|  |
| --- |
|  |
| Arduino LoRaWAN MAC in C (LMIC) |
| Version |
| **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](mailto: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/>.  **© 2018-2024 MCCI Corporation**  Copyright MCCI Corporation, 2018-2024. All rights reserved. Distributed under the terms of the LICENSE file found at <https://github.com/mcci-catena/arduino-lmic/blob/master/LICENSE>.  **© 2014-2016 IBM Corporation**  Copyright International Business Machines Corporation, 2014-2016. All Rights Reserved.  The following are trademarks or registered trademarks of International Business Machines Corporation in the United States, or other countries, or both: IBM, the IBM Logo, Ready for IBM Technology.  MCCI and MCCI Catena are registered trademarks of MCCI Corporation.  LoRa is a registered trademark of Semtech Corporation.  LoRaWAN is a registered trademark of the LoRa Alliance.  Other company, product and service names may be trademarks or service marks of others.  All information contained in this document is subject to change without notice. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary. THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document. |

Table of Contents

[1. Introduction 5](#_Toc182760456)

[1.1 LoRaWAN Versions and Features Supported 6](#_Toc182760457)

[1.2 Class A and Class B Support 6](#_Toc182760458)

[2. Programming Model and API 7](#_Toc182760459)

[2.1 Programming Model 7](#_Toc182760460)

[2.2 Run-time Functions 9](#_Toc182760461)

[2.3 Application callbacks 10](#_Toc182760462)

[2.4 The LMIC Struct 11](#_Toc182760463)

[2.5 API Functions 12](#_Toc182760464)

[3. Hardware Abstraction Layer 23](#_Toc182760465)

[3.1 HAL Interface 23](#_Toc182760466)

[3.2 HAL Reference Implementation for Arduino 25](#_Toc182760467)

[4. Examples 26](#_Toc182760468)

[5. Release History 27](#_Toc182760469)

[5.1 IBM Release History 27](#_Toc182760470)

# Introduction

The Arduino IBM LoRaWAN C-library ([LMIC](https://github.com/mcci-catena/arduino-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, SX1276, SX1261 or SX1262 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 |
| LMIC Library  SX1272/1276/1261/1262 LoRa Radio  Sensor Hardware  MCU  Application Code  Application Drivers  (Sensors, etc.)  Hardware Abstraction Layer  MAC State Engine  Run-time Environment  Arduino Platform |
| *High-level view of all application device components.* |
|  |

## 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 (with the SX1276 radio) 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.

## 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.

# 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.

## 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.

### 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.

### 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.

### 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.

## 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.

void os\_init ()

Initialize the operating system by calling os\_init\_ex(NULL).

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.

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).

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.

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.

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.

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.

ostime\_t os\_getTime ()

Query absolute system time (in ticks).

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.

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.

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.

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.

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.

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.

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.

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.

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.

## 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.

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).

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).

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).

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.

## 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 |

## API Functions

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

void LMIC\_reset ()

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

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.

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.

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.

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 ≤ *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.

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.

void LMIC\_disableChannel (u1\_t *channel*)

Disable specified channel. Default channels cannot be disabled.

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.

void LMIC\_setAdrMode (bit\_t *enabled*)

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

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.

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.

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.

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.

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.

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. |

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. |

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.

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.

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.

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.

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.

#### 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

);

}

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.

void LMIC\_disableTracking ()

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

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.

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.

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.

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.

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.

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).

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);

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().

# 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.

## 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:

void lmic\_hal\_init ()

Initialize the hardware abstraction layer. Configure all components (IO, SPI, TIMER, IRQ) for further use with the lmic\_hal\_xxx() functions. This function is deprecated and obsolete. The LMIC library calls lmic\_hal\_init\_ex() instead. The client cannot call lmic\_hal\_init() or lmic\_hal\_init\_ex() directly,as they are called from os\_init()/os\_init\_ex(), and they must only be called once.

void lmic\_hal\_init\_ex (const void \*pHalData)

Initialize the hardware abstraction layer. Configure all components (IO, SPI, TIMER, IRQ) for further use with the lmic\_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 lmic\_hal\_init\_ex(). The client cannot call lmic\_hal\_init() or lmic\_hal\_init\_ex() directly, as they are called from os\_init()/os\_init\_ex(), and they must only be called once.

void lmic\_hal\_failed (const char \*pFileName, u2\_t line)

Perform “fatal failure” action. This function will be called by code assertions on fatal conditions. Possible actions could be HALT or reboot. pFileName points to a C string containing the name of the source file reporting the failure; line gives the line number of the code reporting the failure.

If a custom failure handler function has been registered (section 3.1.5), the HAL will call it before performing any other actions specific to the HAL implementation.

typedef void lmic\_failure\_handler\_t (const char \* const file, const u2\_t line)

This typedef is used for declaring the prototypes of custom failure handler functions. See lmic\_set\_failure\_handler(), section 3.1.5.

void lmic\_set\_failure\_handler (lmic\_hal\_failure\_handler\_t \*handler)

Set the custom failure handler function to handler, replacing any previous handler. If handler is NULL, then the registration is canceled.

void lmic\_hal\_pin\_rxtx (u1\_t *val*)

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

void lmic\_hal\_pin\_rst (u1\_t *val*)

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

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().

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.

void lmic\_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.

void lmic\_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.

u4\_t lmic\_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.

void lmic\_hal\_waitUntil (u4\_t *time*)

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

u1\_t lmic\_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.

void lmic\_hal\_disableIRQs ()

Disable all CPU interrupts. Might be invoked nested. But will always be followed by matching call to lmic\_hal\_enableIRQs().

void lmic\_hal\_enableIRQs ()

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

void lmic\_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.

s1\_t lmic\_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.

ostime\_t lmic\_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.

## 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. General implementation details are given in the following section.

### lmic\_hal\_failed() behavior

If an error occurs and lmic\_hal\_failed() is called, the Arduino LMIC HAL implementation does the following by default.

1. If a custom error handler has been registered by lmic\_hal\_set\_failure\_handler(), it is called.
2. It prints the message “Failure {file}:{line}\n” to the console.
3. It calls lmic\_hal\_disableIRQs().
4. It enters an infinite loop.

# 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.

# Release History

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

## 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. |