Wyres

LoWAPP Implementation Document

Année 2016

Nathan Olff
07/12/2016

This document

Objectives

This document describes the work plan for implementing “LoRa Wide Area Peer Protocol” (LoWAPP), a small scale communication protocol between LoRa based peer endpoints. It is based on the specification document *lowapp_FS_v1-8*.

Structure

The document presents the different steps of the implementation process for LoWAPP as well at the validation of this implementation using both real boards and simulated environment on a Linux.

Versions

Version	Author	Date	Changes
V0.1	N.Olff	08/08/2016	Creation
V0.2	N.Olff	09/08/2016	Update after planning meeting
V0.3	N.Olff	17/08/2016	Modify Gateway behaviour and add task planning
V0.4	N.Olff	01/12/2016	Update with the latest state machine
V0.5	N.Olff	07/12/2016	Update

Overview

LoWAPP is a LoWPAN type protocol focused on providing peer-to-peer interactions between connected end devices. The precise specification of the protocol is defined in a document called *lowapp_FS_v1-8*. This document describes the network architecture, the attributes of the end points, the way endpoints communicate with each other or with an external network (through an external gateway). It contains a state machine of the way the protocol works as well as the content of the messages sent through the LoRa radio (using Semtech SX1272).

Implementation requirements

The protocol implementation will be tested on a STM32L1XX MCU (ARM Cortex-M3). It should be easily portable to any MCU using a defined interface and a series of callbacks. It is a protocol destined to be used for the Internet of Things and should therefore be energy efficient.

Validation

In order to validate the protocol implementation, a simulated environment will be developed. It will be created before the actual implementation so that the implementation can be tested against the simulation during the development phase. Each device will be represented as a process running on a Linux desktop environment. The radio communication will be represented as a file stored on the filesystem. Each LoRa frequency will correspond to one file. The AT commands (normally sent via UART), will be replaced by standard console input and output.

Once the implementation is validated on the simulated environment, a second phase of validation will happen on the actual microcontrollers with the Semtech LoRa chips.

Simulated environment

The simulation environment will be running on a Linux OS. It will be developed using Ubuntu 16.04 LTS on a virtual machine.

It will consist of a single program, launched multiple times from multiple consoles, each instance representing one device.

Radio transmissions will be simulated using text file: an empty file on the filesystem corresponds to the preamble, a non-empty file corresponds to the actual transmission and the deletion of the file is the end of the transmission. A message is only received when the transmission is finished. Preamble duration will be customizable, whereas transmission duration will be calculated using the spreading factor, the size of the message and Semtech battery life calculator.

A simulation manager will also be developed, allowing us to run automated tests on several nodes with different configurations.

Initialisation

The initialisation of the protocol is done by registering a series of callbacks. All these functions are grouped into a `LOWAPP_SYS_IF_T` structure that is sent to the `lowapp_init` function for initialisation of the protocol. Here is the list of those functions, grouped by types of functions.

Timing functions

Timing functions are used to start or stop timers within the LoWAPP core. Three different timers are required by the LoWAPP core. It also include a function used to get a timestamp for time measuring.

- `SYS_getTimeMs`: Gives the current epoch time in ms (as known by the system).
- `SYS_initTimer, SYS_initTimer2, SYS_initRepetitiveTimer`: Initialise a timer with a callback to be called when it reaches its timeout. Three timers are used by the LoWAPP core.
- `SYS_setTimer, SYS_setTImer2, SYS_setRepetitiveTimer`: Start a timer for specific amount of time (in ms).
- `SYS_cancelTimer, SYS_cancelTimer2, SYS_cancelRepetitiveTimer`: Stop a running timer.

On the simulation, the `time.h` Linux library is used. On the hardware, we are using logic timers, processed by the RTC. Timestamps are retrieved using Linux epoch timestamp in ms.

I/O functions

Communication with the LoWAPP core is done through AT commands sent via UART.

- `SYS_cmdReponse`: Output data to the application

On the simulation, the UART is represented by console I/O. A thread is running to process the inputs and send them to the core when an end of line character is detected.

On the hardware, we would be using actual UART peripheral.

It is up the application layer to forward incoming UART commands to the LoWAPP core by calling the `lowapp_atcmd` function.

Configuration

Configuration values are stored in a temporary structure on the application side of the program. They can be set and retrieved using AT commands. When requested, these values can be written into persistent memory.

- `SYS_getConfig`: Get a device configuration value from the temporary configuration structure
- `SYS_setConfig`: Set a configuration value to the temporary configuration structure
- `SYS_writeConfig`: Write the current temporary configuration to persistent memory.
- `SYS_readConfig`: Load configuration from persistent memory to the temporary configuration structure.

On simulation, the configuration values are stored in device specific text files, with one value per line.

On the hardware, the configuration values are saved in EEPROM.

Radio related functions

Radio functions include all the communications with the Semtech SX1272 LoRa radio chip. On the simulation, we are reading and writing data into text files corresponding to a specific radio frequency. A Linux program called `inotify` is used to poll the filesystem and detect activity in the folder containing the radio text files.

- `SYS_radioInit`: Initialise the radio with the radio callbacks sent as parameters.
- `SYS_radioTx`: Send a frame to the radio for transmission.
- `SYS_radioRx`: Set the radio chip into reception mode.
- `SYS_radioCAD`: Start a Channel Activity Detection on the radio chip.
- `SYS_radioLBT`: Execute a Listen Before Talk operation and return to indicate if a symbol was found or not.
- `SYS_radioSetTxConfig`: Setup the TX configuration of the radio chip. This includes the transmit power, bandwidth, spreading factor, preamble...
- `SYS_radioSetRxConfig`: Setup the RX configuration of the radio chip. This includes the bandwidth, spreading factor, preamble…
- `SYS_radioTimeOnAir`: Computes the packet time on air for a given frame size.
- `SYS_radioSetChannel`: Set the frequency to use for radio transmissions and receptions.
- `SYS_radioSleep`: Put the radio into sleep mode after a transmission or reception is done.

The following functions have been added to allow for simpler way to modify a single configuration value of the radio without doing a full `SYS_radioSetTxConfig` or `SYS_radioSetRxConfig`.

- `SYS_radioSetPreamble`: Setup the preamble length to use for transmissions.
- `SYS_radioSetTxFixLen`: Set fixed length packet attribute for radio transmission.
- `SYS_radioSetRxFixLen`: Set fixed length packet attribute for radio reception. The expected size of the frame is sent as parameter.
- `SYS_radioSetTxTimeout`: Set the timeout value for radio transmission. If a launched transmission takes more that the timeout set by this functions, the radio will call the TxTimeout functions and go to sleep.

- `SYS_radioSetRxContinuous`: Change reception mode for radio reception. If this is true, the radio is kept running. In this case, the SYS_radioSleep needs to be called to stop the radio.
- `SYS_radioSetCallbacks`: Modify the radio callbacks used by the radio to notify the core of an end of transmission, end of reception or error/timeout issue.

On simulation, we are

Other functions

- `SYS_random`: Generate a random number using radio noise as a seed for initialising a random number generator used in the core.
- `SYS_delayMs`: Blocking delay function (in ms).

AT commands

AT commands are a set of commands allowing a user to directly interact with a device via a wired connection.

On the physical board, it uses the UART peripheral. On our simulated environment, it consists in reading and writing the standard input/output streams. A set of functions will be written to initialise, send, and receive data. These functions are called from the `LOWAPP_SYS_IF_T` functions.

Incoming AT commands will be managed by the UART peripheral. The UART receive interruption will store incoming commands into a fifo queue and send a RXAT event to the LoWAPP state machine. When the state machine is in idle state and receives that event, it will pop a value from the AT queue, parse it and interpret its content. A check for the AT queue will also be done directly at the entry of the idle state in case an RXAT event was sent while doing something else.

AT commands are grouped in three types of commands:

- Configuration: read and write characteristics of the devices (like deviceId or panId)
- Validation: get the current configuration of the device as well as radio-related statistics. They can also send ping messages or print statistics of received messages.
- Operation: send messages through the radio and change receive mode (push, pull, join, leave)

Transmit operations will consist in sending TXREQ event to the state machine with the message to send as parameter. We then wait for the state machine to raise a flag telling us if the message was sent correctly, if the receiver did send its ACK and if it perceived duplicate or missing frame.

Reception works in two ways: either we are in push mode, in which case the LoWAPP core will directly send incoming messages to the application as JSON using the `SYS_cmdResponse` function. The other mode is pull mode, in which the application sends an AT+POLLRX command to get the packets received since its last call.

Implementation of the protocol

The core of the LoWAPP protocol is implemented as a state machine. This state machine is described below, as well as in the LoWAPP specification document.

States

The states of the state machine are:

- Idle state
- Receiving state
- Skipping ACK state
- Transmitting state
- Waiting slot for transmitting ACK state
- Transmitting ACK state
- CAD state
- Transmitting state
- Wait before listening to ACK state
- Receiving ACK state
- Restart state

Transitions are done through the use of events (see next section).

We also have definitions of Linked Lists and Fixed Queues in *lowapp_core.h*. Functions to manipulate those lists and queues are present in the *lowapp_utils* folder.

The declaration of the device configuration is done as global variables in *lowapp_api.c* with:

- The radio channel id
- The radio spreading factor
- The gateway attribute
- The endpoint id
- The group id
- The preamble duration
- The AES encryption key

These configuration variables can be modified on the fly by AT commands. They can also be stored into persistent memory using AT&W command (in text files on the simulation, in EEPROM on the hardware)

Events

Events are used to communicate between the radio layer and the state machine but also to move from one state of the SM to another:

- STATE_ENTER: called every time we enter a state from another state.
- STATE_EXIT: called every time we leave a state
- TXREQ: A transmission is required either by the application or by the LoWAPP core itself.
- TXDONE: This event is sent by the radio layer when a transmission is finished.

- TXTIMEOUT: This event is sent by the radio layer when a transmission did not finish in the given timeout value configured.
- RXMSG: This event is sent when an incoming message is received by the radio layer and decoded. It contains the message itself as parameter.
- RXTIMEOUT: This event is sent by the radio layer when the radio was set in reception mode but no frame was received.
- RXERROR: This event is sent by the radio layer when the radio did not manage to receive a frame correctly.
- TXUNBLOCK: When sending a frame, the state machine raises a flag to disable transmission for a small period, corresponding to the transmission time + a random time between 0 and 1s. A timer is launched to clear this flag and post the TXUNBLOCK event after that period.
- RXAT: An AT command has been received and is waiting to be processed in the AT queue
- TIMEOUT: A timer has reached the end of its time slot.

State machine

The implementation is based on the state machine described in the specification, although with some modifications. A more detailed version of the state machine is available as a svg file joined to this document.

Each state of the state machine is represented by a function in the *lowapp_sm.c* file. Those functions take into parameter an event and return the new state the state machine will be after execution.

The state machine itself is an infinite loop in a function called `sm_run`. To start the state machine, the application calls a function called `lowapp_process`. The first step done in the loop is to pop the next event out of the event queue. If no event occurred, we should return to allow the application layer to sleep or to process other tasks. It is up to the application to call the `lowapp_process` function regularly to manage the incoming events.

If an event occurred, we check that the state is valid and execute the current state's function with the event as parameter. Once that function returns, we check if the new state (returned by it) is different from the current state. If so, the current and new state's functions are called respectively with a STATE_EXIT event and a STATE_ENTER event.

Idle state (IDLE)

The idle state has four events to manage:

- `STATE_ENTER`: We first check if the AT queue is empty. If a command is present, we parse, interpret, and respond to it (using existing lowapp_atcmd.c functions). Then we check if we are in PUSH mode (more on that later) and send the received messages to the UART if so. Finally, we check if a frame is pending transmission or if the TX queue contains an element. If so, we call the corresponding `tryTx` function (see below).
- `TXREQ`: We first check if transmission is allowed (`txBlock` flag is clear).
    - After that, we check if a frame is pending transmission. If so, we call the `tryTxFrame` function. This functions executes a radio LBT (Listen Before Talk). If the LBT did not find anything on the channel, the transmission is started. If it found something, a retry variables is incremented and compared to a maximum number of retries. If the retry is less that the max number of retry, the frame is put as pending transmission

and the transmission of blocked for some time using the `txBlock` flag. We then move over to the reception state to try and catch the incoming frame.
    - If no frame is pending transmission, we check the TX queue. If it contains at least one element, we call the `tryTxFromQueue` function that extracts that message and then call the `tryTxCurrent` function. This functions uses the message structure to build a corresponding LoWAPP frame, including the required header, encrypts the frame, and call the `tryTxFrame` function described above.
- `RXAT`: When an AT command is received by the UART peripheral, it adds the command to the AT queue and send an RXAT event to the state for processing of that command.
- `CADTIMEOUT`: This means that the repetitive timer has reached its end. It is therefore time to move to CAD state to check radio activity.

All other events will be discarded and will have no effect on the state machine.

Receive state (RXING)

When we enter the RXING state, a radio reception is launched using `SYS_radioRx`.

When a message is received by the radio layer, an RXMSG event is posted to the state machine, containing the received frame as parameter. The state then decodes the frame, decrypts its content with the AES key and check the CRC value. If the CRC matches the one included in the frame, the different values stored in the header of the frame (source id, destination Id…) are extracted and stored in a message structure.

If the message was a broadcast, it is added to the RX queue. Then we simply move back to the idle state as broadcast are not acknowledged.

If it was a unicast and the destination id matches the device id, we add the message to the RX queue, build a corresponding ACK using the different sequence numbers and move over the Wait slot for transmitting ACK state.

If the id does not match, we go to the skipping ack state.

Skipping ack state

This state just starts a timer and waits for it to reach its value. This is done to allow the expected receiver of the packet to send its ACK while avoiding clashes on the radio (no other transmission during that time). Once the timeout event occurs, we move back to idle state.

Transmitting ack state

When entering this state, we first set configure the radio to transmit ACK messages (fixed message length and shorter preamble). ACK have a more standard 8 symbols preamble instead of a 1s (or other) preamble.

This state waits for the transmission of the ACK to be done via a TXDONE event. This TXDONE event is sent by the radio layer via the registered `RadioEvent.TxDone` function after transmission. Once this event is received, the radio is configured back to standard message (long preamble and variable message length).

If the timer times out, we log the error and go back to idle state.

Transmitting state (TXING)

Transmitting state is enter after a call to `SYS_radioTx`.

Once the transmission is done, a `TXDONE` event is sent and received by this state. The corresponding sequence number is incremented.

If the message sent is a broadcast, we set the `txBlock` flag, set a timer to unblock transmission later, and move back to idle state.

If the message sent is a unicast, we set the `txBlock` flag and move to WAIT_BEFORE_LISTENING_FOR_ACK state.

Wait before listening for ack state

ACK are expected to be sent during a specific ACK slot. This slot starts 1s after the end of a transmission and is 1s long.

The Wait before listening for ACK state corresponds to the waiting period after a transmission done and before the start of that ACK slot. It is materialised with a simple timer and a wait on the TIMEOUT event. After that event is received, we move over to the Receiving ACK state.

Receiving ACK state

This state represents the ACK slot.

When entering the state, the radio is configured to receive ACK frames (with short preamble and fixed length). The radio is then set in reception mode with `SYS_radioRx`.

If a RXMSG event occurs, we check if it is an ack. If so, we update the sequence number and go back to idle. If not, we move back to idle mode and returns a “NOK TX” message to the application.

If a timeout occurs, we inform the application that the message was not acknowledged by the receiver. It is up to the application to decide if it wants to try sending it again.

In both cases, we set a timer for clearing the `txBlock` flag set earlier after a period of time presenting the transmission time of the original message + a random value between 0 and 1s.

CAD state

CAD, or Channel Activity Detection is a radio feature of the SX1272. It turns the radio on for a short period of time (1 symbol) to see if a preamble is being transmitted by another device.

In LoWAPP, the CAD state is entered periodically every one preamble (preamble being a customizable value). It executes the CAD with `SYS_radioCAD` function and then waits for the CADDONE event to indicate if a symbol was detected or not by the radio.

If the CAD found a symbol, we go to the Receiving state. If no CAD if found we go back to idle state.

Restart state

The restart state directly goes to idle state.

Frame format

Messages transmitted between nodes in a LoWAPP network have a specific format. They all contain 4 sections:

- A LoRa header: common to all types of messages, it includes the version of the LoWAPP protocol, the type of message, and the payload length
- A nonce: used in conjunction with the AES key for encryption
- The message, including type specific information like source id, destination id, sequence number, actual payload…
- A CRC16 used to validate the decryption of the packet

Message type

Two different message types are available in the code:

- A classic mode, used when LOWAPP_MSG_FORMAT_CLASSIC is defined. In this type, messages are sent as ASCII characters with the destination id as two hex digits, followed by a comma, and the message to transmit (eg: AT+SEND=A5,Hello World). Received messages are forwarded to the application as JSON strings.
- A special GPS format, used for communicating with an Android application via Bluetooth. When using the GPS format message, send requests are sent in a special binary after the AT+SEND= prefix.
    - B0: 45
    - B1: Message type:
        - 0 : Unicast or broadcast message
    - B2-B6: GPS latitude
    - B7-B10: GPS longitude
    - B1 : Destination id, 0xFF for broadcast
    - B12: Device id of the transmitter
    - B13-BN: Text message to send

  Received messages are forwarded to the application using the following format:
    - B0-B1: 45 02
    - B2: Number of GPS elements following
        - B(2+nx): Device Id of the source device
        - B(2+nx+1)-B(2+nx+4): GPS latitude
        - B(2+nx+5)-B(2+nx+8): GPS longitude
        - Repeated n times for each GPS element
    - For each received message:
        - B(2+9n): Length of the message, including length byte and source id byte
        - B(2+9n+1): deviceId of the source of the message
        - B(2+9n+2)-BN…: Text message received

  Example of received messages :

  4502
  02
  03 34352E31 352E3730
  01 00000001 00000002
  06 01 41424344 (ABCD)

07 03 3132333435 (12345)

Encryption

All frames send in LoWAPP are partially encrypted in AES 128bits. Only the type specific part of a frame is encrypted (between the nonce and the CRC16.

Encryption of a message is done using the output of a XOR operation between an AES 128 key (common to all nodes of the same group) and a 2-bytes random value called a nonce and sent in clear in the messages.

Gateway

Gateway functionality has not yet been implemented into this version of the protocol. There is no way for now to communicate between devices in different groups or to communicate with other types of networks.

The description below explains how the gateway functionality should be implemented in a later version of the protocol:

Each node has a gateway attribute which is a 32-bit mask. If a node is designated as gateway, the bit mask corresponds to the type of messages it can forward. If this attribute is 0, it means that the node is not a gateway.

If the device receives a gateway message, it checks that it can process that type of gateway message using its gateway attribute. If so, it calls the message is stored in the Rx queue like a standard message.

A gateway can manage several types of gateway messages (LoWAPP, LoRaWAN, IPv4, CoAP…) but only one gateway of each type can be present in the group. Because the gateway ACK the messages it receives, we cannot have multiple gateway forwarding the same type of message and a layer in the application managing duplicates.

It is up to the application layer to manage incoming messages from the outside that are destined to be sent to this group. One possible way to do this would be through a TXREQ.

Validation

Once the core of the protocol was developed, it has been validated using the simulated Linux environment.

A series of nodes can be created by executing several times the application. Each node has an additional uuid, used for naming the corresponding configuration file. This allows us to do several tests with the same configurations, as well as changing things like the node's device id. Uuids are sent as an argument when launching the Linux simulation program.

Using the standard input and output for AT commands allows us to automate some tests via scripts and to evaluate load performances. The creation of nodes is also made scriptable so that we can test a network of 2 up to 250 nodes.

Checks have been done that each AT command executes well, that the messages are received and acknowledged by the recipient. We also checked sending broadcast packets.

Gateway tests are described in this document, although as said above, the functionality is not present at the moment and can therefore not be tested.

Some of those tests cases can only be validated on the actual hardware (specifically those related to range and performance).

Logs are written into device specific files.

Test cases

Several cases will be tested, both on the simulation environment and on the real hardware.

Results of those tests can be found in the lowapp_simu_testing_doc document.

AT configuration commands

Test case A-: Send AT configuration commands to a node.

- A-1: AT+GROUPID=groupId and AT+GROUPID: set and retrieve the group id
- A-2: AT+DEVICEID=deviceid and AT+DEVICEID: set and retrieve the device id
- A-3: AT+GWMASK=gwMask and AT+GWMASK: enable/disable gateway mode and set the available gateway message types
- A-4: AT+ENCKEY=enckey and AT+ENCKEY: set and retrieve AES key (identical for the group). As displaying the encryption key is not allowed, a specific error message is returned when AT+ENCKEY is sent to the device.
- A-5: AT&W: write current configuration into persistent memory

AT Validation commands

Test case B-: Send AT validation commands to a node

- B-1: AT&V: print current configuration as json
- B-2: AT+SELFTEST: perform hardware selftest
- B-3: AT+WHO: get RSSI of the recently seen nodes
- B-4: AT+PING: send ping packet to a node and wait for the ACK

- B-5: AT+HELLO: send a hello packet. Responses happen when remote nodes decide to reply

AT Operation commands

Test case C-: Send AT operation commands to a node

- C-1: AT+SEND: Send a message to a node and wait for the ACK
- C-2: AT+POLLRX: Check for received packets since last check
- C-3: AT+RX: Change to push mode
- C-4: AT+DISCONNECT: Stop listening for packets and disable transmissions.
- C-5: AT+CONNECT: Start listening for packets and allow transmissions.

Use cases

Test case D-: Check the general functionality of the protocol.

- D-1: Send a message from one node to another (AT+SEND), both in the same group. The receiver is in **pull** mode. The receiver's application checks for the received packets.
- D-2: Send a message from one node to another, both in the same group. The receiver is **push** mode. The receiver's application is notified by its LoWAPP core.
- D-3: Send a message to a node that is either not existing or out of range.
- D-4: Send a message to a node on another group (should not receive it).
- D-5: Send a message to a node B in the same group. We will check with a third node C on another group, same channel, with the same device id of node B.
- D-6: Send a broadcast message with nodes in and out of the group. Check that the nodes out of the group do not receive the message.

D-x.b : Same test cases but with a group filled with 250 nodes.

Performance testing

Test case E-: Performance testing. Activity (time processing vs sleep time) and efficiency (percentage of message lost) of each nodes will be measured.

- E-1: Application send unicast messages to X nodes, waiting for ACK each time
  - E-1a: 10 nodes
  - E-1b: 50 nodes
- E-2: Application send unicast messages to X nodes, not waiting for ACK (burst tx requests).
  - E-2a: 10 nodes
  - E-2a: 50 nodes
- E-3: Application send X messages to the same node (burst rx).
  - E-3a: 10 messages
  - E-3b: 50 messages
- E-4: Test transmission with a preamble length of
  - E-4a: 1000 ms
  - E-4b: 500 ms
  - E-4c: 2000 ms
  - E-4d: 5000 ms
- E-5: Send one message to a specific node every X seconds and check no message is lost:
  - E-5a: 1 s

    - E-5b: 5 s
    - E-5c: 30 s
    - E-5d: 60 s
    - E-5e: 120 s
- E-6: Send broadcast message every X seconds:
    - E-6a: 1 s
    - E-6b: 5 s
    - E-6c: 30 s
    - E-6d: 60 s
    - E-6e: 120 s
- E-7: Sending messages from several nodes at the same time (AT+SEND at the same time, processing should be delayed)
    - E-7a: 2 nodes at the same time
    - E-7b: 3 nodes at the same time
- E-8: X nodes send messages to other nodes every Y seconds with each node having a different interval
    - E-8a: 10 nodes
        - E-8a1: Between 3 and 10 s
        - E-8a2: Between 10 and 20 s
        - E-8a3: Between 20 and 60 s
    - E-8b: 50 nodes
    - E-8c: 100 nodes
    - E-8d: 250 nodes

Range testing

Test case F-: Range testing

- F-1: Send a message to a node X meters away, wait for ACK
    - F-1a: 1 meter
    - F-1b: 10 meters
    - F-1c: 50 meters
    - F-1d: 100 meters
    - F-1e: 200 meters
    - F-1f: 500 meters
    - F-1g: 1 km
    - F-1h: 5 km
    - F-1i: 10 km
- F-2: Put X nodes, each being Y meter from each other (on a straight line) and send a broadcast
    - F-2a: 10 nodes, 100 meters
    - F-2b: 5 nodes, 200 meters
    - F-2c: 5 nodes, 1 km
- F-3: Put X nodes within the same square meter (or so)
    - F-3a: 10 nodes
    - F-3b: 200 nodes

    - F-3c: 250 nodes
- F-4: Put 250 nodes in the building (within about 100 meters)

All those test cases could be tested on several spreading factors, and different radio channels.

Gateway functionality

Test case G-: Gateway related activity

- G-1: A gateway receives a non-gateway specific message
- G-2: A gateway send a non-gateway specific message
- G-3: A gateway receives an outbound gateway message from a node of its group to an external network (IPv4 for example)
- G-4: A gateway receives an inbound gateway message destined to its group
- G-5: A gateway receives a gateway message of a type it cannot handle
- G-6: A gateway receives an intergroup LoWAPP gateway message and forward it to another group on the same channel.
    - G-6b: Same but on another channel

Hardware implementation of the system calls

The LoWAPP core is in the `src/lowapp/lowapp_core/`. It requires the `src/lowapp/lowapp_utils/` folder to be included as well as it contains some utility functions used by the core. Finally, the `src/lowapp/lowapp_shared_res/` folder contains functions used to lock and unlock access to shared resources from the core. On the simulations, these use mutexes. On the hardware, we are disabling and re-enabling global interrupts.

The `src/lowapp/lowapp_sys/` folder shows an example of the system functions required by the core.

A set of LOWAPP_SYS_IF_T functions have been developed for the STM32L1 microcontroller using hardware timers and the Semtech radio chip instead of the text files of the simulation.

Reimplementing the set of LOWAPP_SYS_IF_T functions should allow the lowapp core to be ported on any microcontroller.