning the signal is either ON or OFF. The modulation frequency is specified to 16384 Hz (half of the RTC clock). The signal is ON for one sample and then OFF. It is ON with a periodicity of 11, 13 or 16 (16384 Hz) samples.

The following OOK frequency dF are possible:

- dF1 = 16384 / 16 = 1024 Hz
- dF2 = 16384 / 13 = 1260.3 Hz
- dF3 = 16384 / 11 = 1489.4 Hz

The 400 ms of the Monarch pattern include two sub-patterns:

- pattern1 (362 ms at a specific dF)
- pattern2 (38 ms at another specific dF)

Table 5. Monarch signal characteristics versus RC

RC	Monarch frequency (Hz)	Pattern1 dF (Hz)	Pattern2 dF (Hz)
RC1	869 505 000	1024	1260.3
RC2	905 180 000		
RC3c	922 250 000		
RC4		1260.3	1489.4
RC5		1024	
RC6	866 250 000		
RC7	869 160 000	1260.3	

2.5.2 Monarch signal demodulation

When a device starts to scan a Monarch signal, the device sweeps during 5 mn onto all Monarch frequencies listed in Table 5: this is called the sweep period.

Note: If the time is known, the sweep time may be reduced about 10 s + some clock drift.

During this period, the device tries to match with one of the pattern1s. When a match is found, the device exits the sweep period to enter a second period called the window period during 400 ms.

The device sets its RF frequency where the pattern1 match occurred. The device then tries to match the pattern2 to confirm a Monarch beacon is found.

3 SubGHz HAL driver

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

The SubGHz HAL is directly on top of the sub-GHz radio peripheral (see Figure 1).

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

This SubGHz HAL driver is composed the following main parts:

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

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

3.1 SubGHz resources

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

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

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

- `HAL_SUBGHZ_IRQHandler` in the SUBGHZ_Radio_IRQHandler.

3.2 SubGHz data transfers

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

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

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

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

4 BSP STM32WL55 Nucleo boards

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

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

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

4.1 Frequency band

Two types of Nucleo board are available on the STM32WL5 devices:

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

If the user tries to run a firmware compiled at 868 MHz on a low-frequency-band board, very poor RF performances are expected.

The firmware does not check the band of the board on which it runs.

4.2 RF switch

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

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

Table 6. BSP radio switch

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

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

Table 7. **RF states versus switch configuration**

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

4.3 TCXO

Various oscillator types can be mounted on the user application. On the STM32WL55 Nucleo boards, a temperature compensated crystal oscillator (TCXO) is used to achieve a better frequency accuracy.

Table 8. **BSP radio TCXO**

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

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

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

```
#define IS_TCXO_SUPPORTED                    1U
```

4.4 Power regulation

Depending on the user application, a LDO or an SMPS (also named DC/DC) is used for power regulation. An SMPS is used on the STM32WL55 Nucleo boards.

Table 9. **BSP radio SMPS**

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

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

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

```
#define IS_DCDC_SUPPORTED                    1U
```

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

4.5 Maximum Tx RF output power

Power amplifier (PA) optimal settings are used to maximize its efficiency when the requested output power is lower than the nominal (22 dBm for HP RFO or 15 dBm for LP RFO). This configuration depends on RF matching network, and can be set in the board support package (BSP), so that the software can select the optimum PA configuration before transmitting a packet.

Table 10. BSP board maximum power configuration

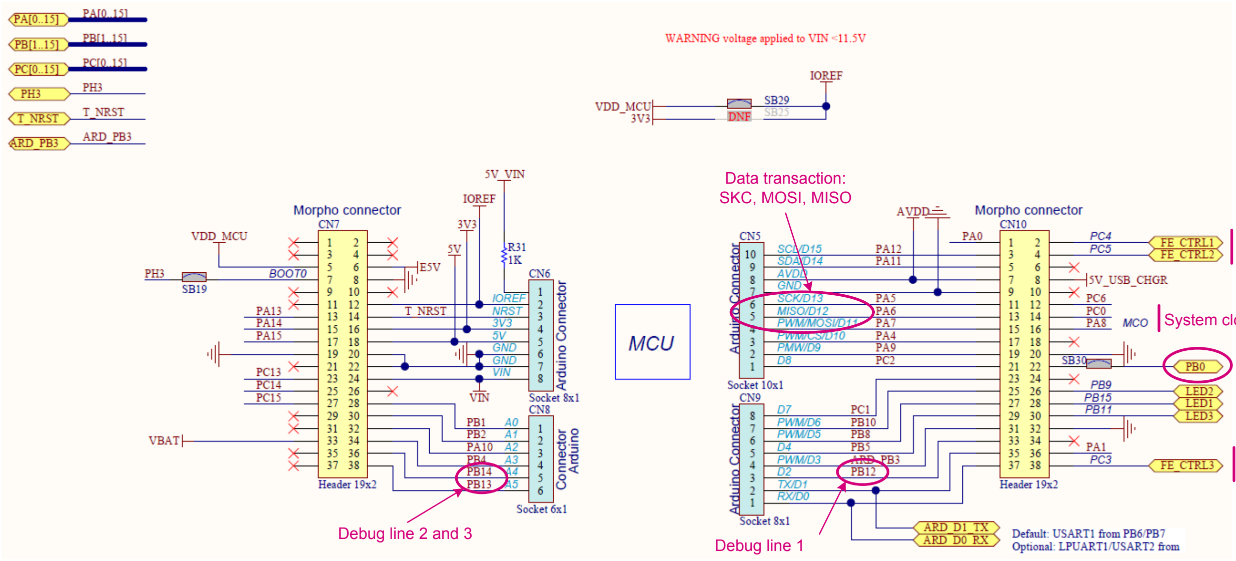
Function	Description
`int32_t BSP_RADIO_GetRFOMaxPowerConfig(BSP_RADIO_RFOMaxPowerConfig_TypeDef Config);`	Return the maximum output power for low-power or high-power RFO.

4.6 STM32WL Nucleo board schematic

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

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

Figure 4. NUCLEO-WL55JC schematic



Data transaction: SKC, MOSI, MISO

Control switchs 1 and 2

System clock

TCXO control voltage

Control switch 3

Debug line 2 and 3

Debug line 1

5 BSP B-WL5M-SUBG1 boards

The BSP driver provides a set of functions to manage radio RF switch settings. It also gives additional components available on B-WL5M-SUBG1 such as external flash memory, LEDs, and sensors (environmental, motion, etc.).

5.1 RF switch

The B-WL5M-SUBG1 board embeds a RF switch integrated in the STM32WL5MOC. There is only a read API to return the board configuration.

Table 11. **BSP radio switch**

Function	Description
`int32_t BSP_RADIO_GetTxConfig(void)`	Returns the board configuration: high power, low power, or both.

5.2 External components

The B-WL5M-SUBG1 board embeds multiples components:

- One User PushButton
- 3 LEDs
- Temperature/Barometer Sensors
- Accelerometer, Gyroscope and Magnetometer Sensors
- External flash memory

For more details about the defined APIs, refer to the User Manual document in the BSP directory.

6 Sigfox Stack description

The firmware of the STM32CubeWL MCU Package includes STM32WL resources such as:

- STM32WLxx_Nucleo drivers
- B-WL5M-SUBG1 drivers
- STM32WLxx HAL drivers
- Sigfox middleware
- SubGHz physical layer middleware
- Sigfox application example
- Utilities

The Sigfox middleware for STM32 microcontrollers is split into several modules:

- Sigfox Core library layer module
- Sigfox crypto module
- Sigfox Monarch (ST algorithm patent)

The Sigfox Core library implements a Sigfox medium access controller that interfaces with the Cmac library encrypting uplink payload and verifying downlink payload. The Cmac library interfaces with the Credentials library holding the cryptographic functions. This medium access controller also interfaces with the ST Monarch library.

The Sigfox Core library interfaces also with `i.e rf_api.c.and` and `mcu_api,cm` porting files in the user directory. It is not advised to modify these files.

The Sigfox Core, Sigfox test, cryptographic and Monarch library modules are provided in compiled object.

The libraries have been compiled with wchar32 and 'short enums'. These settings are used by default in IAR Embedded Workbench® and STM32CubeIDE.

For µVision Keil®, specific care must be taken. Tickbox *Short enums/wchar* must be unchecked and `fshort -enums` must be added in *Misc Controls* field.

Note: For dual-core applications, these settings must be applied to both cores to guaranty same enum formatting.

6.1 Sigfox certification

The system including the NUCLEO-WL55JC board and the STM32CubeWL firmware modem application has been verified by Sigfox Test Lab and passed the Sigfox Verified certification.

Nevertheless, the user end product based on STM32WL5x/Ex MCUs must pass again the Sigfox Verified and the Sigfox Ready™ certification before the user end-product commercialization.

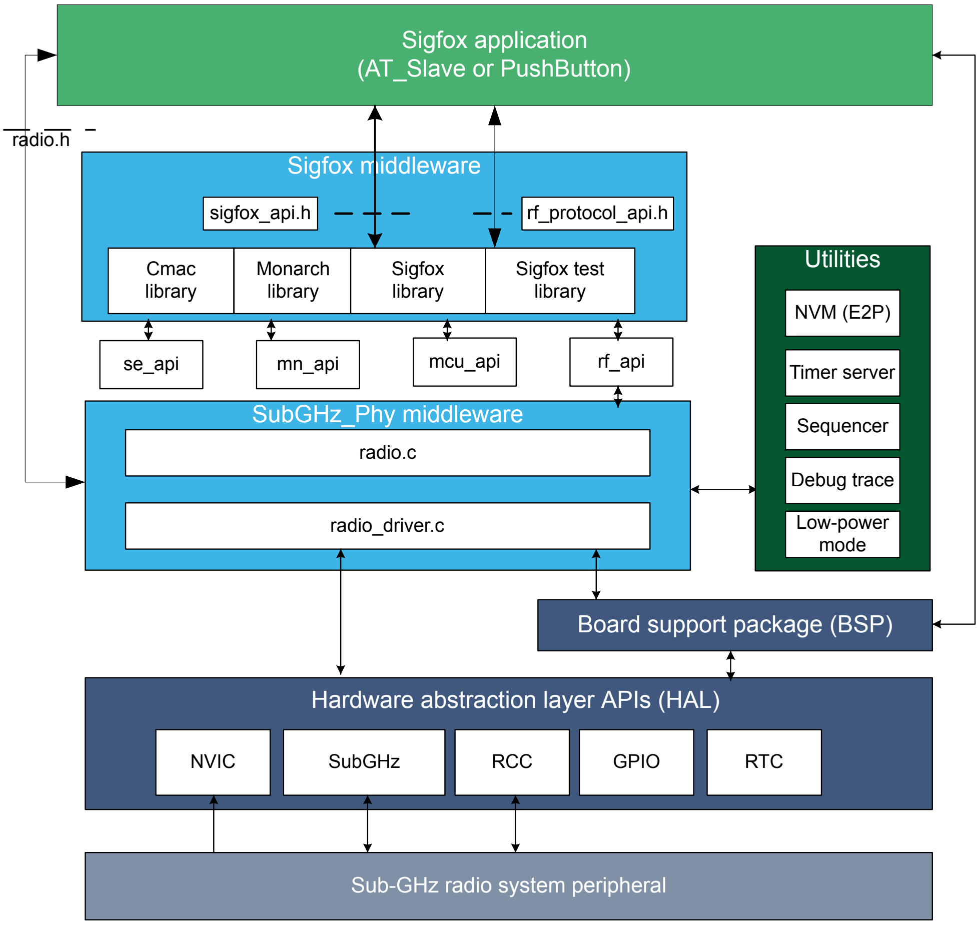
The Sigfox certification requires to provide the value of the rollover_counter 4096 that is implemented into STM32CubeWL.

6.2 Architecture

6.2.1 Static view

The figure below details the main design of the firmware for the Sigfox application.

Figure 5. Static Sigfox architecture



The HAL uses STM32Cube APIs to drive the hardware required by the application.

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

The Sigfox Core library embeds the medium access controller (MAC) as well as some security functions (see Section 7.1: Sigfox Core library for more details).

The application is built around an infinite loop including a scheduler. The scheduler processes tasks and events. When nothing remains to be done, the scheduler transitions to idle state and calls the low-power manager.

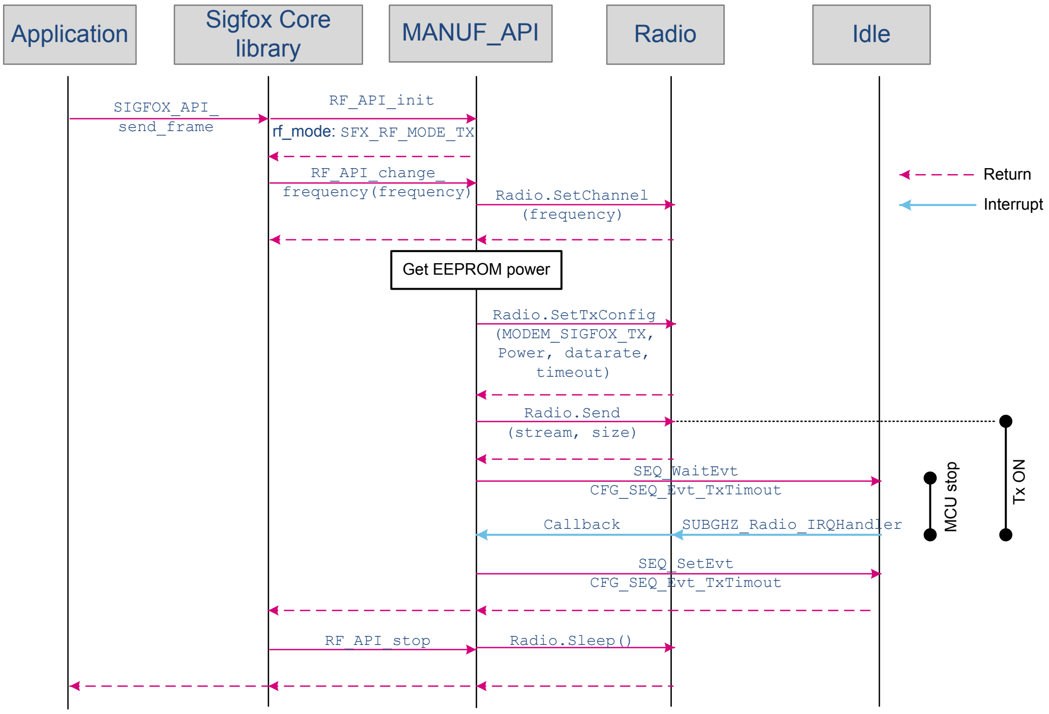
Typical application examples:

- AT layer to interface with external host (see Section 12.2: AT modem application)
- application reading and sending sensor data upon an action (see Section 12.3: PushButton application)

6.2.2 Dynamic view

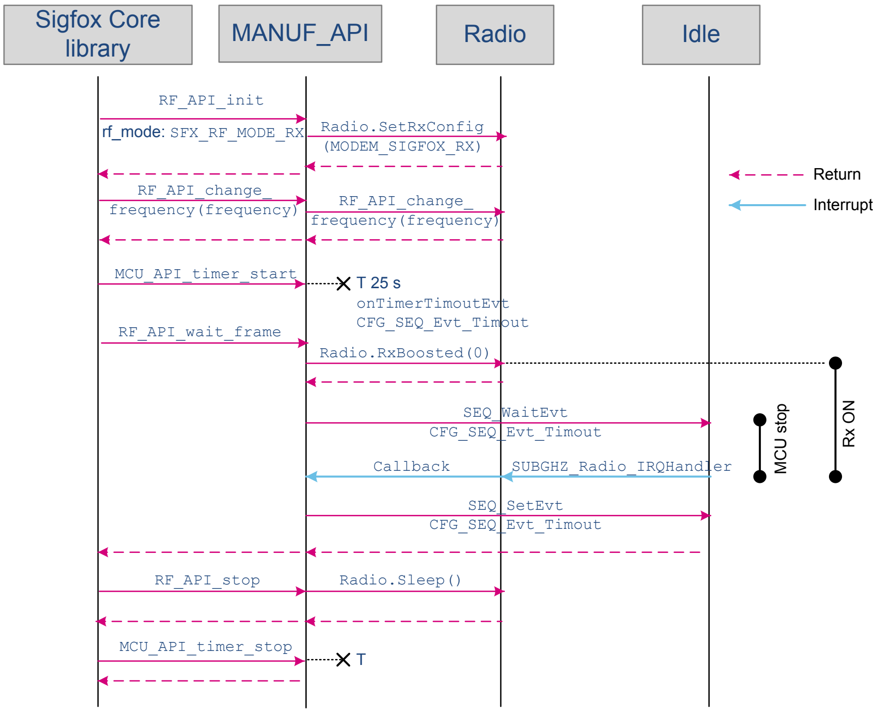
The message sequence chart (MSC) in the figure below depicts the dynamic calls between APIs in Tx mode (for one transmission).

Figure 6. Transmission MSC


Application
Sigfox Core library
MANUF_API
Radio
Idle
SIGFOX_API_send_frame
RF_API_init
rf_mode: SFX_RF_MODE_TX
RF_API_change_frequency(frequency)
Radio.SetChannel (frequency)
Return
Interrupt
Get EEPROM power
Radio.SetTxConfig (MODEM_SIGFOX_TX, Power, datarate, timeout)
Radio.Send (stream, size)
SEQ_WaitEvt CFG_SEQ_Evt_TxTimout
Callback
SUBGHZ_Radio_IRQHandler
SEQ_SetEvt CFG_SEQ_Evt_TxTimout
MCU stop
Tx ON
RF_API_stop
Radio.Sleep()

When a downlink window is requested, an Rx sequence starts after Rxdelay is elapsed (see Figure 2). When Rxdelay is elapsed, the sequence detailed in the figure below occurs.

Figure 7. Reception MSC



6.3 Required STM32 peripherals to drive the radio

Sub-GHz radio

The sub-GHz radio peripheral is accessed through the `stm32wlxx_hal_subghz` HAL.

The sub-GHz radio issues an interrupt through SUBGHZ_Radio_IRQHandler NVIC, to notify a TxDone or RxDone event. More events are listed in the product reference manual.

RTC

The RTC (real-time clock) calendar is used as 32-bit counter running in all power modes from the 32 kHz external oscillator. By default, the RTC is programed to provide 1024 ticks (subseconds) per second. The RTC is programed once at hardware initialization (when the MCU starts for the first time). The RTC output is limited to a 32-bit timer that corresponds to about a 48-day period.

Caution: *When changing the tick duration, the user must keep it below 1 ms.*

LPTIM

The LPTIM (low-power timer) is used for Monarch only. The LPTIM is set when a Monarch scan is requested, uses the LSE clock and issues an interrupt at 16384 Hz.

7 Sigfox middleware programming guidelines

7.1 Sigfox Core library

Embedded applications using the Sigfox Core library call SIGFOX_APIs to manage communication.

Table 12. Application level Sigfox APIs

Function	Description

`sfx_error_t SIGFOX_API_get_device_id`
`(sfx_u8 *dev_id);`
	Copies the ID of the device to the pointer given in parameter. The ID is 4-byte long and in hexadecimal format.

`sfx_error_t`
`SIGFOX_API_get_initial_pac`
`(sfx_u8 *initial_pac);`
	Gets the value of the PAC stored in the device. This value is used when the device is registered for the first time on the backend. The PAC is 8-byte long.

`sfx_error_t SIGFOX_API_open`
`(sfx_rc_t *rc);`
	Initializes the library and saves the input parameters once (cannot be changed until `SIGFOX_API_close()` is called)
	– `rc` is a pointer on the radio configuration zone. It is mandatory to use already existing defined RCs.

`sfx_error_t SIGFOX_API_close(void);`	Closes the library and stops the RF.

`sfx_error_t SIGFOX_API_send_frame`
`(sfx_u8 *customer_data,`
`sfx_u8 customer_data_length,`
`sfx_u8 *customer_response,`
`sfx_u8 tx_repeat,`
`sfx_bool initiate_downlink_flag);`
	Sends a standard Sigfox frame with customer payload.
	• `customer_data` cannot exceed 12 bytes
	• `customer_data_length`: length in bytes
	• `customer_response`: received response
	• `tx_repeat`:
	– when 0, sends one Tx.
	– when 1, sends three Tx.
	• `initiate_downlink_flag`: if set, the frame sent is followed by a receive downlink frame and an out-of-band Tx frame (voltage, temperature and RSSI).

`sfx_error_t SIGFOX_API_send_bit`
`(sfx_bool bit_value,`
`sfx_u8 *customer_response,`
`sfx_u8 tx_repeat,`
`sfx_bool initiate_downlink_flag);`
	Sends a standard Sigfox™ frame with null customer payload (shortest frame that Sigfox library can generate).
	• `bit_value`: bit sent
	• `customer_response`: received response
	• `tx_repeat`:
	– when 0, sends one Tx.
	– when 1, sends three Tx.
	• `initiate_downlink_flag`: if set, the frame sent is followed by a receive downlink frame and an out-of-band Tx frame (voltage, temperature and RSSI).

`sfx_error_t`
`SIGFOX_API_set_std_config`
`(sfx_u32 config_words[3],`
`sfx_bool timer_enable);`
	Configures specific variables for standard. Parameters have different meanings whether in FH or LBT mode.
	Note: this function has no influence in DC (see Section 12.2.21 for details).

Secondary APIs are described in `sigfox_api.h`. The library can be found in the `Middlewares\Third_Party\Sigfox\SigfoxLib` directory.

7.1.1 Open the Sigfox library

`ST_SIGFOX_API_open` must be called to initialize the Sigfox library before any other operation is performed.

This API requires the RC argument number representing the radio configuration zone (see Section 2.2).

For radio control zones 2 and 4, the FCC (federal communications commission) requires frequency hopping so the transmission frequency is not fixed (see Section 7.1.3 for more details on how to map the macro channels).

7.1.2 Send frames/bits

`ST_SIGFOX_API_send_frame` is the main Sigfox library function. This blocking function handles message exchange between the end node and the base stations.

An important parameter of this function is `initiate_downlink_flag` that selects different transmission behaviors as follows:

- `initiate_downlink_flag = 0`: The library requests only uplink frame. The sent frame is transmitted once if `tx_repeat = 0`, or three times if `tx_repeat = 1`, with a 500 ms pause (see Figure 1).The transmit payload can be maximum 12-byte long.
- `initiate_downlink_flag = 1`: The frame to be sent is transmitted three times with a 500 ms pause. A 25 s Rx window opens 20 s after the end of the first repetition (see Figure 2). If the reception is successful, the received 8-byte downlink frame is stored in the buffer location indicated by the `customer_response` buffer.

7.1.3 Set standard configuration

The FCC allows the transmitters to choose certain macro channels to implement a frequency-hopping pattern authorized by the standard. The channel map is specified in the first argument of `SIGFOX_API_set_std_config`, that consists of an array of three 32-bit configuration words.

A macro-channel consists of six micro channels centered about the center frequency of the macro channel and separated by 25 kHz. For example, in the 902.2 MHz macro channel, the six micro channels are 902.1375 MHz, 902.1625 MHz, 902.1875 MHz, 902.2125 MHz, 902.2375 MHz, and 902.2625 MHz.

A typical Sigfox frame lasts between 200 ms and 350 ms at 600 bit/s, and FCC mandates a max dwell time of 400 ms. A transmitter cannot return to a given channel before 20 s. Therefore, at least 20 / 0.4 = 50 channels must be used for continuous transmission.

Actually, a device only transmits a few frames per day (144 messages maximum). Enabling one macro channel only and inserting 10 s delays between two groups of three repeated frames (one frame per micro channel means six micro channels) pass the regulation limits.

Each bit of the config_words[0,1,2] array represents a macro channel according to the mapping described in the table below.

Table 13. **Macro channel mapping**

Bit	config_words[0] Frequency mapping (MHz)	config_words[1] Frequency mapping (MHz)	config_words[2] Frequency mapping (MHz)
0	902.2	911.8	921.4
1	902.5	912.1	921.7
2	902.8	912.4	922
3	903.1	912.7	922.3
4	903.4	913	922.6
5	903.7	913.3	922.9
6	904	913.6	923.2
7	904.3	913.9	923.5
8	904.6	914.2	923.8
9	904.9	914.5	924.1
10	905.2	914.8	924.4
11	905.5	915.1	924.7

Bit	config_words[0] Frequency mapping (MHz)	config_words[1] Frequency mapping (MHz)	config_words[2] Frequency mapping (MHz)
12	905.8	915.4	925
13	906.1	915.7	925.3
14	906.4	916	925.6
15	906.7	916.3	925.9
16	907	916.6	926.2
17	907.3	916.9	926.5
18	907.6	917.2	926.8
19	907.9	917.5	927.1
20	908.2	917.8	927.4
21	908.5	918.1	927.7
22	908.8	918.4	928
23	909.1	918.7	928.3
24	909.4	919	928.6
25	909.7	919.3	928.9
26	910	919.6	929.2
27	910.3	919.9	929.5
28	910.6	920.2	929.8
29	910.9	920.5	930.1
30	911.2	920.8	930.4
31	911.5	921.1	930.7

A macro channel is enabled only when the corresponding config_words[x] bit is set to 1. For example, bit 0 of config_words[0] corresponds to channel 1 while bit 30 of config_words[1] corresponds to channel 63. At least nine macro channels must be enabled to meet the FCC specifications.

In the following long message configuration example, channels 1 to 9 are enabled with frequencies ranging from 902.2 MHz to 904.6 MHz:

- config_words[0] = [0x0000 01FF]
- config_words[1] = [0x0000 0000]
- config_words[2] = [0x0000 0000]

By default, the Sigfox application sets one macro channel with `timer_enable = 1`. The macro channel 1 in RC2 has a 902.2 MHz operational frequency and the macro channel 63 in RC4 has a 920.8 MHz operational frequency. This is the short message configuration operational for Sigfox (see defined `RCx_SM_CONFIG` value in `sigfox_api.h`).

A delay (`timer_enable`) is implemented to avoid one micro channel to be re-used with an interval lower than 20 s. When using one macro channel only (six micro channels) performing three repetitions, this delay corresponds to 10 s. When using two macro channels (12 micro channels), the delay automatically becomes 5 s.

For certification test purposes, `timer_enable` may be set to 0, but must be set to 1 otherwise. The default settings can nevertheless be modified using the ATS400 command (Section 12.2.21) to speed up the certification process.

7.2 Sigfox Addon RF protocol library

This library is used to test the device for Sigfox Verified certification. Ultimately, this library can be removed from the build once certified.

Table 14. **Sigfox Addon Verified library**

Function	Description

`sfx_error_t`
`ADDON_SIGFOX_RF_PROTOCOL_API_test_mode`
`(sfx _rc_enum_t rc_enum, sfx_test_mode_t test_mode);`
	Executes the test modes needed for the Sigfox Verified certification:
	• `rc_enum`: rc at which the test mode is run
	• `test_mode`: test mode to run

`sfx_error_t`
`ADDON_SIGFOX_RF_PROTOCOL_API_monarch_test_ mode`
`(sfx_rc_enum_t rc_enum, sfx_test_mode_t test_mode,`
`sfx_u8 rc_capabilities);`
	This function executes the Monarch test modes needed for Sigfox RF and protocol tests.

This library is located in `Middlewares\Third_Party\Sigfox\SigfoxLibTest`.

7.3 Cmac library

The Cmac library stores the keys, the PAC and the IDs.

Table 15. **Cmac APIs**

Function	Description

`sfx_u8 SE_API_get_device_id`
`(sfx_u8 dev_id[ID_LENGTH]);`
	This function copies the device ID in `dev_id`.

`sfx_u8 SE_API_get_initial_pac (sfx_u8 *initial_pac);`	Gets the initial PAC.

`sfx_u8 SE_API_secure_uplink_message`
`(sfx_u8 *customer_data,`
`sfx_u8 customer_data_length,`
`sfx_bool initiate_downlink_frame,`
`sfx_se_frame_type_t frame_type,`
`sfx_bool *send_rcsync,`
`sfx_u8 *frame_ptr, sfx_u8 *frame_length);`
	Generates an uplink frame bitstream.

`sfx_u8 SE_API_verify_downlink_message`
`(sfx_u8 *frame_ptr, sfx_bool *valid);`
	Authenticates a received message and decrypts its payload.

The Cmac library is located in directory `\Middlewares\Third_Party\Sigfox\Crypto`.

Note:
- *This library interfaces the `se_nvm` functions to store/retrieve `SFX_SE_NVMEM_BLOCK_SIZE` bytes from the non-volatile memory.*
- *`se_api.h` is the interface to the Sigfox secure element that can be either a physical secure element, or emulated by firmware with the Cmac library and the Credentials library.*

7.4 Credentials library

The Credentials library can access the keys, the PAC and the IDs. It can also encrypt data with the Sigfox key.

Table 16. Credentials APIs

Function	Description
`void CREDENTIALS_get_dev_id(uint8_t* dev_id);`	Gets the device ID.
`void CREDENTIALS_get_initial_pac (uint8_t* pac);`	Gets the device initial PAC.
`sfx_bool CREDENTIALS_get_payload_encryption_fl ag(void);`	Gets the encryption flag. Sets to false by default (see Section 12.2.10: ATS411 - Payload encryption).

`sfx_error_t CREDENTIALS_aes_128_cbc_encrypt`
`(uint8 _t* encrypted_data, uint8_t* data_to_encrypt, uint8_t block_len);`
	Encrypts data with the secret key. The secret key can be set to `CMAC_KEY_PRIVATE` or `CMAC_KEY_PUBLIC` (see Section 12.2.9: ATS410 - Encryption key).

`sfx_error_t CREDENTIALS_wrap_session_key`
`(uint8_t *data, uint8_t blocks)`
	Derives a session key based on the Sigfox secret key

`sfx_error_t`
`CREDENTIALS_aes_128_cbc_encrypt_with_session_key`
`(uint8_t *encrypted_data,`
`uint8_t *data_to_encrypt, uint8_t blocks)`
	Encrypts data with the session key.

7.5 Monarch library

The Monarch APIs are defined in `sigfox_monarch_api.h`.

Table 17. Monarch APIs

Function	Description

`sfx_error_t`
`SIGFOX_MONARCH_API_execute_rc_scan`
`(sfx_u8 rc_capabilities_bit_mask, sfx_u16 timer,`
`sfx_timer_unit_enum_t unit, sfx_u8`
`(* app_callback_handler)`
`(sfx_u8 rc_bit_mask, sfx_s16 rssi));`
	Starts a Monarch scan.
	• `sfx_u8 rc_capabilities_bit_mask`
	• `sfx_u16 timer`: scan duration value
	• `sfx_timer_unit_enum_t unit`: unit of timer
	• `app_callback_handler`: function called by the Sigfox library when the scan is completed

`sfx_error_t`
`SIGFOX_MONARCH_API_stop_rc_scan(void);`
	Stops an ongoing Monarch scan.

8 SubGHz_Phy layer middleware description

The radio abstraction layer is composed of two layers:

- high-level layer (radio.c)
  It provides a high-level radio interface to the stack middleware. It also maintains radio states, processes interrupts and manages timeouts. It records callbacks and calls them when radio events occur.
- low-level radio drivers
  It is an abstraction layer to the RF interface. This layer knows about the register name and structure, as well as detailed sequence. It is not aware about hardware interface.

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

The SubGHz_Phy middleware directory is divided in two parts:

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

radio_conf.h contains radio application configuration like RF_WAKEUP_TIME, DC/DC dynamic settings, XTAL_FREQ.

8.1 Middleware radio driver structure

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

Table 18. **Radio_s structure callbacks**

Callback	Description
void Init (RadioEvents_t *events	Initializes the radio.
RadioState_t GetStatus (void	Returns the current radio status.
void SetModem (RadioModems_t modem)	Configures the radio with the given modem.
void SetChannel (uint32_t freq)	Sets the channel frequency.
bool IsChannelFree (uint32_t freq, uint32_t rxBandwidth, int16_t rssiThresh, uint32_t maxCarrierSenseTime)	Checks if the channel is free for the given time.
uint32_t Random (void)	Generates a 32-bit random value based on the RSSI readings.
void SetRxConfig (RadioModems_t modem, uint32_t bandwidth, uint32_t datarate, uint8_t coderate, uint32_t bandwidthAfc, uint16_t preambleLen, uint16_t symbTimeout, bool fixLen, uint8_t payloadLen, bool crcOn, bool freqHopOn, uint8_t hopPeriod, bool iqInverted, bool rxContinuous)	Sets the reception parameters.
void SetTxConfig (RadioModems_t modem, int8_t power, uint32_t fdev, uint32_t bandwidth, uint32_t datarate, uint8_t coderate, uint16_t preambleLen, bool fixLen, bool crcOn, bool freqHopOn, uint8_t hopPeriod, bool iqInverted, uint32_t timeout)	Sets the transmission parameters.
bool CheckRfFrequency (uint32_t frequency)	Checks if the given RF frequency is supported by the hardware.

Callback	Description
`uint32_t TimeOnAir ( RadioModems_t modem, uint32_t bandwidth, uint32_t datarate, uint8_t coderate, uint16_t preambleLen, bool fixLen, uint8_t payloadLen, bool crcOn )`	Computes the packet time on air (in ms), for the given payload.
`radio_status_t Send ( uint8_t *buffer, uint8_t size )`	Prepares the packet to be sent and starts the radio in transmission.
`void Sleep ( void )`	Sets the radio in Sleep mode.
`void Standby ( void )`	Sets the radio in Standby mode.
`void Rx ( uint32_t timeout )`	Sets the radio in reception mode for the given time.
`void StartCad ( void )`	Starts a CAD (channel activity detection).
`void SetTxContinuousWave ( uint32_t freq, int8_t power, uint16_t time )`	Sets the radio in continuous-wave transmission mode.
`int16_t Rssi ( RadioModems_t modem )`	Reads the current RSSI value.
`void Write ( uint16_t addr, uint8_t data )`	Writes the radio register at the specified address.
`uint8_t Read ( uint16_t addr )`	Reads the radio register at the specified address.
`void WriteRegisters ( uint16_t addr, uint8_t *buffer, uint8_t size )`	Writes multiple radio registers starting at address.
`void ReadRegisters ( uint16_t addr, uint8_t *buffer, uint8_t size )`	Reads multiple radio registers starting at address.
`void SetMaxPayloadLength ( RadioModems_t modem, uint8_t max ) )`	Sets the maximum payload length.
`void SetPublicNetwork ( bool enable )`	Sets the network to public or private, and updates the sync byte.
`uint32_t GetWakeupTime ( void )`	Gets the time required for the radio to exit Sleep mode.
`void IrqProcess ( void )`	Processes radio IRQ.
`void RxBoosted ( uint32_t timeout )`	Sets the radio in reception mode with max LNA gain for the given time.
`void SetRxDutyCycle ( uint32_t rxTime, uint32_t sleepTime )`	Sets the Rx duty-cycle management parameters.
`void TxPrbs ( void )`	Sets the transmitter in continuous PRBS mode.
`void TxCw ( int8_t power )`	Sets the transmitter in continuous unmodulated carrier mode.
`int32_t RadioSetRxGenericConfig ( GenericModems_t modem, RxConfigGeneric_t* config, uint32_t rxContinuous, uint32_t symbTimeout )`	Sets the reception parameters with more configuration fields.
`int32_t RadioSetTxGenericConfig ( GenericModems_t modem, TxConfigGeneric_t* config, int8_t power, uint32_t timeout )`	Sets the transmission parameters with more configuration fields.
`int32_t TransmitLongPacket ( uint16_t payload_size, uint32_t timeout,void (*TxLongPacketGetNextChunkCb) ( uint8_t** buffer, uint8_t buffer_size ) )`	Starts sending long Packet, packet may be short.

Callback	Description
`int32_t ReceiveLongPacket ( uint8_t boosted_mode, uint32_t timeout, void (*RxLongStorePacketChunkCb) ( uint8_t* buffer, uint8_t chunk_size ) )`	Starts receiving long Packet, packet maybe short.
`radio_status_t LrFhssSetCfg ( const radio_lr_fhss_cfg_params_t *cfg_params )`	Configures the radio LR-FHSS modem parameters.
`radio_status_t LrFhssGetTimeOnAirInMs ( const radio_lr_fhss_time_on_air_params_t *params, uint32_t *time_on_air_in_ms );`	Gets the time on air in millisecond for LR-FHSS packet.

8.2 Radio IRQ interrupts

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

Table 19. **Radio IRQ bit mapping and definition**

Bit	Source	Description	Packet type	Operation
0	`txDone`	Packet transmission finished	LoRa and GFSK	Tx
1	`rxDone`	Packet reception finished		Rx
2	`PreambleDetected`	Preamble detected		
3	`SyncDetected`	Synchronization word valid	GFSK	
4	`HeaderValid`	Header valid	LoRa	
5	`HeaderErr`	Header error		
6	`Err`	Preamble, sync word, address, CRC or length error	GFSK	
	`CrcErr`	CRC error	LoRa	
7	`CadDone`	Channel activity detection finished		CAD
8	`CadDetected`	Channel activity detected		
9	`Timeout`	Rx or Tx timeout	LoRa and GFSK	Rx and Tx
10	-	RFU	-	-
11				
12				
13				
14	`LrFhssHop`	Asserted at each hop, in Long Range FHSS, after the PA has ramped-up again	LR-FHSS	Tx
15	-	RFU	-	-

For more details, refer to the product reference manual.

9 EEPROM driver

The EEPROM interface (sgfx_eeprom_if.c) is designed above ee.c to abstract the EEPROM driver. The EEPROM is physically placed at EE_BASE_ADRESS defined in the utilities_conf.h.

Table 20. **EEPROM APIs**

Function	Description
void E2P_Init (void);	DEFAULT_FACTORY_SETTINGS is written when the EEPROM is empty.
void E2P_RestoreFs (void);	DEFAULT_FACTORY_SETTINGS are restored .
Void E2P_Write_XXX	Writes data in the EEPROM. For example: void E2P_Write_VerboseLevel(uint8_t verboselevel);
E2P_Read_XXX	Reads XXX from the EEPROM For example: sfx_rc_enum_t E2P_Read_Rc(void);

10 Utilities description

Utilities are located in the `\Utilities` directory.

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

10.1 Sequencer

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

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

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

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

The sequencer provides the following features:

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

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

- Set the number of maximum of supported functions, by defining a value for `UTIL_SEQ_CONF_TASK_NBR`.
- Register a function to be supported by the sequencer with `UTIL_SEQ_RegTask()`.
- Start the sequencer by calling `UTIL_SEQ_Run()` to run a background while loop.
- Call `UTIL_SEQ_SetTask()` when a function needs to be executed.

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

Table 21. Sequencer APIs

Function	Description
`void UTIL_SEQ_Idle( void )`	Called (in critical section - PRIMASK) when there is nothing to execute.
`void UTIL_SEQ_Run(UTIL_SEQ_bm_t mask_bm )`	Requests the sequencer to execute functions that are pending and enabled in the mask `mask_bm`.
`void UTIL_SEQ_RegTask(UTIL_SEQ_bm_t task_id_bm, uint32_t flags, void (*task)( void ))`	Registers a function (task) associated with a signal (`task_id_bm`) in the sequencer. The `task_id_bm` must have a single bit set.
`void UTIL_SEQ_SetTask( UTIL_SEQ_bm_t taskId_bm, uint32_t task_Prio )`	Requests the function associated with the `task_id_bm` to be executed. The `task_prio` is evaluated by the sequencer only when a function has finished. If several functions are pending at any one time, the one with the highest priority (0) is executed.
`void UTIL_SEQ_WaitEvt( UTIL_SEQ_bm_t EvtId_bm );`	Waits for a specific event to be set.
`void UTIL_SEQ_SetEvt( UTIL_SEQ_bm_t EvtId_bm );`	Sets an event that waits with `UTIL_SEQ_WaitEvt()`.

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

Table 22. While-loop standard vs. sequencer implementation

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

Void some_Irq(void) /*handler context*/
{
   flag2=1; /*will execute Fct2*/
}</td><td>/*Flag1 and Flag2 are bitmasks*/
UTIL_SEQ_RegTask(flag1, Fct1());
UTIL_SEQ_RegTask(flag2, Fct2());

While(1)
{
    UTIL_SEQ_Run();
}

void UTIL_SEQ_Idle(void)
{
  LPM_EnterLowPower( );
}

Void some_Irq(void) /*handler context*/
{
 UTIL_SEQ_SetTask(flag2); /*will execute Fct2*/
}</td></tr>
</table>

10.2 Timer server

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

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

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

Table 23. **Timer server APIs**

Function	Description

`UTIL_TIMER_Status_t`
`UTIL_TIMER_Init(void)`
	Initializes the timer server.

`UTIL_TIMER_Status_t UTIL_TIMER_Create`
`(UTIL_TIMER_Object_t *TimerObject, uint32_t PeriodValue,`
`UTIL_TIMER_Mode_t Mode, void (*Callback)`
`(void*), void *Argument)`
	Creates the timer object and associates a callback function when timer elapses.

`UTIL_TIMER_Status_t`
`UTIL_TIMER_SetPeriod(UTIL_TIMER_Object_t *TimerObject,`
`uint32_t NewPeriodValue)`
	Updates the period and starts the timer with a timeout value (milliseconds).

`UTIL_TIMER_Status_t UTIL_TIMER_Start`
`(UTIL_TIMER_Object_t *TimerObject)`
	Starts and adds the timer object to the list of timer events.

`UTIL_TIMER_Status_t UTIL_TIMER_Stop`
`(UTIL_TIMER_Object_t *TimerObject)`
	Stops and removes the timer object from the list of timer events.

10.3 Low-power functions

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

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

Table 24. **Low-power APIs**

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

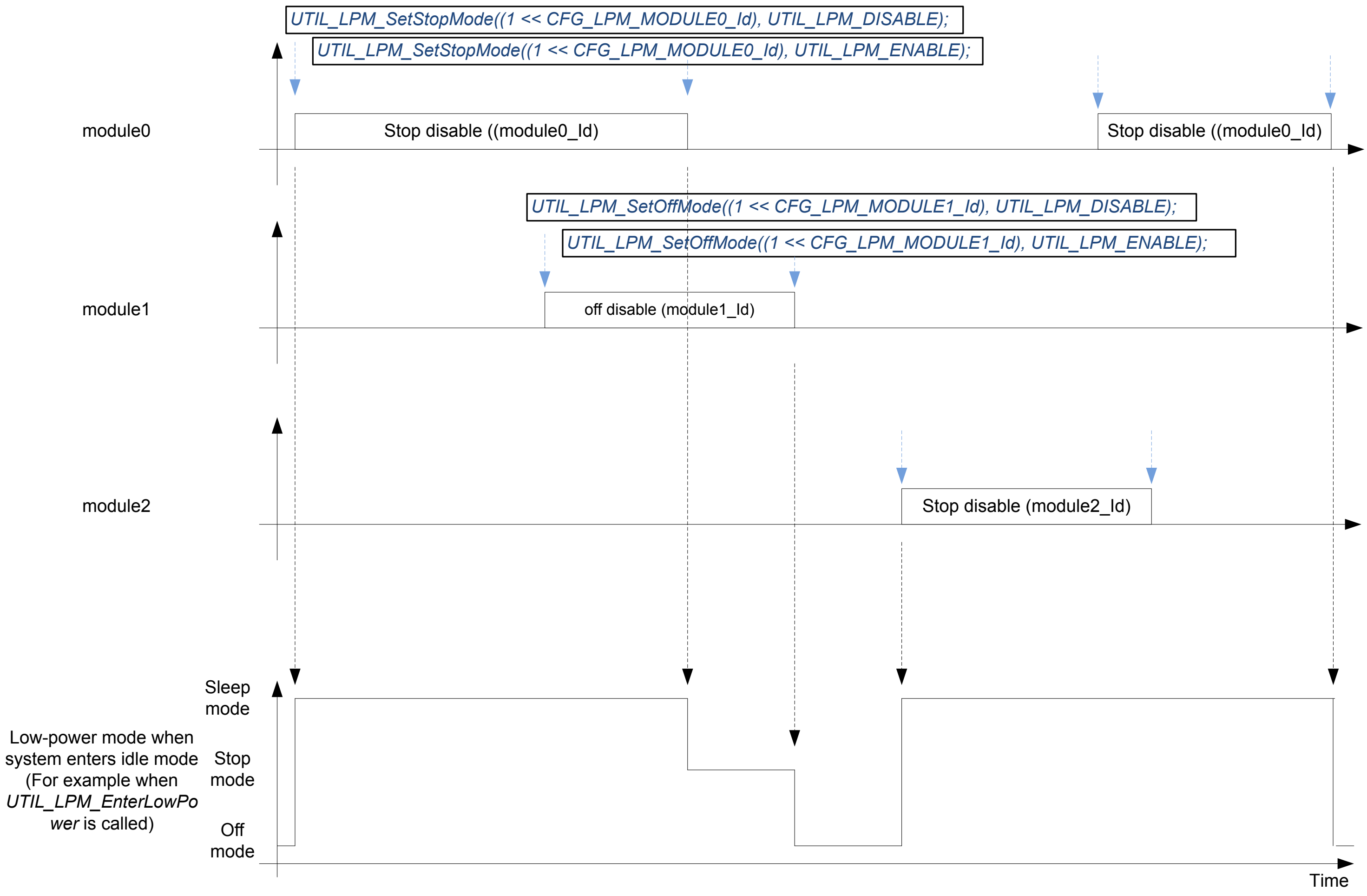
1. *Bitmaps for which the shift values are defined in* `utilities_def.h`.

The default low-power mode is Off mode, which may be Standby or Shutdown mode (defined in `void PWR_EnterOffMode (void)` from Table 25):

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

Figure 8 depicts the behavior with three different firmware modules setting dependently their low-power requirements and low-power mode, selected when the system enters a low-power mode.

Figure 8. **Example of low-power mode dynamic view**


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

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

Table 25. **Low-level APIs**

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

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

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

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

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

10.4 System time

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

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

Table 26. **System time functions**

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

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

1. *The system time reference is the UNIX epoch starting January 1st, 1970.*
2. `SysTimeMkTime` *and* `SysTimeLocalTime` *are also provided to convert epoch into* `tm` *structure as specified by the* `time.h` *interface.*

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

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

10.5 Trace

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

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

Table 27. Trace functions

Function	Description
UTIL_ADV_TRACE_Status_t UTIL_ADV_TRACE_Init(void)	TraceInit must be called at the application initialization. Initializes the com or vcom hardware in DMA mode and registers the callback to be processed at DMA transmission completion.
UTIL_ADV_TRACE_Status_t UTIL_ADV_TRACE_COND_FSend(uint32_t VerboseLevel, uint32_t Region, uint32_t TimeStampState, const char *strFormat, ...)	Converts string format into a buffer and posts it to the circular queue for printing.
UTIL_ADV_TRACE_Status_t UTIL_ADV_TRACE_COND_Send(uint32_t VerboseLevel, uint32_t Region, uint32_t TimeStampState, const uint8_t *pdata, uint16_t length)	Posts data of length = len and posts it to the circular queue for printing.
UTIL_ADV_TRACE_Status_t UTIL_ADV_TRACE_COND_ZCSend_Allocation(uint32_t VerboseLevel, uint32_t Region, uint32_t TimeStampState, uint16_t length,uint8_t **pData, uint16_t *FifoSize, uint16_t *WritePos)	Writes user formatted data directly in the FIFO (Z-Cpy).

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

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

The UTIL_ADV_TRACE_COND_FSend (..) function can be used:

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

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

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

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

  where:

  – VL is the VerboseLevel of the trace.

  – TS allows a timestamp to be added to the trace (TS_ON or TS_OFF).

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

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

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

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

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

```
#define UTIL_ADV_TRACE_TMP_BUF_SIZE 256U
```

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

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

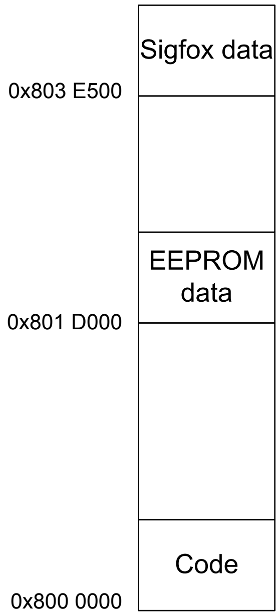
-
  ```
  void UTIL_ADV_TRACE_PostSendHook (void)
  { UTIL_LPM_SetStopMode((1 << CFG_LPM_UART_TX_Id) , UTIL_LPM_ENABLE );}
  ```

11 Memory section

The code is placed at 0x0800 0000. The sigfox_data (Credentials) is placed at 0x0803 E500 (can be modified in the scatter file).

Also the EEPROM is emulated at address 0x0801 D000 (EE_BASE_ADRESS) to store the NVM data that must be retained even if the power supply is lost.

Figure 9. Memory mapping

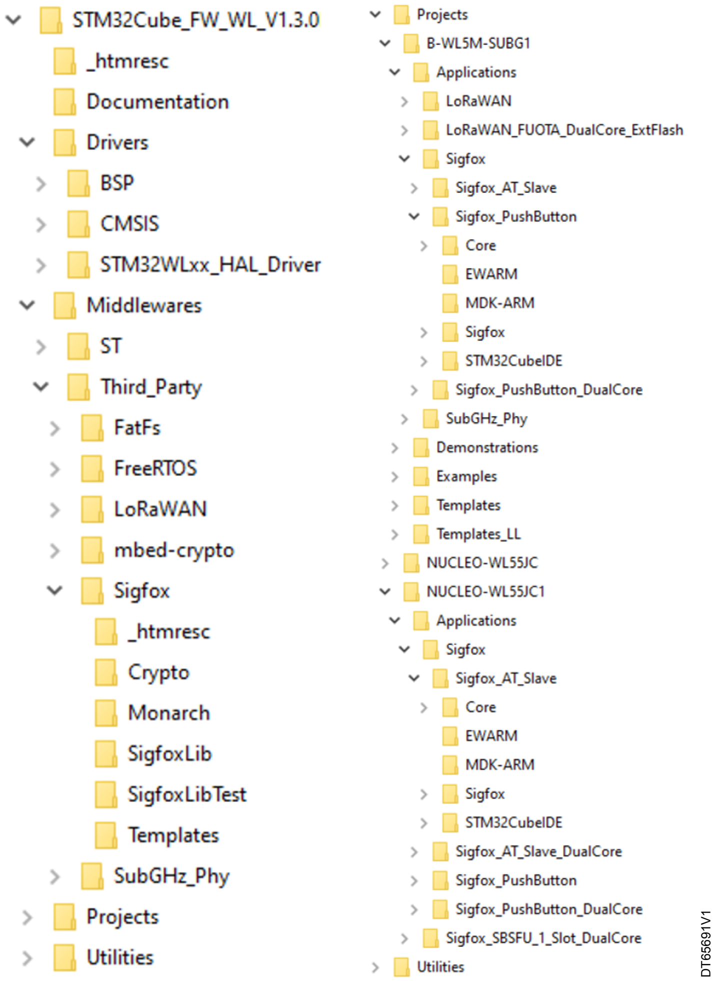


12 Application description

12.1 Firmware package

When the user unzips the firmware of the STM32CubeWL MCU Package, the folder structure is the one shown in the figure below.

Figure 10. Package overview



```
STM32Cube_FW_WL_V1.3.0
  _htmresc
  Documentation
  Drivers
    BSP
    CMSIS
    STM32WLxx_HAL_Driver
  Middlewares
    ST
    Third_Party
      FatFs
      FreeRTOS
      LoRaWAN
      mbed-crypto
      Sigfox
        _htmresc
        Crypto
        Monarch
        SigfoxLib
        SigfoxLibTest
        Templates
      SubGHz_Phy
  Projects
  Utilities
```

```
Projects
  B-WL5M-SUBG1
    Applications
      LoRaWAN
      LoRaWAN_FUOTA_DualCore_ExtFlash
      Sigfox
        Sigfox_AT_Slave
        Sigfox_PushButton
          Core
          EWARM
          MDK-ARM
          Sigfox
          STM32CubeIDE
        Sigfox_PushButton_DualCore
      SubGHz_Phy
    Demonstrations
    Examples
    Templates
    Templates_LL
  NUCLEO-WL55JC
  NUCLEO-WL55JC1
    Applications
      Sigfox
        Sigfox_AT_Slave
          Core
          EWARM
          MDK-ARM
          Sigfox
          STM32CubeIDE
        Sigfox_AT_Slave_DualCore
        Sigfox_PushButton
        Sigfox_PushButton_DualCore
      Sigfox_SBSFU_1_Slot_DualCore
Utilities
```

The firmware of the STM32CubeWL contains two Sigfox applications: Sigfox_AT_Slave and Sigfox_PushButton.

12.2 AT modem application

The purpose of this application is to implement a Sigfox modem controlled though the AT command interface over UART by an external host that can be a host-microcontroller embedding the application and the AT driver or simply a computer executing a terminal. The AT_Slave application implements the Sigfox Stack that 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.

This application targets the NUCLEO_WL55JC1 development board with STM32WL55JC (order codes NUCLEO-WL55JC1 for high-frequency band) and B-WL5M-SUB1 connectivity expansion board with STM32WL5M.

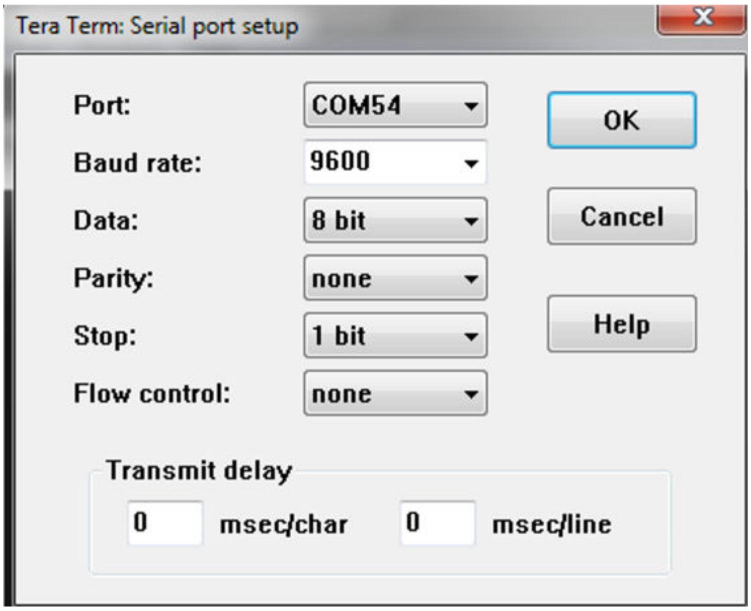
To launch the Sigfox_AT_Slave project, the user must go to `\Projects\<target>\Applications\Sigfox\Sigfox_AT_Slave` and choose one toolchain folder (in the IDE environment).

12.2.1 UART interface

In this example, the LPUART is used at 9600 baud. The device can receive a character while in Stop 2 mode .

Tera Term is used as terminal to control the Sigfox modem, with the settings of the figure below.

Figure 11. Tera Term serial port setup



Tera Term: Serial port setup

Port: COM54

Baud rate: 9600

Data: 8 bit

Parity: none

Stop: 1 bit

Flow control: none

Transmit delay

0 msec/char 0 msec/line

The available commands are given in Section 12.2.3 to Section 12.2.24 with the following format:

- All commands setting parameters are in the form `ATXX=Y<CR>`.
- All commands getting parameters are in the form `ATXX=?<CR>`.

12.2.2 Default parameters

The default parameters when the program starts for the first time (EEPROM empty) are:

- RC1 default values for the region configuration
- 13 dBm output power
- default key to private

These default values can be changed by modifying `E2P_RestoreFs` in the `\Projects\<target>\Applications\Sigfox\Sigfox_AT_Slave\Sigfox\App\sgfx_eeprom_if.c` configuration file.

The default private key and private ID are the test keys described in the Sigfox Test specification. They are stored in the `\Projects\<target>\Applications\Sigfox\Sigfox_AT_Slave\Sigfox\App\sigfox_data.h` file.

12.2.3 AT? - Available commands

Description	Attention is used to check if the link is working properly. AT? provides the short help of all supported commands.
Syntax	`AT?<CR>`
Arguments	None
Response	None
Result code	`<CR><LF>OK<CR><LF>`

General format of the AT commands is described below:

- `AT+<CMD>` runs the `<CMD>\r\n"`.
- `AT+<CMD>?` provides a short help of a given command.
- `AT+<CMD>=<value>` sets the value or runs with parameters `\r\n"`.
- `AT+<CMD>=?` is used to get the value of a given command.

Possible error status are:

- `OK`: command run correctly without error.
- `AT_ERROR`: Generic error
- `AT_PARAM_ERROR`: parameter of the command is wrong.
- `AT_BUSY_ERROR`: Sigfox modem busy, so the command could not complete.
- `AT_TEST_PARAM_OVERFLOW`: parameter is too long.
- `AT_LIB_ERROR`: Sigfox library generic error
- `AT_TX_TIMEOUT`: Tx not possible due to CS (LBT regions only)
- `AT_RX_TIMEOUT`: no Rx frame received during downlink window
- `AT_RX_ERROR`: error detection during the reception of the command
- `AT_RECONF_ERROR`

12.2.4 ATZ - Reset

Description	Generates a NVIC reset impacting the whole system (including radio and microprocessor).
Syntax	`ATZ<CR>`
Arguments	None
Response	None
Result code	None

This command only resets the device. The EEPROM data is maintained (see Section 12.2.5: AT$RFS - Factory settings).

12.2.5 AT$RFS - Factory settings

Description	Restores the factory setting defined in `sgfx_eeprom_if.c` in `E2P_RestoreFs` function.
Syntax	`AT$RFS<CR>`
Arguments	None
Response	None
Result code	`<CR><LF>OK<CR><LF>`

12.2.6 AT+VER - Firmware and library versions

Description	Gets the version of firmware and libraries.
Syntax	`AT+VER<CR>`
Arguments	None
Response	Version of firmware and libraries
Result code	`<CR><LF>OK<CR><LF>`

12.2.7 AT$ID - Device ID

Description	Gets the 32-bit device ID.

Syntax	`AT$ID<CR>`
	`AT$ID=?<CR>`

Arguments	None
Response	`Id<CR><LF`: Id on 4 bytes from MSB to LSB (8 ASCII)
Result code	`<CR><LF>OK<CR><LF>`

12.2.8 AT$PAC - Device PAC

Description	Gets the 8-bit device PAC.

Syntax	`AT$PAC<CR>`
	`AT$PAC=?<CR>`

Arguments	None
Response	`PAC<CR><LF`: PAC on 8 bytes (16 ASCII)
Result code	`<CR><LF>OK<CR><LF>`

12.2.9 ATS410 - Encryption key

Description	Sets or gets the configuration of the device encryption key.

Syntax	`ATS410=Arguments<CR>`
	`ATS410=?<CR>`

Arguments	`0` : use private key `1`: use public key
Response	`Encryption Key Configuration <CR><LF`
Result code	`<CR><LF>OK<CR><LF>`

By default, the payload encryption is OFF.

12.2.10 ATS411 - Payload encryption

Description	Sets or gets the device payload encryption mode.

Syntax	`ATS411=Arguments<CR>`
	`ATS411=?<CR>`

Arguments	`0` : payload encryption OFF `1`: payload encryption ON
Response	`Payload Encryption Configuration <CR><LF>`
Result code	`<CR><LF>OK<CR><LF>`

12.2.11 AT$SB - Bit status

Description	Sends a bit to the Sigfox network.

Syntax	`AT$SB=<bit value>{,{,<Optional NbTxFlag>}<CR>`
	`{,<Optional NbTxFlag>}<CR>`

Arguments	`<bit value>`: 0 or 1 `<Optional ResponseWaited>=0`: no response waited (default) `<Optional ResponseWaited>=1`: response waited `<Optional NbTxFlag>=0`: one Tx frame sent `<Optional NbTxFlag>=1`: three Tx frame sent (default)
Response	None
Result code	`<CR><LF>OK<CR><LF>`

Examples:

- `AT$SB=1` sends bit 1 with no response waited.
- `AT$SB=0,1` sends bit 0 with a response waited.
- `AT$SB=0,1,1` sends bit 0 with a response waited and with three Tx frames sent.

12.2.12 AT$SF - ASCII payload in bytes

Description	Sends a frame to the Sigfox network.
Syntax	`AT$SF=<payload data>{,<Optional ResponseWaited>}` `{,<Optional NbTxFlag> }<CR>` to send payload
Arguments	`<payload data>`: 12 bytes maximum in ASCII format (24 ASCII characters max) `<Optional ResponseWaited>=0`: no response waited (default) `<Optional ResponseWaited>=1`: response waited `<Optional NbTxFlag>=0`: one Tx frame sent `<Optional NbTxFlag>=1`: three Tx frames sent (default)
Response	None
Result code	`<CR><LF>OK<CR><LF>`

Examples:

- `AT$SF=313245` sends 0x31 0x32 0x45 payload with no response waited.
- `AT$SF=010205,1` sends 0x01 0x02 0x05 payload with a response waited.

12.2.13 AT$SH - Hexadecimal payload in bytes

Description	Sends a frame to the Sigfox network.
Syntax	`AT$SH=<payload length><payload data>{,<Optional ResponseWaited>}{,<Optional NbTxFlag> }<CR> to send payload`
Arguments	`<payload length>`: length in bytes `<payload data>`: 12 bytes maximum in hexadecimal format `<Optional ResponseWaited>=0`: no response waited (default) `<Optional ResponseWaited>=1`: response waited `<Optional NbTxFlag>=0`: one Tx frame sent `<Optional NbTxFlag>=1`: three Tx frames sent (default)
Response	None
Result code	`<CR><LF>OK<CR><LF>`

Examples:

- `AT$SH=1,A` sends 0x41 payload with no response waited.
- `AT$SH=1,A,1` sends 0x41 payload with a response waited.

12.2.14 AT$CW - Continuous wave (CW)

Description	Starts/stops a continuous unmodulated carrier for test.
Syntax	`AT$CW=<frequency><CR>`
Arguments	`<frequency>`: frequency (in Hz or MHz) When `<frequency>=0`, the test is stopped.
Response	None
Result code	`<CR><LF>OK<CR><LF>`

The `AT$CW=<input><CR>` command sends a continuous unmodulated carrier.

Note:
- *Default power is 14 dBm in RC1 and can be modified with ATS302 - Radio output power.*
- *This command is mandatory for certification of the device for CE.*
- *Power is stored in EEPROM for the region selected.*

Examples:

- `AT$CW=868` starts a CW at 868 MHz.
- `AT$CW=902000000` starts a CW at 902 MHz.
- `AT$CW=0` stops a CW.

12.2.15 AT$PN - PRBS9 BPBSK test mode

Description	Sends a continuous modulated carrier for test.
Syntax	`AT$PN=<input>,<bitrate><CR>`
Arguments	`<frequency>`: frequency (in Hz or MHz) When `<frequency>=0`, the test is stopped. `<bitrate>=100 or 600` when input within center frequency
Response	None
Result code	`<CR><LF>OK<CR><LF>`

Note:
- *Default power is 14 dBm in RC1 and can be modified with ATS302 - Radio output power.*
- *This command is mandatory for certification of the device for CE.*
- *Power is stored in EEPROM for the region selected.*

Examples:
- `AT$PN=868,100` starts a BPSK modulated continuous carrier at 868 MHz with data rate 100 CW at 868 MHz.
- `AT$PN=902000000,600` starts a BPSK modulated continuous carrier at 902 MHz with data rate 600 CW at 868 MHz
- `AT$PN=0` stops a CW.

12.2.16 AT$MN - Monarch scan

Description	Runs a Monarch scan.
Syntax	`AT$MN={<Optional time>}<CR><CR>`
Arguments	`<Optional time>`: scan duration in seconds (default = 5 s)

Response	`No RC found`
	`RC1 found`
	`RC2 found`
	`RC3c found`
	`RC4 found`
	`RC5 found`
	`RC6 found`
	`RC7 found`

Result code	`<CR><LF>OK<CR><LF>`

Examples:
- `AT$MN` runs a Monarch scan for 5 s.
- `AT$MN=10` runs a Monarch scan for 10 s.

12.2.17 AT$TM - Sigfox test mode

The modem must implement this command. This test mode can be used in front of the Sigfox RSA (radio signal analyzer) and the SDR dongle (more details in Sigfox RSA user guide on https://resources.sigfox.com).

This command is for test-mode purposes only and cannot be used to connect to the Sigfox network.

Sigfox RSA tester must be configured as follows (RSA version 2.0.1):

1. Open *Device Configuration*.
2. Set *Radio Configuration*.
3. Set *Payload Encryption Configuration* to *Payload Encryption Capable*.
4. Set *Oscillator Aging* to1.
5. Set *Oscillator Temperature Accuracy* to 1.
6. Apply *Settings*.
7. *Open* (to start the tester).

Description	Starts a Sigfox test mode.
Syntax	`AT$TM=<rc>,<mode><CR>`
Argument `<rc>`	rc = 1, 2, 3c, 4, 5, 6 or 7 for the RC at which the test must run.
Argument `<mode>`	• `SFX_TEST_MODE_TX_BPSK=0` Sends only BPSK 26-byte packets including synchro bit and PRBS synchro frame at the `Tx_frequency` uplink frequency defined in Table 3. The uplink frequency is RC dependent. RSA test: press start after selecting *UL-RF Analysis* then launch the `AT$TM=x,0` command.
Argument `<mode>` (cont'd)	• `SFX_TEST_MODE_TX_PROTOCOL=1` Full protocol with internal Sigfox key that sends all Sigfox protocol frames with all possible length available with hopping (sends bit with downlink flag set and unset, sends out-of-band frame, sends frame with downlink flag set and unset with all possible payload length 1 to 12 bytes. `config`: number of times the test is done RSA test: – Press start after select *UL-Protocol* then launch the `AT$TM=x,1` command. – Press start after select *UL-Protocol w/Encrypted Payload*, then set `ATS411=1` prior launching the `AT$TM=x,1` command. Do not forget to reset `ATS411=0` before next tests. • `Mode =SFX_TEST_MODE_RX_PROTOCOL=2` Full protocol with internal Sigfox key that sends all Sigfox protocol frames with all possible lengths available with hopping (sends bits with downlink flag set and unset, sends out-of-band frames, sends frames with downlink flag set and unset with all possible payload lengths from 1 to 12 bytes). **Caution:** *This test lasts several minutes.* RSA test – Press start after select *DL-Protocol* then launch the `AT$TM=x,2` command. – Press start after select RSA test *w/Encrypted Payload*, then set `ATS411=1` prior launching the `AT$TM=x,2` command. Do not forget to reset `ATS411=0` before next tests. – Press start after select *Start of listening window* then launch the `AT$TM=x,2` command. – Press start after select *End of listening window then* launch the `AT$TM=x,2` command. • `SFX_TEST_MODE_RX_GFSK=3` Rx mode in GFSK with expected pattern = AA AA B2 27 1F 20 41 84 32 68 C5 BA 53 AE 79 E7 F6 DD 9B sent at the Rx_frequency downlink frequency defined in Table 3. The downlink frequency is RC dependent. The test lasts 30 seconds. RSA test: Press *start send GSK* after selecting *DL-GFSK Receiver* then launch the `AT$TM=x,3` command. This test is only informative, not mandatory. • `SFX_TEST_MODE_RX_SENSI=4` This test is used to measure the real sensitivity of device and requests one uplink and one downlink frame with the Sigfox key, with specific timings. RSA test: Press *start* after selecting *DL-Link Budget* then launch the `AT$TM=x,4` command • `SFX_TEST_MODE_TX_SYNTH =5` Does one uplink frame on each Sigfox channel frequency. This test takes a couple of minutes. RSA test: Press *start* after selecting *UL-Frequency Synthesis* then launch the `AT$TM=x,5` command. • `SFX_TEST_MODE_TX_FREQ_DISTRIBUTION=6` This test consists in calling `SIGFOX_API_send_xxx` functions to test the complete protocol in uplink mode only, with uplink data from 0x40 to 0x4B. RSA test: Press *start* after selecting *UL-Frequency-Distribution* then launch the `AT$TM=x,6` command. **Caution:** *This test lasts several minutes.*

	• `SFX_TEST_MODE_RX_MONARCH_PATTERN_LISTENING_SWEEP=7` This test consists in setting the device in pattern scan for 30 s in `LISTENING_SWEEP` mode and report status `TRUE` or `FALSE` depending on the pattern found against the expected pattern. RSA test: not available on RSA. • `SFX_TEST_MODE_RX_MONARCH_PATTERN_LISTENING_WINDOW=8` This test consists in setting the device in pattern scan for 30 s in `LISTENING_WINDOW` mode and report status `TRUE` or `FALSE` depending on the pattern found against the expected pattern. RSA test: not available on RSA. • `SFX_TEST_MODE_RX_MONARCH_BEACON=9` RSA test: not available on RSA SDR dongle. Press *start* after selecting *Monarch Link Budget* then launch the `AT$TM=x,10` command.
Argument `<mode>` (cont'd)	• `SFX_TEST_MODE_RX_MONARCH_SENSI=10` RSA test: not available on RSA SDR dongle. – Press *start* after selecting *Monarch signal at high power* then launch the `AT$TM=x,10` command. Press *start* after selecting *High Power Level interferer for Monarch* then launch the `AT$TM=x,10` command. Press *start Robustness* to *Low Power Level interferer for Monarch* then launch the `AT$TM=x,10` command. • `SFX_TEST_MODE_TX_BIT=11` This test consists in calling `SIGFOX_API_send_bit` function twice to test part of the protocol in uplink only and LBT. • `SFX_TEST_MODE_PUBLIC_KEY=12` Sends Sigfox frame with public key activated. The uplink frequency is RC dependent. RSA test: Press *start* after select *UL-Public Key,* then launch the `AT$TM=x,12` command. • `SFX_TEST_MODE_PUBLIC_KEY=13` This test consists in calling functions once with the PN of the NVM data and verifies NVM storage. RSA test: Press *start* after select *UL-Non-Volatile Memory,* then launch the `AT$TM=x,13` command, then remove supply and resend the `AT$TM=x,13` command.
Response	None
Result code	`<CR><LF>OK<CR><LF>`

12.2.18 AT+BAT? - Battery level

Description	Gets the battery level (in mV).
Syntax	`AT+BAT?<CR>`
Arguments	None
Response	Returns the battery level (in mV).
Result code	`<CR><LF>OK<CR><LF>`

12.2.19 ATS300 - Out-of-band message

Description	Sends one keep-alive out-of-band message.
Syntax	`ATS300<CR>`
Arguments	None
Response	None
Result code	`<CR><LF>OK<CR><LF>`

Note: *Out-of-band messages have Sigfox network well known format. They can be sent every 24 hours.*

12.2.20 ATS302 - Radio output power

Description	Sets/gets the radio output power.

Syntax	`ATS302=<power><CR>`
	`ATS302=?<CR>`

Arguments	`<power>` in dBm
Response	None
Result code	`<CR><LF>OK<CR><LF>`

Note:
- *Default power is 13 dBm for RC1.*
- *This command is mandatory for certification of the device for CE.*
- *Power is saved in EEPROM for the region selected with `AT$RC` (one power per region).*
- *Firmware does not prevent the user to enter higher power than the recommended ones.*

12.2.21 ATS400 - Enabled channels for FCC

Description	Configure the enabled channels for FCC

Syntax	`ATS400=<8_digit_word0><8_digit_word1><8_digit_word2>,`
	`<timer_enable><CR>`
	0 to disable and 1 to enable

Arguments	`<8_digit_word0>`
	`<8_digit_word1>`
	`<8_digit_word2>`
	`<timer_enable>`

Response	None
Result code	`<CR><LF>OK<CR><LF>`

Note: Default value = `<000003FF><00000000><00000000>,1`

Example

```
ATS400=<000001FF><00000000><00000000>,1
```

The timer between consecutive Tx frames is enabled and the following macro channels are enabled: 902.8 MHz, 903.1 MHz, 903.4 MHz, 903.7 MHz, 904.0 MHz, 904.3 MHz, 904.6 MHz, 904.9 MHz and 905.2 MHz.

*Note: At least nine macro channels must be enabled to ensure the minimum of 50 FCC channels (9 * 6 = 54). The configured `default_sigfox_channel` must be at least enabled in configuration word (see Section 7.1.3: Set standard configuration).*

12.2.22 AT$RC - Region configuration

Description	Sets/gets the region configuration (RC).

Syntax	`AT$RC=<rc><CR>`
	`AT$RC=?<CR>`

Arguments	`<rc>`, `RC1`, `RC2`, `RC3c`, `RC4`, `RC5`, `RC6`, `RC7`
Response	`RC1`, `RC2`, `RC3c`, `RC4`, `RC5`, `RC6`, `RC7`
Result code	`<CR><LF>OK<CR><LF>`

The `AT$RC=<zone><CR>` command can be used to set the current zone (response `OK<CR>`)

12.2.23 ATE - Echo mode

Not used except to set echo mode.

12.2.24 AT+VL - Verbose level

Description	Sets/gets the verbose level.

Syntax	`AT$VL=<verbose level><CR>`
	`AT$VL=?<CR>`

Arguments	`<verbose level>`: 0, 1, 2 or 3
Response	0, 1, 2 or 3
Result code	`<CR><LF>OK<CR><LF>`

The verbose level is stored in the EEPROM.

12.3 PushButton application

The PushButton application is a standalone example. On a user push-button event, this application reads the temperature and battery voltage (mV) and sends then in a message to the Sigfox network.

This application targets the NUCLEO_WL55JC1 development board with STM32WL55JC (order codes NUCLEO-WL55JC1 for high-frequency band) and B-WL5M-SUB1 connectivity expansion board with STM32WL5M.

In order to launch the Sigfox PushButton project, the user must go to `Projects\<target>\Applications\Sigfox\Sigfox_PushButton` and choose a toolchain folder.

Note: The device is always in Stop 2 mode unless the user button 1 is pressed.

The payload content can be changed by updating the file `\Projects\<target>\Applications\Sigfox\Sigfox_PushButton\Sigfox\App\sgfx_app.c` in function SendSigfox. If more sensors' data are required, the file `\Projects\<target>\Applications\Sigfox\Sigfox_PushButton\Core\Src\sys_sensors.c` can be updated.

12.4 Static switches

Static defines are used to switch optional features (such as debug, or trace), to disable low power, or to tune some RF parameters.

12.4.1 Debug switch

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

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

The debug mode enables the SWD debugger pins, even when the MCU goes in low-power.

Note: In order to enable a true low-power, `#define DEBUGGER_ENABLED` must be defined to 0.

12.4.2 Low-power switch

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

```
#define LOW_POWER_DISABLE 0
```

with:

- 0: Low-power Stop 2 mode enabled. MCU enters Stop 2 mode.
- 1: Low-power Stop 2 mode disabled. MCU enters Sleep mode only.

12.4.3 Trace level

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

```
#define APP_LOG_ENABLED 1
```

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

```
#define VERBOSE_LEVEL VLEVEL_M
```

The following trace levels are proposed:

- VLEVEL_OFF (all traces disabled)
- VLEVEL_L (functional traces enabled)
- VLEVEL_M (debug traces enabled)
- VLEVEL_H (all traces enabled)

12.4.4 Probe pins

Four probe pins can be enabled in `\Projects\<target>\Applications\Sigfox\Sigfox_PushButton\Core\Inc\sys_conf.h` with the code below:

```
#define PROBE_PINS_ENABLED 0
```

To enable the probe pins, `PROBE_PINS_ENABLED` must be defined to 1.

12.4.5 Radio configuration

Radio configurations can be updated in `\Projects\<target>\Applications\Sigfox\Sigfox_PushButton\Sigfox\Target\radio_conf.h`.
For Sigfox applications, `RF_WAKEUP_TIME` must be defined to 15 ms to pass the drift-rate specification.

13 Dual-core management

The STM32WL5x devices embed two Cortex:

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

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

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

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

The interface between cores is done by the IPCC peripheral (interprocessor communication controller) and the inter-core memory, as described in Section 13.1.

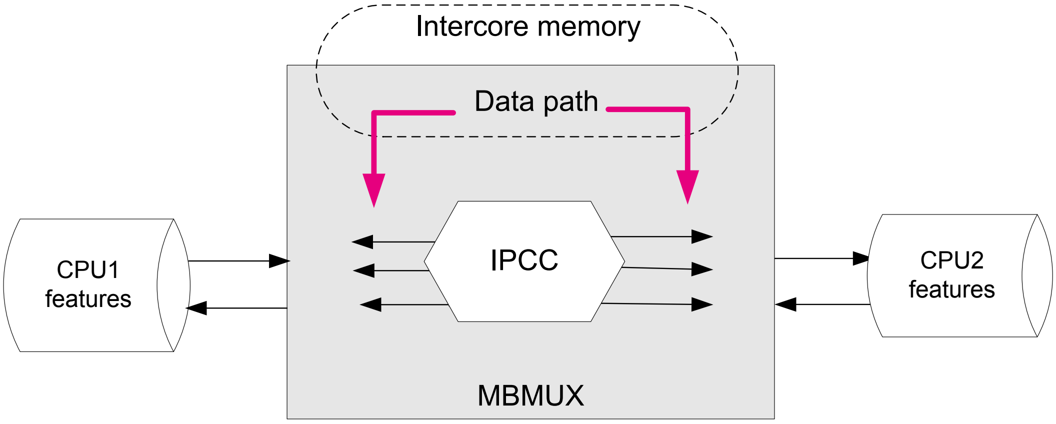
This dual-core implementation has been designed to behave the same way as the single-core program execution, thanks to a message blocking handling through a mailbox mechanism.

13.1 Mailbox mechanism

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

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

Figure 12. Mailbox overview



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

13.1.1 Mailbox multiplexer (MBMUX)

As described in Figure 13, the data to be exchanged need to communicate via the 12 available IPCC channels (six for each direction). This is done via the MBMUX (mailbox multiplexer) that is a firmware component in charge to route the messages. These channels are identified from 1 to 6. An additional channel 0 is dedicated to the system feature.

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

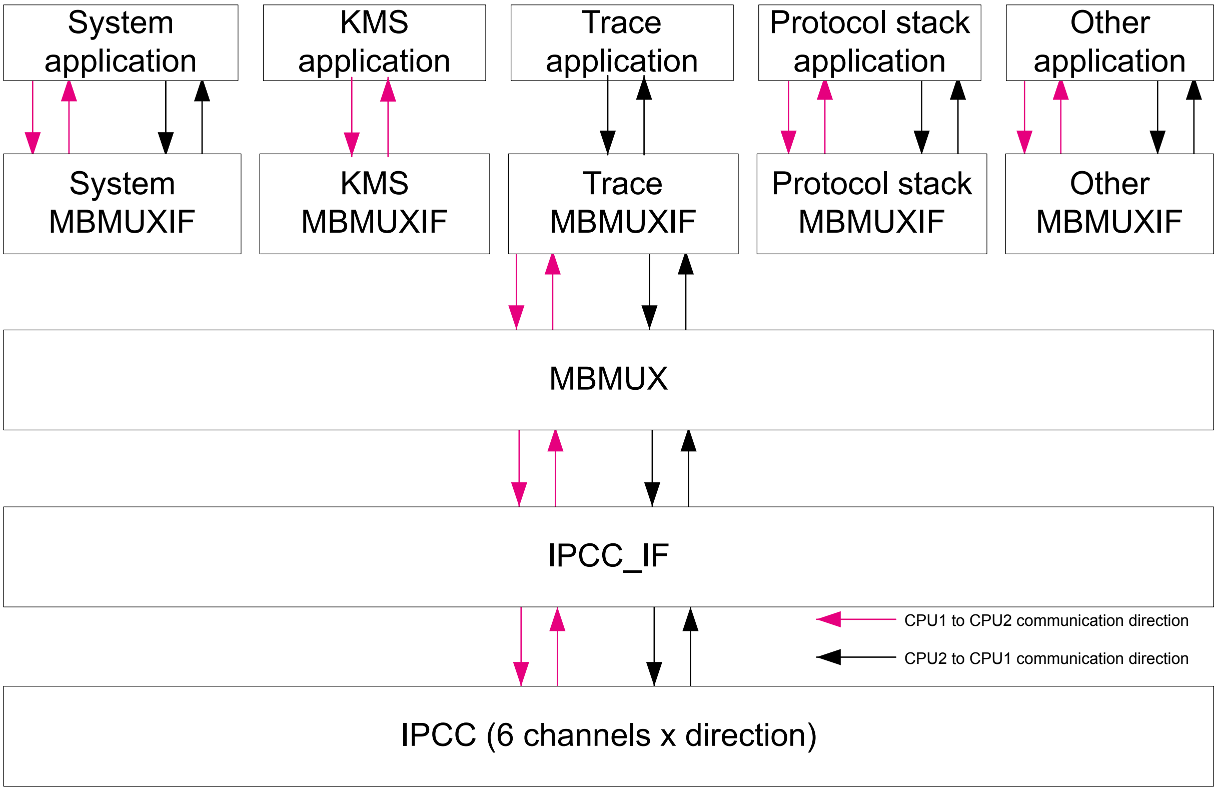
The mailbox is used to abstract a function executed by another core.

13.1.2 Mailbox features

In STM32WL5x devices, the MBMUX has the following features:

- **System**, supporting all communications related to the system
  This includes messages that are either related to one of the supported stacks, or related to none of them. The CPU1 channel 0 is used to notify the CPU2 that a command has been posted, and to receive the response of that command from the CPU2. The CPU2 channel 0 is used to notify CPU1 that an asynchronous event has been posted.
  The following services are mapped on system channel:
  - system initialization
  - IPCC channels versus feature registration
  - information exchanged on feature attributes and capabilities
  - possible additional system channels for high-priority operations (such as RTC notifications)
- **Trace**
  The CPU2 fills a circular queue for information or debug that is sent to the CPU1 via the IPCC. The CPU1 is in charge to handle this information, by outputting it on the same channel used for CPU1 logs (such as the USART).
- **KMS** (key management services)
- **Radio**
  The sub-GHz radio can be interfaced directly without passing by the CPU2 stack. A dedicated mailbox channel is used.
- **Protocol stack**
  This channel is used to interface all protocol stack commands (such as Init or request), and events (response/indication) related to the stack implemented protocol.

Figure 13. **MBMUX - Multiplexer between features and IPCC channels**



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

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

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

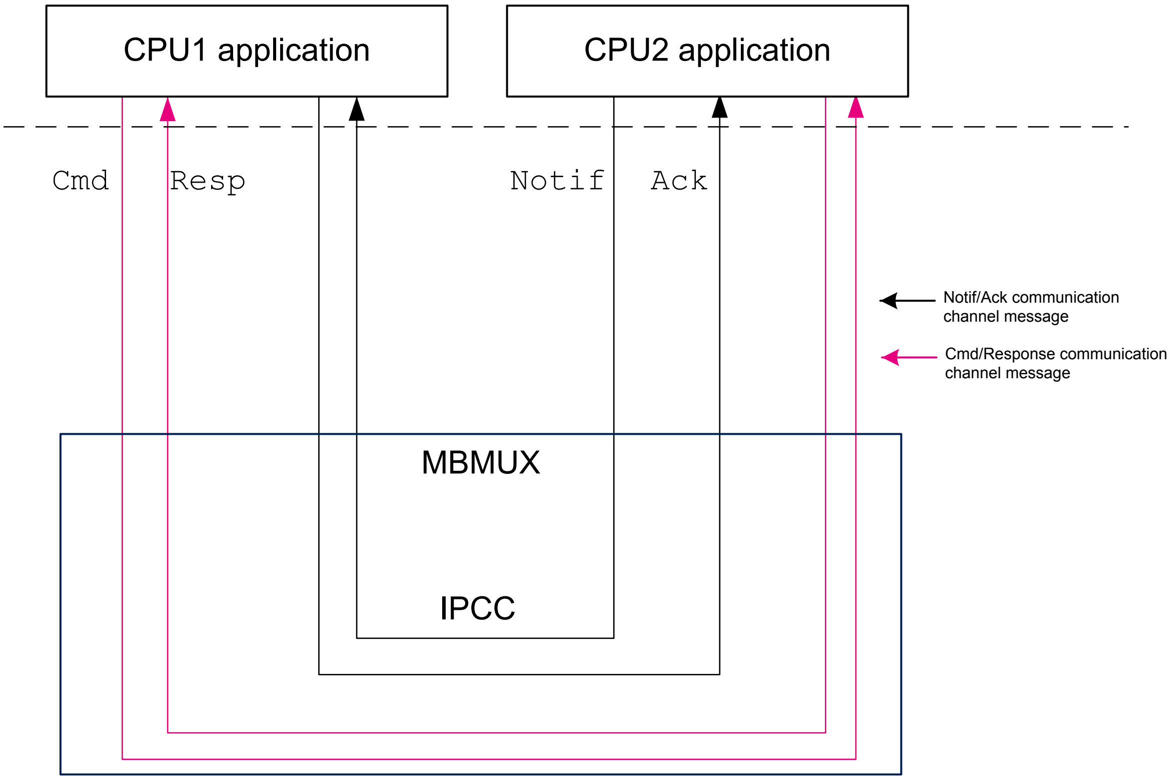
Note:
- *The RTC alarm A transfers the interrupt using one IPCC IRQ, not considered as a feature.*
- *The user must consider adding KMS wrapper to be able to use it as a feature.*

13.1.3 MBMUX messages

The mailbox uses the following types of messages:

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

Figure 14. **Mailbox messages through MBMUX and IPCC channels**


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

13.2 Intercore memory

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

13.2.1 CPU2 capabilities

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

These CPU2 capabilities are stored in the *features_info* table. Data from this table are requested at initialization by the CPU1 to expose CPU2 capabilities, as shown in Section 13.2.5.

The *features_info* table is composed of:

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

MB_MEM2 is used to store these CPU2 capabilities.

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

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

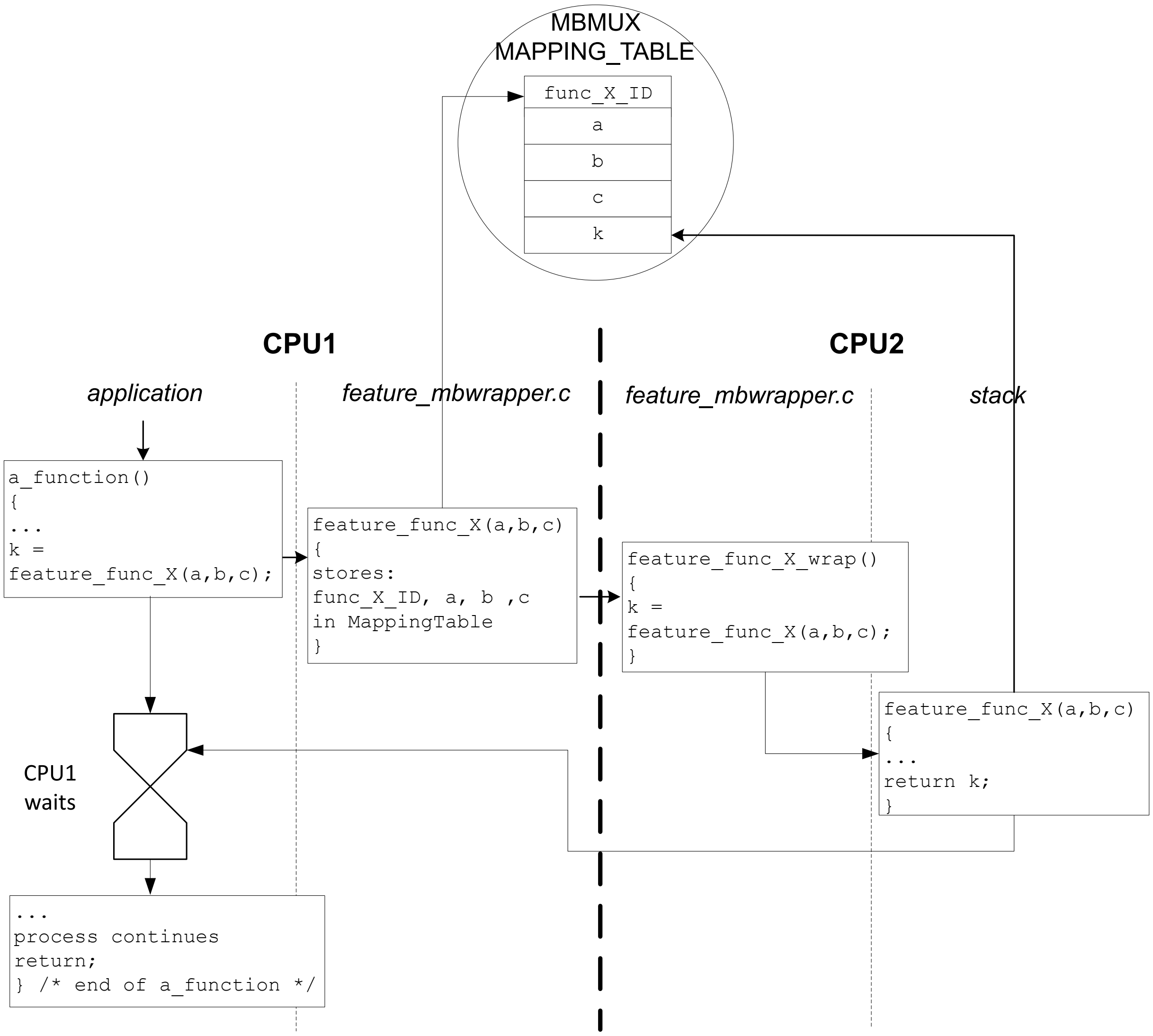
1. The CPU1 sends a **command** containing `feature_func_X()` parameters in the *Mapping* table:
   a. `func_X_ID` that was associated to `feature_func_X()` at initialization during registration, is added in the *Mapping* table. `func_X_ID` has to be known by both cores: this is fixed at compilation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

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

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

The full sequence is shown in the figure below.

Figure 15. **CPU1 to CPU2 feature_func_X() process**



```
a_function()
{
...
k =
feature_func_X(a,b,c);
```

```
feature_func_X(a,b,c)
{
stores:
func_X_ID, a, b ,c
in MappingTable
}
```

```
feature_func_X_wrap()
{
k =
feature_func_X(a,b,c);
}
```

```
feature_func_X(a,b,c)
{
...
return k;
}
```

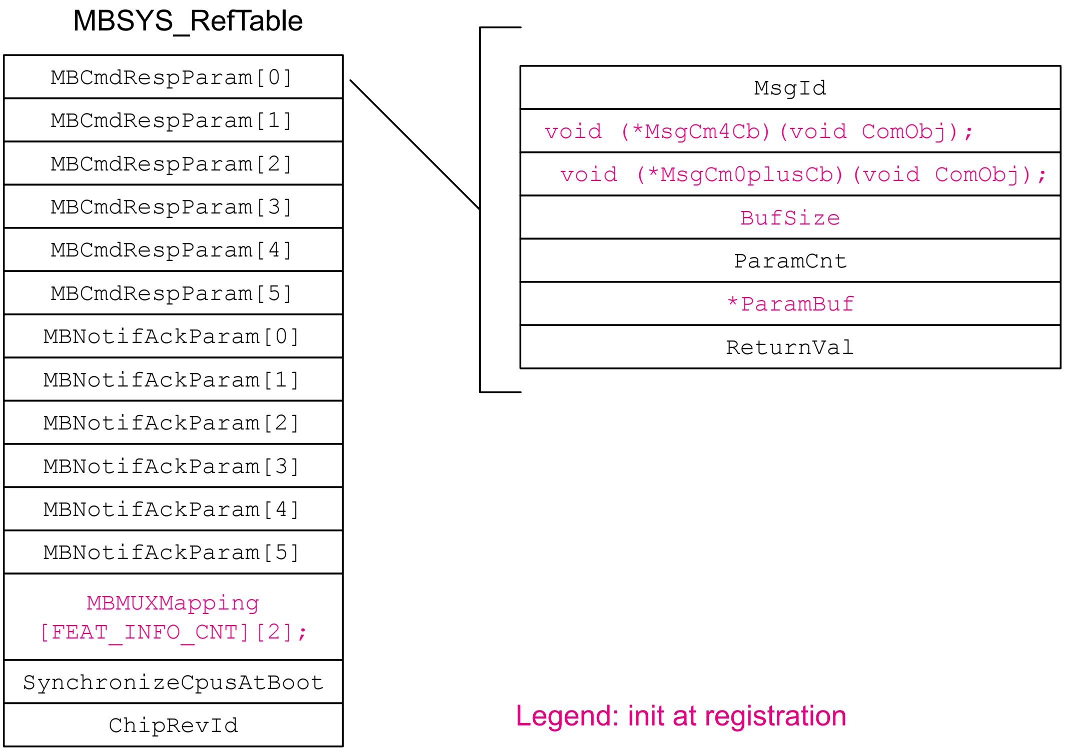
```
...
process continues
return;
} /* end of a_function */
```

13.2.3 Mapping table

The *Mapping* table is a common structure in the MBMUX area of Figure 15. In Section 13.2.5, the memory mapping is referenced as MAPPING_TABLE.

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

Figure 16. **MBMUX communication table**



This MBSYS_RefTable includes:

- two communication parameter structures for both Command/Response and Notification/Acknowledge parameters for each of the six IPCC channels.
  Each communication parameter, as shown in MBMUX *Mapping* table area of Figure 15, is composed of:
  – `MsgId`: message ID of `feature_func_X()`
  – `*MsgCm4Cb`: pointer to CPU1 callback `feature_func_X()`
  – `*MsgCm0plusCb`: pointer to CPU2 callback `feature_func_X()`
  – `BufSize`: buffer size
  – `ParamCnt`: message parameter number
  – `ParamBuf`: message pointer to parameters
  – `ReturnVal`: return value of `feature_func_X()`
- `MBMUXMapping`: chart used to map channels to features
  This chart is filled at the MBMUX initialization during the registration. For instance, if the radio feature is associated to `Cmd/Response channel number = 1`, then `MBMUXMapping` must associate `[FEAT_INFO_RADIO_ID][1]`.
- `SynchronizeCpusAtBoot`: flags used to synchronize CPU1 and CPU2 processing as shown in Figure 17 sequence chart.
- `ChipRevId`: stores the hardware revision ID.

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

13.2.4 Option-byte warning

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

13.2.5 RAM mapping

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

Table 28. **STM32WL5x RAM mapping**

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

1. *2 Kbytes for each page.*

Table 29. **STM32WL5x RAM allocation and shared buffer**

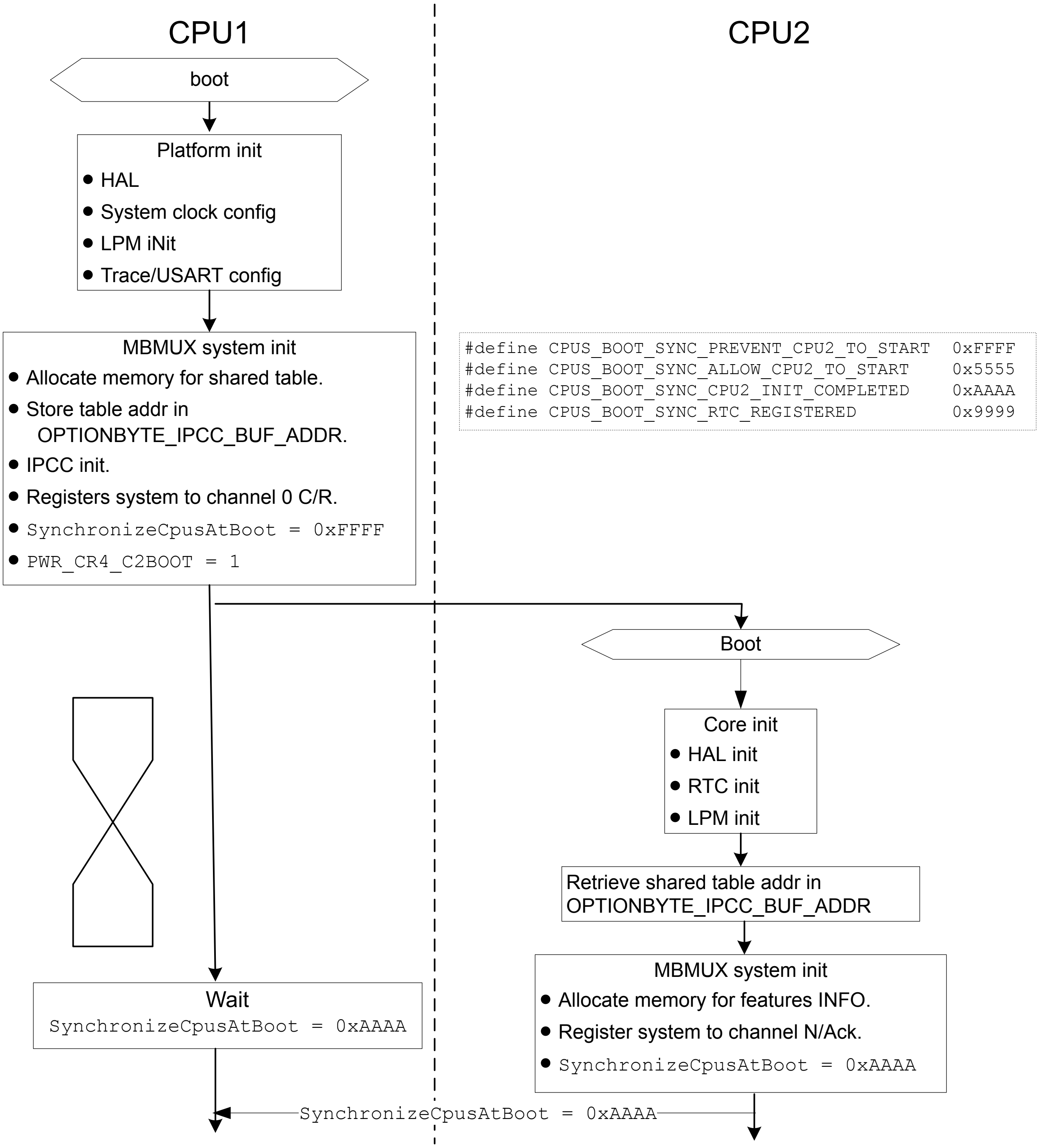
CPU region	Section	Module	Allocated symbol	Size (bytes)	Total (bytes)
CPU1 RAM	readwrite	-			
	CSTACK				
	HEAP				
CPU1 RAM2_Shared	MAPPING_TABLE	MBMUX_SYSTEM	MBMUX_ComTable_t MBSYS_RefTable	316	316
	MB_MEM1	MBMUX_SIGFOX	uint32_t aSigfoxCmdRespBuff[]	60	292
			uint32_t aSigfoxNotifAckBuff[]	20	
		MBMUX_RADIO	uint32_t aRadioCmdRespBuff[]	60	
			uint32_t aRadioNotifAckBuff[]	16	
		MBMUX_TRACE	uint32_t aTraceNotifAckBuff[]	44	
		MBMUX_SYSTEM	uint32_t aSystemCmdRespBuff[]	28	
			uint32_t aSystemNotifAckBuff[]	20	
			uint32_t aSystemPrioACmdRespBuff[]	4	
			uint32_t aSystemPrioANotifAckBuff[]	4	
			uint32_t aSystemPrioBCmdRespBuff[]	4	
			uint32_t aSystemPrioBNotifAckBuff[]	4	
		SGFX_MBWRAPPER	uint8_t aSigfoxMbWrapShareBuffer[]	28	
CPU2 RAM2_Shared	MB_MEM2	MBMUX_TRACE	uint8_t ADV_TRACE_Buffer[]	512	860
		MBMUX_SIGFOX	SigfoxInfo_t SigfoxInfoTable	4	
			FEAT_INFO_Param_t Feat_Info_Table	80	
			FEAT_INFO_List_t Feat_Info_List	8	

CPU region	Section	Module	Allocated symbol	Size (bytes)	Total (bytes)
CPU2 RAM2_Shared	`MB_MEM2`	`SGFX_MBWRAPPER`	`uint8_t aSigfoxMbWrapShare2Buffer[]`	0	860
		`RADIO_MBWRAPPER`	`uint8_t aRadioMbWrapRxBuffer[]`	256	
CPU2 RAM2	`readwrite`	-			
	`CSTACK`				
	`HEAP`				

13.3 Startup sequence

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

Figure 17. Startup sequence


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

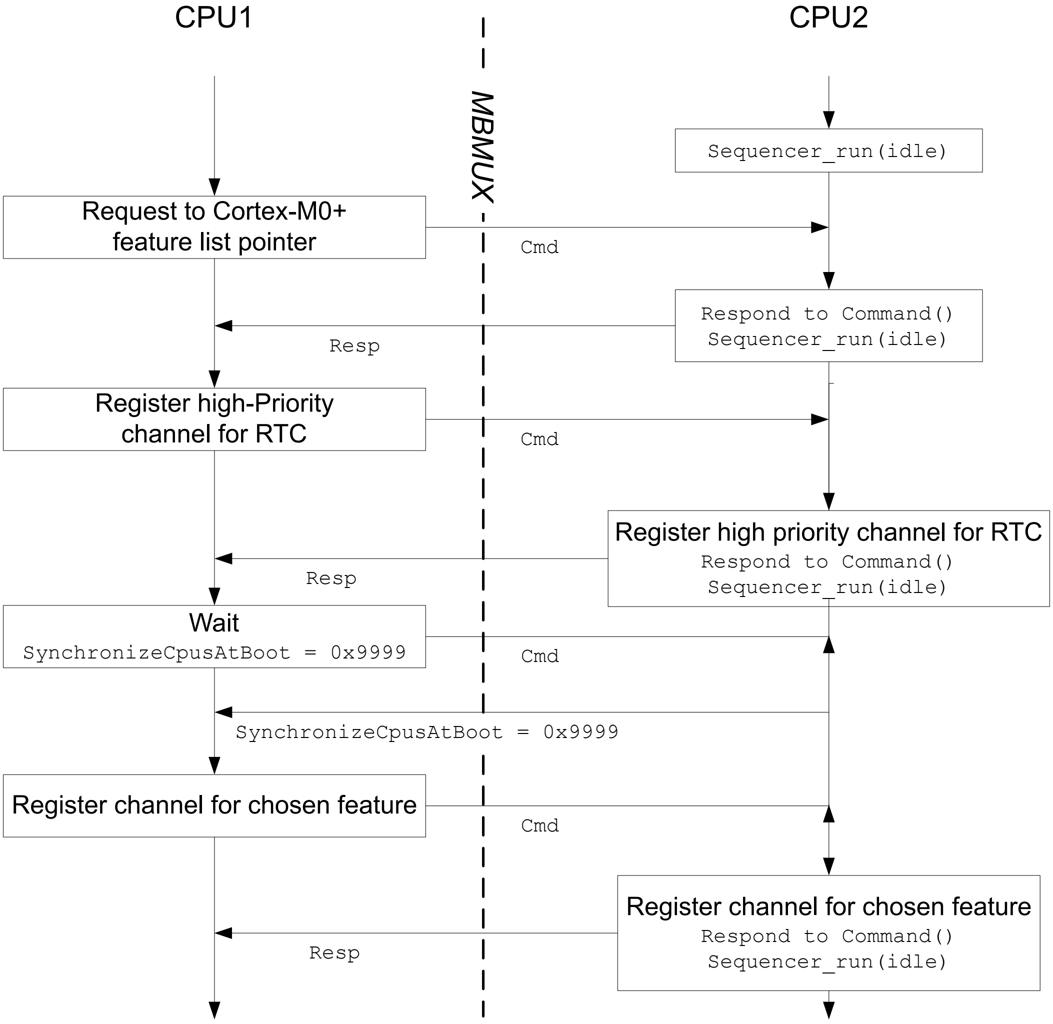
```
#define CPUS_BOOT_SYNC_PREVENT_CPU2_TO_START  0xFFFF
#define CPUS_BOOT_SYNC_ALLOW_CPU2_TO_START    0x5555
#define CPUS_BOOT_SYNC_CPU2_INIT_COMPLETED    0xAAAA
#define CPUS_BOOT_SYNC_RTC_REGISTERED         0x9999
```

The various steps are the following:

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

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

Figure 18. **MBMUX initialization**

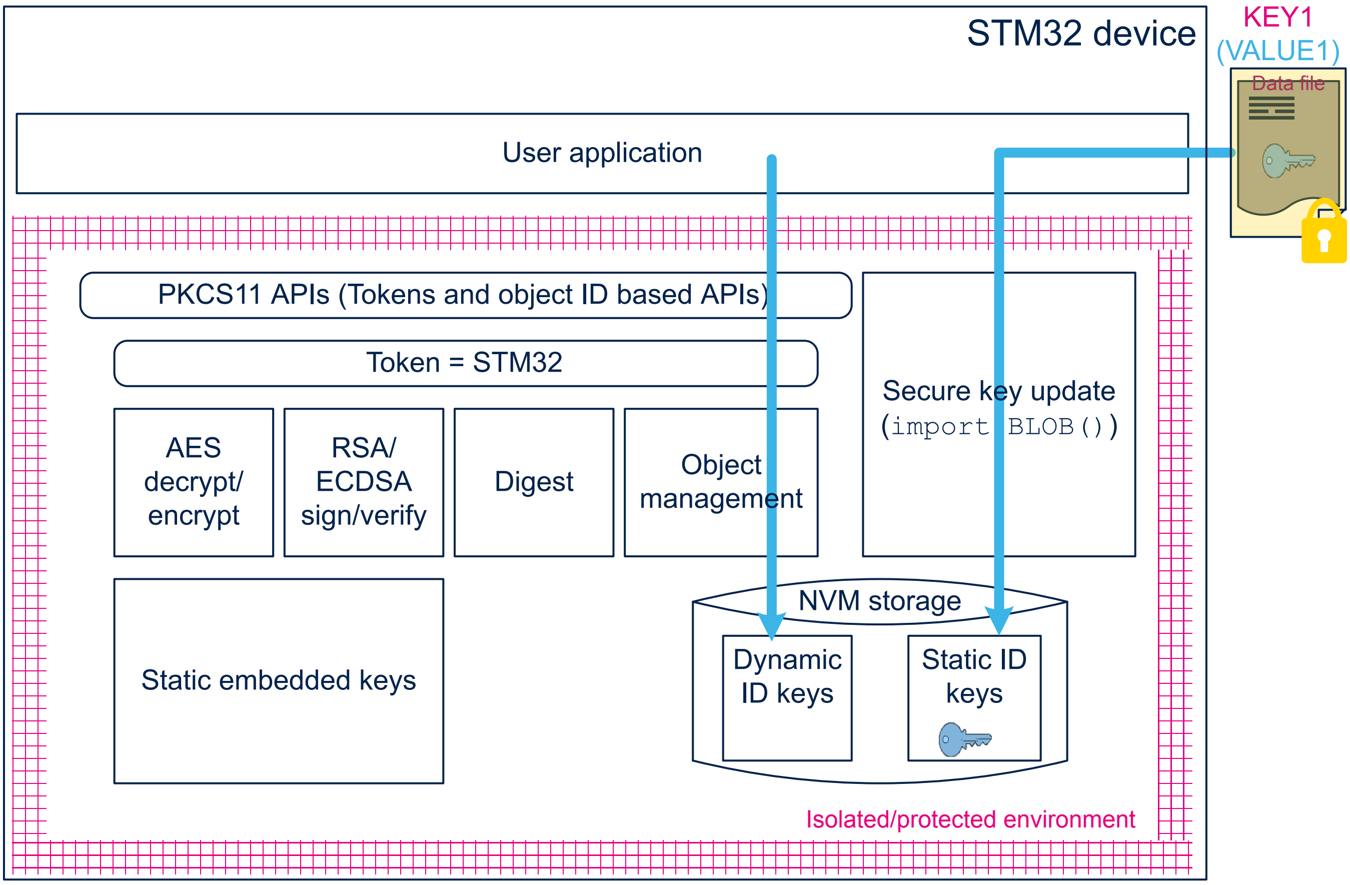

CPU1
CPU2
MBMUX
Sequencer_run(idle)
Request to Cortex-M0+ feature list pointer
Cmd
Respond to Command()
Sequencer_run(idle)
Resp
Register high-Priority channel for RTC
Cmd
Register high priority channel for RTC
Respond to Command()
Sequencer_run(idle)
Resp
Wait
SynchronizeCpusAtBoot = 0x9999
Cmd
SynchronizeCpusAtBoot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

14 Key management services (KMS)

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

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

Figure 19. **KMS overall architecture**



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

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

KMS supports only PKCS #11 APIs listed below:

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

14.1 KMS key types

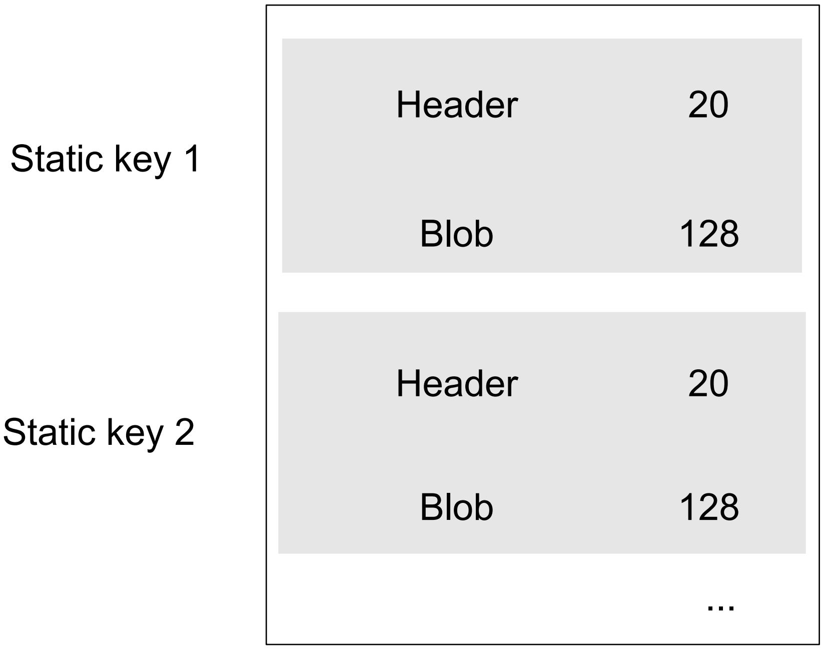
KMS manages three key types but only the two following ones are used:

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

14.2 KMS keys size

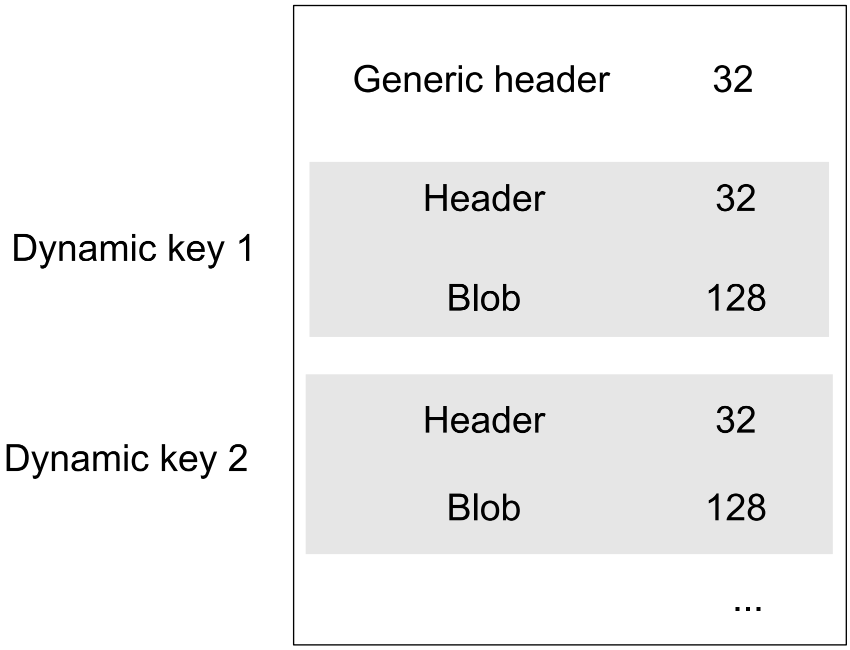
Static and dynamic keys used by Sigfox stack occupies different sizes. As described in the figure below, each static key size is 148 bytes = header(20) + blob(128).

Figure 20. KMS static key size



As described in the figure below, at the top of KMS key storage, there is a KMS generic header (32 bytes), then each dynamic keys size is 160 bytes = header(32) + blob(128).

Figure 21. **KMS dynamic key size**



14.3 Sigfox keys

In the STM32CubeWL application list, the KMS are used on Cortex-CM0+ only, on dual-core application. The root keys are chosen to be static embedded keys. All derived keys are NVM_DYNAMIC keys.

For Sigfox stack, there is one static root key: `Sigfox_Key`.

`Sigfox_pac` and `Sigfox_id` are stored in the KMS but cannot be used as crypto keys.

There is one volatile NVM_DYNAMIC generated key: `Sigfox_Public_Key`.

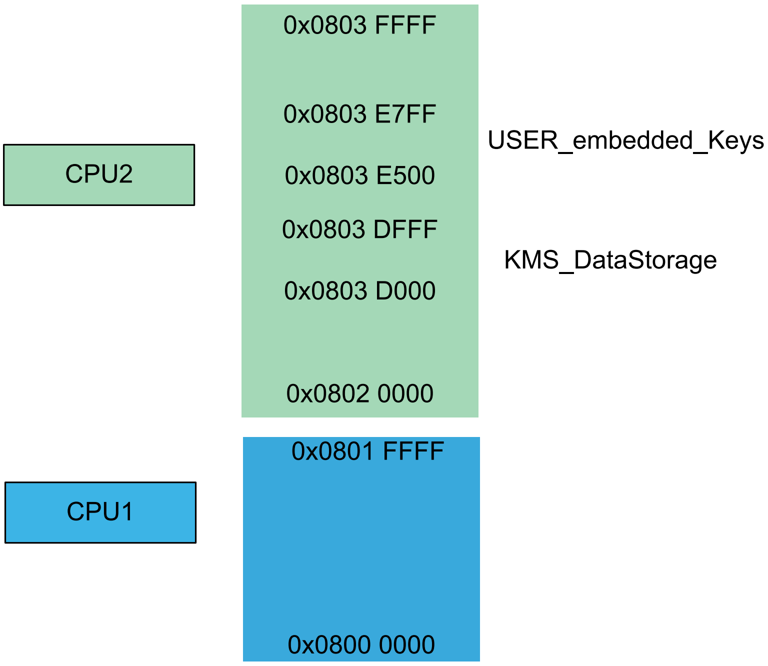
14.4 KMS key memory mapping for user applications

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

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

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

Figure 22. ROM memory mapping



14.5 How to size the NVM for KMS data storage

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

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

User applications use the NVM only to store the KMS keys. A Sigfox key and the related chosen KMS attributes occupy 128 bytes. As described in Figure 21, the KMS header takes 32 bytes for each key and a global header common to all keys takes 32 bytes. Given the above values, it is possible to calculate how many keys can be stored in 2 Kbytes:

(2048 - 32) / (32 + 128) = 12,6 ==> 12 KMS keys (KMS key meaning key value, key attributes, and header).

User applications are configured such that only NVM_DYNAMIC is used. NVM_STATIC can be filled via blob, but not covered by user applications.

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

Sigfox applications use NVM_DYNAMIC only for derived keys. Sigfox_PushButton generates one derived key each time a data is sent (uplink) when payload encryption is set.

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

Destroy a key does not mean that a key is erased but that is tagged as destroyed. This key is not copied at the next flip-flop switch. A destroy flag also occupies some NVM bytes.

The estimation of the life expectation given below corresponds to the case of payload encryption set (one key is generated at each uplink and previous key is destroyed):

- Up to 12 encrypted keys can be generated before a flip-flop transfer is necessary. At the 13th uplink, the derived key is stored at page 2, and page 1 is erased.
- After 24 encrypted uplinks, the key is stored back on page 1 and page 2 is erased.
- After a 240 000 uplinks, the two NVM pages have been erased 10 000 times, which is the estimated lifetime of the FLASH sector.
- Since the maximum amount of Sigfox uplinks is 144 messages per day, the expected lifetime is about 4.5 years. Lifetime can be doubled by doubling the NVM size.

Note:
- *This calculation is not valid when payload encryption is disabled.*
- *Obsolete keys must be destroyed otherwise, if page 1 is fully filled by active keys, the flip-flop switch cannot be done and an error is generated.*

14.6 KMS configuration files to build the application

The KMS are used in the Sigfox example by setting

`SIGFOX_KMS = 1` in `CM0PLUS/Sigfox/App/app_sigfox.h`.

The following files must filled with the SubGhz stack keys information:

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

14.7 Embedded keys

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

These embedded keys are positioned in the ROM as indicated in Figure 22.

15 Personalization and activation

When compiling and loading the firmware using the default `sigfox_data.h`, default Sigfox credentials are loaded in the device. This allows to test the Sigfox device locally in the lab in front of the RSA.

The following steps are needed for the Sigfox device to send data to the Sigfox backend server:

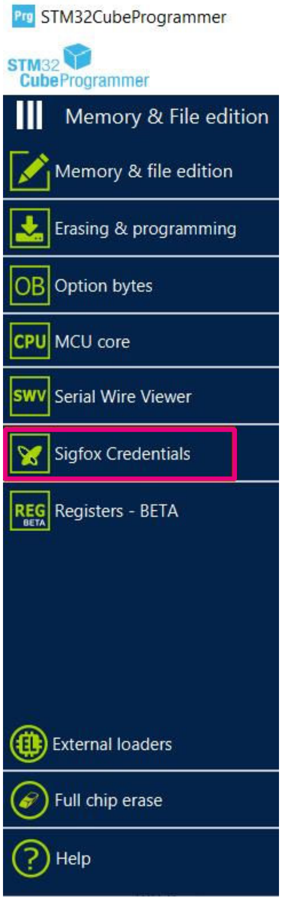
1. Personalization: Every Sigfox device must be loaded with the ID, PAC and private key credentials, that are necessary to activate the device and send data to the Sigfox data server.
2. Activation: Once the device is personalized, it needs to be recorded by the Sigfox backend server. This step requires to log-on the Sigfox backend server.

Note: Steps below require STM32CubeProgrammer version 2.6.0 minimum.

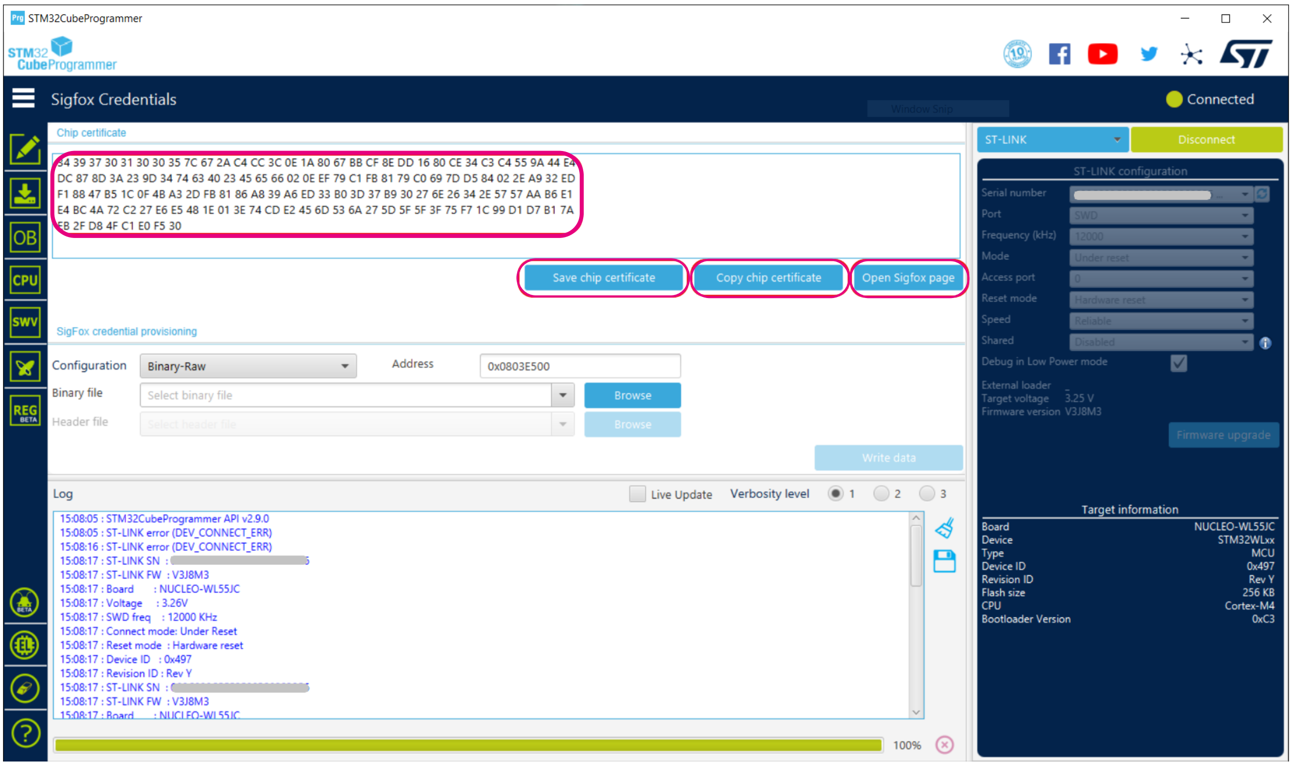
15.1 Personalization

As soon as the user connects the STM32WL5x/Ex devices, a button *Sigfox Credentials* is added on the main menu.

Figure 23. STM32CubeProgrammer Sigfox panel button



After opening the *Sigfox Credentials* window, the chip certificate is extracted automatically with 136-byte size and displayed in *chip certificate* area. This certificate can be saved in a binary file and copied to clipboard to be used in the ST web interface to get Sigfox credentials (see Section 15.1.1). The user have access to the ST web interface using *Open Sigfox page* button integrated in the window.



Chip certificate

```
34 39 37 30 31 30 30 35 7C 67 2A C4 CC 3C 0E 1A 80 67 BB CF 8E DD 16 80 CE 34 C3 C4 55 9A 44 E4
DC 87 8D 3A 23 9D 34 74 63 40 23 45 65 66 02 0E EF 79 C1 FB 81 79 C0 69 7D D5 84 02 2E A9 32 ED
F1 88 47 B5 1C 0F 4B A3 2D FB 81 86 A8 39 A6 ED 33 B0 3D 37 B9 30 27 6E 26 34 2E 57 57 AA B6 E1
E4 BC 4A 72 C2 27 E6 E5 48 1E 01 3E 74 CD E2 45 6D 53 6A 27 5D 5F 5F 3F 75 F7 1C 99 D1 D7 B1 7A
EB 2F D8 4F C1 E0 F5 30
```

SigFox credential provisioning

Configuration: Binary-Raw

Address: 0x0803E500

Binary file: Select binary file

Header file: Select header file

Log □ Live Update Verbosity level ● 1 ○ 2 ○ 3

```
15:08:05 : STM32CubeProgrammer API v2.9.0
15:08:05 : ST-LINK error (DEV_CONNECT_ERR)
15:08:16 : ST-LINK error (DEV_CONNECT_ERR)
15:08:17 : ST-LINK SN  : [redacted]
15:08:17 : ST-LINK FW  : V3J8M3
15:08:17 : Board       : NUCLEO-WL55JC
15:08:17 : Voltage     : 3.26V
15:08:17 : SWD freq    : 12000 KHz
15:08:17 : Connect mode: Under Reset
15:08:17 : Reset mode  : Hardware reset
15:08:17 : Device ID   : 0x497
15:08:17 : Revision ID : Rev Y
15:08:17 : ST-LINK SN  : [redacted]
15:08:17 : ST-LINK FW  : V3J8M3
15:08:17 : Board       : NUCLEO-WL55JC
```

ST-LINK configuration

Serial number	[redacted]
Port	SWD
Frequency (kHz)	12000
Mode	Under reset
Access port	0
Reset mode	Hardware reset
Speed	Reliable
Shared	Disabled
Debug in Low Power mode	☑
External loader	_
Target voltage	3.25 V
Firmware version	V3J8M3

Target information

Board	NUCLEO-WL55JC
Device	STM32WLxx
Type	MCU
Device ID	0x497
Revision ID	Rev Y
Flash size	256 KB
CPU	Cortex-M4
Bootloader Version	0xC3

Figure 24. **STM32CubeProgrammer Sigfox panel - Getting certificate**

Note: *When Sigfox project is generated with STM32CubeMx, the Sigfox data may not be placed at 0x803 E500.*

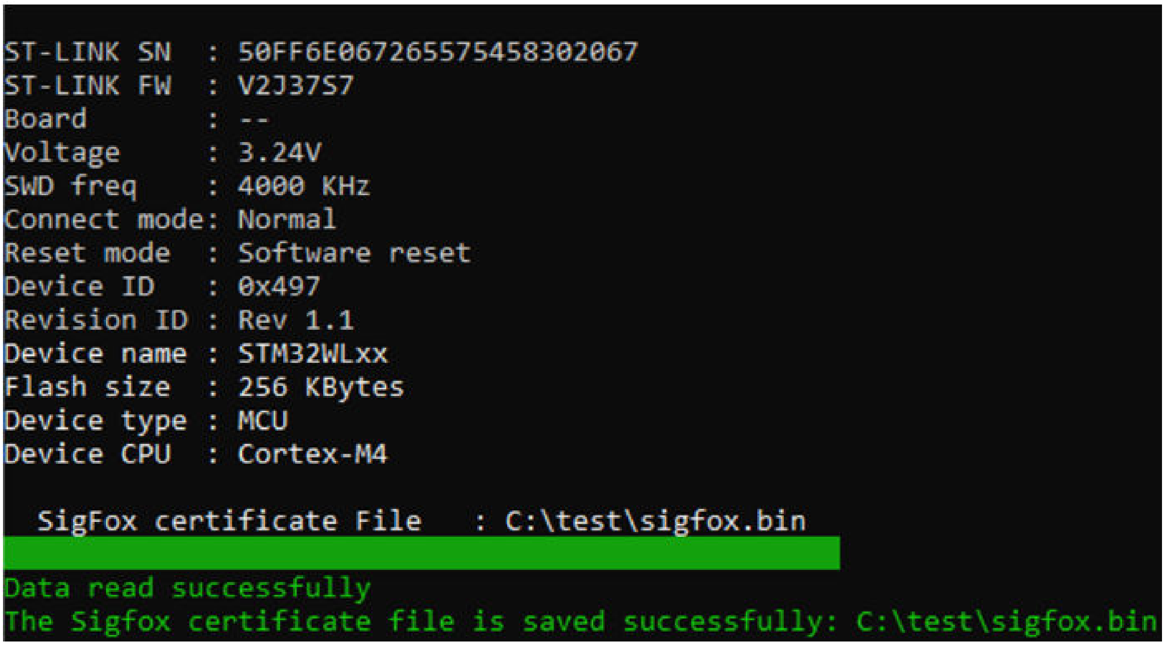
When the device memory is lower than 256-Kbyte flash, the Sigfox data cannot be placed at 0x803 E500.

As a result of these two cases, the user must either place the Sigfox data at another location, or find this location in the mapping file and update the address accordingly.

The command line used to save the chip certificate in a binary file:

- Command: `-ssigfoxc`
- Description: This command allows the user to save the chip certificate to a binary file.
- Syntax: `-ssigfoxc <binary_file_path>`
- Example: `STM32_Programmer_CLI.exe -c port=swd -ssigfoxc /local/user/chip_certif.bin`

Figure 25. STM32CubeProgrammer Sigfox CLI - Getting certificate



```
ST-LINK SN   : 50FF6E067265575458302067
ST-LINK FW   : V2J37S7
Board        : --
Voltage      : 3.24V
SWD freq     : 4000 KHz
Connect mode: Normal
Reset mode   : Software reset
Device ID    : 0x497
Revision ID  : Rev 1.1
Device name  : STM32WLxx
Flash size   : 256 KBytes
Device type  : MCU
Device CPU   : Cortex-M4

  SigFox certificate File    : C:\test\sigfox.bin

Data read successfully
The Sigfox certificate file is saved successfully: C:\test\sigfox.bin
```

15.1.1 Getting the credentials

ST provides a web interface on my.st.com, where the user can get the sigfox trial credentials.

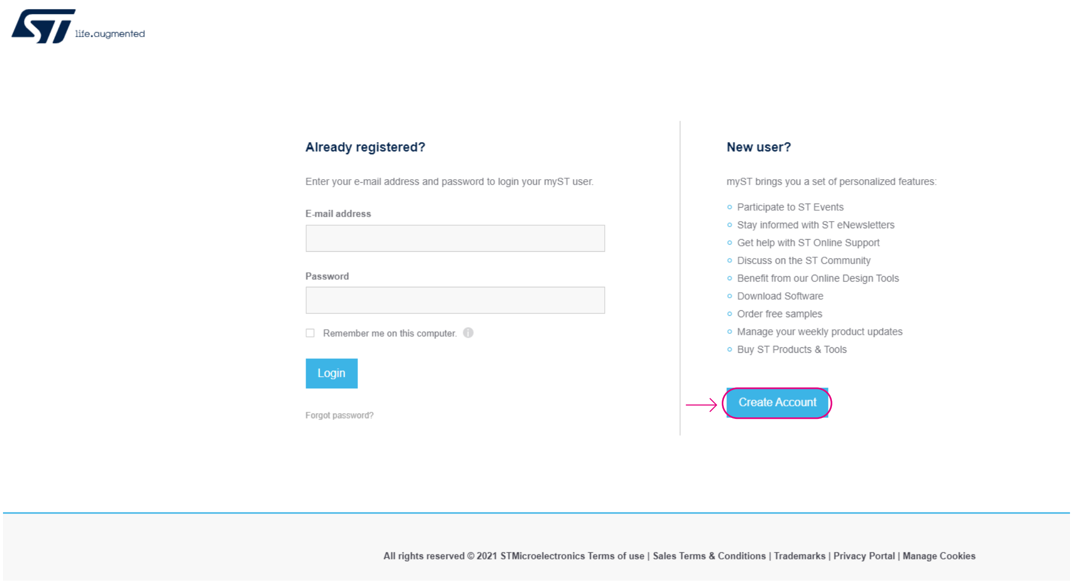
The credentials are delivered as a zip file containing the following fi

- `sigfox_data_XXXXXXXX.h` defining the credentials that can be integrated into the application source code
- `sigfox_data_ XXXXXXXX.bin` to flash the credentials onto the chip, thanks to STM32CubeProgrammer

Follow the steps below to get the credentials:

1. Go to https://my.st.com/sfxp and register on my.st.com to create a specific user account (if not existing yet).

Figure 26. **Login on my.st.com**



life.augmented

Already registered?

Enter your e-mail address and password to login your myST user.

E-mail address

Password

□ Remember me on this computer

Login

Forgot password?

New user?

myST brings you a set of personalized features:

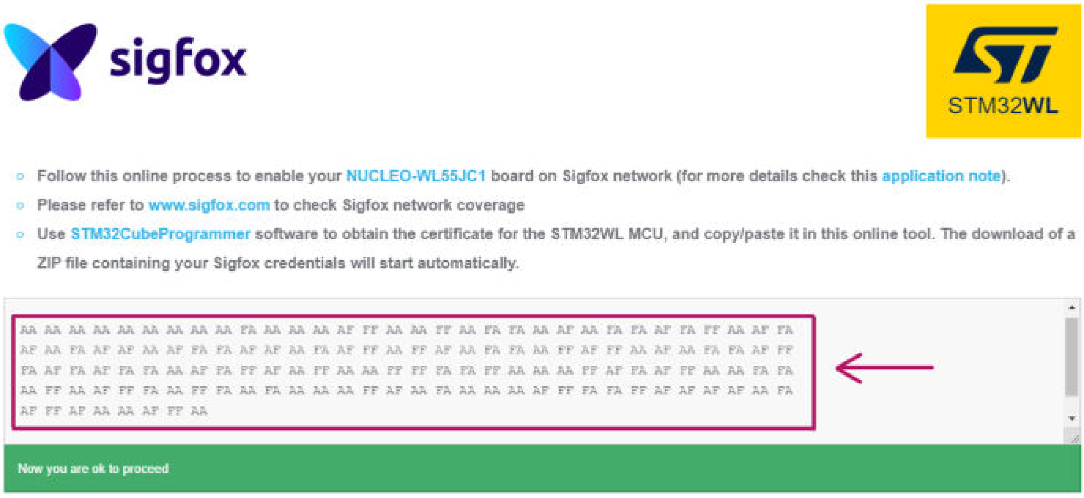
- Participate to ST Events
- Stay informed with ST eNewsletters
- Get help with ST Online Support
- Discuss on the ST Community
- Benefit from our Online Design Tools
- Download Software
- Order free samples
- Manage your weekly product updates
- Buy ST Products & Tools

Create Account

All rights reserved © 2021 STMicroelectronics Terms of use | Sales Terms & Conditions | Trademarks | Privacy Portal | Manage Cookies

2. Paste the certificate extracted with STM32CubeProgrammer into the form.

Figure 27. **Sigfox credential page**



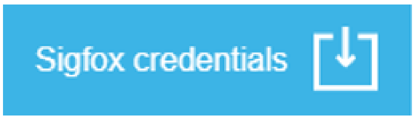
- Follow this online process to enable your NUCLEO-WL55JC1 board on Sigfox network (for more details check this application note).
- Please refer to www.sigfox.com to check Sigfox network coverage
- Use STM32CubeProgrammer software to obtain the certificate for the STM32WL MCU, and copy/paste it in this online tool. The download of a ZIP file containing your Sigfox credentials will start automatically.

```
AA AA AA AA AA AA AA AA AA FA AA AA AA AF FF AA AA FF AA FA FA AA AF AA FA FA AF FA FF AA AF FA
AF AA FA AF AF AA AF FA FA AF AF AA FA AF FF AA FF AF AA FA FA AA FF AF FF AA AF AA FA FA AF FF
FA AF FA AF FA FA AA AF FA FF AF AA FF AA AA FF FF FA FA FF AA AA AA FF AF FA AF FF AA AA FA FA
AA FF AA AF FF FA AA FF FA AA FA AA AA AA FF AF AA FA AA AA AA AF FF FA FA FF AF AF AF AF AA FA
AF FF AF AA AA AF FF AA
```

Now you are ok to proceed

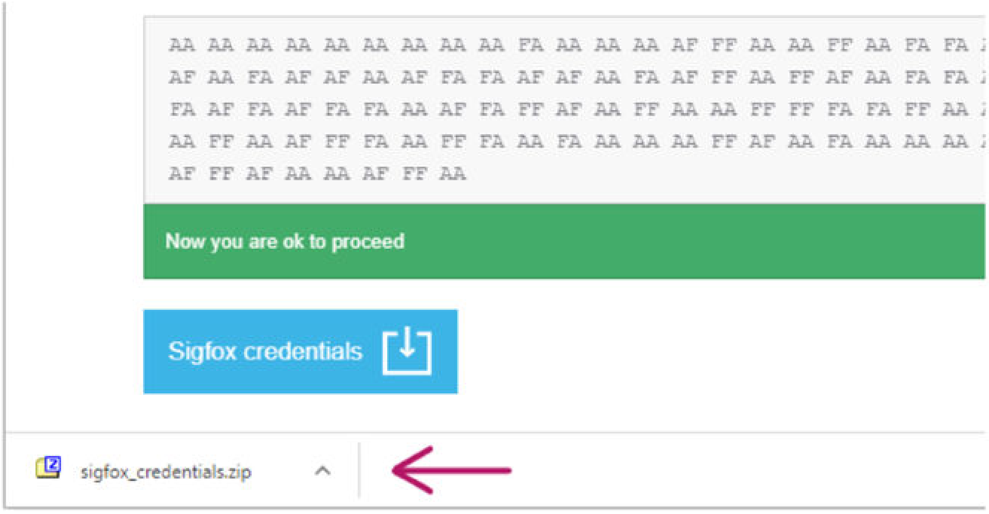
3. Click on the *download* button.

Figure 28. **Download button**



4. A zip file is automatically downloaded on the user computer.

Figure 29. **Sigfox_credetentials download**



```
AA AA AA AA AA AA AA AA AA FA AA AA AA AF FF AA AA FF AA FA FA
AF AA FA AF AF AA AF FA FA AF AF AA FA AF FF AA FF AF AA FA FA
FA AF FA AF FA FA AA AF FA FF AF AA FF AA AA FF FF FA FA FF AA
AA FF AA AF FF FA AA FF FA AA FA AA AA AA FF AF AA FA AA AA AA
AF FF AF AA AA AF FF AA
```

Now you are ok to proceed

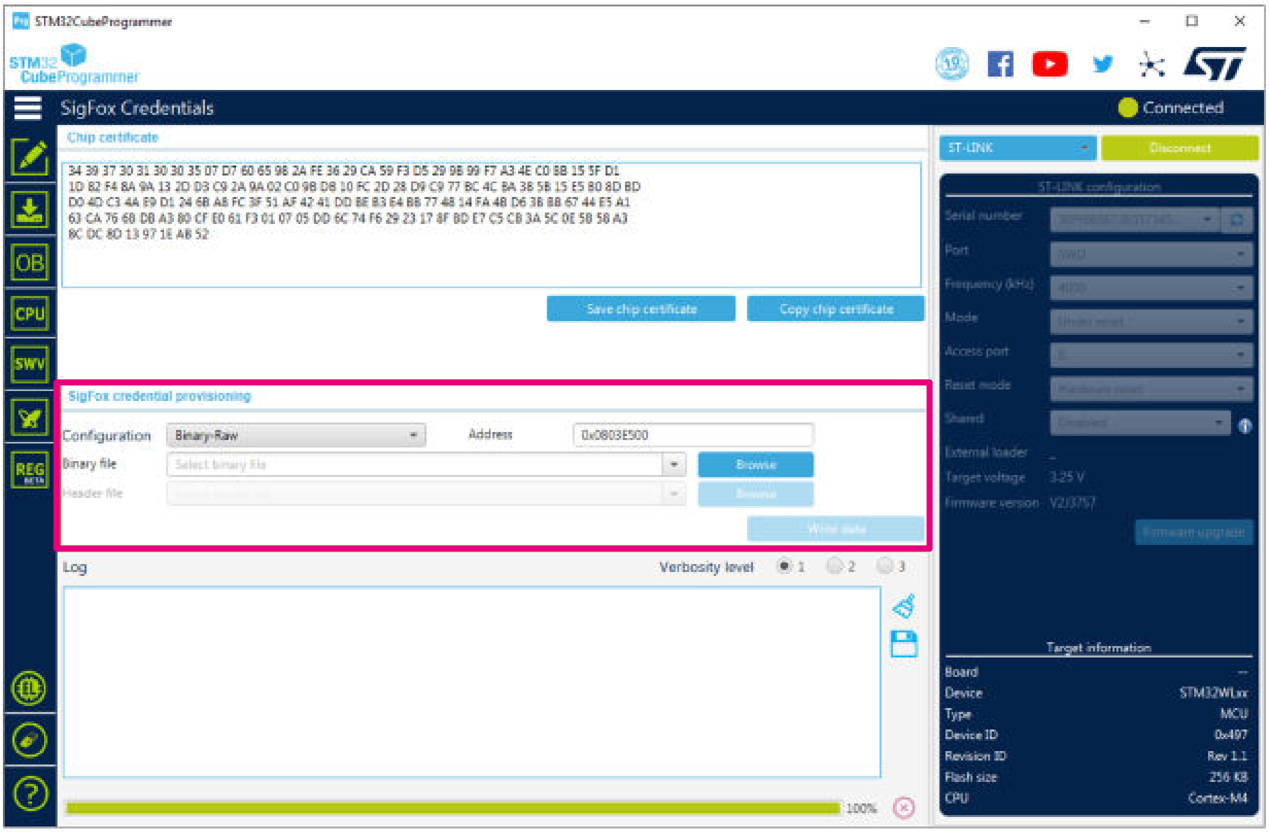
15.1.2 Loading the credentials in the device

As soon as the user gets the Sigfox credential from the ST web interface, the user can load them in the STM32WL5x/Ex devices, at 0x0803 E500, using the *Sigfox credential provisioning* area in STM32CubeProgrammer.

- Case 1: *Binary-Raw*:
  This option can only be used in single-core projects or in dual-core projects where KMS is not enabled. The binary file returned by the ST web interface must be used. This file must be 48-byte size and is written at the default address 0x0803 E500.
- Case 2: *Binary KMS*:
  This option may only be used in dual core projects where KMS is enabled. The header file returned by the ST web interface must be used. It is written at the default address 0x0803 E500.

Note: The address 0x0803 E500 is placed by the linker file (see `.ld` file for STM32CubeIde, `.icf` for IAR Embedded Workbench, or `.sct` for MDK_ARM).

Figure 30. STM32CubeProgrammer Sigfox panel - Flashing credentials



Chip certificate

```
34 39 37 30 31 30 30 35 07 D7 60 65 98 2A FE 36 29 CA 59 F3 D5 29 9B 99 F7 A3 4E C0 BB 15 5F D1
1D B2 F4 8A 9A 13 2D D3 C9 2A 9A 02 C0 9B D8 10 FC 2D 28 D9 C9 77 BC 4C BA 38 5B 15 E5 80 8D BD
D0 4D C3 4A E9 D1 24 6B AB FC 3F 51 AF 42 41 DD BE B3 E4 BB 77 48 14 FA 4B D6 3B BB 67 44 E5 A1
63 CA 76 6B DB A3 80 CF E0 61 F3 01 07 05 DD 6C 74 F6 29 23 17 8F BD E7 C5 CB 3A 5C 0E 5B 58 A3
8C DC 8D 13 97 1E AB 52
```

SigFox credential provisioning

Configuration: Binary-Raw
Address: 0x0803E500
Binary file: Select binary file
Header file

Log

Verbosity level ● 1 ○ 2 ○ 3

100%

ST-LINK configuration

Serial number
Port
Frequency (kHz)
Mode
Access port
Reset mode
Shared
External loader: -
Target voltage: 3.25 V
Firmware version: V2J37S7

Target information

Board	--
Device	STM32WLxx
Type	MCU
Device ID	0x497
Revision ID	Rev 1.1
Flash size	256 KB
CPU	Cortex-M4

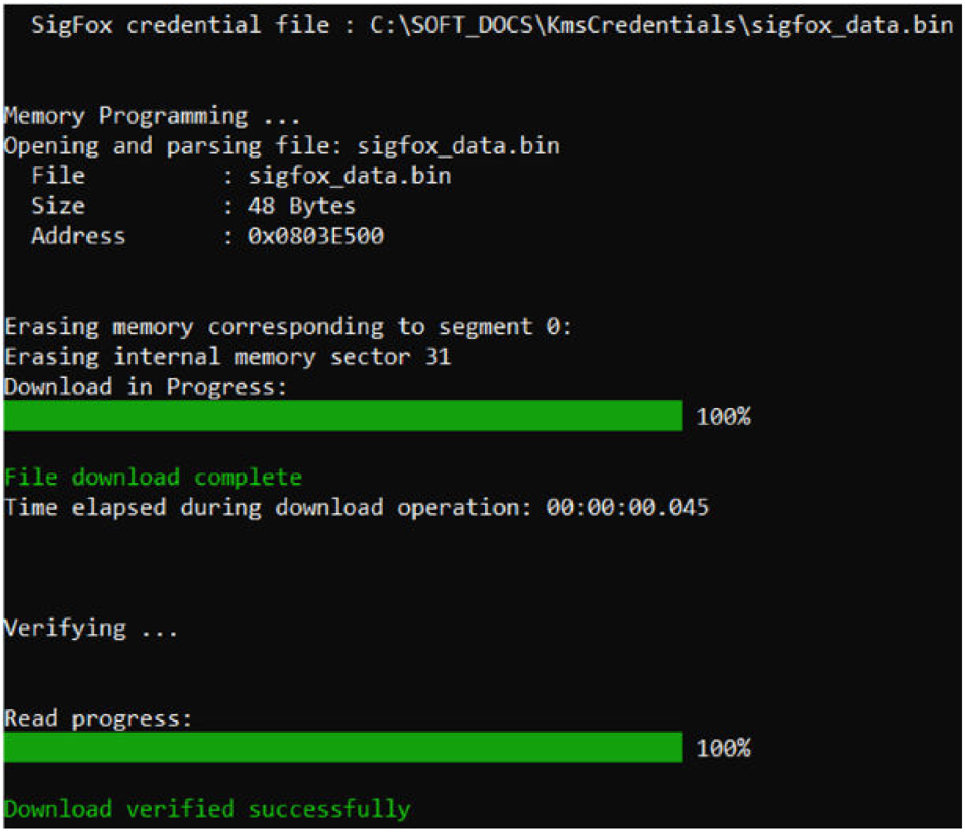
The command line used to write the credentials in the device is defined as follows:

- Command: `-wsigfoxc`
- Description: This command allows the user to write the sigfox credentials at the default address 0x0803 E500.
- Syntax: `-wsigfoxc <sigfox_credential_file_path> <address>`
  - `<address>` is optional (by default 0x0803 E500).
  - `<sigfox_credential_file_path>` can be a binary file (see example 1) or an header file (see example 2 below).

Example 1

```
STM32_Programmer_CLI.exe -c port=swd -wsigfoxc "/local/user/sigfox_data.bin"
0x0803E500
```

Figure 31. STM32CubeProgrammer Sigfox CLI - Flashing raw credentials

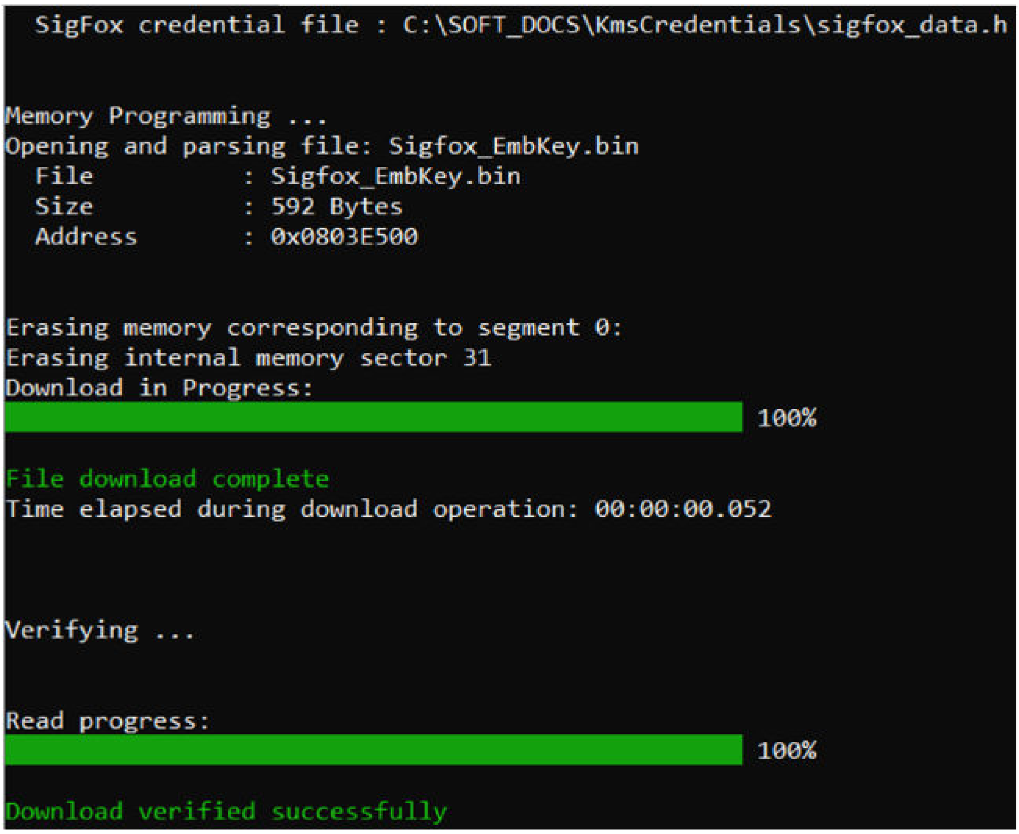


```
 SigFox credential file : C:\SOFT_DOCS\KmsCredentials\sigfox_data.bin


Memory Programming ...
Opening and parsing file: sigfox_data.bin
  File          : sigfox_data.bin
  Size          : 48 Bytes
  Address       : 0x0803E500


Erasing memory corresponding to segment 0:
Erasing internal memory sector 31
Download in Progress:
                                                  100%

File download complete
Time elapsed during download operation: 00:00:00.045



Verifying ...


Read progress:
                                                  100%

Download verified successfully
```

Example 2

```
STM32_Programmer_CLI.exe -c port=swd -wsigfoxc "/local/user/sigfox_data.h"
```

Figure 32. STM32CubeProgrammer Sigfox CLI - Flashing KMS credentials



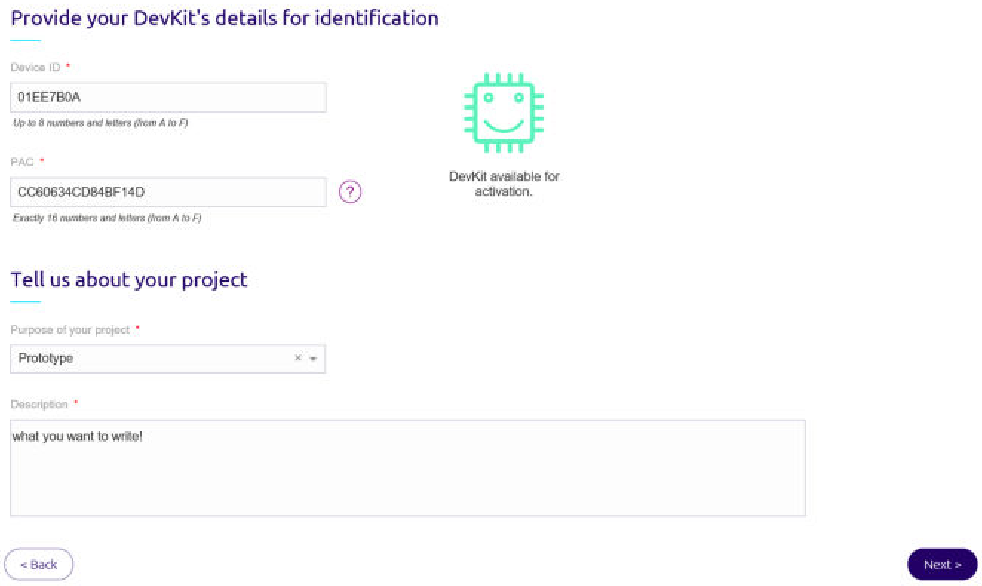
```
  SigFox credential file : C:\SOFT_DOCS\KmsCredentials\sigfox_data.h


Memory Programming ...
Opening and parsing file: Sigfox_EmbKey.bin
  File          : Sigfox_EmbKey.bin
  Size          : 592 Bytes
  Address       : 0x0803E500


Erasing memory corresponding to segment 0:
Erasing internal memory sector 31
Download in Progress:
                                                  100%

File download complete
Time elapsed during download operation: 00:00:00.052



Verifying ...


Read progress:
                                                  100%

Download verified successfully
```

15.2 Activation

Follow these steps:

1. Use `AT$ID?<CR>` and `AT$PAC?<CR>` commands to get Sigfox ID and PAC.
2. Go on https://buy.sigfox.com/activate/ and login.
3. Copy the device ID and PAC into the activate page (see the figure below) and click *Next*.

Figure 33. **Device activation (1/2)**



Provide your DevKit's details for identification

Device ID *

01EE7B0A

Up to 8 numbers and letters (from A to F)

PAC *

CC60634CD84BF14D

Exactly 16 numbers and letters (from A to F)

DevKit available for activation.

Tell us about your project

Purpose of your project *

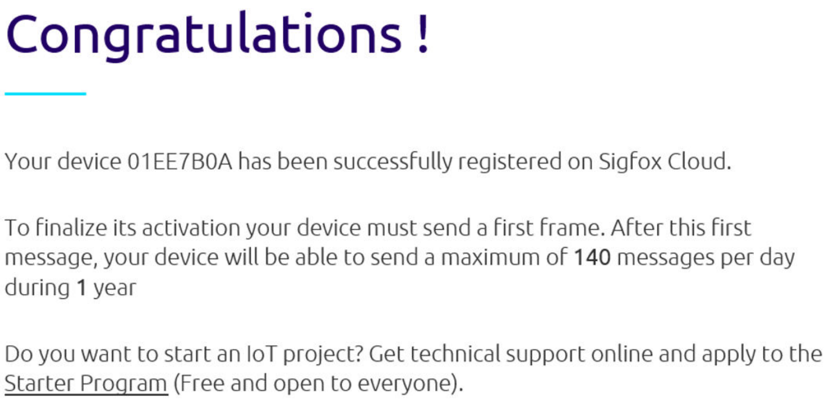
Prototype

Description *

what you want to write!

4. The browser loads the page shown below for the example.

Figure 34. **Device activation (2/2)**



Congratulations !

Your device 01EE7B0A has been successfully registered on Sigfox Cloud.

To finalize its activation your device must send a first frame. After this first message, your device will be able to send a maximum of **140** messages per day during **1** year

Do you want to start an IoT project? Get technical support online and apply to the Starter Program (Free and open to everyone).

5. The device is now activated on the Sigfox network for 1 year (evaluation activation).

15.3 See the message

Go to https://backend.sigfox.com/device/list to see the device listed (click on *DEVICE*). Data can be sent using the `AT$SF` command, for example, on the terminal. The device sends data to the Sigfox network and messages are visible on the backend (click on the device *Id* and the go on the *MESSAGES* tab).

Caution: *The Sigfox backend records a sequence number matching the device sequence number. This sequence number is incremented on both sides every time a new message is sent/received. The backend accepts messages only if the device sequence number is greater or equal to the sequence number of the backend. The device sequence number is stored in the EEPROM emulation of the device on the flash memory. When the application is in development, the EEPROM may be erased, for example with the cube programmer. In this case, the device sequence number is reset to 0, then smaller than the sequence number of the backend. Messages are not displayed but uplinks can still be seen the EVENTS tab. In order to see messages again, press on Disengage sequence number. This resets the sequence number of the backend, allowing the backend to accept new messages.*

16 How to secure a Sigfox application

The application note *How to secure LoRaWAN and Sigfox with STM32CubeWL* (AN5682) describes how to secure a dual-core Sigfox application using the SBSFU framework.

17 System performance

17.1 Memory footprint

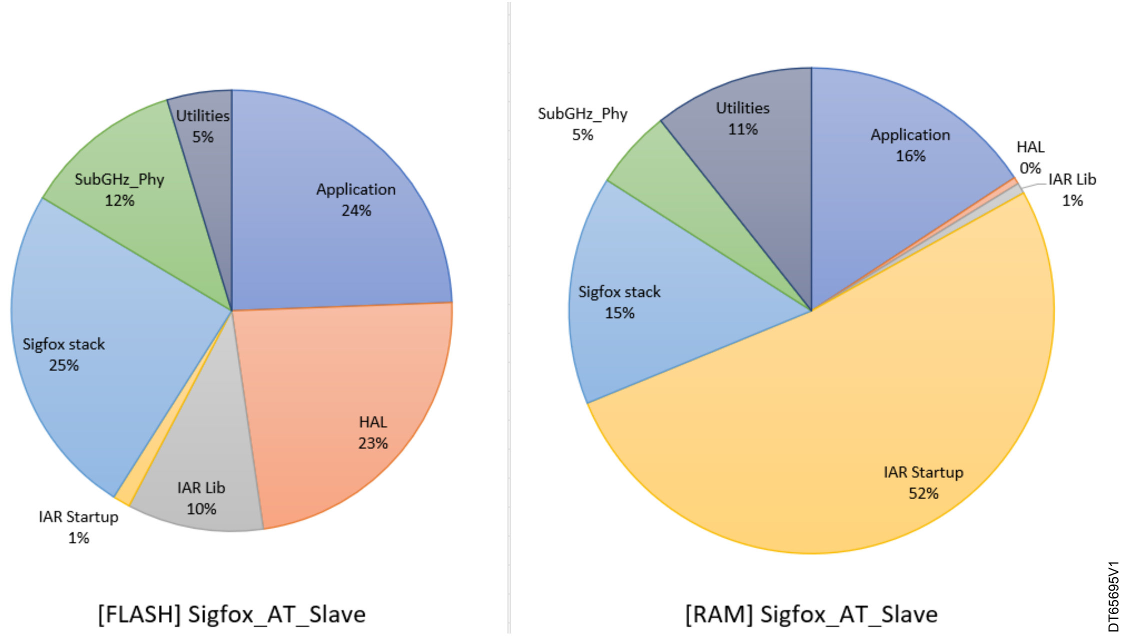
The values of the figure below have been extracted from the map file using the following configuration of the IAR Embedded Workbench compiler (EWARM compiler 9.20.1):

- Optimization: optimized for size level 3
- Debug option: off

Table 30. Memory footprint values for Sigfox_AT_Slave application

Project module	Flash memory (bytes)	RAM (bytes)	Description
Application	16923	1246	Core, application, and target components
HAL	16162	36	STM32WL5x/Ex HAL and LL drivers
IAR Lib	6941	56	Proprietary IAR libraries
IAR Startup	888	4097	Int_vect, init routines, init table, CSTACK, and HEAP
Sigfox stack	17040	1205	Middleware Sigfox and libraries
SubGHz_Phy	8110	417	Middleware radio interface
Utilities	3286	844	All STM32 services (sequencer, time server, low-power mgr, trace, mem)
Total application	69350	7901	Memory footprint for Sigfox_AT_Slave application

Figure 35. Flash memory and RAM footprint



17.2 Real-time constraints

Real-time constraints apply when the Monarch algorithm runs.

17.3 Power consumption

The power consumption has been measured on the STM32WL Nucleo-64 board (NUCLEO-WL55JC) with the following setup:

- No DEBUG
- No TRACE

In these conditions, the typical consumption in Stop mode is 2 µA.

Revision history

Table 31. **Document revision history**

Date	Version	Changes
20-May-2020	1	Initial release.
17-Nov-2020	2	Updated: • Section 10 Memory section • Section 11.1 Firmware package • Intro of Section 11.2 AT modem application • Section 11.2.3 AT? - Available commands • Section 11.2.9 ATS410 - Encryption key • Section 11.2.10 ATS411 - Payload encryption • Section 11.2.22 AT$RC - Region configuration • Section 11.3 PushButton application • Section 11.4 Static switches • Section 14.1 Personalization Added: • Section 12 Dual-core management • Section 13 Key management services (KMS)
18-Jan-2021	3	Updated: • Nucleo-73 corrected in Nucleo-64 in the whole document • RC5 in Table 3. RF parameters for region configurations • Intro of Section 5 Sigfox Stack description • Intro of Section 14 Personalization and activation • Step 5 of Section 14.2 Activation
7-July-2021	4	Updated: • Table 1. Acronyms • Intro of Section 4 BSP STM32WL Nucleo boards • end of Section 6.1 Sigfox Core library introduction • end of Section 6.2 Sigfox Addon RF protocol library • end of Section 6.3 Cmac library • Table 16. Radio_s structure callbacks • Section 9.1 Sequencer • Section 9.3 Low-power functions • Section 11.4 Static switches • Section 14.1.2 Loading the credentials in the device • Figure 36. Memory footprint FLASH Sigfox_AT_Slave Added Section 15 How to secure a Sigfox application Removed: • 'RF wakeup time' in Section 4 BSP STM32WL Nucleo boards • note from Section 2.3 Rx/Tx radio time diagram
01-Feb-2022	5	Added Section 4.5 Maximum Tx RF output power. Updated: • Table 1. Acronyms and terms • Section 5.1 Sigfox certification • Figure 7. Reception MSC • Table 20. Sequencer APIs • Note in Section 14.1 Personalization • Section 9.1 Sequencer

Date	Version	Changes
17-Nov-2022	6	Updated: • Section Introduction • Section 2.1: End-device hardware architecture • Section 4.2: RF switch • Section 6: Sigfox Stack description • Section 8.1: Middleware radio driver structure • Section 8.2: Radio IRQ interrupts • Figure 10. Package overview • Section 12.2: AT modem application • Section 12.2.2: Default parameters • Section 12.3: PushButton application • Section 12.4.1: Debug switch • Section 12.4.2: Low-power switch • Section 12.4.3: Trace level • Section 12.4.4: Probe pins • Section 12.4.5: Radio configuration • Section 17.1: Memory footprint Added: • Section 5: BSP B-WL5M-SUBG1 boards • Section 5.1: RF switch • Section 5.2: External components
06-Dec-2024	7	Updated: • Section 13.2.3: Mapping table • Section 15.1.2: Loading the credentials in the device
20-Mar-2025	8	Changed the scope of this document from STM32WL Series to STM32WL5x/Ex devices.

Contents

List of tables

List of figures