LSM100A Sigfox
API Manual

Rev 1.0

SJI

JAN. 28. 2022

Contents

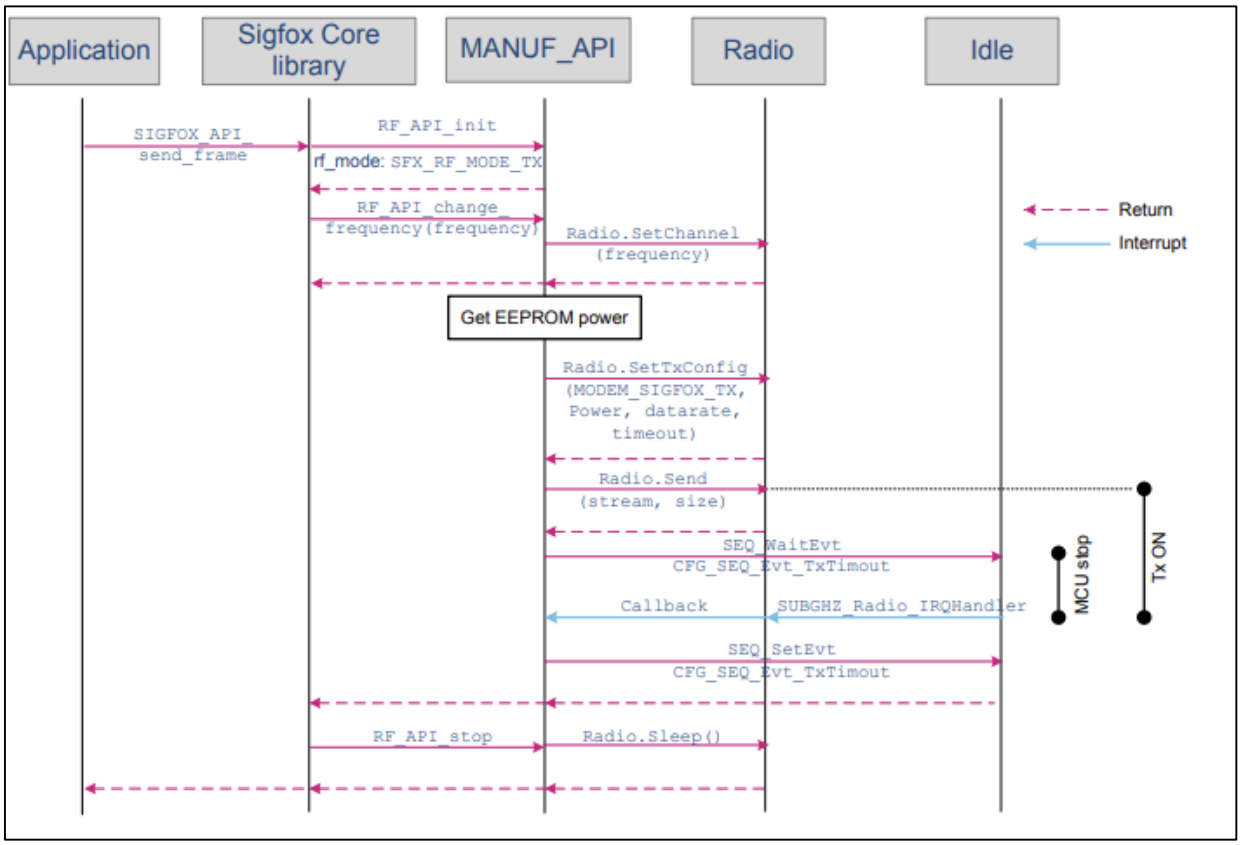
History

Date	Contents	Version	
2022-01-28	Create	V1.0	

1. Dynamic view

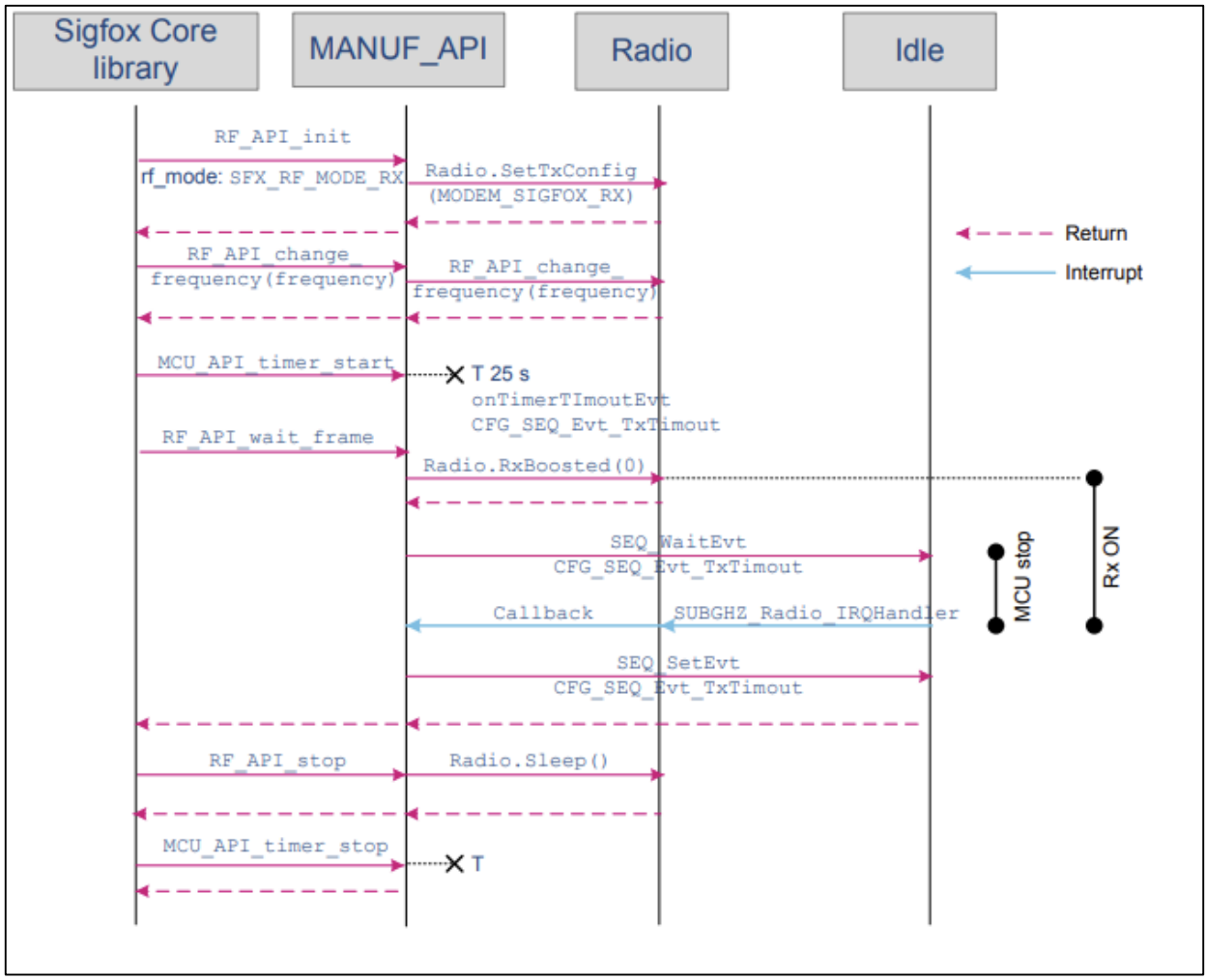
The message sequence chart (MSC) in the figure below depicts the dynamic calls between APIs in Tx mode (for one transmission).

Transmission MSC



When a downlink window is requested, an Rx sequence is started after Rxdelay is elapsed When Rxdelay is elapsed, the sequence detailed in the figure below occurs.

Reception MSC



2. Sigfox middleware programming guidelines

2.1 Sigfox Core library

Embedded applications using the Sigfox Core library call SIGFOX_APIs to manage Communication

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 isused 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 ismandatory 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 sentis 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 sentis 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 havedifferent meanings whether in FH or LBT mode. *Note: this function has no influence in DC (see Section 11.2.21 ATS400 - Enabled channels for FCC fordetails).*

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

2.2 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

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 theSigfox Verified certification: `rc_enum`: rc at which the testmode 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 protocoltests.

This library is located in Middlewares₩Third_Party₩Sgfx₩SigfoxLibTest₩.

3. EEPROM

3.1 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`.

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);`

4. 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.

4.1 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_conf.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 can not switch to another task like a RTOS would do on RTOS tick. 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
- Waiting event and set event
- Task priority setting

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.

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_tmask_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.

4.2 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`.

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 functionwhen timer elapses.
`UTIL_TIMER_Status_t UTIL_TIMER_SetPeriod(UTIL_TIMER_Object_t *TimerObject, uint32_t NewPeriodValue)`	Updates the period and startsthe timer with a timeout value(milliseconds).
`UTIL_TIMER_Status_t UTIL_TIMER_Start ( UTIL_TIMER_Object_t *TimerObject )`	Starts and adds the timer object tothe 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.

4.3 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 in use 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

Low-power APIs

Function	Description
`void UTIL_LPM_EnterLowPower( void )`	Enters the selected low-power mode. Called by idle state ofthe system
`void LPM_SetStopMode(LPM_Id_t id,LPM_SetMode_t mode)`	Sets Stop mode. `id` defines the process mode requested: `LPM_Enable` or `LPM_Disable`.(1)
`void LPM_SetOffMode(LPM_Id_t id,LPM_SetMode_t mode)`	Sets Stop mode. `id` defines the process mode requested: `LPM_Enable` or `LPM_Disable`.
`UTIL_LPM_Mode_t UTIL_LPM_GetMode( void )`	Returns the selected low-power mode.

LPM_Id_t are bitmaps. Their shift values are defined in utilities_def.h of project sub-folder

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

4.4 System time

The MCU time is referenced to the MCU reset. The system time is able to record the UNIX® epoch time.The APIs presented in the table below are used to manage the system time of the core MCU.

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 thebackup 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)

4.5 Trace

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

Trace functions

Function	Description
`UTIL_ADV_TRACE_Status_t UTIL_ADV_TRACE_Init( void )`	`TraceInit` must be called at the applicationinitialization. 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_FSend(uint32_t VerboseLevel,uint32_t Region, uint32_t TimeStampState, const char *strFormat, ...)`	Converts string format into a buffer and posts it tothe circular queue for printing.
`UTIL_ADV_TRACE_Status_t UTIL_ADV_TRACE_Send(uint8_t *pdata, uint16_t len)`	Posts data of length = `len` and posts it to thecircular queue for printing.
`UTIL_ADV_TRACE_Status_tUTIL_ADV_TRACE_ZCSend (uint32_t VerboseLevel, uint32_t Region, uint32_t TimeStampState, uint32_t length, void (*usercb)(uint8_t*, uint16_t, uint16_t))`	Writes user formatted data directly in theFIFO (Z-Cpy).

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