LSM1x0A LoRa API Manual

Rev 1.0

SJI

JAN. 28. 2022

Contents

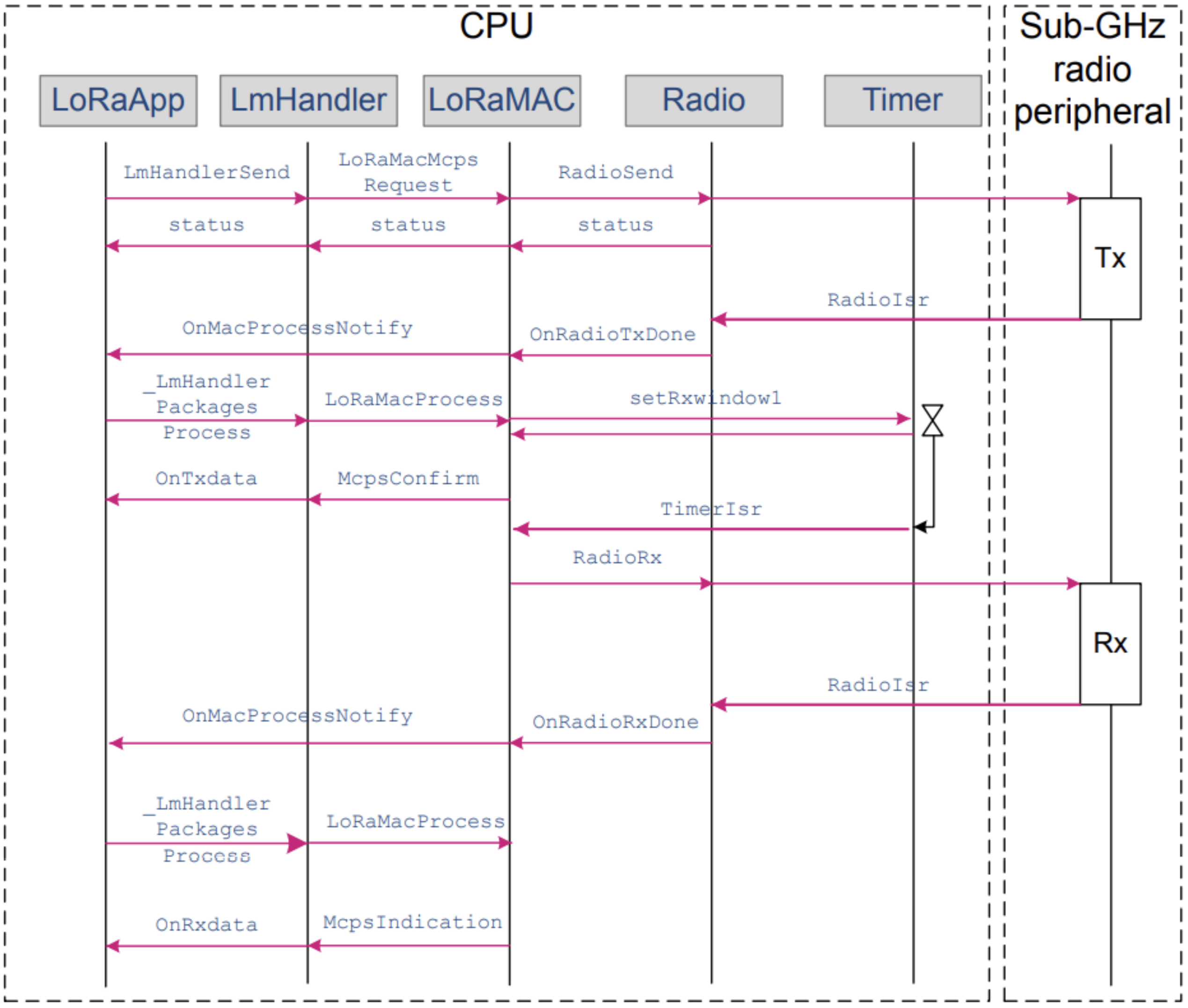
History

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

1. Dynamic view

The MSC (message sequence chart) shown in the figure below depicts a Class A device transmitting an application data and receiving application data from the server

Class A Tx and Rx processing MSC



Once the radio has completed the application data transmission, an asynchronous RadioIRQ wakes up the system. The RadioIsr here calls `txDone` in the handler mode.
All RadioIsr and MAC timer call a `LoRaMacProcessNotify` callback to request the application layer to update the LoRaMAC state and to do further processing when needed.
For instance, at the end of the reception, `rxDone` is called in the ISR (handler), but all the Rx packet processing including decryption must not be processed in the ISR. This case is an example of call sequence. If no data is received into the Rx1 window, then another Rx2 window is launched.

2. LoRaWAN middleware description

2.1 LoRaWAN middleware initialization

The initialization of the LoRaMAC layer is done through the LoRaMacInitialization API, that initializes both the preamble run time of the LoRaMAC layer and the callback primitives of the MCPS and MLME services (see the table below).

LoRaWAN middleware initialization

Function	Description
`LoRaMacStatus_t LoRaMacInitialization (LoRAMacPrimitives_t *primitives, LoRaMacCallback_t *callback, LoRaMacRegion_t region)`	Initializes the LoRaMAC layer module (see Section 6.4 Middleware MAC layer callbacks)

2.2 Middleware MAC layer APIs

The provided APIs follow the definition of "primitive" defined in IEEE802.15.4-2011

The interfacing with the LoRaMAC is made through the request-confirm and the indication-response architecture. The application layer can perform a request that the LoRaMAC layer confirms with a confirm primitive.

Conversely, the LoRaMAC layer notifies an application layer with the indication primitive in case of any event.

The application layer may respond to an indication with the response primitive. Therefore, all the confirm or indication are implemented using callbacks.

The LoRaMAC layer provides the following services:

MCPS services

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

In general, the LoRaMAC layer uses the MCPS services for data transmissions and data receptions

MMLE services

Function	Description
`LoRaMacStatus_t LoRaMacMlmeRequest(MlmeReq_t *mlmeRequest )`	Generates a join request or requests for a link check.

The LoRaMAC layer uses the MLME services to manage the LoRaWAN network.

MIB services

Function	Description
`LoRaMacStatus_t LoRaMacMibSetRequestConfirm (MibRequestConfirm_t *mibSet)`	Sets attributes of the LoRaMAC layer.
`LoRaMacStatus_t LoRaMacMibGetRequestConfirm (MibRequestConfirm_t *mibGet )`	Gets attributes of the LoRaMAC layer.

The MIB stores important runtime information (such as MIB_NETWORK_ACTIVATION or MIB_NET_ID) and holds the configuration of the LoRaMAC layer (for example the MIB_ADR, MIB_APP_KEY).

2.3 Middleware MAC layer callbacks

The LoRaMAC user event functions primitives (also named callbacks) to be implemented by the application are the following:

MCPS primitives

Function	Description
`void (*MacMcpsConfirm ) (McpsConfirm_t *McpsConfirm)`	Response to a McpsRequest
`Void (*MacMcpsIndication) (McpsIndication_t *McpsIndication)`	Notifies the application that a received packet is available.

MLME primitive

Function	Description
`void ( *MacMlmeConfirm ) ( MlmeConfirm_t *MlmeConfirm )`	Manages the LoRaWAN network.

2.4 Middleware MAC layer timers

Delay Rx window

Function	Description
`void OnRxWindow1TimerEvent (void)`	Sets the RxDelay1 (ReceiveDelayX - RADIO_WAKEUP_TIME).
`void OnRxWindow2TimerEvent (void)`	Sets the RxDelay2.

Delay for Tx frame transmission

Function	Description
`void OnTxDelayedTimerEvent (void)`	Sets the timer for Tx frame transmission.

Delay for Rx frame

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

2.5 Main application functions definition

LoRa initialization

Function	Description
`LmHandlerErrorStatus_t LmHandlerInit (LmHandlerCallbacks_t *handlerCallbacks)`	Initialization of the LoRa finite state machine

LoRa configuration

Function	Description
`LmHandlerErrorStatus_t LmHandlerConfigure (LmHandlerParams_t *handlerParams)`	Configuration of all applicative parameters

LoRa End_Node join request entry point

Function	Description
`void LmHandlerJoin (ActivationType_t mode)`	Join request to a network either in OTAA mode orABP mode.

LoRa stop

Function	Description
`void LmHandlerStop (void)`	Stops the LoRa process and waits a new configurationbefore a rejoin action.

LoRa request class

Function	Description
`LmHandlerErrorStatus LmHandlerRequestClass (DeviceClass_t newClass)`	Requests the MAC layer to change LoRaWANclass.

Send an uplink frame

Function	Description
`LmHandlerErrorStatus_t LmHandlerSend (LmHandlerAppData_t *appData, LmHandlerMsgTypes_t isTxConfirmed) TimerTime_t *nextTxIn, bool allowDelayedTx)`	Sends an uplink frame. This frame can be eitheran unconfirmed empty frame or an unconfirmed/ confirmed payload frame.

2.6 Application callbacks

Callbacks in the tables below are used for both End_Node and AT_Slave applications

Current battery level

Function	Description
`uint8_t GetBatteryLevel (void)`	Gets the battery level.

Current temperature

Function	Description
`uint16_t GetTemperature (void)`	Gets the current temperature (in °C) of the device inq7.8 format.

LmHandler process

Function	Description
`void (*OnMacProcess)(void)`	Calls LmHandler Process when a Radio IRQ isreceived.

Join status

Function	Description
`void OnJoinRequest (LmHandlerJoinParams_t *joinParams)`	Notifies the upper layer that a network has beenjoined.

Tx frame done

Function	Description
`void OnTxData (LmHandlerTxParams_t *params)`	Notifies the upper layer that a frame has beentransmitted

Rx frame received

Function	Description
`void OnRxData ( LmHandlerAppData_t *appData, LmHandlerRxParams_t *params)`	Notifies the upper layer that an applicative frame hasbeen received.

2.7 Extended application functions

These callbacks are used for both LoRaWAN_End-Node and LoRaWAN_AT-Slave applications.

Getter/setter functions

Function	Description
`int32_t LmHandlerGetCurrentClass( DeviceClass_T *deviceClass)`	Gets the current LoRaWAN class.
`int32_t LmHandlerGetDevEUI( uint8_t *devEUI)`	Gets the LoRaWAN device EUI.
`int32_t LmHandlerSetDevEUI( uint8_t *devEUI)`	Sets the LoRaWAN device EUI (if OTAA).
`int32_t LmHandlerGetAppEUI( uint8_t *appEUI)`	Gets the LoRaWAN App EUI.
`int32_t LmHandlerSetAppEUI( uint8_t *appEUI)`	Sets the LoRaWAN App EUI.
`int32_t LmHandlerGetAppKey( uint8_t *appKey)`	Gets the LoRaWAN App Key.
`int32_t LmHandlerSetAppKey( uint8_t *appKey)`	Sets the LoRaWAN App Key.
`int32_t LmHandlerGetNetworkID( uint32_t *networkId)`	Gets the LoRaWAN Network ID.
`int32_t LmHandlerSetNetworkID uint32_t networkId)`	Sets the LoRaWAN Network ID.
`int32_t LmHandlerGetDevAddr( uint32_t *devAddr)`	Gets the LoRaWAN deviceaddress.
`int32_t LmHandlerSetDevAddr( uint32_t devAddr)`	Sets the LoRaWAN deviceaddress (if ABP).
`int32_t LmHandlerSetNwkSKey( uint8_t *nwkSKey)`	Sets the LoRaWAN NetworkSession Key.
`int32_t LmHandlerSetAppSKey( uint8_t *appSKey)`	Sets the LoRaWAN Application Session Key.
`int32_t LmHandlerGetActiveRegion( LoRaMacRegion_t *region)`	Gets the active region.
`int32_t LmHandlerSetActiveRegion( LoRaMacRegion_t region)`	Sets the active region.
`int32_t LmHandlerGetAdrEnable( bool *adrEnable)`	Gets the adaptive data rate state.
`int32_t LmHandlerSetAdrEnable( bool adrEnable)`	Sets the adaptive data rate state.
`int32_t LmHandlerGetTxDatarate( int8_t *txDatarate)`	Gets the current Tx data rate.
`int32_t LmHandlerSetTxDatarate( int8_t txDatarate)`	Sets the Tx data rate (if adaptiveDR disabled).
`int32_t LmHandlerGetDutyCycleEnable( bool *dutyCycleEnable)`	Gets the current Tx duty cyclestate.
`int32_t LmHandlerSetDutyCycleEnable( bool dutyCycleEnable)`	Sets the Tx duty cycle state.
`int32_t LmHandlerGetRX2Params( RxChannelParams_t *rxParams)`	Gets the current Rx2 data rate and frequency conf.
`int32_t LmHandlerSetRX2Params( RxChannelParams_t *rxParams)`	Sets the Rx2 data rate andfrequency conf.
`int32_t LmHandlerGetTxPower( int8_t *txPower)`	Gets the current Tx power value.
`int32_t LmHandlerSetTxPower( int8_t txPower)`	Sets the Tx power value.
`int32_t LmHandlerGetRx1Delay( uint32_t *rxDelay)`	Gets the current Rx1 delay (afterTx window).
`int32_t LmHandlerSetRx1Delay( uint32_t rxDelay)`	Sets the Rx1 delay (after Txwindow).

Function	Description
`int32_t LmHandlerGetRx2Delay( uint32_t *rxDelay)`	Gets the current Rx2 delay (afterTx window).
`int32_t LmHandlerSetRx2Delay( uint32_t rxDelay)`	Sets the Rx2 delay (after Txwindow).
`int32_t LmHandlerGetJoinRx1Delay( uint32_t *rxDelay)`	Gets the current Join Rx1 delay(after Tx window).
`int32_t LmHandlerSetJoinRx1Delay( uint32_t rxDelay)`	Sets the Join Rx1 delay (after Tx window).
`int32_t LmHandlerGetJoinRx2Delay( uint32_t *rxDelay)`	Get the current Join Rx2 delay(after Tx window)
`int32_t LmHandlerSetJoinRx2Delay( uint32_t rxDelay)`	Sets the Join Rx2 delay (after Tx window).
`int32_t LmHandlerGetPingPeriodicity( uint8_t *pingPeriodicity)`	Gets the current Rx Ping Slot periodicity (If `LORAMAC_CLASSB_ENABLED`)
`int32_t LmHandlerSetPingPeriodicity( uint8_t pingPeriodicity)`	Sets the Rx Ping Slot periodicity (If `LORAMAC_CLASSB_ENABLED`)
`int32_t LmHandlerGetBeaconState( BeaconState_t *beaconState)`	Gets the beacon state (If `LORAMAC_CLASSB_ENABLED`)

3. 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.

3.1 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.

3.2 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.

3.3 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.

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

3.4 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)

3.5 Trace

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

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).