|  |
| --- |
| IBM Long Range Signaling and Control |
| IBM LoRaWAN in C (LMiC) |
| Version |
| **LMiC Product Information**  LMiC is 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).  **© 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.  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](#_Toc456268601)

[2. Programming Model and API 6](#_Toc456268602)

[2.1 Programming Model 6](#_Toc456268603)

[2.2 Run-time Functions 7](#_Toc456268604)

[2.3 Application callbacks 7](#_Toc456268605)

[2.4 The LMIC Struct 9](#_Toc456268606)

[2.5 API Functions 10](#_Toc456268607)

[3. Hardware Abstraction Layer 13](#_Toc456268608)

[3.1 HAL Interface 13](#_Toc456268609)

[3.2 HAL Reference Implementation for STM32/Cortex-M3 14](#_Toc456268610)

[4. Examples 16](#_Toc456268611)

[4.1 Example 1: hello 17](#_Toc456268612)

[4.2 Example 2: join 17](#_Toc456268613)

[4.3 Example 3: transmit 18](#_Toc456268614)

[4.4 Example 4: periodic 19](#_Toc456268615)

[4.5 Example 5: interrupt 20](#_Toc456268616)

[4.6 Example 6: beacon 20](#_Toc456268617)

[4.7 Example 7: ping 21](#_Toc456268618)

[4.8 Example 8: modem 22](#_Toc456268619)

[4.9 Debug library 22](#_Toc456268620)

[5. Release History 24](#_Toc456268621)

# Introduction

The IBM LoRaWAN C-library (LMiC) is a portable implementation of the LoRa™ MAC specification for the C programming language. It supports both the EU-868 and the US-915 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 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. For the STM32/Cortex-M3 platform, a reference implementation of the HAL is supplied and the overall code footprint of all components on this platform is less than 20K.

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 their code will not be provided by IBM.

|  |
| --- |
| Figure . Application device components |
| LMiC Library  SX1272 LoRa Radio  Sensor Hardware  STM32 MCU  Application Code  Application Drivers  (Sensors, etc.)  Hardware Abstraction Layer  MAC State Engine  Run-time Environment |
| *High-level view of all application device components. On top of the STM32 MCU with the connected SX1272 radio and other sensor hardware runs the LMiC library and the application code.* |
|  |

# 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”

To identify the version of the LMiC library two constants are defined in this header file.

#define LMIC\_VERSION\_MAJOR 1

#define LMIC\_VERSION\_MINOR 6

## 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 2.2.4). These application jobs are coded as normal C functions and can be managed using the run-time functions described in section 2.2. 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 has to do is to initialize the run-time environment using the os\_init() function and to run the job scheduler function os\_runloop(), which does not return. 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.

void main () {

osjob\_t initjob;

*// initialize run-time env*

os\_init();

*// setup initial job*

os\_setCallback(&initjob, initfunc);

*// execute scheduled jobs and events*

os\_runloop();

*// (not reached)*

}

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.

## Run-time Functions

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

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.

ostime\_t os\_getTime ()

Query absolute system time (in ticks).

## Application callbacks

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

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*)

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 is still connected to the old network.
* EV\_TXCOMPLETE  
  The data prepared via LMIC\_setTxData() has been sent, and eventually downstream data has been received in return. If confirmation was requested, the acknowledgement has been received.
* EV\_RXCOMPLETE  
  Downstream data has been received.
* 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.

Details for specific events can be obtained from the global LMIC structure decribed in the next section.

## 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 2.5.11).

For the EV\_RXCOMPLETE and EV\_TXCOMPLETE events, the txrxFlags field sould be evaluated and 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 field is contained in the received frame
* 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

For the EV\_TXCOMPLETE event the fields have the following values:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Received frame** | **LMIC.txrxFlags** | | | | | | **LMIC.dataLen** | **LMIC.dataBeg** |
| **ACK** | **NACK** | **PORT** | **DNW1** | **DNW2** | **PING** |
| nothing | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| empty frame | x | x | 0 | x | x | 0 | 0 | x |
| port only | x | x | 1 | x | x | 0 | 0 | x |
| port+payload | x | x | 1 | x | x | 0 | x | x |

For the EV\_RXCOMPLETE event the fields have the following values:

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

Create new band with the specified transmit power and duty cycle (1/*txcap*) properties.

bit\_t LMIC\_setupChannel (u1\_t *channel*, u4\_t *freq*, u2\_t *drmap*, s1\_t *band*)

Create new channel in the given band using the specified frequency and allowing the data rates defined in the data rate bitmask (1 << DRx).

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

Disable specified channel.

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.

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.

int 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.

void LMIC\_clrTxData ()

Remove data previously prepared for upstream transmission.

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.

# 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:

* 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 is needed to forward interrupts generated by the digital input lines.

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

void hal\_init ()

Initialize the hardware abstraction layer. Configure all components (IO, SPI, TIMER, IRQ) for further use with the hal\_xxx() functions.

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.

void hal\_pin\_rxtx (u1\_t *val*)

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

void hal\_pin\_nss (u1\_t *val*)

Drive the digital output pin NSS (0=low/selected, 1=high/deselected).

void hal\_pin\_rst (u1\_t *val*)

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

void radio\_irq\_handler (u1\_t *dio*)

The three input lines DIO0, DIO1 and DIO2 must be configured to trigger an interrupt on the rising edge and the corresponding interrupt handlers must invoke the function radio\_irq\_handler() and pass the line which generated the interrupt as argument (0, 1, 2).

u1\_t hal\_spi (u1\_t *outval*)

Perform 8-bit SPI transaction. Write given byte *outval* to radio, read byte from radio and return value.

u4\_t hal\_ticks ()

Return 32-bit system time in ticks.

void hal\_waitUntil (u4\_t *time*)

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

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.

void hal\_disableIRQs ()

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

void hal\_enableIRQs ()

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

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.

## HAL Reference Implementation for STM32/Cortex-M3

The source code of the LMiC library includes a reference implementation of the HAL for the STM32/Cortex-M3 platform. This implementation demonstrates the required semantics of the HAL function interface. For brevity’s sake it is kept as simple as possible and it is not optimized (e.g. for power consumption). We will describe here the hardware resources used by this implementation. Applications using the library must be aware of the use of these resources and must not interfere with them! Either applications have to use different resources availbale on the platform, or they have to modify the HAL implementation and multiplex access to the required resources!

### Output I/O Lines

The following generic output lines are used to control the radio.

|  |  |
| --- | --- |
| **Function** | **GPIO** |
| NSS | PB 0 |
| TX | PA 4 |
| RX | PC 13 |
| RST | PA 2 |

### Input I/O Lines

The following generic input lines are used to track the transmitter and receiver state. These lines are programmed to generate interrupts on the rising edge (see section 3.1.6 and 3.2.5).

|  |  |
| --- | --- |
| **Function** | **GPIO** |
| DIO 0 | PB 1 |
| DIO 1 | PB 10 |
| DIO 2 | PB 11 |

### SPI

The SPI1 peripheral is connected to the radio as shown in the table below.

|  |  |
| --- | --- |
| **Function** | **GPIO** |
| SCK | PA 5 |
| MISO | PA 6 |
| MOSI | PA 7 |

### Timer

The TIMER 9 peripheral is used to provide 32kHz clock ticks and to generate comparator interrupts for scheduled protocol actions.

### Interupts

One EXTI interrupt handler is used to handle all external I/O line interrupt groups (0, 1, 2, 3, 4, 5-9, 10-15). The EXTI handler checks the source of the interrupt and eventually invokes the radio\_irq\_handler().

The handler for TIMER 9 interrupts updates the system clock ticks on roll-over of the counter. No specific action has to be performed by the handler when the interrupt is triggered by the comparator. It is sufficient that the CPU wakes from sleep and the run-time environment of LMiC can check for pending actions.

# 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. The shipped examples are ready to run on the demo board of the *IMST / WiMOD LoRa™ Radio Starter Kit*.

The examples can be built using different compiler toolchains and makefiles are provided for *IAR*, *Keil* and *GCC*. Additionally project files for the integrated development environments *IAR Workbench* and *Keil µVision* are provided in the examples directory of the ZIP file.

**Note:** all projects should have the following preprocessor defines set in the project options:

CFG\_eu868, CFG\_wimod\_board, CFG\_sx1272\_radio

In addition to driving the radio using the LMiC APIs, the examples are written to give local feedback via a LED and a serial console using the debug library described in section 4.8. The demo board will make the serial console available as “USB Serial Port” on the connected PC and the output of the debug functions can be easily viewed with the terminal application of your choice. Communication parameters are 115200bps 8/N/1.

|  |
| --- |
| Figure . IMST / WiMOD LoRa™ Radio Starter Kit |
|  |
| *The examples have been tested on IMST WiMOD SK-iM880A.* |
|  |

For brevity’s sake only the relevant portions of the code are included in the snippets shown for each example in this section. In most cases, this is the application’s onEvent() callback function plus some utility glue contained in the example’s main.c file.

## Example 1: hello

The first example (*hello*) can be used to verify that your development environment is up and running and all components are well connected. The example doesn’t use the radio, it only uses the run-time functions and the debug library to periodically log a counter value to the serial console and to blink the LED.

// counter

static int cnt = 0;

// log text to USART and toggle LED

static void initfunc (osjob\_t\* job) {

// say hello

debug\_str("Hello World!\r\n");

// log counter

debug\_val("cnt = ", cnt);

// toggle LED

debug\_led(++cnt & 1);

// reschedule job every second

os\_setTimedCallback(job, os\_getTime()+sec2osticks(1), initfunc);

}

If everything is set up correctly and the program is executed you should see the LED blink in a one second interval and see the following output on the terminal:

============== DEBUG STARTED ==============

Hello World!

cnt = 00000000

Hello World!

cnt = 00000001

Hello World!

cnt = 00000002

. . . . . .

## Example 2: join

The next example (*join*) can be used to verify that the radio is working and that the node settings are correct and match your network infrastructure. For the example to work, the application callbacks os\_getArtEui(), os\_getDevEui() and os\_getDevKey() have to return correct values for the application router id, the device id, and the device key!

static osjob\_t blinkjob;

static u1\_t ledstate = 0;

static void blinkfunc (osjob\_t\* j) {

// toggle LED

ledstate = !ledstate;

debug\_led(ledstate);

// reschedule blink job

os\_setTimedCallback(j, os\_getTime()+ms2osticks(100), blinkfunc);

}

void onEvent (ev\_t ev) {

debug\_event(ev);

switch(ev) {

// starting to join network

case EV\_JOINING:

// start blinking

blinkfunc(&blinkjob);

break;

// network joined, session established

case EV\_JOINED:

// cancel blink job

os\_clearCallback(&blinkjob);

// switch on LED

debug\_led(1);

// (don't schedule any new actions)

break;

}

}

On execution the LED should start blinking fast, and after about five seconds (if the network can be successfully joined), it should become ON permanently. The output on the terminal should be JOINING at the beginning, and after about five seconds JOINED.

============== DEBUG STARTED ==============

JOINING

JOINED

## Example 3: transmit

After joining the network, the *transmit* example will start sending upstream frames containing one byte with the last known signal-to-noise ratio. Once a transmission is complete, a new transmission will be scheduled immediately, and hence the frames will be sent with the maximum rate permitted by the duty cycle. If downstream data has been received in the receive slot after the transmission, it will be logged to the console.

void onEvent (ev\_t ev) {

debug\_event(ev);

switch(ev) {

// network joined, session established

case EV\_JOINED:

debug\_val("netid = ", LMIC.netid);

goto tx;

// scheduled data sent (optionally data received)

case EV\_TXCOMPLETE:

if(LMIC.dataLen) { // data received in rx slot after tx

debug\_buf(LMIC.frame+LMIC.dataBeg, LMIC.dataLen);

}

tx:

// immediately prepare next transmission

LMIC.frame[0] = LMIC.snr;

// schedule transmission (port 1, datalen 1, no ack requested)

LMIC\_setTxData2(1, LMIC.frame, 1, 0);

// (will be sent as soon as duty cycle permits)

break;

}

}

The upstream frames should be delivered to the application router and the following output should be seen on the node’s console:

============== DEBUG STARTED ==============

JOINING

JOINED

netid = 00000001

TXCOMPLETE

TXCOMPLETE

TXCOMPLETE

## Example 4: periodic

The next example (*periodic*) will periodically report a sensor value to the network. After joining, a job is run which reads the sensor, prepares an upstream transmission with the sensor’s value, and reschedules the job for repeated execution in 60 seconds. To implement the sensor, this example uses platform-specific functions initsensor() and readsensor() contained in the file sensor.c. The sample sensor simply reads the position of the “DIP switch 1” on the demo board (PB 12) as 1-bit value.

static osjob\_t reportjob;

// report sensor value every minute

static void reportfunc (osjob\_t\* j) {

// read sensor

u2\_t val = readsensor();

debug\_val("val = ", val);

// prepare and schedule data for transmission

LMIC.frame[0] = val << 8;

LMIC.frame[1] = val;

LMIC\_setTxData2(1, LMIC.frame, 2, 0); // (port 1, 2 bytes, unconfirmed)

// reschedule job in 60 seconds

os\_setTimedCallback(j, os\_getTime()+sec2osticks(60), reportfunc);

}

void onEvent (ev\_t ev) {

debug\_event(ev);

switch(ev) {

// network joined, session established

case EV\_JOINED:

// switch on LED

debug\_led(1);

// kick-off periodic sensor job

reportfunc(&reportjob);

break;

}

}

Depending on the position of DIP switch 1, this example should generate output similar to this:

============== DEBUG STARTED ==============

JOINING

JOINED

val = 00000001

TXCOMPLETE

val = 00000001

TXCOMPLETE

val = 00000000

TXCOMPLETE

## Example 5: interrupt

This example (*interrupt*) uses the same sensor as in the previous example but it doesn’t read the sensor periodically. Instead, it is interrupt-driven and only sends the sensor value when the sensor has changed. An application-defined interrupt handler has been added in the sensor.c file to run a registered job callback when the interrupt is triggered:

// called by EXTI\_IRQHandler

// (set preprocessor option CFG\_EXTI\_IRQ\_HANDLER=sensorirq)

void sensorirq () {

if((EXTI->PR & (1<<INP\_PIN)) != 0) { // pending

EXTI->PR = (1<<INP\_PIN); // clear irq

// run application callback function in 50ms (debounce)

os\_setTimedCallback(&irqjob, os\_getTime()+ms2osticks(50), irqjob.func);

}

}

## Example 6: beacon

The following example (*beacon*) enables beacon tracking after joining the network. It drives the LED depending on the TRACKED/MISSED events in each period. If the beacon is successfully tracked, the GPS time contained in the beaon is logged to the console.

void onEvent (ev\_t ev) {

debug\_event(ev);

switch(ev) {

// network joined, session established

case EV\_JOINED:

// enable tracking mode, start scanning...

LMIC\_enableTracking(0);

debug\_str("SCANNING...\r\n");

break;

// beacon found by scanning

case EV\_BEACON\_FOUND:

// switch LEN on

debug\_led(1);

break;

// beacon tracked at expected time

case EV\_BEACON\_TRACKED:

debug\_val("GPS time = ", LMIC.bcninfo.time);

// switch LEN on

debug\_led(1);

break;

// beacon missed at expected time

case EV\_BEACON\_MISSED:

// switch LEN off

debug\_led(0);

break;

}

}

Depending on the reception quality the console output should look similar to this:

============== DEBUG STARTED ==============

JOINING

JOINED

SCANNING...

BEACON\_FOUND

BEACON\_TRACKED

GPS time = 545CE201

BEACON\_TRACKED

GPS time = 545CE281

BEACON\_TRACKED

GPS time = 545CE301

## Example 7: ping

The next example (*ping*) joins the network and repeatedly listens for downstream data. This is achieved by enabling the beacon-based ping mode with an interval of two seconds. The call to LMIC\_setPingable() sets the ping mode locally and starts scanning for the beacon. Once the first beacon has been found, an upstream frame needs to be sent (in this case an empty frame via LMIC\_sendAlive()) to transport the MAC options and to notify the server of the ping mode and interval. Whenever the server has sent downstream data in one of the receive slots, the EV\_RXCOMPLETE event is triggered and the received data can be evaluated in the frame field of the LMIC struct. The sample code logs the received data to the console and, in the special case when exactly one byte is received, it drives the LED depending on the received value.

void onEvent (ev\_t ev) {

debug\_event(ev);

switch(ev) {

// network joined, session established

case EV\_JOINED:

// enable pinging mode, start scanning...

// (set local ping interval configuration to 2^1 == 2 sec)

LMIC\_setPingable(1);

debug\_str("SCANNING...\r\n");

break;

// beacon found by scanning

case EV\_BEACON\_FOUND:

// send empty frame up to notify server of ping mode and interval!

LMIC\_sendAlive();

break;

// data frame received in ping slot

case EV\_RXCOMPLETE:

// log frame data

debug\_buf(LMIC.frame+LMIC.dataBeg, LMIC.dataLen);

if(LMIC.dataLen == 1) {

// set LED state if exactly one byte is received

debug\_led(LMIC.frame[LMIC.dataBeg] & 0x01);

}

break;

}

}

## Example 8: modem

The next example *(modem)* is a full modem application and provides easy access to LoRaWAN networks via high-level ASCII commands exchanged over a serial interface. For complexity reasons the documentation of the modem has been moved to a separate file LMiC-Modem.pdf.

## Debug library

A small debug library is provided to enable local text output for the examples shown in this chapter. This library is not required by LMiC but is useful for development and debugging. The library functions offer simple serial console logging and access to a LED for diagnostic output. The debug library is platform-specific and is implemented for STM32/Cortex-M3.

void debug\_init ()

Initilize the peripherals required for the debug functions. USART1 and LED4 are used in the reference implementation for STM32/Cortex-M3. Serial communication settings are 115200 8/N/1.

|  |  |
| --- | --- |
| **Function** | **GPIO** |
| USART 1 TX | PA 9 |
| LED 4 | PA 8 |

void debug\_led (u1\_t *val*)

Drive LED (0=off, 1=on).

void debug\_char (u1\_t c)

Log single character to serial console.

void debug\_hex (u1\_t *b*)

Log byte value as two hexadecimal characters to serial console.

void debug\_buf (const u1\_t\* *buf*, u2\_t *len*)

Log multiple bytes as space-separated hex characters to serial console.

void debug\_uint (u4\_t *v*)

Log 32-bit unsigned int value as eight hexadecimal digits to serial console.

void debug\_str (const u1\_t\* *str*)

Log arbitrary nul-terminated string to serial console.

void debug\_event (int *ev*)

Log name of event followed by “\r\n” to serial console.

void debug\_val (const u1\_t\* *label*, u4\_t *val*)

Log label string plus hexadecimal integer value followed by “\r\n” to serial console.

# 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. |