IBM Сигнализация и управление на большом расстоянии

IBM LoRaWAN на языке C (LMiC)

Версия 1.6

13 - Июль - 2016

Информация о продукте LMiC

LMiC разработан и продается исследовательской лабораторией IBM Zurich (IBM Research GmbH), 8803 Rü schlikon, Switzerland. Для получения дополнительной информации обращайтесь по адресу: lrsc@zurich.ibm.com.

Оглавление

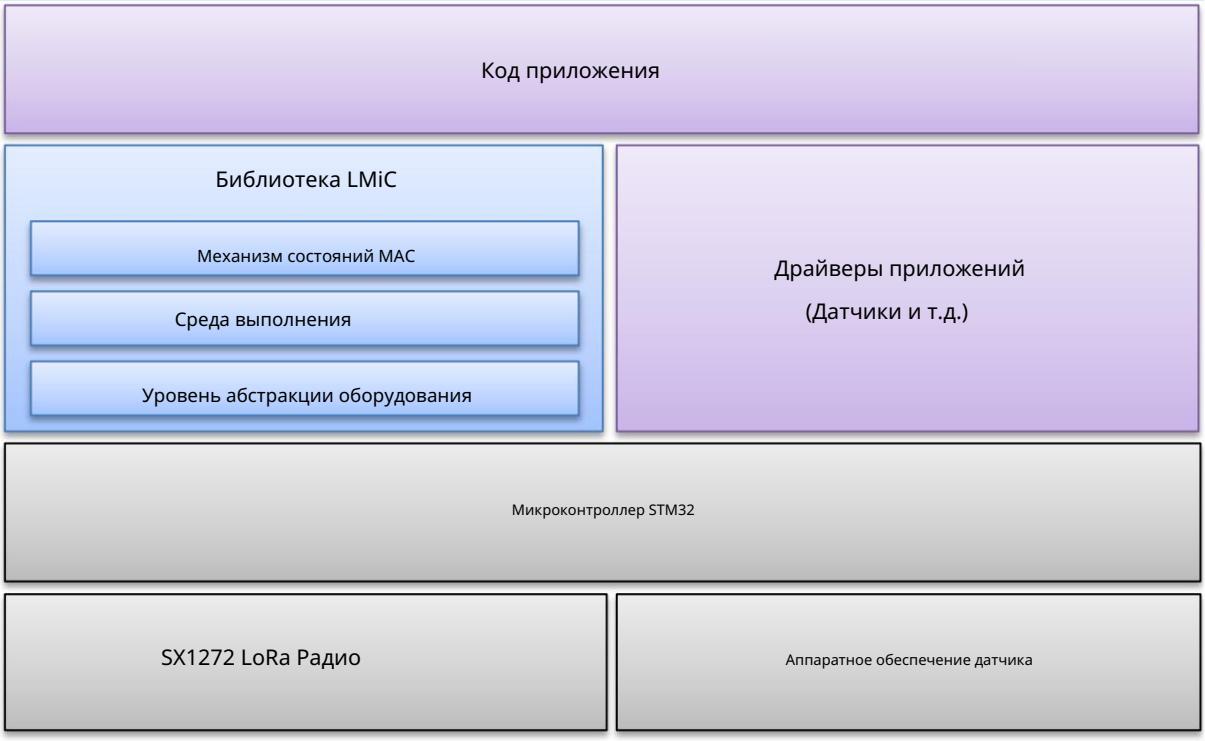
1. Введение

Библиотека IBM LoRaWAN C (LMiC) — это переносимая реализация спецификации LoRa™ MAC для языка программирования C. Она поддерживает варианты спецификации EU-868 и US-915 и может работать с устройствами классов A и B. Библиотека заботится обо всех логических состояниях MAC и ограничениях по времени и управляет радио SEMTECH SX1272. Таким образом, приложения могут свободно выполнять другие задачи, а библиотека гарантирует соответствие протоколу. Чтобы обеспечить соответствие спецификации и связанным с ней правилам, механизм состояний был протестирован и проверен с использованием среды логического моделирования. Библиотека была тщательно спроектирована для точного соответствия ограничениям по времени протокола MAC и даже для учета возможных отклонений часов при расчетах по времени.

Приложения могут получать доступ ко всем функциям и настраивать их с помощью простой модели программирования на основе событий и не должны иметь дело с деталями, специфичными для платформы, такими как обработчики прерываний. Используя тонкий уровень аппаратной абстракции (HAL), библиотеку можно легко переносить на новые аппаратные платформы. Для платформы STM32/Cortex-M3 поставляется эталонная реализация HAL, а общий объем кода всех компонентов на этой платформе составляет менее 20 КБ.

В дополнение к предоставленной библиотеке LMiC реальному приложению также нужны драйверы для датчиков или другого оборудования, которым оно хочет управлять. Эти драйверы приложений выходят за рамки этого документа, и их код не будет предоставлен IBM.

Рисунок 1. Компоненты прикладного устройства



Высокоуровневый вид всех компонентов устройства приложения. Поверх микроконтроллера STM32 с подключенным радио SX1272 и другим сенсорным оборудованием работает библиотека LMiC и код приложения.

2. Модель программирования и API

Библиотека LMiC доступна через набор функций API, функции времени выполнения, функции обратного вызова и глобальную структуру данных LMIC. Интерфейс определен в одном заголовочном файле «lmic.h», который должны включать все приложения.

```
#включить «lmic.h»
```

Для идентификации версии библиотеки LMiC в этом заголовочном файле определены две константы.

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

2.1 Модель программирования

Библиотека LMiC предлагает простую модель программирования на основе событий, где все события протокола отправляются в функцию обратного вызова onEvent() приложения (см. 2.3.4). Чтобы освободить приложение от деталей, таких как тайминги или прерывания, библиотека имеет встроенную среду выполнения, которая заботится об очередях таймеров и управлении заданиями.

2.1.1 Прикладные задачи

В этой модели весь код приложения выполняется в так называемых заданиях, которые выполняются в главном потоке функцией планировщика времени выполнения os_runloop() (см. 2.2.4). Эти задания приложения кодируются как обычные функции C и могут управляться с помощью функций времени выполнения, описанных в разделе 2.2. Для управления заданиями требуется дополнительная структура управления для каждого задания osjob_t, которая идентифицирует задание и хранит контекстную информацию. Задания не должны быть долго выполняющимися , чтобы обеспечить бесперебойную работу!
Они должны обновлять только состояние и планировать действия, которые вызовут новые обратные вызовы заданий или событий.

2.1.2 Основной цикл событий

Все, что нужно сделать приложению, это инициализировать среду выполнения с помощью функции os_init() и запустить функцию планировщика заданий os_runloop(), которая не возвращает результат. Для того чтобы загрузить действия протокола и сгенерировать события, необходимо настроить начальное задание. Поэтому запланировано задание запуска
с помощью функции os_setCallback().

```
пустая основная () {
    osjob_t initjob;

    // инициализация среды выполнения
    os_init();
    // настройка начального задания
    os_setCallback(&initjob, initfunc);
    // выполнение запланированных заданий и событий
    os_runloop();
    // (не достигнуто)
}
```

Код запуска, показанный в функции initfunc() ниже, инициализирует MAC и начинает подключение к сети.

```
// начальная работа
static void initfunc (osjob_t* j) {
        // сбросить состояние MAC
        LMIC_reset();
        // начать присоединение
        LMIC_startJoining();
        // инициализация выполнена - будет вызван обратный вызов onEvent()...
}
```

Функция initfunc() вернется немедленно, а функция обратного вызова onEvent() будет вызвана планировщиком позже для событий EV_JOINING, EV_JOINED или EV_JOIN_FAILED.

2.2 Функции времени выполнения

Функции времени выполнения, упомянутые выше, используются для управления средой времени выполнения. Это включает инициализацию, планирование и выполнение заданий времени выполнения.

2.2.1 void os_setCallback (osjob_t* задание, osjobcb_t cb)

Подготовить немедленно запускаемую задачу. Эту функцию можно вызвать в любое время, в том числе из контекстов обработчика прерываний (например, если стало доступно новое значение датчика).

2.2.2 void os_setTimedCallback (osjob_t* задание, ostime_t время, osjobcb_t cb)

Запланировать выполнение задания по времени в указанную временную метку (абсолютное системное время). Эту функцию можно вызвать в любое время, в том числе из контекстов обработчика прерываний.

2.2.3 void os_clearCallback (osjob_t* задание)

Отменить задание времени выполнения. Ранее запланированное задание времени выполнения удаляется из очередей таймера и выполнения. Задание идентифицируется адресом структуры задания. Функция не оказывает никакого эффекта, если указанное задание еще не запланировано.

2.2.4 void os_runloop ()

Выполнять задания времени выполнения из таймера и из очередей выполнения. Эта функция является основным диспетчером действий. Она не возвращает управление и должна выполняться в основном потоке.

2.2.5 ostime_t os_getTime ()

Запрос абсолютного системного времени (в тиках).

2.3 Обратные вызовы приложений

Библиотека LMiC требует, чтобы приложение реализовывало несколько функций обратного вызова. Эти функции будут вызываться механизмом состояний для запроса специфической для приложения информации и для доставки событий состояния в приложение.

2.3.1 void os_getDevEui (u1_t* buf)

Реализация этой функции обратного вызова должна предоставить EUI устройства и скопировать его в указанный буфер. EUI устройства имеет длину 8 байт и хранится в формате little-endian, то есть, наименее значимый байт первым (LSBF).

2.3.2 void os_getDevKey (u1_t* buf)

Реализация этой функции обратного вызова должна предоставить криптографический ключ приложения, специфичный для устройства, и скопировать его в указанный буфер. Ключ приложения, специфичный для устройства, представляет собой 128-битный ключ AES (длиной 16 байт).

2.3.3 void os_getArtEui (u1_t* buf)

Реализация этой функции обратного вызова должна предоставить EUI приложения и скопировать его в указанный буфер. EUI приложения имеет длину 8 байт и хранится в формате little-endian, то есть, наименее значимый байт первым (LSBF).

2.3.4 void onEvent (ev_t ev)

Реализация этой функции обратного вызова может реагировать на определенные события и запускать новые действия на основе события и состояния LMIC. Обычно реализация обрабатывает интересующие ее события и планирует дальнейшие действия протокола с использованием API LMIC. Будут сообщены следующие события:

- EV_JOINING
  Узел начал подключаться к сети.
- EV_JOINED
  Узел успешно подключился к сети и теперь готов к обмену данными.
- EV_JOIN_FAILED
  Узел не смог подключиться к сети (после повторной попытки).
- EV_REJOIN_FAILED
  Узел не присоединился к новой сети, но все еще подключен к старой сети.
- EV_TXCOMPLETE
  Данные, подготовленные через LMIC_setTxData(), были отправлены, и в конечном итоге данные нисходящего потока были получены в ответ. Если было запрошено подтверждение, то подтверждение было получено.
- EV_RXCOMPLETE
  Получены данные по нисходящему течению.
- EV_SCAN_TIMEOUT
  После вызова LMIC_enableTracking() в течение интервала маяка не было получено ни одного сигнала.
  Необходимо перезапустить отслеживание.
- EV_BEACON_FOUND
  После вызова LMIC_enableTracking() был получен первый маяк в пределах интервала маяка.
- EV_BEACON_TRACKED
  Следующий маяк был получен в ожидаемое время.
- EV_BEACON_MISSED
  В ожидаемое время сигнал маяка не был получен.

- EV_LOST_TSYNC

  Маяк был пропущен неоднократно, и синхронизация времени была потеряна. Трекинг или пингование необходимо перезапустить.

- EV_RESET

  Сессия сброшена из-за сброса счетчиков последовательностей. Сеть будет автоматически переподключена для получения новой сессии.

- EV_LINK_DEAD

  Подтверждение от сетевого сервера не поступало в течение длительного периода времени.
  Передачи все еще возможны, но их прием ненадежен.

Подробную информацию о конкретных событиях можно получить из глобальной структуры СНСД, описанной в следующем разделе.

2.4 Структура стран с низким и средним уровнем дохода

Вместо передачи многочисленных параметров туда и обратно между API и функциями обратного вызова, информация о состоянии протокола может быть доступна через глобальную структуру LMIC, как показано ниже. Все поля, кроме явно указанных ниже, доступны только для чтения и не должны изменяться.

```
структура lmic_t {
        y1_т                кадр[MAX_LEN_FRAME];
        y1_т                dataLen; // 0 нет данных или данные нулевой длины, >0 байт данных
        y1_т                dataRequest; // 0 или начало данных (dataBeg-1 — это порт)

        y1_т                txCnt;
        y1_т                txrxFlags; // флаги транзакции (комбинация TX-RX)

        y1_т                pendTxPort;
        y1_т                pendTxConf; // подтвержденные данные
        y1_т                pendTxLen;
        y1_т                pendTxData[MAX_LEN_PAYLOAD];

        u1_t bcnChnl;
        u1_t bcnRxsyms;
        ostime_t bcnRxtime;
        bcninfo_t bcninfo; // Последняя полученная информация о маяке

        ...
        ...
};
```

В этом документе не описывается вся структура подробно, поскольку большинство полей структуры LMIC используется только внутри. Наиболее важными полями для проверки при приеме (событие EV_RXCOMPLETE или EV_TXCOMPLETE) являются txrxFlags для информации о состоянии и frame[] и dataLen / dataBeg для полученных данных полезной нагрузки приложения. Для передачи данных наиболее важными полями являются pendTxData[], pendTxLen, pendTxPort и pendTxConf, которые используются в качестве входных данных для функции API LMIC_setTxData() (см. 2.5.11).

Для событий EV_RXCOMPLETE и EV_TXCOMPLETE необходимо оценить поле txrxFlags и определить следующие флаги:

- TXRX_ACK: подтверждено, что кадр UP был подтвержден (взаимоисключающее с TXRX_NACK)
- TXRX_NACK: подтвержденный UP-кадр не был подтвержден (взаимоисключающее с TXRX_ACK)
- TXRX_PORT: поле порта содержится в полученном кадре

- TXRX_DNW1: получено в первом слоте DOWN (взаимоисключающее с TXRX_DNW2)
- TXRX_DNW2: получено во втором слоте DOWN (взаимоисключающее с TXRX_DNW1)
- TXRX_PING: получено в запланированном слоте приема

Для события EV_TXCOMPLETE поля имеют следующие значения:

Полученный кадр	LMIC.txrxФлаги						LMIC.dataLen	LMIC.dataBeg
	ACK	NACK	PORT		DNW1	DNW2 ПИНГ		
ничего	0	0	0	0	0	0	0	0
пустой порт кадра	x	x	0	x	x	0	0	x
только	x	x	1	x	x	0	0	x
порт+полезная нагрузка	x	x	1	x	x	0	x	x

Для события EV_RXCOMPLETE поля имеют следующие значения:

Полученный кадр	LMIC.txrxФлаги						LMIC.dataLen	LMIC.dataBeg
	ACK	NACK	PORT		DNW1	DNW2 ПИНГ		
пустой порт кадра	0	0	0	0	0	1	0	x
только	0	0	1	0	0	1	0	x
порт+полезная нагрузка	0	0	1	0	0	1	x	x

2.5 Функции API

Библиотека LMiC предлагает набор функций API для управления состоянием MAC и запуска действий протокола.

2.5.1 недействительный LMIC_reset ()

Сбросьте состояние MAC. Сеанс и ожидающие передачи данных будут отменены.

2.5.2 bit_t LMIC_startJoining ()

Немедленно начать присоединение к сети. Будет вызвано неявно другими функциями API, если сеанс еще не установлен. Будут сгенерированы события EV_JOINING и EV_JOINED или EV_JOIN_FAILED.

2.5.3 недействительный LMIC_tryRejoin ()

Проверьте, нет ли поблизости других сетей, к которым можно присоединиться. Сеанс с текущей сетью сохраняется, если не найдена новая сеть. Будут сгенерированы события EV_JOINED или EV_REJOIN_FAILED.

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

Установите статические параметры сеанса. Вместо динамического установления сеанса путем присоединения к сети можно предоставить предварительно вычисленные параметры сеанса. Для возобновления сеанса с предварительно вычисленными параметрами счетчики последовательности кадров (LMIC.seqnoUp и LMIC.seqnoDn) должны быть восстановлены до последних значений.

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

Создайте новый диапазон с указанными свойствами мощности передачи и рабочего цикла (1/txcap) .

2.5.6 bit_t LMIC_setupChannel (канал u1_t , частота u4_t , карта drmap u2_t , диапазон s1_t)

Создать новый канал в заданном диапазоне, используя указанную частоту и позволяя использовать скорости передачи данных, определенные в битовой маске скорости передачи данных (1 << DRx).

2.5.7 void LMIC_disableChannel (канал u1_t)

Отключить указанный канал.

2.5.8 void LMIC_setAdrMode (bit_t включен)

Включить или отключить адаптацию скорости передачи данных. Следует отключить, если устройство мобильное.

2.5.9 void LMIC_setLinkCheckMode (bit_t включен)

Включить/отключить проверку ссылок. Режим проверки ссылок включен по умолчанию и используется для периодической проверки сетевого подключения. Должен вызываться только в случае установления сеанса.

2.5.10 void LMIC_setDrTxpow (dr_t dr, s1_t txpow)

Установите скорость передачи данных и мощность передачи. Следует использовать только если отключена адаптация скорости передачи данных.

2.5.11 недействительный LMIC_setTxData ()

Подготовить передачу данных вверх по течению в ближайшее возможное время. Предполагается, что pendTxData, pendTxLen, pendTxPort и pendTxConf уже установлены. Данные длиной LMIC.pendTxLen из массива LMIC.pendTxData[] будет отправлено в порт LMIC.pendTxPort. Если LMIC.pendTxConf имеет значение true, будет запрошено подтверждение сервера. Событие EV_TXCOMPLETE будет сгенерировано после завершения транзакции, т. е. после отправки данных и получения возможных данных down или запрошенного подтверждения.

2.5.12 int LMIC_setTxData2 (порт u1_t , данные xref2u1_t , dlen u1_t , подтверждено u1_t)

Подготовьте передачу данных вверх по течению в следующее возможное время. Удобная функция для LMIC_setTxData(). Если данные равны NULL, будут использоваться данные в LMIC.pendTxData[].

2.5.13 недействительный LMIC_clrTxData ()

Удалить данные, ранее подготовленные для передачи по восходящему течению.

2.5.14 bit_t LMIC_enableTracking (u1_t tryBcnInfo)

Включить отслеживание маяков. Значение 0 для tryBcnInfo указывает на немедленное начало сканирования маяков. Ненулевое значение указывает количество попыток запроса сервера для точного времени прибытия маяков. Запросы будут отправлены в следующих кадрах восходящего потока (кадр не будет сгенерирован). Если ответ не получен, будет запущено сканирование. События EV_BEACON_FOUND или EV_SCAN_TIMEOUT будут сгенерированы для первого маяка, а события EV_BEACON_TRACKED, EV_BEACON_MISSED или EV_LOST_TSYNC будут сгенерированы для последующих маяков.

2.5.15 недействительный LMIC_disableTracking ()

Отключить отслеживание маяка. Маяк больше не будет отслеживаться, и, следовательно, пингование также будет отключено.

2.5.16 void LMIC_setPingable (u1_t intvExp)

Включить пинг и установить интервал прослушивания нисходящего потока. Пинг будет включен со следующим восходящим потоком кадр (кадр не будет сгенерирован). Интервал прослушивания составляет 2^intvExp секунд, допустимые значения для intvExp 0-7. Эта функция API требует действительного сеанса, установленного с сетевым сервером либо через функции LMIC_startJoining(), либо LMIC_setSession() (см. разделы 2.5.2 и 2.5.4). Если отслеживание маяка еще не включено, сканирование будет запущено немедленно. Чтобы избежать сканирования, маяк можно найти более эффективно, выполнив предшествующий вызов LMIC_enableTracking() с ненулевым параметром. В дополнение к событиям, упомянутым для LMIC_enableTracking(), событие EV_RXCOMPLETE будет генерироваться всякий раз, когда в слоте ping будут получены данные нисходящего потока.

2.5.17 недействительный LMIC_stopPingable ()

Остановить прослушивание нисходящих данных. Периодический прием отключен, но маяки все равно будут отслеживаться. Чтобы остановить отслеживание, маяку требуется вызвать LMIC_disableTracking().

2.5.18 недействительный LMIC_sendAlive ()

Отправьте один пустой восходящий MAC-кадр как можно скорее. Может использоваться для сигнализации жизнеспособности или для передачи ожидающих MAC-опций, а также для открытия окна приема.

2.5.19 недействительный LMIC_shutdown ()

Остановить всю активность MAC. После этого MAC необходимо сбросить с помощью вызова LMIC_reset() и инициировать новые действия протокола.

3. Уровень абстракции оборудования

Библиотека LMiC разделена на большую часть переносимого кода и небольшую часть, специфичную для конкретной платформы. Реализуя функции этого уровня аппаратной абстракции с указанной семантикой, библиотеку можно легко переносить на новые аппаратные платформы.

3.1 Интерфейс HAL

Должны поддерживаться следующие группы аппаратных компонентов:

- Для управления антенным переключателем радиоприемника (RX и TX), выбор микросхемы SPI (NSS) и линия сброса (RST).
- Для обнаружения передатчика и приемника радиостанции в режиме ввода необходимы три цифровые линии ввода/вывода. состояния (DIO0, DIO1 и DIO2).
- Для чтения и записи регистров радиостанции необходим модуль SPI.
- Для точной регистрации событий и планирования новых протокольных действий необходим таймер.
- Контроллер прерываний необходим для пересылки прерываний, генерируемых цифровыми входными линиями.

В этом разделе описывается функциональный интерфейс, необходимый для доступа к этим аппаратным компонентам:

3.1.1 void hal_init ()

Инициализируйте уровень абстракции оборудования. Настройте все компоненты (IO, SPI, TIMER, IRQ) для дальнейшего использования с функциями hal_xxx().

3.1.2 void hal_failed ()

Выполнить действие «фатального сбоя». Эта функция будет вызываться утверждениями кода при фатальных условиях. Возможные действия: ОСТАНОВКА или перезагрузка.

3.1.3 void hal_pin_rxtx (значение u1_t)

Управляйте цифровыми выходными контактами RX и TX (0=прием, 1=передача).

3.1.4 void hal_pin_nss (значение u1_t)

Управляйте цифровым выходным контактом NSS (0=низкий/выбран, 1=высокий/не выбран).

3.1.5 void hal_pin_rst (значение u1_t)

Управление выводом RST радио (0=низкий, 1=высокий, 2=плавающий)

3.1.6 void radio_irq_handler (u1_t dio)

Три входные линии DIO0, DIO1 и DIO2 должны быть настроены на запуск прерывания по нарастающему фронту, а соответствующие обработчики прерываний должны вызывать функцию radio_irq_handler() и передавать линию, сгенерировавшую прерывание, в качестве аргумента (0, 1, 2).

3.1.7 u1_t hal_spi (u1_t outval)

Выполнить 8-битную транзакцию SPI. Записать заданный байт outval в радио, прочитать байт из радио и вернуть значение.

3.1.8 u4_t hal_ticks ()

Возвращает 32-битное системное время в тиках.

3.1.9 void hal_waitUntil (время u4_t)

Ожидание в режиме занятости до достижения указанной временной метки (в тиках).

3.1.10 u1_t hal_checkTimer (u4_t targettime)

Проверить и перемотать таймер на заданное targettime. Возвратить 1, если targettime близко (не стоит программировать таймер). В противном случае перемотать таймер на точное targettime или на полный период таймера и вернуть 0. Единственное действие, которое требуется при достижении targettime , — это пробуждение ЦП из возможных состояний сна.

3.1.11 void hal_disableIRQs ()

Отключить все прерывания ЦП. Может вызываться вложенно. Но всегда будет сопровождаться соответствующим вызовом hal_enableIRQs().

3.1.12 void hal_enableIRQs ()

Включить прерывания ЦП. При вложенном вызове только самый внешний вызов должен фактически включить прерывания.

3.1.13 недействительный hal_sleep ()

Спать до прерывания. Предпочтительно, чтобы компоненты системы были переведены в режим пониженного энергопотребления перед сном и повторно инициализированы после сна.

3.2 Реализация эталонного HAL для STM32/Cortex-M3

Исходный код библиотеки LMiC включает в себя эталонную реализацию HAL для платформы STM32/Cortex-M3. Эта реализация демонстрирует требуемую семантику интерфейса функции HAL. Для краткости она максимально упрощена и не оптимизирована (например, по энергопотреблению). Здесь мы опишем аппаратные ресурсы, используемые этой реализацией.

Приложения, использующие библиотеку, должны знать об использовании этих ресурсов и не должны вмешиваться в них! Либо приложения должны использовать другие ресурсы, доступные на платформе, либо им необходимо изменить реализацию HAL и мультиплексировать доступ к требуемым ресурсам!

3.2.1 Выходные линии ввода/вывода

Для управления радио используются следующие общие выходные линии.

Функция	GPIO
НСС	ПБ 0
Техас	ПА 4
РХ	ПК 13
РСТ	ПА 2

3.2.2 Входные линии ввода/вывода

Следующие общие входные линии используются для отслеживания состояния передатчика и приемника. Эти линии запрограммированы на генерацию прерываний по переднему фронту (см. разделы 3.1.6 и 3.2.5).

Функция	GPIO
ЧАСТЬ 0	ПБ 1
ЧАСТЬ 1	ПБ 10
ЧАСТЬ 2	ПБ 11

3.2.3 ИСП

Периферийное устройство SPI1 подключается к радио, как показано в таблице ниже.

Функция	GPIO
ССК	ПА 5
МИСО	ПА 6
ДЕНЬГИ	ПА 7

3.2.4 Таймер

Периферийное устройство TIMER 9 используется для обеспечения тактовой частоты 32 кГц и генерации прерываний компаратора для запланированных действий протокола.

3.2.5 Прерывание

Один обработчик прерываний EXTI используется для обработки всех групп прерываний внешней линии ввода-вывода (0, 1, 2, 3, 4, 5-9, 10-15). Обработчик EXTI проверяет источник прерывания и в конечном итоге вызывает radio_irq_handler().

Обработчик прерываний TIMER 9 обновляет тики системных часов при перевороте счетчика. Обработчику не нужно выполнять никаких специальных действий, когда прерывание инициируется компаратором. Достаточно, чтобы ЦП вышел из спящего режима, а среда выполнения LMiC могла проверить наличие ожидающих действий.

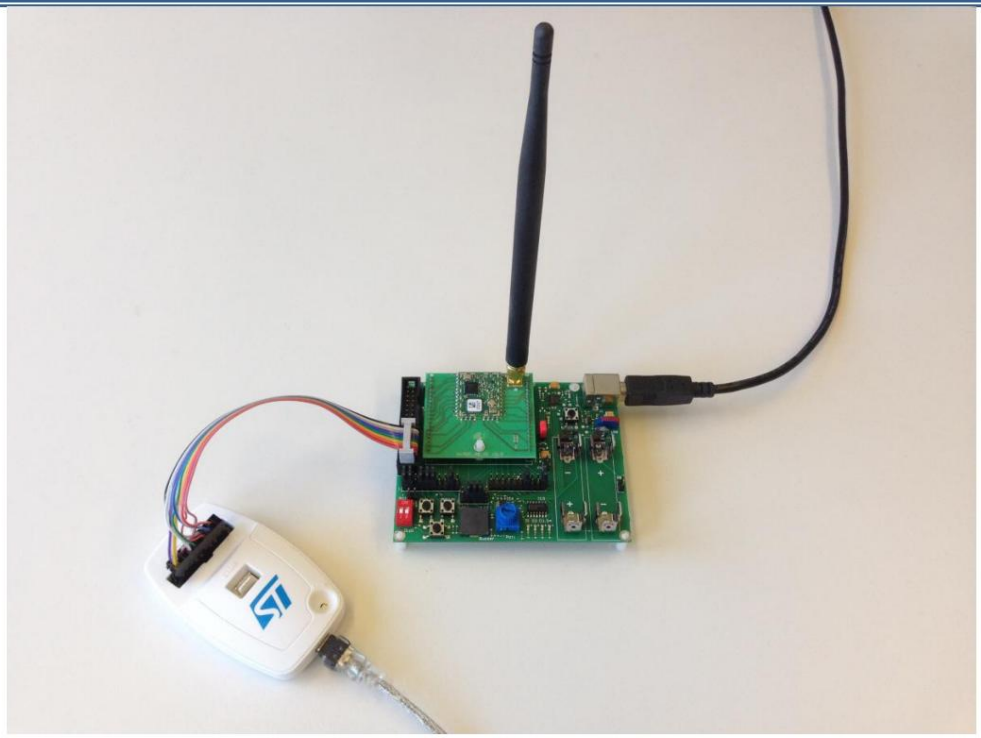
4. Примеры

Набор примеров представлен для демонстрации того, как типичные приложения узлов могут быть реализованы всего несколькими строками кода с использованием библиотеки LMiC. Поставляемые примеры готовы к запуску на демонстрационной плате стартового комплекта радио IMST / WiMOD LoRa™.

Примеры могут быть построены с использованием различных наборов инструментов компилятора, а файлы makefile предоставляются для IAR, Keil. и GCC. Кроме того, в каталоге примеров ZIP-файла предоставлены файлы проектов для интегрированных сред разработки IAR Workbench и Keil µVision .
Примечание: все проекты должны иметь следующие определения препроцессора, установленные в параметрах проекта:
CFG_eu868, CFG_wimod_board, CFG_sx1272_radio

В дополнение к управлению радио с использованием API LMiC, примеры написаны для предоставления локальной обратной связи через светодиод и последовательную консоль с использованием библиотеки отладки, описанной в разделе 4.8. Демонстрационная плата сделает последовательную консоль доступной как «USB Serial Port» на подключенном ПК, а вывод функций отладки можно легко просмотреть с помощью терминального приложения по вашему выбору. Параметры связи: 115200 бит/с 8/N/1.

Рисунок 2. Стартовый комплект радиостанции IMST/WiMOD LoRa™



Примеры были протестированы на IMST WiMOD SK-iM880A.

Для краткости только соответствующие части кода включены в фрагменты, показанные для каждого примера в этом разделе. В большинстве случаев это функция обратного вызова onEvent() приложения плюс некоторый служебный клей, содержащийся в файле main.c примера.

4.1 Пример 1: привет

Первый пример (привет) можно использовать для проверки работоспособности среды разработки и правильности подключения всех компонентов. В примере не используется радио, а только функции времени выполнения и отладочная библиотека для периодической записи значения счетчика на последовательную консоль и мигания светодиода.

```
// прилавок
статическое целое число cnt = 0;

// регистрировать текст в USART и переключать светодиод
static void initfunc (osjob_t* задание) {
        // скажи привет
        debug_str("Привет, мир!\r\n");
        // счетчик журнала
        debug_val("cnt = ", cnt);
        // переключить светодиод
        debug_led(++cnt & 1);
        // перепланировать задание каждую секунду
        os_setTimedCallback(job, os_getTime()+sec2osticks(1), initfunc);
}
```

Если все настроено правильно и программа выполняется, вы должны увидеть мигание светодиода с интервалом в одну секунду и следующий вывод на терминале:

```
============== ОТЛАДКА НАЧАЛАСЬ ==============
Привет, мир!
кнт = 00000000
Привет, мир!
кнт = 00000001
Привет, мир!
кнт = 00000002
. . . .. .
```

4.2 Пример 2: присоединение

Следующий пример (присоединение) можно использовать для проверки работоспособности радио и корректности настроек узла, соответствующих вашей сетевой инфраструктуре. Чтобы пример работал, обратные вызовы приложения os_getArtEui(), os_getDevEui() и os_getDevKey() должны возвращать правильные значения для идентификатора маршрутизатора приложения, идентификатора устройства и ключа устройства!

```
статический osjob_t blinkjob;
статический u1_t ledstate = 0;

статическая пустота мигающая функция (osjob_t* j) {
        // переключить светодиод
        ledstate = !ledstate;
        debug_led(ledstate);
        // перепланировать задание мигания
        os_setTimedCallback(j, os_getTime()+ms2osticks(100),blinkfunc);
}

void onEvent (ev_t ev) {
```

```
        debug_event(ev);

        переключатель(ев) {

            // начинаем присоединение к сети
            case EV_JOINING: //
                    начинаем мигать
                    blinkfunc(&blinkjob); break;


            // сеть подключена, сеанс установлен case
            EV_JOINED: // отмена
                    задания мигания
                    os_clearCallback(&blinkjob); // включение
                    светодиода
                    debug_led(1); //
                    (не планировать никаких новых действий) break;

        }
}
```

При выполнении светодиод должен начать быстро мигать, и примерно через пять секунд (если сеть успешно подключена) он должен загореться постоянно. Выход на терминале должен быть JOINING в начале, и примерно через пять секунд JOINED.

```
============== ОТЛАДКА НАЧАЛАСЬ ==============
ПРИСОЕДИНЕНИЕ
ПРИСОЕДИНИЛСЯ
```

4.3 Пример 3: передача

После присоединения к сети пример передачи начнет отправлять восходящие кадры, содержащие один байт с последним известным отношением сигнал/шум. После завершения передачи будет немедленно запланирована новая передача, и, следовательно, кадры будут отправляться с максимальной скоростью, разрешенной рабочим циклом. Если нисходящие данные были получены в слоте приема после передачи, они будут записаны в консоль.

```
void onEvent (ev_t ev)
        { debug_event(ev);

        переключатель(ев) {

            // сеть подключена, сеанс установлен case EV_JOINED:
            debug_val("netid = ",
                    LMIC.netid); goto tx;


            // запланированные данные отправлены (опционально данные
            получены) case
                    EV_TXCOMPLETE: if(LMIC.dataLen) { // данные получены в слоте приема после
                            передачи debug_buf(LMIC.frame+LMIC.dataBeg, LMIC.dataLen);
                    }
                техас:
                    // немедленно подготовить следующую передачу
                    LMIC.frame[0] = LMIC.snr;
```

```
                // запланировать передачу (порт 1, данные 1, подтверждение не запрошено)
                LMIC_setTxData2(1, LMIC.frame, 1, 0); // (будет
                 отправлено, как только позволит рабочий цикл) break;

        }
}
```

Кадры восходящего потока должны быть доставлены на маршрутизатор приложения, а на консоли узла должен отображаться следующий вывод:

```
============== ОТЛАДКА НАЧАЛАСЬ ==============
ПРИСОЕДИНЕНИЕ
ПРИСОЕДИНИЛСЯ
сетевой идентификатор = 00000001
TXCOMPLETE
TXCOMPLETE
TXCOMPLETE
```

4.4 Пример 4: периодический

Следующий пример (периодический) будет периодически сообщать значение датчика в сеть. После присоединения запускается задание, которое считывает датчик, подготавливает восходящую передачу со значением датчика и перепланирует задание для повторного выполнения через 60 секунд. Для реализации датчика в этом примере используются специфичные для платформы функции initsensor() и readsensor(), содержащиеся в файле sensor.c.
Датчик образца просто считывает положение «DIP-переключателя 1» на демонстрационной плате (PB 12) как 1-битное значение.

```
статический osjob_t reportjob;

// сообщать значение датчика каждую минуту
static void reportfunc (osjob_t* j) { // считать датчик
        u2_t val =
        readsensor(); debug_val("val = ",
        val); // подготовить и запланировать
        данные для передачи LMIC.frame[0] = val << 8; LMIC.frame[1] =
        val; LMIC_setTxData2(1, LMIC.frame,
        2, 0); // (порт 1, 2 байта,
        неподтверждено) // перепланировать задание через 60 секунд os_setTimedCallback(j, os_getTime()
        +sec2osticks(60), reportfunc);

}

void onEvent (ev_t ev)
        { debug_event(ev);

        переключатель(ев) {

            // сеть подключена, сеанс установлен case EV_JOINED: //
            включить светодиод
                debug_led(1); //
                запустить
                периодическое задание датчика
                reportfunc(&reportjob); break;
```

```
        }
}
```

В зависимости от положения DIP-переключателя 1 этот пример должен генерировать вывод, аналогичный следующему:

```
============== ОТЛАДКА НАЧАЛАСЬ ==============
ПРИСОЕДИНЕНИЕ
ПРИСОЕДИНИЛСЯ
значение = 00000001
TXCOMPLETE
значение = 00000001
TXCOMPLETE
значение = 00000000
TXCOMPLETE
```

4.5 Пример 5: прерывание

В этом примере (прерывание) используется тот же датчик, что и в предыдущем примере, но он не считывает датчик периодически. Вместо этого он управляется прерываниями и отправляет значение датчика только тогда, когда датчик изменился. Определенный приложением обработчик прерываний был добавлен в файл sensor.c для запуска зарегистрированного обратного вызова задания при срабатывании прерывания:

```
// вызывается EXTI_IRQHandler //
(устанавливает опцию препроцессора CFG_EXTI_IRQ_HANDLER=sensorirq) void
sensorirq () { if((EXTI->PR
        & (1<<INP_PIN)) != 0) { // ожидание EXTI->PR = (1<<INP_PIN); //
                очистить irq // запустить функцию обратного
                 вызова приложения через 50 мс (устранение дребезга)
                 os_setTimedCallback(&irqjob, os_getTime()+ms2osticks(50), irqjob.func);
        }
}
```

4.6 Пример 6: маяк

Следующий пример (маяк) включает отслеживание маяка после присоединения к сети. Он управляет светодиодом в зависимости от событий TRACKED/MISSED в каждом периоде. Если маяк успешно отслежен, время GPS, содержащееся в маяке, регистрируется на консоли.

```
void onEvent (ev_t ev)
        { debug_event(ev);

        переключатель(ев) {

            // сеть подключена, сеанс установлен case
            EV_JOINED: //
                    включить режим отслеживания, начать сканирование...
                    LMIC_enableTracking(0);
                    debug_str("СКАНИРОВАНИЕ...\r\n");
                    break;

            // маяк найден при сканировании
            случая EV_BEACON_FOUND:
```

```
                // включить LEN
                debug_led(1);
                break;

        // маяк отслеживается в ожидаемое время case
        EV_BEACON_TRACKED:
                debug_val("GPS time = ", LMIC.bcninfo.time); // включаем LEN
                debug_led(1); break;




        // маяк пропущен в ожидаемое время case
        EV_BEACON_MISSED: //
                выключить LEN
                debug_led(0);
                break;
    }
}
```

В зависимости от качества приема вывод консоли должен выглядеть примерно так:

```
============== ОТЛАДКА НАЧАЛАСЬ ==============
ПРИСОЕДИНЕНИЕ
ПРИСОЕДИНИЛСЯ
СКАНИРОВАНИЕ...
МАЯК_НАЙДЕН
BEACON_TRACKED
Время GPS = 545CE201
BEACON_TRACKED
Время GPS = 545CE281
BEACON_TRACKED
Время GPS = 545CE301
```

4.7 Пример 7: пинг

Следующий пример (ping) присоединяется к сети и многократно прослушивает нисходящие данные. Это достигается путем включения режима ping на основе маяка с интервалом в две секунды. Вызов LMIC_setPingable() устанавливает режим ping локально и начинает сканирование на предмет маяка. После того, как первый маяк найден, необходимо отправить восходящий кадр (в данном случае пустой кадр через LMIC_sendAlive()) для передачи параметров MAC и уведомления сервера о режиме ping и интервале. Всякий раз, когда сервер отправляет нисходящие данные в один из слотов приема, запускается событие EV_RXCOMPLETE, и полученные данные можно оценить в поле кадра структуры LMIC. Пример кода регистрирует полученные данные на консоли и, в особом случае, когда получен ровно один байт, он управляет светодиодом в зависимости от полученного значения.

```
void onEvent (ev_t ev)
        { debug_event(ev);

        переключатель(ев) {

            // сеть подключена, сеанс установлен case
            EV_JOINED: //
                включить режим пингования, начать сканирование... //
                (установить локальную конфигурацию интервала пингования на 2^1 == 2 сек)
```

```
                LMIC_setPingable(1);
                debug_str("СКАНИРОВАНИЕ...\r\n");
                перерыв;

        // маяк найден при сканировании
        случай EV_BEACON_FOUND:
                // отправьте пустой кадр, чтобы уведомить сервер о режиме и интервале пинга!
                LMIC_sendAlive();
                перерыв;

        // кадр данных получен в слоте ping
        случай EV_RXCOMPLETE:
                // данные лог-фрейма
                debug_buf(LMIC.frame+LMIC.dataBeg, LMIC.dataLen);
                если (LMIC.dataLen == 1) {
                        // установить состояние светодиода, если получен ровно один байт
                        debug_led(LMIC.frame[LMIC.dataBeg] & 0x01);
                }
                перерыв;
    }
}
```

4.8 Пример 8: модем

Следующий пример (модем) представляет собой полноценное приложение модема и обеспечивает простой доступ к сетям LoRaWAN посредством высокоуровневых команд ASCII, обмениваемых через последовательный интерфейс. Из соображений сложности документация модема была перемещена в отдельный файл LMiC-Modem.pdf.

4.9 Отладочная библиотека

Для обеспечения локального текстового вывода примеров, показанных в этой главе, предусмотрена небольшая отладочная библиотека. Эта библиотека не требуется LMiC, но полезна для разработки и отладки. Функции библиотеки предлагают простую последовательную консольную регистрацию и доступ к светодиоду для диагностического вывода. Отладочная библиотека специфична для платформы и реализована для STM32/Cortex-M3.

4.9.1 void debug_init ()

Инициализируйте периферию, необходимую для функций отладки. USART1 и LED4 используются в эталонной реализации для STM32/Cortex-M3. Настройки последовательной связи: 115200 8/N/1.

Функция	GPIO
USART 1 TX	ПА 9
Светодиод 4	ПА 8

4.9.2 void debug_led (u1_t val)

Светодиод привода (0=выкл, 1=вкл).

4.9.3 void debug_char (u1_t c)

Вывод одного символа на последовательную консоль.

4.9.4 void debug_hex (u1_t b)

Запишите значение байта в виде двух шестнадцатеричных символов на последовательную консоль.

4.9.5 void debug_buf (const u1_t* buf, u2_t len)

Запись нескольких байтов в виде шестнадцатеричных символов, разделенных пробелами, на последовательную консоль.

4.9.6 void debug_uint (u4_t v)

Вывести 32-битное беззнаковое целое значение в виде восьми шестнадцатеричных цифр на последовательную консоль.

4.9.7 void debug_str (const u1_t* str)

Вывод произвольной строки с нулевым завершением на последовательную консоль.

4.9.8 void debug_event (целое_событие)

Запишите имя события, за которым следует «\r\n», на последовательную консоль.

4.9.9 void debug_val (const u1_t* метка, u4_t значение)

Строка метки журнала плюс шестнадцатеричное целое значение, за которым следует «\r\n» на последовательную консоль.

5. История релизов

Версия и дата	Описание
Версия 1.0 Ноябрь 2014 г.	Первоначальная версия.
Версия 1.1 Январь 2015 г.	Добавлен API LMIC_setSession(). Незначительные внутренние исправления.
Версия 1.2 Февраль 2015 г.	Добавлены API LMIC_setupBand(), LMIC_setupChannel(), LMIC_disableChannel(), LMIC_setLinkCheckMode(). Незначительные внутренние исправления.
В 1.4 Март 2015 г.	Измененный API: флаг индикатора порта в LMIC.txrxFlags был инвертирован (теперь TXRX_PORT, ранее TXRX_NOPORT). Внутренние исправления ошибок. Форматирование документа.
В 1.5 Май 2015 г.	Исправления ошибок и обновление документации.
В 1.6 Июль 2016	Изменена лицензия на BSD. Включено приложение модема (см. examples/modem и LMiC-Modem.pdf). Добавлены драйверы оборудования STM32 и периферийный код, специфичный для платы Blipper.