Arduino LoRaWAN MAC in C (LMIC)

Version 4.1.0
2021-10-10

LMIC Product Information

The LMIC library was originally developed and marketed by the IBM Zurich Research Laboratory (IBM Research GmbH), 8803 Rüschlikon, Switzerland. For additional information please contact: lrsc@zurich.ibm.com. This document was taken from V1.6, from July 2016.

The library was adapted for Arduino by Mathijs Kooijman and Thomas Telkamp. This version of the document describes the version being maintained by MCCI Corporation at https://github.com/mcci-catena/arduino-lmic/.

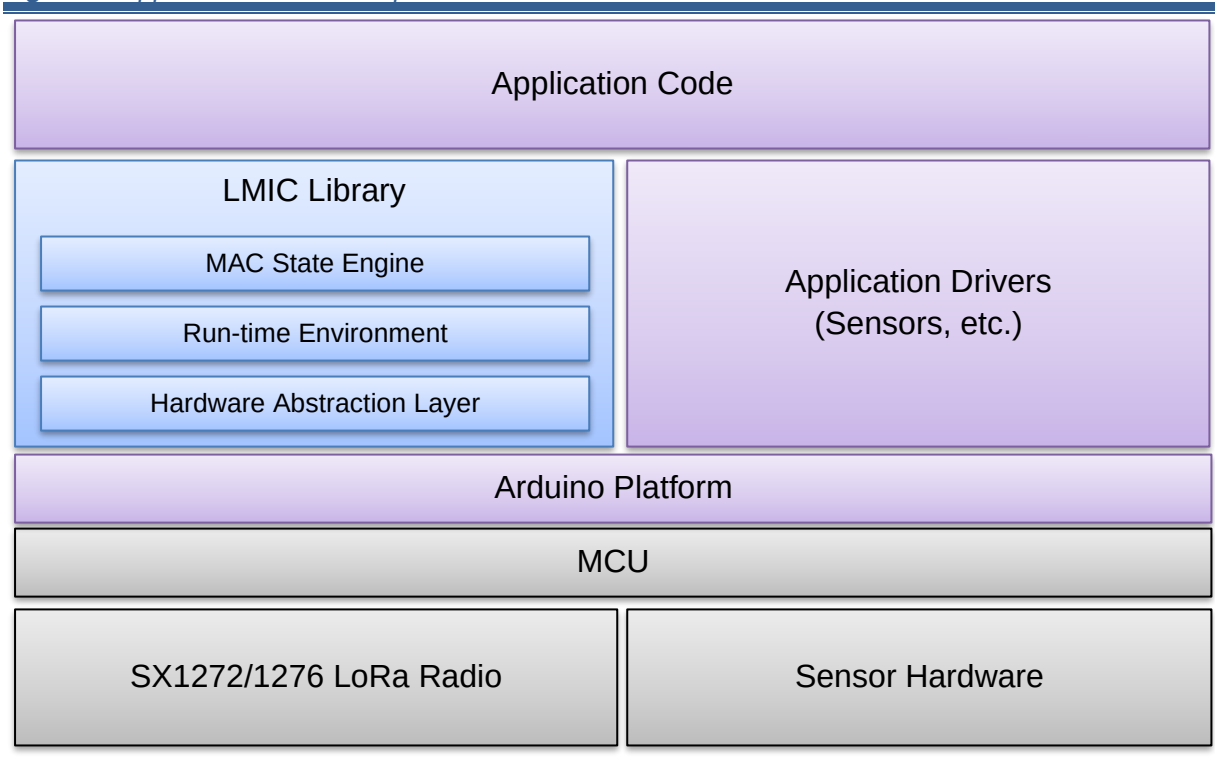
Table of Contents

1. Introduction

The Arduino IBM LoRaWAN C-library (LMIC) is a portable implementation of the LoRaWAN™ 1.0.3 end-device specification in the C programming language. ("LMIC" stands for "LoRaWAN MAC in C"). It supports the EU-868, US-915, AU-915, AS-923, KR-920 and IN-866 variants of the specification and it can handle class A and class B devices. The library takes care of all logical MAC states and timing constraints and drives the SEMTECH SX1272 or SX1276 radio. This way, applications are free to perform other tasks and the protocol compliance is guaranteed by the library. In order to ensure compliance with the specification and associated regulations, the state engine has been tested and verified using a logic simulation environment. The library has been carefully engineered to precisely satisfy the timing constraints of the MAC protocol and to even consider possible clock drifts in the timing computations. Applications can access and configure all functionality via a simple event-based programming model and do not have to deal with platform-specific details like interrupt handlers. By using a thin hardware abstraction layer (HAL), the library can be easily ported to new hardware platforms. An Arduino HAL is provided which allows for easy integration with most Arduino variants. Eight-bit AVR platforms are supported as well as 32-bit platforms.

In addition to the provided LMIC library, a real-world application also needs drivers for the sensors or other hardware it desires to control. These application drivers are outside the scope of this document and are not part of this library.

Figure 1. *Application device components*



High-level view of all application device components.

1.1 LoRaWAN Versions and Features Supported

The LMIC library supports Class A operation as described by the LoRaWAN specification V1.0.3 and V1.0.2. It does not support V1.1.

Class B support code is provided, but is not tested.

LoRaWAN 1.0.3 Class A multicast downlinks are not supported.

Class C operation is not supported.

The library has been tested for compliance with the following LoRaWAN 1.0.2 test specifications as implemented by RedwoodComm in their RWC5020A tester, using firmware version 1.170. The MCCI Catena 4610 was used as the reference device.

- EU868 V1.5 (excluding optional data rates)
- US915 V1.3 (excluding optional data rates; testing was done using channels 0~7 and 64). The TxPower test fails because the LMIC complies with LoRaWAN V1.0.3. The TxPower test uses a value that is defined for V1.0.3, but not for V1.0.2, and expects the device to reject the value.
- AS923 V1.1 (excluding optional data rates)
- KR920 V1.2
- IN865 V1.0

All tests were performed using a tethered connection.

1.2 Class A and Class B Support

The Arduino LMIC library can be configured to support LoRaWAN Class A and Class B operation. A Class A device receives only at fixed times after transmitting a message. This allows for low power operation, but means that downlink latency is controlled by how often the device transmits. A Class B device synchronizes to beacons transmitted by the network, and listens for messages at certain intervals (“ping slots”) during the interval between beacons.

Devices (and the LMIC library) start out as Class A devices, and switch to Class B based on requests from the application layer of the device.

This document uses the term “pinging” to mean that the LMIC is operating in Class B and also listening for downlink during ping slots. If a device is pinging, then the LMIC must also be tracking the beacon. It is possible to track the beacon (perhaps for time synchronization purposes) without enabling pinging.

Since many devices and networks only support Class A operation, the library can be configured at compile time to omit support for tracking and pinging. It is possible to omit support for pinging without omitting support for tracking, but this is not a tested configuration.

2. Programming Model and API

The LMIC library can be accessed via a set of API functions, run-time functions, callback functions, and a global `LMIC` data structure. The interface is defined in a single header file "`lmic.h`" which all applications should include.

```
#include "lmic.h"
```

The library version follows Semantic Versioning 2.0.0 (https://semver.org/). A single symbol represents the version. Bits 31..24 represent the major version, bits 23..15 the minor version, and bits 15..8 represent the patch. Bits 7..0 are used for representing the pre-release, called "LOCAL" for historical reasons.

A function-like macro, `ARDUINO_LMIC_VERSION_CALC()`, is used to construct version numbers.

```
#define ARDUINO_LMIC_VERSION     ARDUINO_LMIC_VERSION_CALC(2, 3, 2, 0)
```

For convenience, the library supplies function-like macros `ARDUINO_LMIC_VERSION_GET_MAJOR()`, `ARDUINO_LMIC_VERSION_GET_MINOR()`, `ARDUINO_LMIC_VERSION_GET_PATCH()`, and `ARDUINO_LMIC_VERSION_GET_LOCAL()`, which extract the appropriate field from a version number.

Version numbers are represented naively – the four fields are simply packed together into a 32-bit word. This makes them easy to display, but hard to compare. Pre-releases are numbered higher than releases, but compare less. Several macros are provided to make it easy to compare versions numbers. `ARDUINO_LMIC_VERSION_TO_ORDINAL()` converts a version number computed by `ARDUINO_LMIC_VERSION_CALC()` into an integer that can be compared using the normal C comparison operators. `ARDUINO_LMIC_VERSION_COMPARE_LT(v1, v2)` compares two version numbers and returns non-zero if `v1` is less than `v2` (after converting both to ordinals). `ARDUINO_LMIC_VERSION_COMPARE_LE(v1, v2)`, `ARDUINO_LMIC_VERSION_COMPARE_GT(v1, v2)`, and `ARDUINO_LMIC_VERSION_COMPARE_GE(v1, v2)` test for less-than-or-equal, greater-than, or greater-than-or-equal relationships.

To identify the original version of the IBM LMIC library two constants are defined in this header file.

```
#define LMIC_VERSION_MAJOR 1
#define LMIC_VERSION_MINOR 6
```

These version strings identify the base version of the library, and will not change.

2.1 Programming Model

The LMIC library offers a simple event-based programming model where all protocol events are dispatched to the application's `onEvent()` callback function (see 2.3.4). In order to free the application of details like timings or interrupts, the library has a built-in run-time environment to take care of timer queues and job management.

2.1.1 Application jobs

In this model all application code is run in so-called jobs which are executed on the main thread by the run-time scheduler function `os_runloop()` (see 2.2.6). These application jobs are coded as normal C functions and can be managed using the run-time functions described in section 2.1.3. For the job management an additional per job control struct `osjob_t` is required which identifies the job and stores

context information. **Jobs must not be long-running** in order to ensure seamless operation**!** They should only update state and schedule actions, which will trigger new job or event callbacks.

2.1.2 Main event loop

All an application must do is to initialize the run-time environment using the `os_init()` or `os_init_ex()` function and then periodically call the job scheduler function `os_runloop_once()`. In order to bootstrap protocol actions and generate events, an initial job needs to be set up. Therefore, a startup job is scheduled using the `os_setCallback()` function.

```
osjob_t initjob;

void setup () {
    // initialize run-time env
    os_init();
    // setup initial job
    os_setCallback(&initjob, initfunc);
}

void loop () {
    // execute scheduled jobs and events
    os_runloop_once();
}
```

The startup code shown in the `initfunc()` function below initializes the MAC and starts joining the network.

```
// initial job
static void initfunc (osjob_t* j) {
    // reset MAC state
    LMIC_reset();
    // start joining
    LMIC_startJoining();
    // init done - onEvent() callback will be invoked...
}
```

The `initfunc()` function will return immediately, and the `onEvent()` callback function will be invoked by the scheduler later on for the events `EV_JOINING`, `EV_JOINED` or `EV_JOIN_FAILED`.

2.1.3 OS time

The LMIC uses values of the type `ostime_t` to represent time in ticks. The rate of these ticks defaults to 32768 ticks per second, but may be configured at compile time to any value between 10000 ticks per second and 64516 ticks per second.

In general, one tick is not an integral number of microseconds or milliseconds. Convenience functions are provided for switching back and forth.

```
typedef int32_t ostime_t;
```

Note that this is a *signed integer* value; care must be taken when computing differences to avoid being fooled by overflow. OS time starts at zero, and increments uniformly to `INT32_MAX`; it then wraps to `INT32_MIN` and increments uniformly up to zero, and repeats. Rather than comparing two `ostime_t` values, recommended practice is to subtract them and see if the result is positive or negative.

2.2 Run-time Functions

The run-time functions mentioned before are used to control the run-time environment. This includes initialization, scheduling and execution of the run-time jobs.

2.2.1 `void` **`os_init`** `()`

Initialize the operating system by calling `os_init_ex(NULL)`.

2.2.2 `void` **`os_init_ex`** `(const void * pHalData)`

To facilitate use of this library on multiple platforms, the os_init_ex() routine takes an arbitrary pointer to platform data. The Arduino LMIC default HAL implementation expects this pointer to be a reference to a C++ `struct lmic_pinmap` object. See README.md for more information.

2.2.3 `void` **`os_setCallback`** `(osjob_t*` *`job`*`, osjobcb_t` *`cb`*`)`

Prepare an immediately runnable job. This function can be called at any time, including from interrupt handler contexts (e.g. if a new sensor value has become available).

2.2.4 `void` **`os_setTimedCallback`** `(osjob_t*` *`job`*`, ostime_t time, osjobcb_t` *`cb`*`)`

Schedule a timed job to run at the given timestamp (absolute system time). This function can be called at any time, including from interrupt handler contexts.

2.2.5 `void` **`os_clearCallback`** `(osjob_t*` *`job`*`)`

Cancel a run-time job. A previously scheduled run-time job is removed from timer and run queues. The job is identified by the address of the job struct. The function has no effect if the specified job is not yet scheduled.

2.2.6 `void` **`os_runloop`** `()`

Execute run-time jobs from the timer and from the run queues. This function is the main action dispatcher. It does not return and must be run on the main thread. This routine is normally not used in Arduino environments, as it disables the normal calling of the Arduino `loop()` function.

2.2.7 `void` **`os_runloop_once`** `()`

Execute run-time jobs from the timer and from the run queues. This function is just like os_runloop(), except that it returns after dispatching the first available job.

2.2.8 `ostime_t` **`os_getTime`** `()`

Query absolute system time (in ticks).

2.2.9 `ostime_t` **`us2osticks`**`(s4_t us)`

Returns the ticks corresponding to the integer value `us`. This may be a function-like macro, so `us` may be evaluated more than once. Any fractional part of the calculation is discarded.

2.2.10 ostime_t us2osticksCeil(s4_t us)

Returns the ticks corresponding to the integer value `us`. This may be a function-like macro, so `us` may be evaluated more than once. If the fractional part of the calculation is non-zero, the result is increased towards positive infinity.

2.2.11 ostime_t us2osticksRound(s4_t us)

Returns the ticks corresponding to the integer value `us`. This may be a function-like macro, so `us` may be evaluated more than once. The result is rounded to the nearest tick.

2.2.12 ostime_t ms2osticks(s4_t ms)

Returns the ticks corresponding to the integer millisecond value `ms`. This may be a function-like macro, so `ms` may be evaluated more than once. If the fractional part of the calculation is non-zero, the result is increased towards positive infinity.

2.2.13 ostime_t ms2osticksCeil(s4_t ms)

Returns the ticks corresponding to the integer millisecond value `ms`. This may be a function-like macro, so `ms` may be evaluated more than once.

2.2.14 ostime_t ms2osticksRound(s4_t ms)

Returns the ticks corresponding to the integer millisecond value `ms`. This may be a function-like macro, so `ms` may be evaluated more than once. The result is rounded to the nearest tick.

2.2.15 ostime_t sec2osticks(s4_t sec)

Returns the ticks corresponding to the integer second value `sec`. This may be a function-like macro, so `sec` may be evaluated more than once.

2.2.16 S4_t osticks2ms(ostime_t os)

Returns the milliseconds corresponding to the tick value `os`. This may be a function-like macro, so `os` may be evaluated more than once.

2.2.17 S4_t osticks2us(ostime_t os)

Returns the microseconds corresponding to the tick value `os`. This may be a function-like macro, so `os` may be evaluated more than once.

2.3 Application callbacks

The LMIC library requires the application to implement a few callback functions. These functions are called by the state engine to query application-specific information and to deliver state events to the application.

Upcalls by name from the LMIC to application code are deprecated and will be removed in future versions of the LMIC. The provisioning APIs (os_getDevEui, os_getDevKey and os_getArtEui) will be replaced by secure element APIs in version 4. The onEvent API will be disabled by default in version 4, and removed in version 5.

2.3.1 void os_getDevEui (u1_t* *buf*)

The implementation of this callback function has to provide the device EUI and copy it to the given buffer. The device EUI is 8 bytes in length and is stored in little-endian format, that is, least-significant-byte-first (LSBF).

2.3.2 void os_getDevKey (u1_t* *buf*)

The implementation of this callback function has to provide the device-specific cryptographic application key and copy it to the given buffer. The device-specific application key is a 128-bit AES key (16 bytes in length).

2.3.3 void os_getArtEui (u1_t* *buf*)

The implementation of this callback function has to provide the application EUI and copy it to the given buffer. The application EUI is 8 bytes in length and is stored in little-endian format, that is, least-significant-byte-first (LSBF).

2.3.4 void onEvent (ev_t *ev*)

This function, if provided, is called to report LMIC events (such as transmission complete or downlink message received). It is a legacy function. In V3, if this name conflicts with a name in your application, you can disable the use of this name by putting the following in your configuration file:

```
#define LMIC_ENABLE_onEvent 0
```

Upcalls by name from the LMIC to application code are deprecated and will be removed in future versions of the LMIC. A suitable substitute is available. See the discussion of `LMIC_registerEventCb()`, below.

2.4 The LMIC Struct

Instead of passing numerous parameters back and forth between API and callback functions, information about the protocol state can be accessed via a global `LMIC` structure as shown below. All fields besides the ones explicitly mentioned below are read-only and should not be modified.

```
struct lmic_t {
    u1_t        frame[MAX_LEN_FRAME];
    u1_t        dataLen;    // 0 no data or zero length data, >0 byte count of data
    u1_t        dataBeg;    // 0 or start of data (dataBeg-1 is port)

    u1_t        txCnt;
    u1_t        txrxFlags;  // transaction flags (TX-RX combo)

    u1_t        pendTxPort;
    u1_t        pendTxConf; // confirmed data
    u1_t        pendTxLen;
    u1_t        pendTxData[MAX_LEN_PAYLOAD];

    u1_t        bcnChnl;
    u1_t        bcnRxsyms;
    ostime_t    bcnRxtime;
    bcninfo_t   bcninfo;    // Last received beacon info
```

```
    …
    …
};
```

This document does not describe the full struct in detail since most of the fields of the `LMIC` struct are used internally only. The most important fields to examine on reception (event `EV_RXCOMPLETE` or `EV_TXCOMPLETE`) are the `txrxFlags` for status information and `frame[]` and `dataLen` / `dataBeg` for the received application payload data. For data transmission the most important fields are `pendTxData[]`, `pendTxLen`, `pendTxPort` and `pendTxConf`, which are used as input to the `LMIC_setTxData()` API function (see 2.5.13).

For the `EV_RXCOMPLETE` and `EV_TXCOMPLETE` events, the `txrxFlags` field should be evaluated. The following flags are defined:

- `TXRX_ACK`: confirmed UP frame was acked (mutually exclusive with `TXRX_NACK`)
- `TXRX_NACK`: confirmed UP frame was not acked (mutually exclusive with `TXRX_ACK`)
- `TXRX_PORT`: a port byte is contained in the received frame at offset `LMIC.dataBeg – 1`.
- `TXRX_NOPORT`: no port byte is available.
- `TXRX_DNW1`: received in first DOWN slot (mutually exclusive with `TXRX_DNW2`)
- `TXRX_DNW2`: received in second DOWN slot (mutually exclusive with `TXRX_DNW1`)
- `TXRX_PING`: received in a scheduled RX slot
- `TXRX_LENERR`: the transmitted message was abandoned because it was longer than the established data rate.

For the `EV_TXCOMPLETE` event the fields have the following values:

Received frame	**LMIC.txrxFlags**								**LMIC.dataLen**	**LMIC.dataBeg**
	ACK	**NACK**	**PORT**	**NOPORT**	**DNW1**	**DNW2**	**PING**	**LENERR**		
nothing	0	0	0	1	0	0	0	0	0	0
empty frame	x	x	0	1	x	x	0	0	0	x
port only	x	x	1	0	x	x	0	0	0	x
port+payload	x	x	1	0	x	x	0	0	x	X
No message received, transmit message too long	0	0	0	1	x	x	x	1	0	0

For the `EV_RXCOMPLETE` event the fields have the following values:

Received frame	**LMIC.txrxFlags**							**LMIC.dataLen**	**LMIC.dataBeg**
	ACK	**NACK**	**PORT**	**NOPORT**	**DNW1**	**DNW2**	**PING**		
empty frame	0	0	0	1	0	0	1	0	x
port only	0	0	1	0	0	0	1	0	x
port+payload	0	0	1	0	0	0	1	x	x

2.5 API Functions

The LMIC library offers a set of API functions to control the MAC state and to trigger protocol actions.

2.5.1 void LMIC_reset ()

Reset the MAC state. Session and pending data transfers will be discarded.

2.5.2 bit_t LMIC_startJoining ()

Immediately start joining the network. Will be called implicitly by other API functions if no session has been established yet. The events `EV_JOINING` and `EV_JOINED` or `EV_JOIN_FAILED` will be generated.

2.5.3 void LMIC_tryRejoin ()

Check if other networks are around which can be joined. The session to the current network is kept if no new network is found. The events `EV_JOINED` or `EV_REJOIN_FAILED` will be generated.

2.5.4 void LMIC_setSession (u4_t *netid*, devaddr_t *devaddr*, u1_t* *nwkKey*, u1_t* *artKey*)

Set static session parameters. Instead of dynamically establishing a session by joining the network, precomputed session parameters can be provided. To resume a session with precomputed parameters, the frame sequence counters (`LMIC.seqnoUp` and `LMIC.seqnoDn`) must be restored to their latest values.

2.5.5 bit_t LMIC_setupBand (u1_t *bandidx*, s1_t txpow, u2_t *txcap*)

Initialize a specific duty-cycle band with the specified transmit power and duty cycle ($1/txcap$) properties. Returns non-zero for success, zero for failure. Some regions don't allow you to define duty cycle bands, and in that case this function will always return zero.

The argument *bandidx* selects the band; it must be in the range $0 \le bandidx <$ `MAX_BANDS`. For EU-like regions, the following band names are defined: `BAND_MILLI`, intended to be used for channels with a duty cycle of 0.1%; `BAND_CENTI`, intended to be used by channels with a duty cycle of 1%; and `BAND_DECI`, intended to be used by channels with a duty cycle of 10%. A fourth band, `BAND_AUX`, is available for use by custom code.

2.5.6 bit_t LMIC_setupChannel (u1_t *channel*, u4_t *freq*, u2_t *drmap*, s1_t *band*)

Create a new channel, or modify an existing channel. If *freq* is non-zero, the channel is enabled; otherwise it's disabled. The argument *drmap* is a bitmap indicating which data rates (0 through 15) are enabled for this channel. Band, if in the range 0..3, assigns the channel to the specified duty-cycle band – the meaning of this is region-specific. If set to -1, then the appropriate duty-cycle band is selected based on the value of *freq*.

The result is non-zero if the channel was successfully set up; zero otherwise.

For any region, there are some channels that cannot be modified (the "default channels"). You can't disable or change these, but it's not an error to try to set a default channel to its default value.

Some regions (such as US915 or AU915) do not allow any channels to be set up.

2.5.7 void LMIC_disableChannel (u1_t *channel*)

Disable specified channel. Default channels cannot be disabled.

2.5.8 u1_t LMIC_queryNumDefaultChannels ()

Return the number of default channels defined by this region. In the EU868, KR920 and IN866 regions, the result is 3; in the AS923 region, the result is 2; in the US and AU regions, the result is 72.

2.5.9 void LMIC_setAdrMode (bit_t *enabled*)

Enable or disable data rate adaptation. Should be turned off if the device is mobile.

2.5.10 void LMIC_setLinkCheckMode (bit_t *enabled*)

Enable/disable link check validation. Link check mode is enabled by default and is used to periodically verify network connectivity. Must be called only if a session is established.

2.5.11 void LMIC_setDrTxpow (dr_t *dr*, s1_t *txpow*)

Set data rate and transmit power. Should only be used if data rate adaptation is disabled.

2.5.12 bit_t LMIC_queryTxReady ()

Return non-zero if the LMIC is ready to accept transmitted data, zero if an attempt to transmit will be rejected.

2.5.13 void LMIC_setTxData ()

Prepare upstream data transmission at the next possible time. It is assumed, that `pendTxData`, `pendTxLen`, `pendTxPort` and `pendTxConf` have already been set. Data of length `LMIC.pendTxLen` from the array `LMIC.pendTxData[]` will be sent to port `LMIC.pendTxPort`. If `LMIC.pendTxConf` is true, confirmation by the server will be requested. The event `EV_TXCOMPLETE` will be generated when the transaction is complete, i.e. after the data has been sent and eventual down data or a requested acknowledgement has been received.

If data rate adaptation is enabled, this function will check whether the message being transmitted is feasible with the current data rate. If not, the data rate will be increased, if possible. This is to provide maximum software compatibility with older applications that do not comprehend the side-effects of rate changes on longer messages. However, it is not guaranteed always to be possible to send a message of a given size, due to regional plan restrictions and current network operating requirements (as provided by downlinks). Prior to V3.2, the LMIC would transmit anyway; as of V3.2, it will report an error. Thus, applications may need to be modified.

Because of the numerous post-conditions that must be checked after calling **`LMIC_setTxData`**, we strongly recommend using **`LMIC_setTxData2()`** instead.

2.5.14 void LMIC_setTxData_strict ()

Prepare upstream data transmission at the next possible time. The caller must first initialize `LMIC.pendTxData[]`, `LMIC.pendTxLen`, `LMIC.pendTxPort` and `LMIC.pendTxConf`. Data of length `LMIC.pendTxLen` from the array `LMIC.pendTxData[]` will be sent to port `LMIC.pendTxPort`. If `LMIC.pendTxConf` is true, confirmation by the network server will be requested. The event `EV_TXCOMPLETE` will be generated when the transaction is complete, i.e. after the data has been sent and eventual down data or a requested acknowledgement has been received.

Unlike **`LMIC_setTxData()`**, this API will not try to adjust data rate if the message is too long for the current data rate. See 2.5.13 for a complete discussion.

This API is preferred for new applications, for two reasons. First, automatically increasing data rates is arguably not compliant. Second, it isn't always possible. It's better for applications to be designed to control operation in the face of slow data rates themselves.

Because of the numerous post-conditions that must be checked after calling **`LMIC_setTxData_strict()`**, we strongly recommend using **`LMIC_setTxData2_strict()`** or **`LMIC_sendWithCallback_strict()`** instead.

2.5.15 `lmic_tx_error_t` **`LMIC_setTxData2`** `(u1_t` *`port`*`, xref2u1_t` *`data`*`, u1_t` *`dlen`*`, u1_t` *`confirmed`*`)`

Prepare upstream data transmission at the next possible time. Convenience function for `LMIC_setTxData()`. If *`data`* is `NULL`, the data in `LMIC.pendTxData[]` will be used.

For compatibility with existing applications, if data rate adaptation is enabled, this function will check whether the message being transmitted is feasible with the current data rate. If not, the data rate will be increased, if possible. See 2.5.13 for a complete discussion.

The result type `lmic_tx_error_t` is a synonym for `int`. The following values may be returned.

Name	Value	Description
`LMIC_ERROR_SUCCESS`	0	No error occurred, `EV_TXCOMPLETE` will be posted.
`LMIC_ERROR_TX_BUSY`	-1	The LMIC was busy sending another message. This message was not sent. `EV_TXCOMPLETE` will not be posted for this message.
`LMIC_ERROR_TX_TOO_LARGE`	-2	The queued message is too large for the any data rate for this region. This message was not sent. `EV_TXCOMPLETE` will not be posted for this message.
`LMIC_ERROR_TX_NOT_FEASIBLE`	-3	The queued message is not feasible for any enabled data rate. This message was not sent. `EV_TXCOMPLETE` will not be posted for this message.
`LMIC_ERROR_TX_FAILED`	-4	The queued message failed for some other reason than data length, during the initial call to the LMIC to transmit it. This message was not sent. `EV_TXCOMPLETE` will not be posted for this message.

2.5.16 `lmic_tx_error_t` **`LMIC_setTxData2_strict`** `(u1_t` *`port`*`, xref2u1_t` *`data`*`, u1_t` *`dlen`*`, u1_t` *`confirmed`*`)`

Prepare upstream data transmission at the next possible time. This function is identical to LMIC_setTxData2(), except that the current data rate will never be changed. Thus, the error return `LMIC_ERROR_TX_NOT_FEASIBLE` has a slightly different meaning. See 2.5.13 for a complete discussion.

Name	Value	Description
`LMIC_ERROR_TX_NOT_FEASIBLE`	-3	The queued message is not feasible for the current data rate. This message was not sent. `EV_TXCOMPLETE` will not be posted for this message.

2.5.17 lmic_tx_error_t LMIC_sendWithCallback (u1_t *port*, u1_t *data, u1_t *dlen*, u1_t *confirmed*, lmic_txmessage_cb_t *pCb, void *pUserData)

Prepare upstream data transmission at the next possible time, and call a specified function when the transmission completes. Convenience function for `LMIC_setTxData()`. If *`data`* is `NULL`, the data in `LMIC.pendTxData[]` will be used. The arguments *`dlen`*, *`confirmed`*, and *`port`* have the same meaning as they do for `LMIC_setTxData2()`.

If the initial call succeeds, the callback will be called, and the event EV_TXCOMPLETE will also be issued. See section 2.5.21 for a complete discussion.

For compatibility with existing applications, if data rate adaptation is enabled, this function will check whether the message being transmitted is feasible with the current data rate. If not, the data rate will be increased, if possible. See 2.5.13 for a complete discussion.

The callback function has the prototype:

```
typedef void (lmic_txmessage_cb_t)(void *pUserData, int fSuccess);
```

It is called by the LMIC when transmission of the message is completed. *`fSuccess`* will be non-zero for a transmission that's judged to be successful, or zero if the transmission is judged to have failed. *`pUserData`* is set to the value passed in the corresponding `LMIC_sendWithCallback()` invocation.

See section 2.5.15 for a list of the possible result codes. In all cases, if the result is other than `LMIC_ERROR_SUCCESS`, the user's callback function has not been called, and will not be called, and `EV_TXCOMPLETE` will not be reported.

2.5.18 lmic_tx_error_t LMIC_sendWithCallback_strict (u1_t *port*, u1_t *data, u1_t *dlen*, u1_t *confirmed*, lmic_txmessage_cb_t *pCb, void *pUserData)

Prepare upstream data transmission at the next possible time, and call a specified function when the transmission completes. This function is identical to `LMIC_sendWithCallback()`, except that it will not attempt to change the current data rate if the current transmission is not feasible. (See 2.5.13 for a complete discussion.) Thus, the error return `LMIC_ERROR_TX_NOT_FEASIBLE` has a slightly different meaning, as described in section 2.5.16.

2.5.19 void LMIC_clrTxData ()

Remove data previously prepared for upstream transmission. If `LMIC_sendWithCallback()` or `LMIC_sendWithCallback_strict()` operations are pending, the callback function will be called with `fSuccess` set to zero. If transmit messages are pending, the event `EV_TXCOMPLETE` will be reported.

2.5.20 void LMIC_registerRxMessageCb (lmic_rxmessage_cb_t *pRxMessageCb, void *pUserData)

This function registers a callback function to be called when the LMIC detects the reception of a message.

The callback function has the prototype:

```
typedef void (lmic_rxmessage_cb_t)(
        void *pUserData, u1_t port, const u1_t *pMessage, size_t nMessage
        );
```

This function is called from within the LMIC; it should avoid calling LMIC APIs, and avoid time-critical operations.

The argument *pUserData* is set to the value passed to `LMIC_registerRxMessageCb()`. The argument *port* is set to the port number of the message. If zero, then a MAC message was received, and *nMessage* will be zero. The argument *pMessage* is set to point to the first byte of the message, in the LMIC's internal buffers, and *nMessage* is set to the number of bytes of data.

The following conditions can be distinguished.

port	nMessage	Meaning
0	≠ 0	A MAC message was received on port zero. It has already been processed, but it's delivered to the client for inspection. The LMIC will discard any messages with piggy-backed MAC data targeting port 0.
0	0	An empty payload (no port, no frame) was received. Most likely there is piggybacked MAC data. See below.
≠ 0	0	An empty payload was received for a specific port. It's up to the application to interpret this.
≠ 0	≠ 0	A non-empty payload was received for a specific port. It's up to the application to interpret this.

If *port* is zero and *nMessage* is zero, piggybacked MAC data can be detected and inspected by checking the value of `LMIC.dataBeg`. If non-zero, there are `LMIC.dataBeg` bytes of piggybacked data, and the data can be found at `LMIC.frame[0]` through `LMIC.frame[LMIC.dataBeg-1]`.

If port is non-zero, piggybacked MAC data can also be checked using the value of `LMIC.dataBeg`. If greater than 1, there are `(LMIC.dataBeg-1)` bytes of piggybacked data, and the data can be found at `LMIC.frame[0]` through `LMIC.frame[LMIC.dataBeg-2]`.

Be aware that if you are also using event callbacks, events will also be reported to the event listening functions. See section 2.5.21 for a complete discussion.

2.5.21 `void LMIC_registerEventCb (lmic_event_cb_t *pEventCb, void *pUserData)`

This function registers a callback function to be called when the LMIC detects an event.

The callback function has the prototype:

```
typedef void (lmic_event_cb_t)(void *pUserData, ev_t ev);
```

The argument *ev* is set to a numeric code indicating the type of event that has occurred. The argument *pUserData* is set according to the value passed to `LMIC_registerEventCb()`.

The implementation of this callback function may react on certain events and trigger new actions based on the event and the `LMIC` state. Typically, an implementation processes the events it is interested in and schedules further protocol actions using the LMIC API. The following events will be reported:

- **`EV_JOINING`**
  The node has started joining the network.
- **`EV_JOINED`**
  The node has successfully joined the network and is now ready for data exchanges.

- **EV_JOIN_FAILED**
  The node could not join the network (after retrying).
- **EV_REJOIN_FAILED**
  The node did not join a new network but might still be connected to the old network. This feature (trying to join a new network while connected to an old one) is deprecated and will be removed in future versions.
- **EV_TXCOMPLETE**
  The data prepared via `LMIC_setTxData()` has been sent, and the receive window for downstream data is complete. If confirmation was requested, the acknowledgement has been received. When handling this event, the code can also check for data reception. See 2.5.21.1 for details. If using `LMIC_registerRxMessageCb()`, it's best to leave that to the LMIC, which will call the client's receive-message function as appropriate.
- **EV_RXCOMPLETE**
  A downlink has been received, either in a Class A RX slot, in a Class B ping slot, or (in the future) in a Class C receive window. The code should check the received data. See 2.5.21.1 for details. If using `LMIC_sendWithCallback()` and `LMIC_registerRxMessageCb()`, ignore `EV_RXCOMPLETE` in your event processing function.
- **EV_SCAN_TIMEOUT**
  After a call to `LMIC_enableTracking()` no beacon was received within the beacon interval. Tracking needs to be restarted.
- **EV_BEACON_FOUND**
  After a call to `LMIC_enableTracking()` the first beacon has been received within the beacon interval.
- **EV_BEACON_TRACKED**
  The next beacon has been received at the expected time.
- **EV_BEACON_MISSED**
  No beacon was received at the expected time.
- **EV_LOST_TSYNC**
  Beacon was missed repeatedly, and time synchronization has been lost. Tracking or pinging needs to be restarted.
- **EV_RESET**
  Session reset due to rollover of sequence counters. Network will be rejoined automatically to acquire new session.
- **EV_LINK_DEAD**
  No confirmation has been received from the network server for an extended period of time. Transmissions are still possible, but their reception is uncertain.
- **EV_LINK_ALIVE**
  The link was dead, but now is alive again.
- **EV_SCAN_FOUND**
  This event is reserved for future use, and is never reported.
- **EV_TXSTART**
  This event is reported just before telling the radio driver to start transmission.
- **EV_TXCANCELED**
  A pending transmission was canceled, either because of a request to cancel, or as a side effect of an API request, or as a side-effect of a change at the MAC layer (such as frame count overflow).

- **EV_RXSTART**
  The LMIC is about to open a receive window. It's very important that the event processing routine do as little work as possible – no more than one millisecond of real time should be consumed, otherwise downlinks may not work properly. Don't print anything out while processing this event; save data to be printed later. This event is not sent to the `onEvent()` subroutine, section 2.3.4; it's only sent to event handlers registered via `LMIC_registerEventCb()`, section 2.5.21.
- **EV_JOIN_TXCOMPLETE**
  This event indicates the end of a transmission cycle for JOINs. It indicates that both receive windows of the join have been processed without receiving a JoinAccept message from the network.

Information about the LMIC state for specific events can be obtained from the global `LMIC` structure described in section 2.4.

Events functions and the transmit/receive call back functions are orthogonal. Multiple routines may be called for a given event.

The sequence is as follows.

1. If using an `onEvent()` function (the LMIC is configured to use `onEvent`, and a function named `onEvent` is provided when using compilers that support weak references), the `onEvent()` function is called. (For compatibility, the event `EV_RXSTART` is never sent to the `onEvent()` function.)
2. If the event indicates that a message was received, and a receive callback is registered, the receive callback is called.
3. If the event indicates that a transmission has completed, and the message was sent with one of the callback APIs, the client callback is invoked.
4. Finally, if the client has registered an event callback, the registered callback is invoked.

2.5.21.1 Receiving Downlink Data with an Event Function

When `EV_TXCOMPLETE` or `EV_RXCOMPLETE` is received, the event-processing code should check for downlink data, and pass it to the application. To do this, use code like the following.

```
    // Any data to be received?
    if (LMIC.dataLen != 0) {
        // Data was received. Extract port number if any.
        u1_t bPort = 0;
        if (LMIC.txrxFlags & TXRX_PORT)
            bPort = LMIC.frame[LMIC.dataBeg - 1];
        // Call user-supplied function with port #, pMessage, nMessage
        receiveMessage(
            bPort, LMIC.frame + LMIC.dataBeg, LMIC.dataLen
            );
    }
```

If you wish to support alerting the client for zero-length messages, slightly-more complex code must be used.

```
    // Any data to be received?
    if (LMIC.dataLen != 0 || LMIC.dataBeg != 0) {
```

```
        // Data was received. Extract port number if any.
        u1_t bPort = 0;
        if (LMIC.txrxFlags & TXRX_PORT)
            bPort = LMIC.frame[LMIC.dataBeg - 1];
        // Call user-supplied function with port #, pMessage, nMessage;
        // nMessage might be zero.
        receiveMessage(
            bPort, LMIC.frame + LMIC.dataBeg, LMIC.dataLen
            );
    }
```

2.5.22 bit_t LMIC_enableTracking (u1_t *tryBcnInfo*)

Enable beacon tracking. A value of 0 for *tryBcnInfo* indicates to start scanning for the beacon immediately. A non-zero value specifies the number of attempts to query the server for the exact beacon arrival time. The query requests will be sent within the next upstream frames (no frame will be generated). If no answer is received scanning will be started. The events `EV_BEACON_FOUND` or `EV_SCAN_TIMEOUT` will be generated for the first beacon, and the events `EV_BEACON_TRACKED`, `EV_BEACON_MISSED` or `EV_LOST_TSYNC` will be generated for subsequent beacons.

2.5.23 void LMIC_disableTracking ()

Disable beacon tracking. The beacon will be no longer tracked and, therefore, also pinging will be disabled.

2.5.24 void LMIC_setPingable (u1_t *intvExp*)

Enable pinging and set the downstream listen interval. Pinging will be enabled with the next upstream frame (no frame will be generated). The listen interval is 2^*intvExp* seconds, valid values for `intvExp` are 0-7. This API function requires a valid session established with the network server either via `LMIC_startJoining()` or `LMIC_setSession()` functions (see sections 2.5.2 and 2.5.4). If beacon tracking is not yet enabled, scanning will be started immediately. In order to avoid scanning, the beacon can be located more efficiently by a preceding call to `LMIC_enableTracking()` with a non-zero parameter. Additionally to the events mentioned for `LMIC_enableTracking()`, the event `EV_RXCOMPLETE` will be generated whenever downstream data has been received in a ping slot.

2.5.25 void LMIC_stopPingable ()

Stop listening for downstream data. Periodical reception is disabled, but beacons will still be tracked. In order to stop tracking, the beacon a call to `LMIC_disableTracking()` is required.

2.5.26 void LMIC_sendAlive ()

Send one empty upstream MAC frame as soon as possible. Might be used to signal liveness or to transport pending MAC options, and to open a receive window.

2.5.27 void LMIC_shutdown ()

Stop all MAC activity. Subsequently, the MAC needs to be reset via a call to `LMIC_reset()` and new protocol actions need to be initiated.

2.5.28 void LMIC_requestNetworkTime (lmic_request_network_time_cb_t *, void *pUserData)

Register a network time MAC request to be forwarded with the next uplink frame. The first argument is a pointer to a function, with the following signature.

```
typedef void LMIC_ABI_STD lmic_request_network_time_cb_t (void * pUserData, int
flagSuccess);
```

This function is called after processing is complete. `flagSuccess` will be non-zero if time was successfully obtained, zero otherwise. `pUserData` in the callback will be set according to the value of `pUserData` in the original call to `LMIC_requestNetworkTime()`. If not used, please use a NULL pointer.

2.5.29 int LMIC_getNetworkTimeReference (lmic_time_reference_t *pReference)

Fetch a time reference for the most-recently obtained network time. The `lmic_time_reference_t` structure at `pReference` is updated with the relevant information. The result of this call is a Boolean indicating whether a valid time was returned if non-zero, a valid time was returned, otherwise no valid time was available (or `pReference` was `NULL`). If unsuccessful, `*pReference` is not modified.

The structure lmic_time_reference_t has the following fields.

```
typedef struct {
    ostime_t tLocal;
    lmic_gpstime_t tNetwork;
} lmic_time_reference_t;
```

`tNetwork` is set to the GPS time transmitted by the network in the `DeviceTimeAns` message. `tLocal` is calculated, by converting the fractional part of the `DeviceTimeAns` message into OS ticks, and subtracting that from the completion time of the `DeviceTimeReq` message.

The two fields establish a relationship between a given OS time `tLocal` and a given GPS time `tNetwork`. From this, you can work out the current OS time corresponding to a given GPS time, using a formula like `ostime = ref.tLocal + sec2osticks(gpstime – ref.tNetwork)`.

2.5.30 uint8_t LMIC_setBatteryLevel (uint8_t uBattLevel)

Set the battery level that will be returned to the network server by the MAC in `DevStatusAns` messages. The result of this call is the previous value. The internal value is initialized by LMIC_init() to `MCMD_DEVS_NOINFO`. The internal value is not changed by `LMIC_reset()`.

The possible values of `uBattLevel` are:

Value	Name	Meaning
0	MCMD_DEVS_EXT_POWER	Device is operating on external power.
0x01	MCMD_DEVS_BATT_MIN	Device is operating on battery power; battery is at minimum value.
…		
0xFE	MCMD_DEVS_BATT_MAX	Device is operating on battery power; battery is at maximum value
0xFF	MCMD_DEVS_BATT_NOINFO	The device doesn't know its battery / power state, and was unable to measure the battery level.

If the application knows the battery level as a % of capacity (from 0% to 100%, inclusive), it should calculate as follows.

```
uBattLevel = (MCMD_DEVS_BAT_MAX – MCMD_DEVS_BAT_MIN253 + 50) * uBatteryPercent / 100;
uBattLevel += MCMD_DEVS_BAT_MIN;

LMIC_setBatteryLevel(uBattLevel);
```

2.5.31 uint8_t LMIC_getBatteryLevel (void)

This function returns the battery level currently stored for use by the MAC. The value is as described in `LMIC_setBatteryLevel()`.

3. Hardware Abstraction Layer

The LMIC library is separated into a large portion of portable code and a small platform-specific part. By implementing the functions of this hardware abstraction layer with the specified semantics, the library can be easily ported to new hardware platforms.

3.1 HAL Interface

The following groups of hardware components must be supported:

- Up to four digital I/O lines are needed in output mode to drive the radio's antenna switch (`RX` and `TX`), the SPI chip select (`NSS`), and the reset line (`RST`).
- Three digital I/O lines are needed in input mode to sense the radio's transmitter and receiver states (`DIO0, DIO1` and `DIO2`).
- A SPI unit is needed to read and write the radio's registers.
- A timer unit is needed to precisely record events and to schedule new protocol actions.
- An interrupt controller cab be used to forward interrupts generated by the digital input lines.

This section describes the function interface required to access these hardware components:

3.1.1 `void hal_init ()`

Initialize the hardware abstraction layer. Configure all components (IO, SPI, TIMER, IRQ) for further use with the `hal_xxx()` functions. This function is deprecated and obsolete. The LMIC library calls `hal_init_ex()` instead. The client cannot call `hal_init()` or `hal_init_ex()` directly,as they are called from `os_init()`/`os_init_ex()`, and they must only be called once.

3.1.2 `void hal_init_ex (const void *pHalData)`

Initialize the hardware abstraction layer. Configure all components (IO, SPI, TIMER, IRQ) for further use with the `hal_xxx()` functions. pHalData is a pointer to HAL-specific data. When running with the Arduino HAL, this must be a pointer to a `lmic_pinmap` structure. The LMIC library calls `hal_init_ex()`. The client cannot call `hal_init()` or `hal_init_ex()` directly, as they are called from `os_init()`/`os_init_ex()`, and they must only be called once.

3.1.3 `void hal_failed ()`

Perform "fatal failure" action. This function will be called by code assertions on fatal conditions. Possible actions could be HALT or reboot.

3.1.4 `void hal_pin_rxtx (u1_t val)`

Drive the digital output pins `RX` and `TX` (0=receive, 1=transmit).

3.1.5 `void hal_pin_rst (u1_t val)`

Control the radio `RST` pin (0=low, 1=high, 2=floating)

3.1.6 void radio_irq_handler (u1_t *dio*)

When the HAL detects a rising edge on any of the three input lines `DIO0`, `DIO1` and `DIO2`, it must notify the LMIC. It may do this by calling the function `radio_irq_handler()`. It must set `dio` to indicate the line which generated the interrupt (`0, 1, 2`). This routine is a wrapper for `radio_irq_handler_v2()`, and just calls `os_getTime()` to get the current time. If your hardware can capture the interrupt time more accurately, your HAL should use `radio_irq_handler_v2()`.

3.1.7 void radio_irq_handler_v2 (u1_t *dio*, os_time_t *tIrq*)

When the HAL detects a rising edge on any of the three input lines `DIO0`, `DIO1` and `DIO2`, it must notify the LMIC. If the HAL has a high-accuracy time-stamp for when the line changed state, it should call the function `radio_irq_handler_v2()`. Set `dio` to indicate the line which changed (`0, 1, 2`). Set `tIrq` to the time-stamp of when the line changed state.

3.1.8 void hal_spi_read(u1_t cmd, u1_t* buf, size_t len)

Perform a SPI read. Write the command byte `cmd`, then read `len` bytes into the buffer starting at `buf`.

3.1.9 void hal_spi_write(u1_t cmd, const u1_t* buf, size_t len)

Perform a SPI write. Write the command byte `cmd`, followed `len` bytes from buffer starting at `buf`.

3.1.10 u4_t hal_ticks ()

Return 32-bit system time in ticks (same units as `ostime_t`) – but note that this is unsigned, whereas `ostime_t` is signed.

3.1.11 void hal_waitUntil (u4_t *time*)

Busy-wait until specified timestamp (in ticks) is reached.

3.1.12 u1_t hal_checkTimer (u4_t *targettime*)

Check and rewind timer for given *targettime*. Return 1 if *targettime* is close (not worthwhile programming the timer). Otherwise rewind timer for exact *targettime* or for full timer period and return 0. The only action required when *targettime* is reached is that the CPU wakes up from possible sleep states.

3.1.13 void hal_disableIRQs ()

Disable all CPU interrupts. Might be invoked nested. But will always be followed by matching call to `hal_enableIRQs()`.

3.1.14 void hal_enableIRQs ()

Enable CPU interrupts. When invoked nested, only the outmost invocation actually must enable the interrupts.

3.1.15 void hal_sleep ()

Sleep until interrupt occurs. Preferably system components can be put in low-power mode before sleep, and be re-initialized after sleep. When using the Arduino reference implementation, this is a no-op; the LMIC returns to the caller, who is responsible for arranging to sleep.

3.1.16 `s1_t` **`hal_getRssiCal`** `()`

Get the RSSI calibration for the radio, in dB. The radio driver adds this to the indicated RSSI to convert to absolute dB, for doing listen-before-talk computations. Not used unless listen-before-talk is configured for this region.

3.1.17 `ostime_t` **`hal_setModulePower`** `(bool val)`

Request that the module be powered up or down. If true, TCXO power should be activated, and any normally high-Z control lines should be activated. This function returns the number of ticks of delay that must be inserted before using the radio. If module-level power control is not implemented, or if the radio is already in the desired state, this routine can just return zero. Normally a delay of a few milliseconds is needed when turning power on, but no delay is needed if power is already on or if turning power off.

3.2 HAL Reference Implementation for Arduino

The Arduino LMIC library includes a reference implementation of the HAL for the Arduino. Please refer to README.md for information about the implementation.

4. Examples

A set of examples is provided to demonstrate how typical node applications can be implemented with only a few lines of code using the LMIC library.

5. Release History

The Arduino LMIC release history is in the README.md file at https://github.com/mcci-catena/arduino-lmic.

5.1 IBM Release History

Version and date	Description
V 1.0 November 2014	Initial version.
V 1.1 January 2015	Added API `LMIC_setSession()`. Minor internal fixes.
V 1.2 February 2015	Added APIs `LMIC_setupBand()`, `LMIC_setupChannel()`, `LMIC_disableChannel()`, `LMIC_setLinkCheckMode()`. Minor internal fixes.
V 1.4 March 2015	Changed API: port indicator flag in `LMIC.txrxFlags` has been inverted (now `TXRX_PORT`, previously `TXRX_NOPORT`). Internal bug fixes. Document formatting.
V 1.5 May 2015	Bug fixes and documentation update.
V 1.6 Jul 2016	Changed license to BSD. Included modem application (see examples/modem and LMIC-Modem.pdf). Added STM32 hardware drivers and Blipper board-specific peripheral code.