LoRa Basics™ Modem
Porting Guide

Table of Contents

List of Figures

List of Tables

1 Introduction

The LoRa Basics™ Modem has been designed for easy portability and use with a variety of microcontrollers and Semtech transceivers. To this end, it implements a stacked architecture in which the microcontroller and transceiver interact via abstraction layers.

1.1 Purpose of this Manual

This document describes how to port the LoRa Basics Modem to a microcontroller or board.

1.2 Scope

This document applies to the LoRa Basics Modem (SWL2001). This version of this document applies to version 3.2.4 (https://github.com/Lora-net/SWL2001/releases/tag/v3.2.4).

It should be read in conjunction with the following documents:

- LoRa Basics Modem SDK User Manual and associated SDK (SWSD001), version 2.0.0 (https://github.com/Lora-net/SWSD001/releases/tag/v2.0.0)
- LoRa Basics Modem User Manual, which contains information about the API (Refer to the LoRa Edge web page: (https://www.semtech.com/products/wireless-rf/lora-edge/lr1110#documentation)
- LoRa Development Portal which contains information about LoRa Cloud Modem & Geolocation Services, Application server code, and other resources (https://lora-developers.semtech.com/)

2 Overview

The LoRa Basics Modem (SWL2001) runs on top of a radio driver and a Radio Abstraction Layer (RAL).

To port the LoRa Basics Modem to a microcontroller, the LoRa Basics Modem Hardware Abstraction Layer (HAL) must be implemented for that microcontroller.

For each transceiver used on that microcontroller, it is necessary to implement the radio driver HAL. It is also necessary to implement the board support package for the Radio Abstraction Layer.

In what follows, `LBM_DIR` refers to the directory containing the LoRa Basics Modem, which is named `lora_basics_modem` in the SDK (SWSD001).

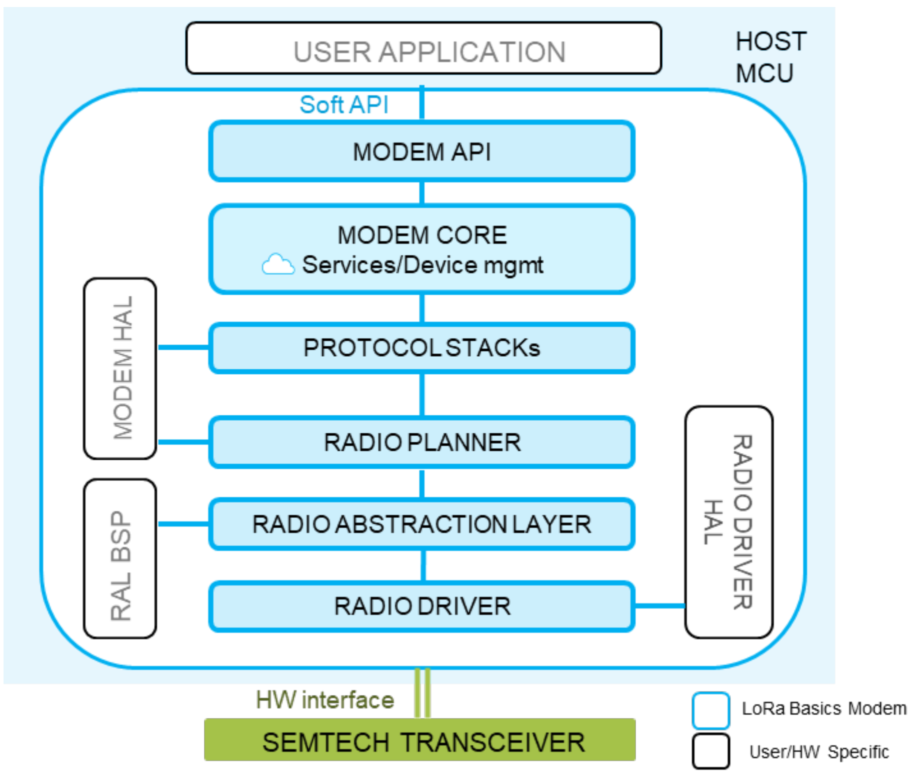

Figure 1: LoRa Basics™ Modem Software Stack

2.1 MCU Requirements

The LoRa Basics Modem SDK contains an implementation for the STMicroelectronics STM32L476 Nucleo board, however, LoRa Basics Modem can be easily ported to other MCUs. The following MCU features are required:

- 32-bit native operation (No specific CPU core needed)
- Refer to section 2.3 for detailed memory requirements
- Little-endian
- Software MCU reset
- A timer with 100µs resolution or better (Timer accuracy compensation is possible by widening the LoRaWAN reception windows)
- A random number generator (can be implemented in software)
- Non-volatile storage for modem state storage (refer to section 2.3)
- An SPI controller with MISO, MOSI, SCK, NSS
- GPIO lines for the transceiver IRQ, BUSY, and RESET lines
- A dedicated (non-shared) GPIO MCU interrupt for the transceiver IRQ line is recommended

Note that reliable Class A LoRaWAN communication can be obtained without any major time constraints on the MCU oscillator, or the oscillator used to clock the devices that implement the time-related LoRa Basics Modem HAL functions. To compensate for time-related oscillator frequency errors, calling the `smtc_modem_set_crystal_error_ppm()` modem API function with an appropriate value is sufficient. This results in a widening of the LoRaWAN reception window and increased power consumption. In the case of Class B, however, it is desirable to be able to remain synchronized with the beacon over relatively long time intervals, even if beacons are sometimes not received due to poor RF conditions. In this case, it is recommended to use an accurate crystal oscillator or TCXO to clock the devices used to implement the time-related LoRa Basics Modem HAL functions.

2.2 Transceiver Requirements

This version of LoRa Basics Modem supports the following transceivers:

- LR1110 transceiver with firmware version 0x0307 or greater
- LR1120 transceiver with firmware version 0x0101 or greater
- SX1261
- SX1262
- SX1268
- SX1280 (not yet supported by the LoRa Basics Modem SDK)
- SX1281 (not yet supported by the LoRa Basics Modem SDK)

In certain situations, such as the use of GNSS reception with the LR11xx, a transceiver TCXO is required. When using GNSS advanced scan on the LR1110, the TCXO must have a relatively fast settling time, and the 32.768 kHz crystal oscillator must have 20ppm accuracy at 25 degrees. For more information, see Application Note AN1200.59.

Additionally, when transmitting at high power with little thermal insulation between the transceiver and its oscillator, self-heating of the latter may occur, resulting in drift that may interfere with communication. For more information, consult your transceiver documentation and Application Note AN1200.59.

2.3 Release Build Resource Use

The RAM and flash use of a release build for LR1110 on STM32L476 are listed below. These values were determined by building the SDK "lorawan" example with the Arm® GNU Toolchain version 8-2018-q4-major, after modifying the file `apps/examples/lorawan/makefile/Makefile`, as follows:

```
APP_TRACE ?= no
MODEM_TRACE ?= no
DEBUG ?= no
OPT ?= -Os

$ make REGION=EU_868,AS_923,US_915,AU_915,CN_470,AS_923_GRP2,AS_923_GRP3,IN_865,KR_920,RU_
864,CN_470_RP_1_0

   text data    bss    dec    hex filename
    522    0      0    522    20a lr11xx_bootloader.o
   1242    0      0   1242    4da lr11xx_crypto_engine.o
     15    0      0     15      f lr11xx_driver_version.o
   2818    0      0   2818    b02 lr11xx_radio.o
    486    0      0    486    1e6 lr11xx_regmem.o
   1554    0      0   1554    612 lr11xx_system.o
   2286    0      0   2286    8ee lr11xx_wifi.o
    326    0      0    326    146 lr11xx_lr_fhss.o
   1636    0      0   1636    664 lr11xx_gnss.o
   2925    0      0   2925    b6d ral_lr11xx.o
    238    0      0    238     ee ralf_lr11xx.o
   3154    0      0   3154    c52 radio_planner.o
     36    0      0     36     24 radio_planner_hal.o
   2476    0   3512   5988   1764 lorawan_api.o
    800    0      0    800    320 dm_downlink.o
   4935   15    433   5383   1507 modem_context.o
   5757    0   2764   8521   2149 smtc_modem.o
   2612    0    276   2888    b48 smtc_modem_test.o
    756    0      0    756    2f4 fifo_ctrl.o
     52    0      0     52     34 modem_utilities.o
    144    0      0    144     90 smtc_modem_services_hal.o
   1326    0      0   1326    52e lorawan_certification.o
   3724    3    675   4402   1132 modem_supervisor.o
    948    0      0    948    3b4 smtc_clock_sync.o
     58    0      0     58     3a almanac_update.o
    230    0      0    230     e6 stream.o
   1882    0      0   1882    75a rose.o
   1342    0      0   1342    53e file_upload.o
    740    0      0    740    2e4 alc_sync.o
   1676    0     24   1700    6a4 lr11xx_ce.o
   1134   16      0   1150    47e smtc_modem_crypto.o
   1406    0      0   1406    57e region_as_923.o
   2564    0      0   2564    a04 region_au_915.o
   2739    0      0   2739    ab3 region_cn_470.o
   1370    0      0   1370    55a region_cn_470_rp_1_0.o
   1553    0      0   1553    611 region_eu_868.o
   1147    0      0   1147    47b region_in_865.o
   1058    0      0   1058    422 region_kr_920.o
   1269    0      0   1269    4f5 region_ru_864.o
   2509    0      0   2509    9cd region_us_915.o
```

```
  7884    0     0   7884  1ecc lr1_stack_mac_layer.o
  2878    0     0   2878   b3e lr1mac_core.o
   678    0     0    678   2a6 lr1mac_utilities.o
  7165    0     0   7165  1bfd smtc_real.o
  1313    0     0   1313   521 smtc_duty_cycle.o
   860    0     0    860   35c smtc_lbt.o
  1919    0     0   1919   77f lr1mac_class_c.o
  3332    0     0   3332   d04 smtc_beacon_sniff.o
  3215    0     0   3215   c8f smtc_ping_slot.o
   226    0     0    226    e2 smtc_multicast.o
 92915   34  7684 100633 18919 (TOTALS)

  text data   bss    dec   hex filename
124648 3576 42252 170476 299ec build/lorawan.elf
```

The "(TOTALS)" line above indicates the resource use of the LoRa Basics Modem, the radio drivers, and the RAL. Certain features of LoRa Basics Modem that are not used by the application will be removed at link time, so the effective size of LoRa Basics Modem is likely to be smaller for a given application.

The "build/lorawan.elf" line indicates the resource use of the entire project, including the simple LoRaWAN® demo application, and the LoRa Basics Modem HAL implementation, based on the STM32Cube MCU HAL implementation. It is possible to reduce resource use by using a size-optimized HAL for a given platform.

The worst-case stack use is currently unknown. A simple example that includes joining a device to the network and then sending a few uplinks uses approximately 2kB of stack RAM.

We can conclude that a release build of the stack, drivers, and RAL uses 92915 bytes of flash and 9766 bytes of RAM (34+7684+2048). A release build of the entire demo uses 124648 bytes of flash and 47876 bytes of RAM (3576+42252+2048).

Component	Flash	RAM
Stack, drivers, RAL	92915	9766
Entire STM32L476 demo (including stack, drivers, RAL, LoRa Basics Modem HAL, STM32Cube MCU HAL)	124648	47876

Table 1: Release Build Resource Use for All Supported Regions

If some supported regions are not needed, the approximate flash use may be obtained by subtracting the flash use of the unneeded regions:

Region	Flash	RAM
EU_868	1553	0
US_915	2509	0
CN_470	2739	0
CN_470_RP_1_0	1370	0
AS_923	1406	0
AU_915	2564	0
IN_865	1147	0
KR_920	1058	0
RU_864	1269	0

Table 2: Resource Use Values to Subtract for Unused Regions

For example, a complete LoRa Basics Modem stack for the US_915 region uses approximately 79809 bytes of flash (92915-1553-2739-1370-1406-2564-1147-1058-1269). Note that this is an approximate calculation. By compiling with REGION= US_915, one can determine that the exact flash use for a complete LoRa Basics Modem stack for this region is 77864 bytes.

2.4 Debug Build Resource Use

For a debug build for the LR1110 on the STM32L476, the RAM and flash requirements for LoRa Basics Modem, the radio drivers, and the RAL are listed below. These values were determined by building the SDK "lorawan" example with the Arm® GNU Toolchain version 8-2018-q4-major, after modifying the file `apps/lorawan/makefile`, as follows:

```
APP_TRACE ?= yes
MODEM_TRACE ?= yes
DEBUG ?= yes
OPT ?= -O0

$ make REGION=EU_868,AS_923,US_915,AU_915,CN_470,AS_923_GRP2,AS_923_GRP3,IN_865,KR_920,RU_
864,CN_470_RP_1_0
```

```
   text data    bss    dec    hex filename
   1014    0      0   1014    3f6 lr11xx_bootloader.o
   2444    0      0   2444    98c lr11xx_crypto_engine.o
     19    0      0     19     13 lr11xx_driver_version.o
   5442    0      0   5442   1542 lr11xx_radio.o
   1110    0      0   1110    456 lr11xx_regmem.o
   2982    0      0   2982    ba6 lr11xx_system.o
   5634    0      0   5634   1602 lr11xx_wifi.o
    802    0      0    802    322 lr11xx_lr_fhss.o
   3230    0      0   3230    c9e lr11xx_gnss.o
   6187    0      0   6187   182b ral_lr11xx.o
   1156    0      0   1156    484 ralf_lr11xx.o
   8788    0      0   8788   2254 radio_planner.o
    100    0      0    100     64 radio_planner_hal.o
   6588    0   3512  10100   2774 lorawan_api.o
   4690   56      0   4746   128a dm_downlink.o
  12681   15    442  13138   3352 modem_context.o
  17226    0   2772  19998   4e1e smtc_modem.o
   8042    0    276   8318   207e smtc_modem_test.o
   1556    0      0   1556    614 fifo_ctrl.o
    124    0      0    124     7c modem_utilities.o
    375    0      0    375    177 smtc_modem_services_hal.o
   6221    0      0   6221   184d lorawan_certification.o
  10800    3    676  11479   2cd7 modem_supervisor.o
   3945    0      0   3945    f69 smtc_clock_sync.o
    108    0      0    108     6c almanac_update.o
    516    0      0    516    204 stream.o
   4561    0      0   4561   11d1 rose.o
   2484    0      0   2484    9b4 file_upload.o
   2972    0      0   2972    b9c alc_sync.o
   3083    0     24   3107    c23 lr11xx_ce.o
   2356   16      0   2372    944 smtc_modem_crypto.o
   4916    0      0   4916   1334 region_as_923.o
   8246    0      0   8246   2036 region_au_915.o
   8118   16      0   8134   1fc6 region_cn_470.o
   5266    0      0   5266   1492 region_cn_470_rp_1_0.o
   4878    0      0   4878   130e region_eu_868.o
   4496    0      0   4496   1190 region_in_865.o
   4432    0      0   4432   1150 region_kr_920.o
   4676    0      0   4676   1244 region_ru_864.o
   8214    0      0   8214   2016 region_us_915.o
```

```
 22973   64      0  23037   59fd lr1_stack_mac_layer.o
 10297  168      0  10465   28e1 lr1mac_core.o
  2208    0      0   2208    8a0 lr1mac_utilities.o
 17774    0      0  17774   456e smtc_real.o
  2904    0      0   2904    b58 smtc_duty_cycle.o
  4053    0      0   4053    fd5 smtc_lbt.o
  6989    0      0   6989   1b4d lr1mac_class_c.o
  9215    0      0   9215   23ff smtc_beacon_sniff.o
 10828    0      0  10828   2a4c smtc_ping_slot.o
   689    8      0    697    2b9 smtc_multicast.o
268408  346   7702 276456 437e8 (TOTALS)

  text data    bss    dec    hex filename
270720 3888 42276 316884 4d5d4 build/lorawan.elf
```

The "(TOTALS)" line above indicates the resource use of the LoRa Basics Modem, the radio drivers, and the RAL. The "build/lorawan.elf" line indicates the resource use of the entire project, including the simple LoRaWAN® demo application, and the LoRa Basics Modem HAL implementation, based on the STM32Cube MCU HAL implementation. Depending on the features used by the final application, parts of the LoRa Basics Modem library may be removed by the linker and take no space in the final executable.

The worst-case stack use is currently unknown. A simple example that includes joining a device to the network and then sending a few uplinks uses approximately 4kB of stack RAM.

We can conclude that a debug build of the stack, drivers, and RAL uses 268408 bytes of flash and 12144 bytes of RAM (346+7702+4096). A debug build of the entire demo uses 270720 bytes of flash and 50260 bytes of RAM (3888+42276+4096).

Component	Flash	RAM
Stack, drivers, RAL	268408	13317
Entire STM32L476 demo (including stack, drivers, RAL, LoRa Basics Modem HAL, STM32Cube MCU HAL)	270720	50260

Table 3: Debug Build Resource Use for All Supported Regions

If some supported regions are not needed, the approximate flash use may be obtained by subtracting the flash use of the unneeded regions:

Region	Flash	RAM
EU_868	4878	0
US_915	8214	0
CN_470	8134	0
CN_470_RP_1_0	5266	0
AS_923	4916	0
AU_915	8246	0
IN_865	4496	0
KR_920	4432	0
RU_864	4676	0

Table 4: Resource Use Values to Subtract for Unused Regions

2.5 System Design Considerations

There are numerous requirements and options to consider when developing a device that implements LoRaWAN. For the LoRa Basics Modem, it is important to consider the transceiver and timer interrupt behavior and configuration. The LoRa Basics Modem is designed to use a specific transceiver DIO line as the radio interrupt source. For SX126x, this is the DIO1 line, and for LR11xx, this is the DIO9 line.

Two principal interrupt sources interact with the LoRa Basics Modem: a timer interrupt, and a radio interrupt.

The system interrupt priorities must be configured in such a way that the timer and radio interrupts do not nest or interrupt each another.

The current implementation of the LoRa Basics Modem has been designed to perform certain radio operations in the MCU's interrupt context. For this reason, HAL API commands are provided to disable and enable these two interrupt sources.

Therefore, when designing hardware that will run LoRa Basics Modem, it is recommended that the MCU GPIO lines selected for the transceiver's DIO interrupt request line do not share an MCU interrupt flag with other timing-critical hardware. If MCU interrupt flags are shared, it may not always be possible to react immediately to interrupts originating from these other devices.

The LoRa Basics Modem timer and radio interrupt service routines may perform radio operations over the transceiver SPI bus. If the MCU hardware SPI controller is used to communicate with other devices, interference to that communication may occur due to the timer and radio interrupt service routines that might reconfigure the MCU hardware SPI controller at an unexpected time. Therefore, it is recommended that the radio has exclusive use of its MCU hardware SPI controller device. In certain circumstances, it may be possible to coordinate the communication between devices sharing the SPI controller. However, that is beyond the scope of this document.

3 Radio Driver HAL Implementation

The LoRa Basics Modem depends on Semtech's radio driver, which, in turn, requires a radio driver HAL implementation. A brief description of the necessary steps for this implementation follows.

The HAL implementation must provide platform-specific read, write, reset, and wakeup implementations.

- Radio driver API functions call the HAL implementation to perform the actual reset, wake, and communication operations needed by the driver.
- For the LR11xx, these functions are documented in `LBM_DIR/smtc_modem_core/radio_drivers/lr11xx_driver/src/lr11xx_hal.h`.
- For the SX126x, these functions are documented in `LBM_DIR/smtc_modem_core/radio_drivers/sx126x_driver/src/sx126x_hal.h`.

All radio driver API functions take a `'const void* context'` argument:

- This argument is opaque to both the radio driver and LoRa Basics Modem.
- It may be used by the HAL implementer to differentiate between different transceivers, which makes it easy to communicate with several radios inside the same application.
- Driver API functions do not use the `context` argument but pass it directly to the HAL implementation.

The LoRa Basics Modem imposes a specific requirement on the radio driver HAL implementation:

- If a radio driver API function is called while the transceiver is in sleep mode, the HAL implementation must properly wake the transceiver and wait until it is ready before initiating any SPI communication.
- This typically requires that the HAL keeps track of whether the radio is awake or asleep, potentially by monitoring any commands sent to the transceiver to detect the `SetSleep` command.
- For a concrete LR11xx example, see the SDK file: `shields/LR11XX/radio_drivers_hal/lr11xx_hal.c`.
- For a concrete SX126x example, see the SDK file: `shields/SX126X/radio_drivers_hal/sx126x_hal.c`.

When compiling the radio driver HAL implementation, it is necessary to add the radio driver source directory to the include path. For example, for LR11xx:

- `LBM_DIR/smtc_modem_core/radio_drivers/lr11xx_driver/src`

4 RAL BSP Implementation

When porting the LoRa Basics Modem to a new radio + MCU implementation, a Radio Abstraction Layer (RAL) board support package (BSP) implementation is necessary. A brief description of the necessary steps follows.

The RAL provides radio-independent API functions that are similar to those provided by each radio driver. The RAL, and a complementary layer called the RALF, are described in the following header functions:

- `LBM_DIR/smtc_modem_core/smtc_ral/src/ral.h`
- `LBM_DIR/smtc_modem_core/smtc_ralf/src/ralf.h`

The RAL requires the implementer to define a few BSP API functions for the selected transceiver, by providing platform or radio-specific information to the RAL.

- For the LR11xx, these functions are described in `LBM_DIR/smtc_modem_core/smtc_ral/src/ral_lr11xx_bsp.h`.
- For the SX126x, these functions are described in `LBM_DIR/smtc_modem_core/smtc_ral/src/ral_sx126x_bsp.h`.
- An LR11xx sample implementation is in the SDK file `shields/LR11XX/LR1110MB1LxKS/BSP/ral_bsp/ral_lr11xx_bsp.c`.
- An SX126x sample implementation is in the SDK file `shields/SX126X/SX1262MB1CAS/ral_sx126x_bsp.c`.

The role of the `'const void* context'` variable is described in Section 3. It is typically used to store radio-specific information, but depending on the radio driver BSP implementation, it may be `NULL` if a single transceiver is used. The RAL and RALF need to store the `'const void* context'` variable, and keep track of functions implementing the RAL and RALF for a given radio, as described below:

- Typically, on startup, an application creates a `ralf_t` structure, storing both the `'const void* context'` address and pointers to RAL and RALF API functions. The only information required from the application developer is the `context` variable.
- On startup, instead of taking the `'const void* context'` variable as a startup argument, LoRa Basics Modem requires the address of the `ralf_t` structure. This gives the modem full access to all RAL and RALF API functions.
- All the SDK examples call a function named `smtc_board_initialise_and_get_ralf()` which creates the `ralf_t` structure that gets passed to `smtc_modem_init()`. For details, see the following files:
  - `apps/examples/lorawan/main_lorawan.c`
  - `shields/LR11XX/LR1110MB1LxKS/BSP/board/lr1110_mb1lxks_board.c`

When compiling the RAL BSP implementation, it is necessary to add the radio driver source directory and the RAL source directory to the include path. For example, for LR11xx:

- `LBM_DIR/smtc_modem_core/radio_drivers/lr11xx_driver/src`
- `LBM_DIR/smtc_modem_core/smtc_ral/src`

5 LoRa Basics Modem HAL Implementation

Porting LoRa Basics Modem to a new MCU architecture requires implementing the modem Hardware Abstraction Layer (HAL) API commands described by the prototypes in the header file `LBM_DIR/smtc_modem_hal/smtc_modem_hal.h`.

Among other things, these API implementations define how timing information is provided to the LoRa Basics Modem, how random numbers are generated, and how data is stored in non-volatile memory.

If a TCXO is used, its startup timing behavior should be specified in the RAL BSP implementation, and the documentation of the `smtc_modem_hal_start_radio_tcxo()`, `smtc_modem_hal_stop_radio_tcxo()`, and `smtc_modem_hal_get_radio_tcxo_startup_delay_ms()` functions, should be consulted.

The following sections provide the list and more details on the different modem HAL APIs.

5.1 smtc_modem_hal_reset_mcu()

```
void smtc_modem_hal_reset_mcu( void );
```

Brief

Reset the MCU.

LoRa Basics Modem may need to reset the MCU on initial startup, or if a state arises from which the modem cannot recover without restarting.

5.2 smtc_modem_hal_reload_wdog()

```
void smtc_modem_hal_reload_wdog( void );
```

Brief

Reload the watchdog timer.

If the HAL implementation configures a watchdog timer, then this function should be implemented to reload the watchdog timer. Currently, the only code in LoRa Basics Modem that calls this HAL API command is the test code in `smtc_modem_test.c`.

5.3 smtc_modem_hal_get_time_in_s()

```
uint32_t smtc_modem_hal_get_time_in_s( void );
```

Brief

Provide the time since startup, in seconds.

LoRa Basics Modem uses this command to help perform various LoRaWAN® activities that do not have significant time accuracy requirements, such as `NbTrans` retransmissions.

Return

The current system uptime in seconds.

5.4 smtc_modem_hal_get_time_compensation_in_s()

```
int32_t smtc_modem_hal_get_time_compensation_in_s( void );
```

Brief

Provide a time-correcting term, in seconds.

Suppose that, due to MCU clock inaccuracy, the principal time source used for `smtc_modem_hal_get_time_in_s()` significantly lags behind or runs ahead of the real time. If the LoRa Basics Modem HAL developer can quantify this deviation and calculate an integer number of seconds that additively corrects the time source, it should be returned by this HAL API command. Otherwise, this command should return the value 0.

For example, consider an MCU clock that loses one second per day. If after exactly one day of runtime (86400 seconds), the API function `smtc_modem_hal_get_time_in_s()` returns 86399, then the API function `smtc_modem_hal_get_time_compensation_in_s()` should be implemented to return the value 1 after the first day, return 2 after the second day, and so on. This effectively corrects the error so that `smtc_modem_hal_get_compensated_time_in_s()` returns the correct time.

Return

Additive correction of the time source. Return zero, if unknown.

5.5 smtc_modem_hal_get_compensated_time_in_s()

```
uint32_t smtc_modem_hal_get_compensated_time_in_s( void );
```

Brief

Provide the compensated time since startup, in seconds.

This command should be implemented as follows:

```
uint32_t smtc_modem_hal_get_compensated_time_in_s()
{
    return smtc_modem_hal_get_time_compensation_in_s() + smtc_modem_hal_get_time_i
n_s();
}
```

If active, the ALC Sync service obtains accurate time from the network GPS clock. Currently, the ALC Sync implementation is the only LoRa Basics Modem code that uses the compensated time, as described in the brief for `smtc_modem_hal_get_time_compensation_in_s()`. This may seem unnecessary since the purpose of ALC Sync is to provide an accurate clock. However, if the time is accurately compensated by `smtc_modem_hal_get_time_compensation_in_s()` and `smtc_modem_hal_get_compensated_time_in_s()`, ALC Sync requires less network activity to keep the clock perfectly synchronized. In the future, this HAL API command may be removed.

Return

Additive correction of the time source. Return zero, if unknown.

5.6 smtc_modem_hal_get_time_in_ms()

```
uint32_t smtc_modem_hal_get_time_in_ms( void );
```

Brief

Provide the time since startup, in milliseconds.

Return

The system uptime, in milliseconds. The value returned by this function must monotonically increase all the way to 0xFFFFFFFF and then overflow to 0x00000000.

5.7 smtc_modem_hal_get_time_in_100us()

```
uint32_t smtc_modem_hal_get_time_in_100us( void );
```

Brief

Provide the time since startup, in 100µs units.

This command is used for Class B ping slot openings and must use the same timer as the one used for `smtc_modem_hal_get_radio_irq_timestamp_in_100us()`.

Return

The system uptime, in tenths of milliseconds. The value returned by this function must monotonically increase all the way to 0xFFFFFFFF, and then overflow to 0x00000000.

5.8 smtc_modem_hal_get_radio_irq_timestamp_in_100us()

```
uint32_t smtc_modem_hal_get_radio_irq_timestamp_in_100us( void );
```

Brief

Provide the time of the last radio interrupt (i.e.: the end of TX), in 100µs units.

Return

The timestamp, in tenths of milliseconds, of the last radio IRQ event. This must use the same timer as the one used for `smtc_modem_hal_get_time_in_100us()`.

5.9 smtc_modem_hal_start_timer()

```
void smtc_modem_hal_start_timer(
    const uint32_t milliseconds,
    void ( *callback )( void* context ),
    void* context
    );
```

Brief

Start a timer that will expire at the requested time.

Upon expiration, the provided callback is called with `context` as its sole argument.

The current design of the LoRa Basics Modem has only been tested in the case where the provided callback is executed in an interrupt context, with interrupts disabled. Also, note that this callback may communicate with the radio using the MCU SPI device.

Parameters

[in]	`milliseconds`	Number of milliseconds before `callback` execution
[in]	`callback`	Callback to execute
[in]	`context`	Argument that is passed to `callback`

5.10 smtc_modem_hal_stop_timer()

```
void smtc_modem_hal_stop_timer( void );
```

Brief

Stop the timer that may have been started with `smtc_modem_hal_start_timer()`.

5.11 smtc_modem_hal_disable_modem_irq()

```
void smtc_modem_hal_disable_modem_irq( void );
```

Brief

Disable the two interrupt sources that execute the LoRa Basics Modem code: the timer, and the transceiver DIO interrupt source.

Please also refer to System Design Considerations.

5.12 smtc_modem_hal_enable_modem_irq()

```
void smtc_modem_hal_enable_modem_irq( void );
```

Brief

Enable the two interrupt sources that execute the LoRa Basics Modem code: the timer, and the transceiver DIO interrupt source.

Please also refer to Section 2.5.

5.13 smtc_modem_hal_context_restore()

```
void smtc_modem_hal_context_restore(
    const modem_context_type_t ctx_type,
    uint8_t* buffer,
    const uint32_t size
    );
```

Brief

Restore to RAM a data structure of type `ctx_type` that has previously been stored in non-volatile memory by calling `smtc_modem_hal_context_store()`.

Parameters

[in]	`ctx_type`	Type of modem context to be restored
[out]	`buffer`	Buffer where context must be restored
[in]	`size`	Number of bytes of context to restore

5.14 smtc_modem_hal_context_store()

```
void smtc_modem_hal_context_store(
    const modem_context_type_t ctx_type,
    const uint8_t* buffer,
    uint32_t size
    );
```

Brief

Store a data structure of type `ctx_type` from RAM to non-volatile memory.

Parameters

[in]	`ctx_type`	Type of modem context to be saved
[in]	`buffer`	Buffer which must be saved
[in]	`size`	Number of bytes of context to save

5.15 smtc_modem_hal_store_crashlog()

```
void smtc_modem_hal_store_crashlog( uint8_t crashlog[CRASH_LOG_SIZE] );
```

Brief

Store the modem crash log to non-volatile memory.

On most MCUs, RAM is preserved upon reset, so it may be possible to use RAM for this purpose.

Parameters

[in]	`crashlog`	Buffer pointer to write from

5.16 smtc_modem_hal_restore_crashlog()

```
void smtc_modem_hal_restore_crashlog( uint8_t crashlog[CRASH_LOG_SIZE] );
```

Brief

Retrieve the modem crash log from non-volatile memory.

On most MCUs, RAM is preserved upon reset, so it may be possible to use RAM for this purpose.

Parameters

[out]	`crashlog`	Buffer pointer to write to

5.17 smtc_modem_hal_set_crashlog_status()

```
void smtc_modem_hal_set_crashlog_status( bool available );
```

Brief

Store the modem crash log status to non-volatile memory. True indicates that a crash log has been stored and is available for retrieval.

On most MCUs, RAM is preserved upon reset, so it may be possible to use RAM for this purpose.

Parameters

[in]	`available`	True if a crash log is available; false otherwise

5.18 smtc_modem_hal_get_crashlog_status()

```
bool smtc_modem_hal_get_crashlog_status( void );
```

Brief

Get the modem crash log status from non-volatile memory.

Return

The crash log status, as previously written using `smtc_modem_hal_set_crashlog_status()`.

5.19 smtc_modem_hal_assert_fail()

```
void smtc_modem_hal_assert_fail( uint8_t* func, uint32_t line ) ;
```

Brief

Indicate the location of an unrecoverable error and reset the MCU.

Parameters

[in]	`func`	String indicating the name of the function
[in]	`line`	Line number

5.20 smtc_modem_hal_get_random_nb()

```
uint32_t smtc_modem_hal_get_random_nb( void );
```

Brief
Return a uniformly-distributed 32-bit unsigned random integer.

Return
The random integer.

5.21 smtc_modem_hal_get_random_nb_in_range()

```
uint32_t smtc_modem_hal_get_random_nb_in_range(
    const uint32_t val_1,
    const uint32_t val_2
    );
```

Brief
Return a uniformly-distributed unsigned random integer from the closed interval [val_1, ..., val_2] or [val_2, ..., val_1].

This command may be implemented as follows:

```
uint32_t smtc_modem_hal_get_random_nb_in_range( const uint32_t val_1, const
uint32_t val_2 )
{
    if( val_1 <= val_2 )
    {
        return ( uint32_t )( ( smtc_modem_hal_get_random_nb( ) % ( val_2 - val_1 +
1 ) ) + val_1 );
    }
    else
    {
        return ( uint32_t )( ( smtc_modem_hal_get_random_nb( ) % ( val_1 - val_2 +
1 ) ) + val_2 );
    }
}
```

In the future, this HAL API command may be removed.

Return
The random integer.

5.22 smtc_modem_hal_get_signed_random_nb_in_range()

```
int32_t smtc_modem_hal_get_signed_random_nb_in_range(
    const int32_t val_1,
    const int32_t val_2
    );
```

Brief

Return a uniformly-distributed signed random integer from the closed interval [val_1, ..., val_2] or [val_2, ..., val_1].

This command may be implemented as follows:

```
int32_t smtc_modem_hal_get_signed_random_nb_in_range( const int32_t val_1, const
int32_t val_2 )
{
    uint32_t tmp_range = 0;  // ( val_1 <= val_2 ) ? ( val_2 - val_1 ) : ( val_1 -
val_2 );

    if( val_1 <= val_2 )
    {
        tmp_range = ( val_2 - val_1 );
        return ( int32_t )( ( val_1 + smtc_modem_hal_get_random_nb_in_range( 0,
tmp_range ) ) );
    }
    else
    {
        tmp_range = ( val_1 - val_2 );
        return ( int32_t )( ( val_2 + smtc_modem_hal_get_random_nb_in_range( 0,
tmp_range ) ) );
    }
}
```

In the future, this HAL API command may be removed.

Return

The random integer.

5.23 smtc_modem_hal_irq_config_radio_irq()

```
void smtc_modem_hal_irq_config_radio_irq(
    void ( *callback )( void* context ),
    void* context
);
```

Brief

Store the `callback` and `context` argument that must be executed when a radio event occurs.

Parameters

[in]	`callback`	Callback that is executed upon radio interrupt service request
[in]	`context`	Argument that is provided to `callback`

5.24 smtc_modem_hal_radio_irq_clear_pending ()

```
void smtc_modem_hal_radio_irq_clear_pending( void );
```

Brief

Clear interrupt pending status, if an interrupt service request is pending inside the MCU hardware interrupt controller or stored as a flag in software.

After this function is called, the HAL implementation must guarantee that an interrupt that was raised before this function was called, will not be processed by the `callback` provided to the API function `smtc_modem_hal_irq_config_radio_irq()`.

5.25 smtc_modem_hal_start_radio_tcxo()

```
void smtc_modem_hal_start_radio_tcxo( void );
```

Brief

If the TCXO is not controlled by the transceiver, powers up the TCXO.

If no TCXO is used, or if the TCXO has been configured in the RAL BSP to start up automatically, then implement an empty command. If the TCXO is not controlled by the transceiver, then this function must power up the TCXO, and then *busywait* until the TCXO is running with the proper accuracy.

5.26 smtc_modem_hal_stop_radio_tcxo()

```
void smtc_modem_hal_stop_radio_tcxo( void );
```

Brief

If the TCXO is not controlled by the transceiver, stop the TCXO.

If no TCXO is used, or if the TCXO has been configured in the RAL BSP to start up automatically, implement an empty command.

5.27 smtc_modem_hal_get_radio_tcxo_startup_delay_ms()

```
uint32_t smtc_modem_hal_get_radio_tcxo_startup_delay_ms( void );
```

Brief

Return the time, in milliseconds, that the TCXO needs to start up with the required accuracy.

This does not implement a delay but is used to perform certain calculations in the LoRa Basics Modem so that this time will be taken into consideration when opening the Rx window.

If the TCXO is configured by the RAL BSP to start up automatically, then the value used here should be the same as the startup delay used in the RAL BSP.

Return

The needed TCXO startup time, in milliseconds. Return 0 if no TCXO is used.

5.28 smtc_modem_hal_get_battery_level()

```
uint8_t smtc_modem_hal_get_battery_level( void );
```

Brief

Indicate the current battery state.

Return

A value between 0 (for 0%) and 255 (for 100%).

5.29 smtc_modem_hal_get_temperature()

```
int8_t smtc_modem_hal_get_temperature( void );
```

Brief

Indicate the current system temperature.

Return

The temperature, in degrees Celsius.

5.30 smtc_modem_hal_get_voltage()

```
uint8_t smtc_modem_hal_get_voltage( void );
```

Brief

Indicates the current battery voltage.

Return

The battery voltage, in units of 20mV.

5.31 smtc_modem_hal_get_board_delay_ms()

```
int8_t smtc_modem_hal_get_board_delay_ms( void );
```

Brief

Return the amount of time that passes between the moment the MCU calls `ral_set_tx()` or `ral_set_rx()`, and the moment the radio transceiver enters RX or TX state.

This varies depending on the MCU clock speed and SPI bus speed. See Section 8 for more information.

Return

The board delay, in milliseconds.

5.32 smtc_modem_hal_print_trace()

```
void smtc_modem_hal_print_trace(
    const char* fmt,
    ...
    );
```

Brief

Output a printf-style variable-length argument list to the logging subsystem.

Parameters

[in]	`fmt`	printf-style string
[in]	...	Arguments that accompany `fmt`

6 Building with GNU Make

If GNU Make is available, it offers the easiest way to build the LoRa Basics Modem library. Command line arguments can be used to select the region, transceiver, logging (MODEM_TRACE), and other options.

For more information about building with GNU Make, type:

```
$ make help
```

For example, to build the LoRa Basics Modem library for an LR1110 transceiver with EU_868 regional support, type:

```
$ make basic_modem_lr1110 REGION=EU_868
```

When not building for an STM32L476, use the MCU_FLAGS make argument to specify any compilation flags needed by the MCU. For example:

```
$ make basic_modem_lr1110 REGION=EU_868 MCU_FLAGS="-mcpu=cortex-m4 –mthumb –mfpu=fpv4-sp-d16 –mfloat-abi=hard"
```

To compile the modem HAL implementation, it is necessary to add the following include directory:

- LBM_DIR/smtc_modem_hal

To compile the modem application code, it is necessary to add the following include directory:

- LBM_DIR/smtc_modem_api

The project must then link with the LoRa Basics Modem HAL implementation, the radio driver HAL implementation, the RAL BSP implementation, and the LoRa Basics Modem library. The latter will have one of these names, depending on the selected transceiver, and whether or not MODEM_TRACE has been chosen:

- LBM_DIR/build/basic_modem_<transceiver>_trace.a
- LBM_DIR/build/basic_modem_<transceiver>_notrace.a

For more information about the radio driver HAL implementation and RAL BSP implementation, see Sections 3 and 4.

7 Building without GNU Make

When building without `GNU Make`, the various source code files, include directories, and common preprocessor definitions can be found by looking through the files in the `LBM_DIR/makefiles` directory.

`LBM_DIR/makefiles/regions.mk` lists the source code files, include directories, and preprocessor definitions needed for all transceivers.

`LBM_DIR/makefiles/sx126x.mk` lists the source code files, include directories, and preprocessor definitions needed for the SX126x transceivers.

`LBM_DIR/makefiles/lr11xx.mk` lists the source code files, include directories, and preprocessor definitions needed for the LR11xx transceivers.

`LBM_DIR/makefiles/regions.mk` lists the source files and preprocessor definitions needed to select a set of regions.

7.1 Logging

To disable logging, define `MODEM_HAL_DBG_TRACE` to be equal to 0.

To enable additional logging of radio-related operations, define `MODEM_HAL_DBG_TRACE_RP` to be equal to 1.

It is preferable to use a high-speed UART to implement the trace because logging can potentially interfere with modem communication.

8 Rx Window Debugging

LoRaWAN® requires accurate receive window timing. This section provides tips to verify that the window timing is good.

Having an accurate MCU clock facilitates debugging, so when getting started with LoRa Basics Modem it is recommended to configure the MCU to provide the most accurate possible clock to the various time-related HAL API functions.

Enable logging, as described in Sections 6 and 7.1. A high baud rate is recommended, such as 921600 baud, since the logging code may interfere with the timing.

8.1 Clock Error Compensation

To provide an upper bound on crystal error, the modem API command `smtc_modem_set_crystal_error_ppm()` can be called to specify the crystal error, in parts per million. Large crystal error values result in wider Rx windows.

For more information, see Application Note AN1200.24.

8.2 Rx Window Fine-Tuning

The LoRa Basics Modem can use an algorithm to fine-tune the Rx window position.

To understand how this algorithm works, consider the case where the LoRa Basics Modem is running in the absence of a packet forwarder, or without an appropriate configuration on the network server. In this case, uplinks are not responded to and result in an `RxTimeout` interrupt. Since LoRa Basics Modem knows the reception timeout value that was used, the time elapsed between the `TxDone` interrupt and the `RxTimeout` interrupt can be used to position the start of the Rx window. This is the purpose of the fine-tuning algorithm found in `LBM_DIR/smtc_modem_core/lr1mac/src/lr1_stack_mac_layer.c.`

On every reception failure, the fine-tuning algorithm generates log messages like this:

```
DR3 Fine tune correction (ms) = 1, error fine tune (ms) = 0, lr1_mac->rx_offset_ms = -18
```

If this algorithm is working properly, on every reception failure for a given data rate, the *fine tune correction* value for that data rate will be incremented or decremented until it converges to a value that results in reliable reception. From this point on, *error fine tune* should stay close to zero. This approach works in many cases. To work well, the HAL `smtc_modem_hal_get_time…()` and `smtc_modem_hal_get_radio_irq_timestamp…()` functions must provide accurate time. Timing inaccuracies due to crystal oscillator aging or temperature change may cause a previously tuned system to malfunction.

8.2.1 Rx Window Debugging Configuration

Fine-tuning convergence may be slow, or not occur. Debugging this type of problem, and determining what value to use for `smtc_modem_hal_get_board_delay_ms()`, is the purpose of the following sections of this chapter.

In order to know if something is interfering with Rx window placement, it is important to know the desired length of the window, as requested by the MCU. This desired window length can then be compared to the actual window length, as measured by a logic analyzer.

With this in mind, *temporarily* deactivate the window fine-tuning feature by globally defining the preprocessor definition `BSP_LR1MAC_DISABLE_FINE_TUNE`.

8.2.2 Add IRQ Timing Log Information

The following change to the `rp_radio_irq()` function in `LBM_DIR/smtc_modem_core/radio_planner/src/radio_planner.c` makes it possible to observe in the log the MCU time at which every radio IRQ arrives. Change the following line of code:

```
SMTC_MODEM_HAL_RP_TRACE_PRINTF( " RP: INFO - Radio IRQ received for hook #%u\n", rp->radio_task_id );
```

to read:

```
SMTC_MODEM_HAL_RP_TRACE_PRINTF( " RP: INFO - Radio IRQ received for hook #%u at time %u\n", rp->radio_task_id, rp->irq_timestamp_ms[rp->radio_task_id] );
```

8.2.3 Add Ready and Trigger Timing Log Information

The variable `start_time_ms` contains the MCU time at which the `SetRx` command should be sent to the MCU. Shortly after, this provokes the opening of the Rx window.

When functioning properly, the command `lr1_stack_mac_rx_lora_launch_callback_for_rp()` is expected to be executed a short time before `start_time_ms`. After preparing the radio for reception, a while-loop inside this command waits until the current time is equal to `start_time_ms`. At this point in time, called the *trigger time*, the command `ral_set_rx()` is called. The point at which this while-loop was entered is called the *ready time*.

The following change to command `lr1_stack_mac_rx_lora_launch_callback_for_rp()` in `LBM_DIR/smtc_modem_core/lr1mac/src/lr1_stack_mac_layer.c` makes it possible to observe the MCU ready time and the MCU trigger time in the log.

Change the following block of code:

```
// Wait the exact time
while( ( int32_t )( rp->tasks[id].start_time_ms - smtc_modem_hal_get_time_in_ms( ) ) > 0 )
{
}
smtc_modem_hal_assert( ral_set_rx( &( rp->radio->ral ), rp->radio_params[id].rx.timeout_in_ms ) == RAL_STATUS_OK );
rp_stats_set_rx_timestamp( &rp->stats, smtc_modem_hal_get_time_in_ms( ) );
```

to read:

```
// Wait the exact time
uint32_t tcurrent_ms = smtc_modem_hal_get_time_in_ms( );
while( ( int32_t )( rp->tasks[id].start_time_ms - smtc_modem_hal_get_time_in_ms( ) ) > 0 )
{
}
smtc_modem_hal_assert( ral_set_rx( &( rp->radio->ral ), rp->radio_params[id].rx.timeout_in_ms ) == RAL_STATUS_OK );
rp_stats_set_rx_timestamp( &rp->stats, smtc_modem_hal_get_time_in_ms( ) );
SMTC_MODEM_HAL_TRACE_PRINTF( "RX ready at %d, triggered at %d\n", tcurrent_ms, rp->tasks[id].start_time_ms );
```

8.2.4 Perform a Debugging Session

Once the logging is configured as described above, connect a logic analyzer and run the LoRaWAN® example.

1. First, confirm that the MCU ready time is less than the MCU trigger time. If not, this indicates that there is no margin for error because either `lr1_stack_mac_rx_lora_launch_callback_for_rp()` is being entered too late, or the radio preparations are taking too long. Debugging with additional trace calls should be done in such a way as to not interfere with LoRa Basics Modem timing. Other possibilities for debugging include using LED diagnostics.
2. Referring back to section 8.2.2, observe the log to determine the MCU time at which the `TxDone` interrupt was timestamped by the MCU. Define this value *tm1*.
3. Referring back to section 8.2.3, observe the log to determine the MCU time at which the `SetRx` call was initiated. Define this value *tm2*.
4. Define *delta1* = tm2 - tm1.
5. Using a logic analyzer that can decode the radio SPI bus communication, search for the last transceiver command preceding the first transceiver post-reset radio interrupt service request. The command should be `SetTx`, which can be verified by looking at the SPI bus data and the transceiver user manual.
6. Define *ta1* to be the time, according to the logic analyzer, at which the DIO line rises.
7. Shortly after this moment when the DIO line rises, the MCU software should read and clear the IRQ status, causing the DIO line to fall.
8. Now, search for the last transceiver command preceding the second post-reset transceiver interrupt service request. This command should be `SetRx`, which can be verified by looking at the transceiver user manual. Define *ta2* to be the time, according to the logic analyzer, at which the NSS line fell right before sending `SetRx`. This corresponds approximately to the trigger time.
9. Define *delta2* = ta2 – ta1.
    a. delta1 is the MCU time between the `TxDone` interrupt and the initiation of the `SetRx` command.
    b. delta2 is the logic analyzer time between the `TxDone` interrupt and the initiation of the `SetRx` command.
    c. delta1 and delta2 should be within 1 ms of one another, after correcting for the MCU clock accuracy.
10. Recall that the board delay is the amount of time between the ready time and the moment the transceiver initiates reception. Consider the `SetRx` command on the logic analyzer, and observe the amount of time between the moment that NSS falls and the moment that NSS rises. This value, in milliseconds, is reasonably close to the board delay. Edit the `smtc_modem_hal_get_board_delay_ms()` HAL command so that it returns this value, after rounding up.

The window fine-tuning feature can now be reactivated, if desired, by undefining the preprocessor definition `BSP_LR1MAC_DISABLE_FINE_TUNE`.

9 Revision History

User Manual Version	ECO	Date	Applicable to	Changes
1.0	-	Apr 2020	Use Case: 01 FW Version: 03.07 or later	First Release
2.0	060217	Jan-2022	Use Case: 01 FW Version: 03.07 or later	Major updates for LoRa Basics Modem changes.
3.0	062410	Jul-2022	Use Case: 01 FW Version: 03.07 or later	Updated according to LoRa Basics Modem version 3.1.7.
4.0	064024	Oct-2022	-	Updated according to LoRa Basics Modem version 3.2.4.

Important Notice

Contact Information

Semtech Corporation
Wireless & Sensing Products
200 Flynn Road, Camarillo, CA 93012
E-mail: sales@semtech.com
Phone: (805) 498-2111, Fax: (805) 498-3804
www.semtech.com