IBM Long Range Signaling and Control

IBM LoRa™ MAC in C (LMiC)

Version 1.0
17 - November - 2014

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.

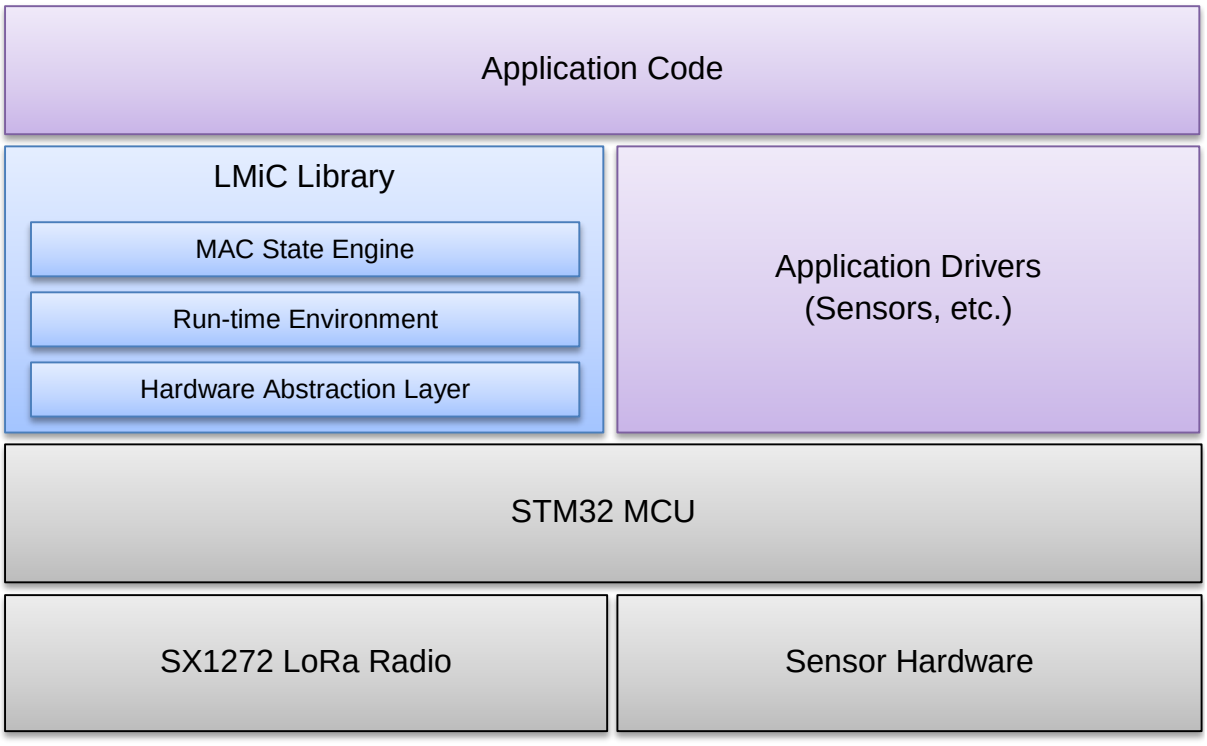
Table of Contents

1. Introduction

The IBM LoRa™ MAC in (LMiC) C-library 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. 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 fulfill the timing constraints of the MAC protocol and even considers 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 small hardware abstraction layer (HAL) the library can be easily ported to new hardware platforms. For the STM32 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 1. *Application device components*



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.

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"
```

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). 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 2.2.4). 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 control struct `osjob_t` is required per job which identifies the job and stores context information. To ensure seamless processing **all jobs must not be long-running!** Jobs should only update state and schedule actions, which will trigger new job or event callbacks.

2.1.2 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 to 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`.

2.2 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.

2.2.1 void os_setCallback (osjob_t* job, osjobcb_t cb);

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

2.2.2 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 interrupt handlers.

2.2.3 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.

2.2.4 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. It must be run on the main thread.

2.2.5 ostime_t os_getTime ();

Query absolute system time (in ticks).

2.3 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.

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);

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 dispatches 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 a longer 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.

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. Unless explicitly mentioned all fields 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` wich are used as input for the `LMIC_setTxdata()` API function (see 2.5.5).

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 implicitely by other API functions if no session has been established yet.

2.5.3 `void LMIC_setAdrMode (bit_t enabled);`

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

2.5.4 `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.

2.5.5 `void LMIC_setTxData ();`

Prepare upstream data transmission at the next possible time. 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.

2.5.6 `int LMIC_setTxData2 (u1_t port, xref2u1_t data, u1_t dlen, u1_t confirmed);`

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

2.5.7 void LMIC_clrTxData ();

Remove data previously prepared for upstream transmission.

2.5.8 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.

2.5.9 void LMIC_disableTracking ();

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

2.5.10 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 are 0-7. If beacon tracking is not yet enabled, scanning will be started immediately. To avoid scanning the beacon can be located more efficiently by a previous call to `LMIC_enableTracking()` with a non-zero parameter.

2.5.11 void LMIC_stopPingable ();

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

2.5.12 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.

2.5.13 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.

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:

- Four digital I/O lines are needed in output mode to drive the radio's antenna switch (`RX` and `TX`), the SPI chip select (`SS`), 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:

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.

3.1.2 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.

3.1.3 void hal_pin_rxtx (u1_t val);

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

3.1.4 void hal_pin_nss (u1_t val);

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

3.1.5 void hal_pin_rst (u1_t val);

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

3.1.6 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`).

3.1.7 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.

3.1.8 u4_t hal_ticks ();

Return 32-bit system time in ticks.

3.1.9 void hal_waitUntil (u4_t time);

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

3.1.10 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.

3.1.11 void hal_disableIRQs ();

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

3.1.12 void hal_enableIRQs ();

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

3.1.13 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.

3.2 HAL STM32 Reference Implementation

With the source code of the LMiC library a reference implementation for the HAL for the STM32 platform is provided. 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 resources used by this implementation. Applications using the library must be aware of these resources and must not interfere with them! Either applications have to use different resources of the platform, or they have to modify the implementation and multiplex access to the required resources!

3.2.1 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

3.2.2 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

3.2.3 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

3.2.4 Timer

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

3.2.5 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 for the source and eventually invokes the `radio_irq_handler()`.
The handler for `TIMER 9` interrupt 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.

3.3 Debug API

Next to the HAL functions a set of development and debugging functions is provided. These functions implement simple serial console logging and access to a LED for diagnostic output. The functions are not required by the LMiC library but are useful for development and are used by the examples shown in the next chapter. If the preprocessor symbol `CFG_DEBUG` is defined the `DEBUG_xxx()` macros implement the behavior described below. Otherwise no code will be generated.

3.3.1 DEBUG_INIT ();

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

Function	GPIO
USART 1 TX	PA 9
LED 4	PA 8

3.3.2 DEBUG_LED (u1_t val);

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

3.3.3 DEBUG_CHAR (u1_t c);

Log single character to serial console.

3.3.4 DEBUG_HEX (u1_t b);

Log byte value as two hexadecimal characters to serial console.

3.3.5 DEBUG_BUF (const u1_t* buf, u2_t len);

Log multiple bytes to serial console (space-separated).

3.3.6 DEBUG_UINT (u4_t v);

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

3.3.7 DEBUG_STR (const u1_t* str);

Log arbitrary nul-terminated string to serial console.

3.3.8 DEBUG_EVENT (int ev);

Log name of event followed by **"**`\r\n`**"** to serial console.

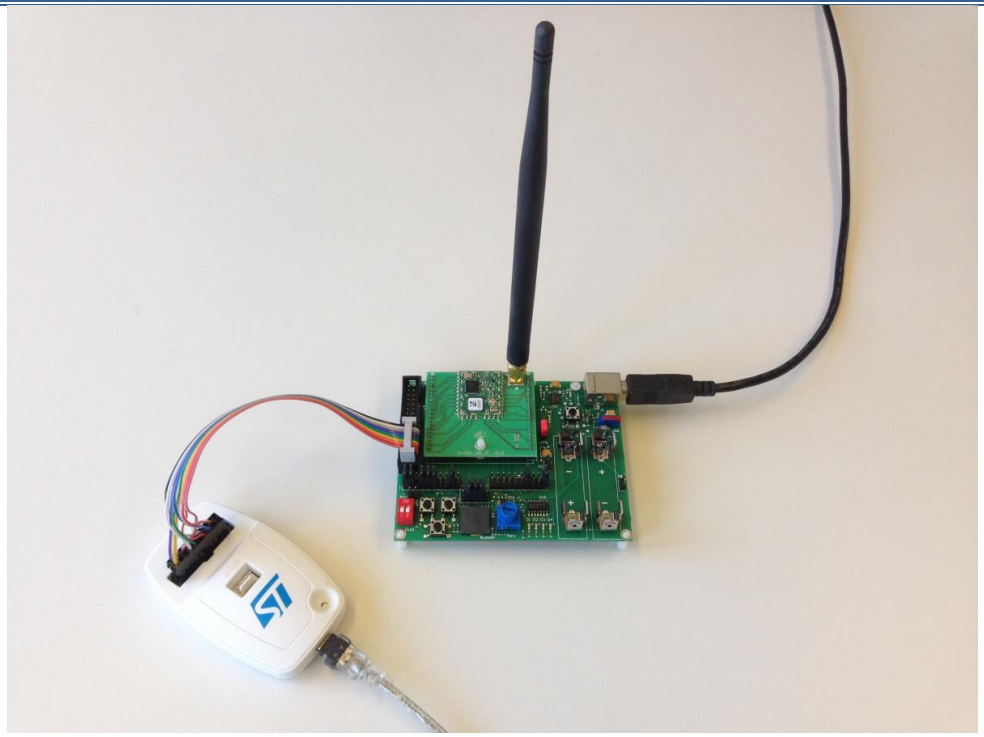
3.3.9 DEBUG_VAL (const u1_t* label, u4_t val);

Log label string plus hexadecimal integer value followed by **"**`\r\n`**"** to serial console.

4. Examples

A set of examples is provided to demonstrate how typical node applications can be implemented in only a few lines of code using the LMiC library. The shipped examples are ready to run using the *IAR Workbench* together with the *IMST / WiMOD LoRa™ Radio Starter Kit*. The `examples.eww` file in the `examples` directory of the ZIP file can be used to launch the workbench with the sample projects.

Figure 2. *IMST / WiMOD LoRa™ Radio Starter Kit*



The STM32 HAL implementation and the examples have been tested on WiMOD SK-iM880A.

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.

Note: all projects should have the following preprocessor defines set in the project options:
`CFG_DEBUG, CFG_eu868, CFG_wimod_board, CFG_sx1272_radio`

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

4.1 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 and the DEBUG functions 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
......
```

4.2 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
```

4.3 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.rxq.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
```

4.4 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 one 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
```

4.5 Example 5: interrupt

This example (*interrupt*) uses the same sensor as in the previous example but it doesn't read the sensor periodically. 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);
    }
}
```

4.6 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
```

4.7 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 the 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;
    }
```

```
}
```

5. Document Revision History

Version and date	Author	Description
V 1.0 November 2014	fhr	Initial version.