Open Innovation Platform for IoT-Big data in Sub-Sahara Africa

Grant Agreement Nº 687607

Document

WAZIUP end-device/gateway IoT developer's guide

Responsible Editor:	UPPA
Contributors:	UPPA
Document Reference:	WAZIUP end-device/gateway IoT developer's guide
Distribution:	Public
Version:	1.1
Date:	March 9th, 2018

CONTRIBUTORS TABLE

DOCUMENT SECTION	AUTHOR(S)	REVIEWER(S)
Section 1	C. PHAM, UPPA	C. DUPONT
Section 2	M. EHSAN, UPPA	S. FATNASSI
Section 3	C. PHAM, UPPA	C. DUPONT

DOCUMENT REVISION HISTORY

Version	Date	Changes
v1.1	MARCH 9TH, 2018	PUBLIC RELEASE
v1.0	MARCH 2ND, 2018	VERSION FOR INTERNAL APPROVAL
v0.3	JAN 3RD, 2018	INTEGRATION OF REVIEWS
v0.2	DEC 26TH , 2017	FIRST RELEASE FOR REVIEW
v0.1	DEC 10TH , 2017	FIRST DRAFT

EXECUTIVE SUMMARY

This document entitled « WAZIUP end-device/gateway IoT developer's guide » describes the WAZIUP LoRa communication library and presents the developer's API. It will be structured as follows:

- 1. BRIEF PRESENTATION OF THE LOW-COST IOT PLATFORM : reviews the WAZIUP LoRa communication library and low-cost IoT platform.

- 2. ADVANCED UNDERSTANDING OF THE LONG-RANGE COMMUNICATION LIBRARY: presents the LoRa developer's API.

- 3. UNDERSTANDING THE LOW-COST GATEWAY: presents the low-cost gateway main components and architecture.

TABLE OF CONTENTS

LIST OF FIGURES

1. BRIEF PRESENTATION OF THE LOW-COST IOT PLATFORM

1.1. Long-range communication library

The long-range communication library is based on an open-source library developed by the Libelium company for their Libelium SX1272 radio module. We enhanced it with various mechanisms for WAZIUP as explained in the next paragraphs. Note that the developed library works for end-devices and gateways, therefore simplifying maintenance, updates and future common developments.

1.1.1. Improvements to the Libelium SX1272 library

The following improvements have been performed on the initial SX1272 library:

```
*  Feb 13th, 2018
*      - fix bug in availableData() to set back the LoRa module into standby mode.
*      - This affected only some radio modules
*  Jan 19th, 2018
*      - add a setCSPin(uint8_t cs) function to set the Chip Select (CS) pin
*      - call sx1272.setCSPin(18) for instance before calling sx1272.ON()
*      - by default, the CS pin will be set to SX1272_SS defined in SX1272.h
*  November 10th, 2017
*       - change the way packet's RSSI is computed
*  November 7th, 2017
*      - bug fix in how the CRC is checked at receiver in getPacket() function
*  November 3rd, 2017
*      - IMPORTANT: the CS pin is now always pin number 10 on Arduino boards
*      - if you use the Libelium Multiprotocol shield to connect a Libelium LoRa
          then change the CS pin to pin 2 in SX1272.h
*      - CRC (RxPayloadCrcOn) is now ON by default for transmitter side (end-device)
*  June, 22th, 2017
*      - setPowerDBM(uint8_t dbm) calls setPower('X') when dbm is set to 20
*  Apr, 21th, 2017
*      - change the way timeout are detected: exitTime=millis()+(unsigned long)wait; then
          millis() < exitTime;
*  Mar, 26th, 2017
*      - insert delay(100) before setting radio module to sleep mode. Remove unstability
          issue
*      - (proposed by escyes -
           https://github.com/CongducPham/LowCostLoRaGw/issues/53#issuecomment-289237532)
*  Jan, 11th, 2017
*      - fix bug in getRSSIpacket() when SNR < 0 thanks to John Rohde from Aarhus University
*  Dec, 17th, 2016
*      - fix bug making -DPABOOST in radio.makefile inoperant
*  Dec, 1st, 2016
*      - add RSSI computation while performing CAD with doCAD()
*      - WARNING: the SX1272 lib for gateway (Raspberry) does not have this functionality
*  Now, 26th, 2016
*      - add preliminary support for ToA limitation
*      - when in "production" mode, uncomment #define LIMIT_TOA
*  Now, 16th, 2016
*      - provide better power management mechanisms
*      - manage PA_BOOST and dBm setting
*  Jan, 23rd, 2016
*      - the packet format at transmission does not use the original Libelium format anymore
*      - the format is now dst(1B) ptype(1B) src(1B) seq(1B) payload(xB)
*          - ptype is decomposed in 2 parts type(4bits) flags(4bits)
*          - type can take current value of DATA=0001 and ACK=0010
*          - the flags are from left to right: ack_requested|encrypted|with_appkey|is_binary
*          - ptype can be set with setPacketType(), see constant defined in SX1272.h
*          - the header length is then 4 instead of 5
*  Jan, 16th, 2016
*      - add support for SX1276, automatic detect
*      - add LF/HF calibaration copied from LoRaMAC-Node.
*      - change various radio settings
*  Dec, 10th, 2015
*      - add SyncWord for test with simple LoRaWAN
*      - add mode 11 that have BW=125, CR=4/5, SF=7 on channel 868.1MHz
*  Nov, 13th, 2015
*      - add CarrierSense() to perform some Listen Before Talk procedure
```

```
*       - add dynamic ACK suport
*   Jun, 2015
*       - Add time on air computation and CAD features
```

Notable improvements are:

1. The support of both Arduino-based end-devices as well as Raspberry-based gateway to reduce maintenance complexity and third-party appropriation and training,
2. The support of both SX1272, 1276 and 1278 transceivers which makes the library able to drive most available SPI-based radio modules on the market,
3. a more flexible power management with selection of PA_BOOST or RFO amplifier lines to drive most available SPI-based radio modules on the market,
4. a carrier sense mechanism with customizable back-off procedure,
5. a Time on Air limitation for duty-cycle regulation enforcement,
6. additional frequency bands for Africa countries,
7. a simplification of developper's API

1.1.2. LoRa modules that have been tested

There are many SX1272/76/78-based radio modules available and we currently tested with 7 modules: the Libelium SX1272 LoRa, the HopeRF RFM92W(SX1272) & RFM95W(SX1276), the Modtronix inAir9(SX1276) & inAir9B(SX1276) & inAir4(SX1278 or SX1276) and the NiceRF SX1276. Actually, most native SPI-based LoRa modules are supported without modifications as reported by many users. In most cases, only a minimum soldering work is necessary to connect the required SPI pins of the radio (MISO, MOSI, CS, CLK) to the corresponding pins on the microcontroller board.

1.1.3. Radio regulations such as frequency bands and duty-cycle

In Europe, electromagnetic transmissions in the 868MHz ISM Band used by Semtech's LoRa technology falls into the Short Range Devices (SRD) category. The ETSI EN300-220-1 and ERC/REC 70-03 documents specify various requirements for SRD devices, especially those on unlicensed frequency bands and radio activity limitation.

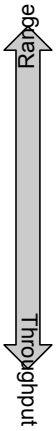
Regarding frequency bands, the library has 6 predefined channels (from 4 to 9) in the 863-865MHz band, 8 pre-defined channels (from 10 to 17) in the 865-868MHz band and 13 pre-defined channels (from 0 to 12) in the 903-915MHz band.

Regarding regulation under duty-cycle limitation, generally transmitters are constrained to a maximum of 0.1%, 1% or 10% every hour depending on the transmission power and the frequency band. For instance, a 1% duty-cycle means 36s of radio activity time per period of 1 hour. This duty cycle limit applies to the total transmission time, even if the transmitter can change to another channel. The rationale for such constraints is to avoid saturating the radio channel as this is an unlicensed band (free for everybody to use as opposed to most of frequency bands used in commercial mobile phone networks).

1.1.4. LoRa modes

As indicated previously, the 3 main LoRa parameters are BW, CR and SF. BW and SF being the 2 most important. However, you do not need to act on these parameters directly. The

communication library defines 10 so-called LoRa modes (from 1 to 10) that are various combinations of BW and SF. For instance, LoRa mode 1 defines BW=125kHz, CR=4/5 and SF=12. This combination provides the highest sensitivity at the receiver therefore it is suitable to achieve the longest range. However, the transmission time is the highest. Practically a real deployment can use this mode for all deployed devices to be sure to get the larger coverage. For the other modes, the range is generally decreased but transmission time is reduced. Figure 1 shows the various LoRa mode as combination of BW and SF.



				time on air in second for payload size of						
LoRa mode	BW	CR	SF	5 bytes	55 bytes	105 bytes	155 Bytes	205 Bytes	255 Bytes	max thr. for 255B in bps
1	125	4/5	12	0.95846	2.59686	4.23526	5.87366	7.51206	9.15046	223
2	250	4/5	12	0.47923	1.21651	1.87187	2.52723	3.26451	3.91987	520
3	125	4/5	10	0.28058	0.69018	1.09978	1.50938	1.91898	2.32858	876
4	500	4/5	12	0.23962	0.60826	0.93594	1.26362	1.63226	1.95994	1041
5	250	4/5	10	0.14029	0.34509	0.54989	0.75469	0.95949	1.16429	1752
6	500	4/5	11	0.11981	0.30413	0.50893	0.69325	0.87757	1.06189	1921
7	250	4/5	9	0.07014	0.18278	0.29542	0.40806	0.5207	0.63334	3221
8	500	4/5	9	0.03507	0.09139	0.14771	0.20403	0.26035	0.31667	6442
9	500	4/5	8	0.01754	0.05082	0.08154	0.11482	0.14554	0.17882	11408
10	500	4/5	7	0.00877	0.02797	0.04589	0.06381	0.08301	0.10093	20212

Figure 1 – LoRa mode as combination of BW and SF

1.1.5. Minimum function set to build a long-range end-device

Here is an example of a minimal setting for an end-device to actually send packets to a gateway.

```
sx1272.setMode(1);
sx1272.setChannel(CH_10_868);
sx1272.setPower('M'); // or sx1272.setPowerDBM(14);
sx1272.setNodeAddress(6);
sx1272.setPacketType(PKT_TYPE_DATA);
sx1272.sendPacketTimeout(1,"TC/18.56",8);
```

1.1.6. Packet format

The LoRa PHY layer packet format is unchanged and managed by the radio module. Then, there is a 4-byte header before the real user data. The header is organized as follows:

[DST(1B), PTYPE(4bits), FLAGS(4bits), SRC(1B), SN(1B)] [DATA(nB)]

DST is the destination address. With the gateway-centric topology, the gateway usually has address 1 so DST will most likely be 1. SRC is the source address on 1-byte [2..255]. SN is the packet sequence number. PTYPE has currently 2 values: 0001 for DATA packet and 0010 for an ACK packet. FLAGS is a 4-bit array that defines the following options:

- 1000: ack requested
- 0100: data is encrypted

- 0010: data has application key
- 0001: reserved

This 4-byte header is managed by our communication library and only the n bytes of [DATA] will be provided to the programmer. If the programmer wants to implement application key to perform further filtering, he can set the application key flag and insert in [DATA] a sequence of bytes that can be further checked at the post-processing stage. In general, one can implement its own packet format, variants or new functionalities (such as AES encryption) by defining a specific format in [DATA] and make the appropriate decoding at the post-processing stage.

1.2. Quick build of a low-cost IoT device

As indicated previously, the availability of low-cost, open-source hardware platforms such as Arduino-like boards is clearly an opportunity for building low-cost IoT devices from consumer market components. Our long-range communication library therefore targets such Arduino-like boards: original Arduino boards (Uno, MEGA, Due, Micro, Pro Mini, Nano, M0) but also many other Arduino-compatible boards from Sparkfun, Teensy, Adafruit Feather, RFduino, Ideetron Nexus, Sodaq,... if they have compatibility with the Arduino IDE. One main issue for an easy eligible board being the availability of a built-in 3.3v pin to power the radio to avoid an extra voltage regulator. Figure 2 shows our long-range communication library long-range library supporting various Arduino boards and LoRa radio modules. All of these boards and radio modules have been successfully tested and we are continuously testing new boards and radio modules.

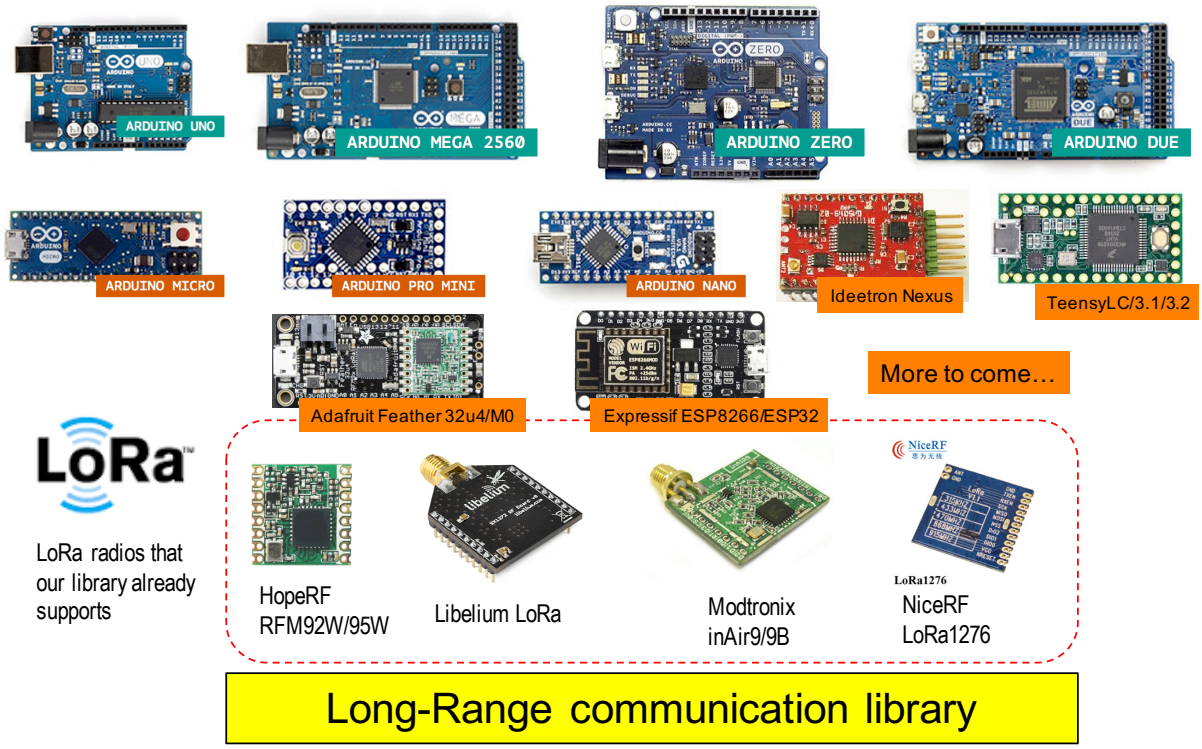


Figure 2 – The long-range library for Arduino-compatible boards and LoRa radio modules

Additionally, some radio shields from the market are actually based on the radio modules supported by WAZIUP. This is the case for the quite interesting Dragino product line which is based on the HopeRF RFM95W : LoRaBee, LoRa/GPS shield, LoRa Shield and Dragino LoRa GPS Hat shield. Figure 3 below, with images taken from Dragino, shows from left to right the aforementioned products.

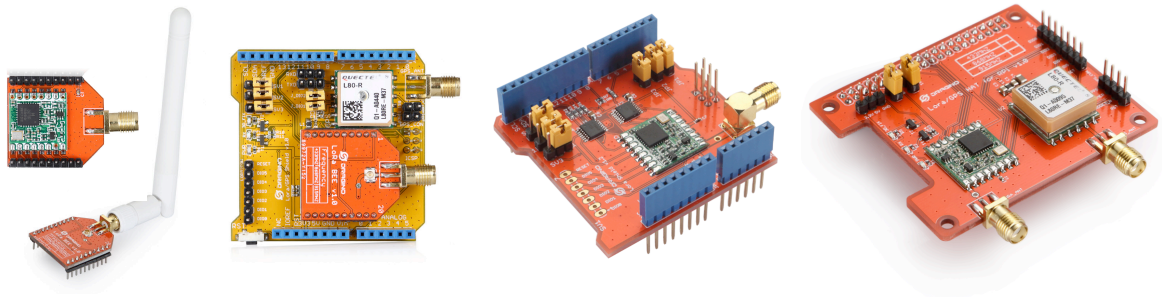


Figure 3 – Dragino LoRa product line based on HopeRF RFM95W

In the context of WAZIUP, we use **the Arduino Pro Mini in its 3.3v and 8MHz version** for simple, small-memory applications such as telemetry applications. Such Arduino Pro Mini can be purchased for about 1.5€ a piece from Chinese manufacturers. We then use **the Teensy31/3.2/LC for more power/memory demanding applications**.

1.2.1. Software integration for long-range IoT device

At the end-device, the main software components are the long-range communication library and the predefined building blocks for realizing generic tasks such as duty-cycled behaviour, physical sensor management, low-power management and encryption. Figure 4 shows the software building blocks for easy integration of long-range IoT devices.

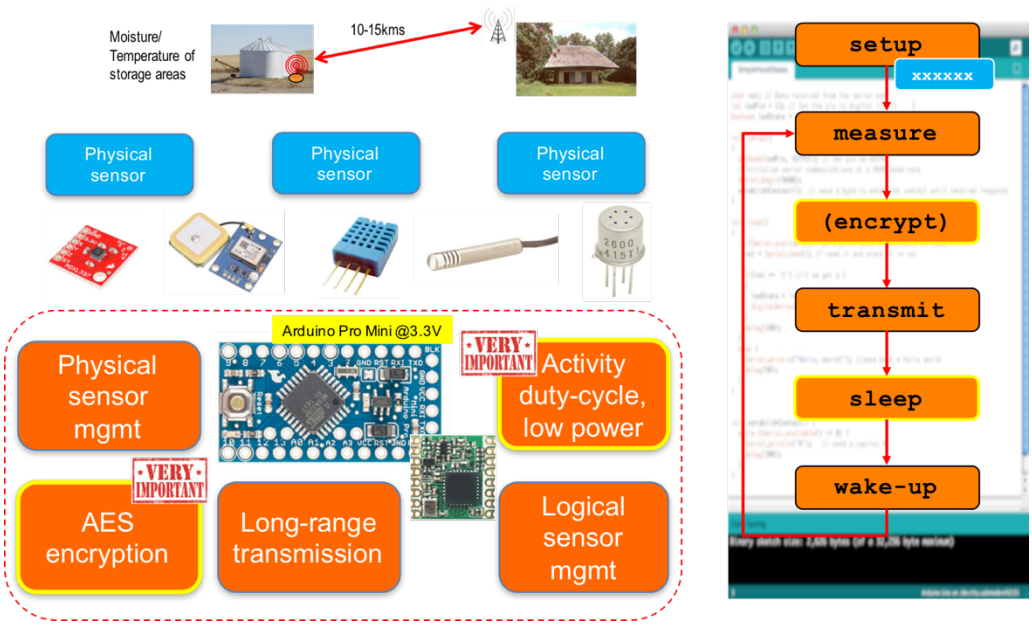


Figure 4 – Software building blocks for easy integration of long-range IoT devices

Low-power is an important feature for WAZIUP and IoT in general. One advantage of using mass-market components is also the availability of a large variety of software libraries. For

the generic sensor device based on the Arduino Pro Mini or the Teensy, we use specific low-power libraries that are capable of considerably reducing the power consumption of the device by providing deep sleep or hibernate modes. The security building block is also an important feature that WAZIUP provides: AES 128-bit encryption mode is supported to provide both security and compatibility with LoRaWAN if necessary.

1.2.2. Software templates for quick and easy appropriation

We have developed a number of examples to demonstrate how simple, yet effective, low-cost LoRa IoT device can be programmed. For instance, they show how LoRa radio modules are configured and how a device can send sensed data to a gateway. They actually serve as template for future developments.

Arduino_LoRa_Ping_Pong shows a simple ping-pong communication between a LoRa device and a gateway by requesting an acknowlegment for data messages sent to the gateway.

Arduino_LoRa_Simple_temp illustrates how a simple LoRa device with temperature data can be flashed to an Arduino board. The example illustrates in a simple manner how to implement most of the features of a real IoT device: periodic sensing, transmission to gateway, duty-cycle and low-power mode to run on battery for months.

Arduino_LoRa_temp illustrates a more complex example by adding a custom Carrier Sense mechanism that you can easily modify, AES encryption and the possibility to send LoRaWAN packet. It can serve as a template for a more complex LoRa IoT device.

Arduino_LoRa_Generic_Sensor is a very generic sensor template where a large variety of new physical sensors can be added. All physical sensors must be derived from a base Sensor class (defined in Sensor.cpp and Sensor.h) and should provide a get_value() and get_nomenclature() function. All the periodic task loop with duty-cycle low-power management is already there as in previous examples : the duty-cycle building block can be configured to trigger sensor reading every M minutes. All sensors connected to the board will be polled and the returned values concatenated into a message string for transmission. Some predefined physical sensors are also already defined:

- very simple LM35DZ analog temperature sensor
- digital DHT22 temperature and humidity sensor
- digital DS18B20 temperature sensor
- ultra-sonic HC-SR04 distance sensor
- Davies Leaf Wetness sensor
- general raw analog sensor

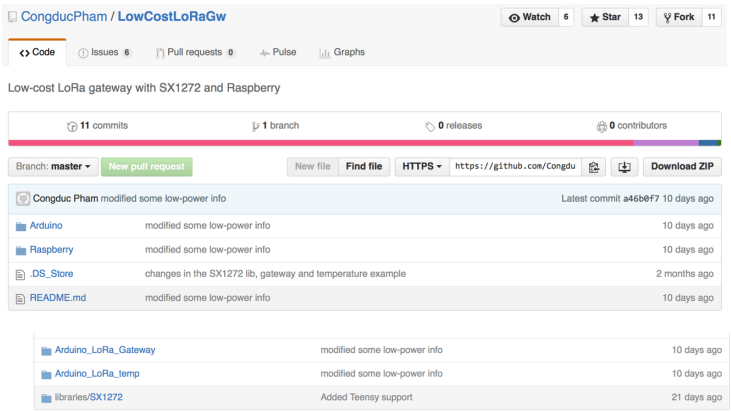
Arduino_LoRa_InteractiveDevice is a tool that turns an Arduino board to an interactive device where a user can interactively enter data to be sent to the gateway. AES encryption and the possibility to send LoRaWAN packet is included. There are also many parameters that can dynamically be configured. This example can serve for test and debug purposes as well.

1.2.3. Programming the board

Programming and deploying the low-cost end-device is made very simple thanks to the usage of Arduino boards and Arduino IDE. These steps are described in details in the various

tutorials that were developed for WAZIUP. We show here some figures to illustrate the simplicity that we want to provide to end-users or third-parties.

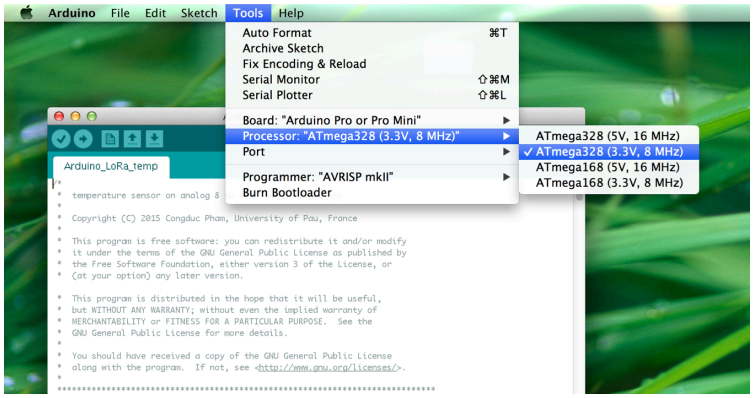
Getting the software



Fisrt, you will need the Arduino IDE 1.6.6 or later (left). Then get the LoRa library from our github: https://github.com/CongducPham/LowCostLoRaGw (right).

Get into the Arduino folder and get both Arduino_LoRa_temp and SX1272 folder. Copy Arduino_LoRa_temp into your "sketch" folder and SX1272 into "sketch/libraries"

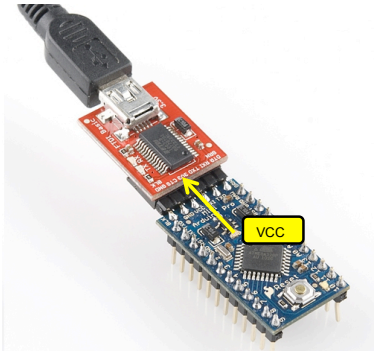
Compiling



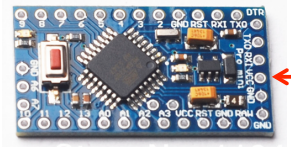
Open the Arduino_LoRa_temp sketch and select the Arduino Pro Mini board with its 3.3V & 8MHz version.

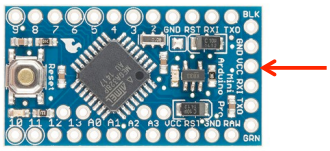
Then, click on the « verify » button 

Uploading (1)



Some clone version, check the VCC pin

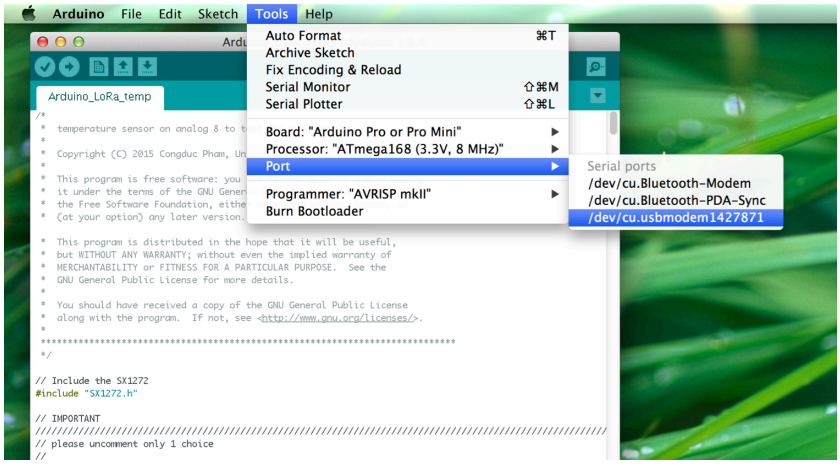




Original Sparkfun version

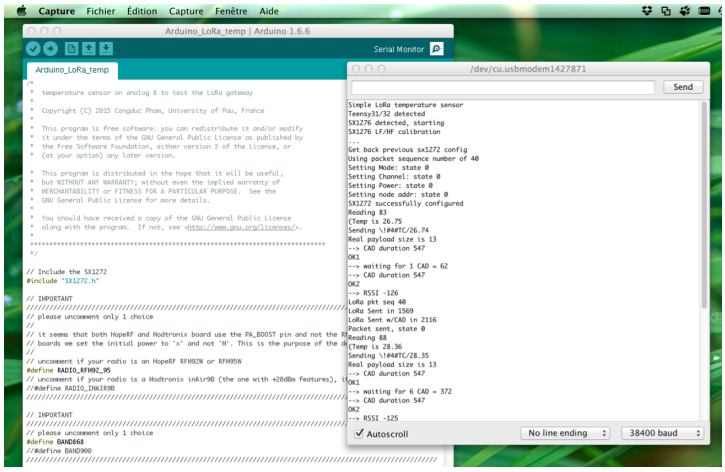
For the Pro Mini, you need to have an FTDI breakout cable working at 3.3v level (there is also 5v version but our advised Pro Mini version is running at 3.3v to reduce energy consumption). Be careful, on some low-cost Pro Mini version (Chinese manufacturer for instance) the pins may be in reversed order. The simplest way in to check the VCC pin and make it to correspond to the VCC pin of the FTDI breakout.

Uploading (2)



Connect the USB end to your computer and the USB port should be detected in the Arduino IDE. Select the serial port for your device. It may have another name than what is shown in the example. Then click on the « upload » button

Serial monitor



You can see the output from the sensor if it is connected to your computer. Use the Arduino IDE « serial monitor » to get such output, just to verify that the sensor is running fine, or to debug new code. Be sure to use 38400 baud.

Once programmed with for instance the **Arduino_LoRa_Simple_temp** template, the end-device is fully operational and starts the periodic sensing and data transmission. In Figure 5, the temperature is 18.5 °C.

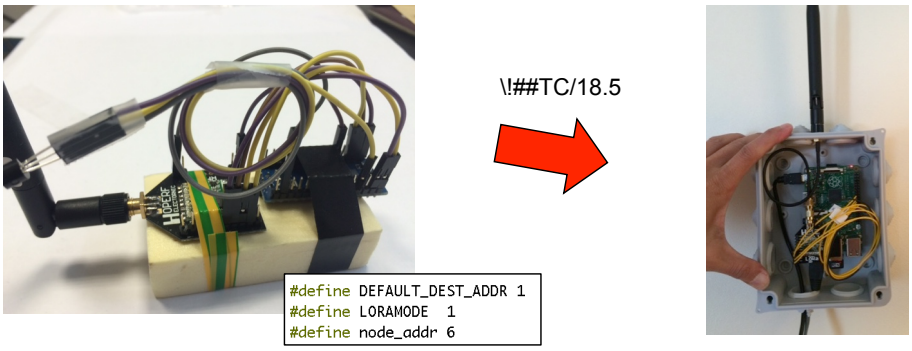


Figure 5 – Simple temperature sensor with periodic sensing and transmission

2. ADVANCED UNDERSTANDING OF THE LONG-RANGE COMMUNICATION LIBRARY

2.1. Important functionalities

The open-source long-range communication library developed by Libelium has been enhanced with mechanisms for WAZIUP by the UPPA team.

For Arduino:
https://github.com/CongducPham/LowCostLoRaGw/tree/master/Arduino/libraries/SX1272
For Raspberry:
https://github.com/CongducPham/LowCostLoRaGw/blob/master/gw_full_latest/SX1272.cpp
https://github.com/CongducPham/LowCostLoRaGw/blob/master/gw_full_latest/SX1272.h

The developed library works for Arduino (and Arduino-compatible boards) based end-devices and Raspberry-based gateways, therefore simplifying maintenance, updates and future developments. Some of the important functions in the library are defined below and is mainly intended for building end-devices:

Initialization:

```
uint8_t ON()
```

`ON()` function switches the module `ON`. This function configures the MISO, MOSI, CS and SPCR. It sets LoRa mode and activates CRC.

The function returns an integer that determines if there has been any error.

`state = 2` The command has not been executed
`state = 1` There has been an error while executing the command
`state = 0` The command has been executed with no errors

LoRa settings:

```
int8_t setMode(uint8_t mode)
```

`setMode()` function sets the bandwidth, coding rate and spreading factor of the LoRa modulation.

The function returns an integer that determines if there has been any error.

`state = 2` indicates the command has not been executed.
`state = 1` indicates there has been an error while executing the command.
`state = 0` indicates the command has been executed with no errors.
`state = -1` indicates a forbidden command for the protocol.

Mode number is used to set different values of the configured parameters, i.e. BW, SF and CR with this function. Different LoRa modes with BW and SF values are shown in Figure 1.

int8_t setChannel(uint32_t ch)

`setChannel()` function sets the indicated frequency channel in the module.

The function returns the Integer that determines if there has been any error.

`state = 2` indicates the command has not been executed.
`state = 1` indicates there has been an error while executing the command.
`state = 0` indicates the command has been executed with no errors.
`state = -1` indicates a forbidden command for the protocol.

This function take "`ch`" as the only parameter to select the frequency channel value to set in configuration. Pre-defined frequency values are defined in SX1272.h.

```
const uint32_t CH_04_868 = 0xD7CCCC; // channel 04, central freq = 863.20MHz
const uint32_t CH_05_868 = 0xD7E000; // channel 05, central freq = 863.50MHz
const uint32_t CH_06_868 = 0xD7F333; // channel 06, central freq = 863.80MHz
const uint32_t CH_07_868 = 0xD80666; // channel 07, central freq = 864.10MHz
const uint32_t CH_08_868 = 0xD81999; // channel 08, central freq = 864.40MHz
const uint32_t CH_09_868 = 0xD82CCC; // channel 09, central freq = 864.70MHz
//
const uint32_t CH_10_868 = 0xD84CCC; // channel 10, central freq = 865.20MHz
const uint32_t CH_11_868 = 0xD86000; // channel 11, central freq = 865.50MHz
const uint32_t CH_12_868 = 0xD87333; // channel 12, central freq = 865.80MHz
const uint32_t CH_13_868 = 0xD88666; // channel 13, central freq = 866.10MHz
const uint32_t CH_14_868 = 0xD89999; // channel 14, central freq = 866.40MHz
const uint32_t CH_15_868 = 0xD8ACCC; // channel 15, central freq = 866.70MHz
const uint32_t CH_16_868 = 0xD8C000; // channel 16, central freq = 867.00MHz
const uint32_t CH_17_868 = 0xD90000; // channel 17, central freq = 868.00MHz

// added by C. Pham
const uint32_t CH_18_868 = 0xD90666; // 868.1MHz for LoRaWAN test

const uint32_t CH_00_900 = 0xE1C51E; // channel 00, central freq = 903.08MHz
const uint32_t CH_01_900 = 0xE24F5C; // channel 01, central freq = 905.24MHz
const uint32_t CH_02_900 = 0xE2D999; // channel 02, central freq = 907.40MHz
const uint32_t CH_03_900 = 0xE363D7; // channel 03, central freq = 909.56MHz
const uint32_t CH_04_900 = 0xE3EE14; // channel 04, central freq = 911.72MHz
const uint32_t CH_05_900 = 0xE47851; // channel 05, central freq = 913.88MHz
const uint32_t CH_06_900 = 0xE5028F; // channel 06, central freq = 916.04MHz
const uint32_t CH_07_900 = 0xE58CCC; // channel 07, central freq = 918.20MHz
const uint32_t CH_08_900 = 0xE6170A; // channel 08, central freq = 920.36MHz
const uint32_t CH_09_900 = 0xE6A147; // channel 09, central freq = 922.52MHz
const uint32_t CH_10_900 = 0xE72B85; // channel 10, central freq = 924.68MHz
const uint32_t CH_11_900 = 0xE7B5C2; // channel 11, central freq = 926.84MHz
const uint32_t CH_12_900 = 0xE4C000; // default channel 915MHz, the module is
configured with it

// added by C. Pham
const uint32_t CH_00_433 = 0x6C5333; // 433.3MHz
const uint32_t CH_01_433 = 0x6C6666; // 433.6MHz
const uint32_t CH_02_433 = 0x6C7999; // 433.9MHz
const uint32_t CH_03_433 = 0x6C9333; // 434.3MHz
```

int8_t setPower(char p)

`setPower()` function sets the signal power indicated in the module.

The function returns an integer that determines if there has been any error.

`state = 2` indicates the command has not been executed.
`state = 1` indicates there has been an error while executing the command.
`state = 0` indicates the command has been executed with no errors.
`state = -1` indicates a forbidden command for the protocol.

The function takes character "`p`" as a parameter to set power option in configuration.

L = **L**ow. On SX1272/76: PA0 on RFO setting. Equivalent to 2dBm.
H = **H**igh. On SX1272/76: PA0 on RFO setting. Equivalent to 6dBm.
M = **M**ax. On SX1272/76: PA0 on RFO setting. Equivalent to 14dBm.

x = e**x**treme. On SX1272/76: PA1&PA2 PA_BOOST setting. Equivalent to 14dBm.
X = e**X**treme. On SX1272/76: PA1&PA2 PA_BOOST setting + 20dBm settings.

int8_t setPowerDBM(uint8_t dbm)

`setPowerDBM()` function is used to set output power in given DBM. It takes '`dbm`' as the only parameter used to set power to '`dbm`' level. '`dbm`' should be between 1 and 20. It is recommended to use this function instead of the previous `setPower()` function provided by Libelium.

The function returns an integer that determines if there has been any error.

`state = 0` indicates that the output power has been successfully set.
`state = 1` indicates there has been an error in setting output power.

int8_t setNodeAddress(uint8_t addr)

`setNodeAddress()` function sets the node address in the module. Device's address starts at `2` as `1` is reserved for the gateway, and stop at `255`.

The function returns an Integer that determines if there has been any error.

`state = 2` indicates the command has not been executed.
`state = 1` indicates there has been an error while executing the command.
`state = 0` indicates the command has been executed with no errors.
`state = -1` indicates a forbidden command for the protocol.

It takes "`addr`" as a sole parameter to set the address value as node address.

LoRa transmission settings:

```
void setPacketType(uint8_t type)
```

`setPacketType()` function sets the type of the packet to send. This function takes "`type`" of the packet to be set as the only parameter. There are 2 types which are `PKT_TYPE_DATA` (simple data packet) and `PKT_TYPE_ACK` (Acknowledgement packet).

There are pre-defined flags to indicate additional information: `PKT_FLAG_ACK_REQ` (Acknowledgement request), `PKT_FLAG_DATA_ENCRYPTED` (indicates that payload is encrypted), `PKT_FLAG_DATA_WAPPKEY` (indicates that payload is prefixed by a 4-byte AppKey) and `PKT_FLAG_DATA_ISBINARY` (indicates that data is binary). Flags can be added as follows: `PKT_TYPE_DATA | PKT_FLAG_ACK_REQ`

```
uint8_t sendPacketTimeout(uint8_t dest, uint8_t *payload, uint16_t length16)
```

`sendPacketTimeout()` function sends a LoRa packet.

The function returns an integer that determines if there has been any error.

`state = 2` indicates the command has not been executed.
`state = 1` indicates there has been an error while executing the command.
`state = 0` indicates the command has been executed with no errors.

This function takes three parameters. `dest` (address of destination), a pointer to the `payload` and `length16` (the message size).

For example, `sendPacketTimeout(1,(uint8_t*)"20.5",4)` sends the message to gateway, the message is `"20.5"` and packet length is `4`.

```
uint8_t sendPacketTimeoutACK(uint8_t dest, uint8_t *payload, uint16_t length16)
```

`sendPacketTimeoutACK()` function sends a LoRa packet and indicate at the receiver (e.g. gateway) that an ACK is requested.

The function returns an integer that determines if there has been any error.

`state = 3` indicates that the packet has been sent but ACK has not been received.
`state = 2` indicates the command has not been executed.
`state = 1` indicates there has been an error while executing the command.
`state = 0` indicates the command has been executed with no errors.

`sendPacketTimeoutACK()` takes three parameters as `sendPacketTimeout()`: address of destination, a pointer to the payload and the message size.

Advanced configuration parameters:

int8_t setBW(uint16_t band)

In case we need different configurations then the ones provided, we can set the BW, CR and SF values by using `setBW()`, `setCR()` and `setSF()` functions. In most cases, you do not need to use them and would use `setMode()` instead.

`setBW()` function sets the indicated BW in the module.

The function returns an integer that determines if there has been any error.

`state = 2` The command has not been executed
`state = 1` There has been an error while executing the command
`state = 0` The command has been executed with no errors

`setBW()` uses `band` as its sole parameter which is the bandwidth value to set in LoRa modem configuration: `BW_125`, `BW_250` or `BW_500`.

`getBW()` function retrieves the BW value from the module to store the value in the `_bandwidth` variable. `isBW(uint16_t band)` function can be used to check if BW has a valid value, returning '`true`' means BW has a valid value or '`false`' in case the BW value does not exist or has an invalid value.

int8_t setCR(uint8_t cod)

`setCR()` function sets the indicated Coding Rate in the module.

The function returns an integer value that determines if there has been any error

`state = 2` indicates the command has not been executed.
`state = 1` indicates there has been an error while executing the command.
`state = 0` indicates the command has been executed with no errors.
`state = -1` indicates a forbidden command for the protocol.

`cod` is the only parameter for the coding rate value to set in LoRa modem configuration: `CR_5`, `CR_6`, `CR_7` or `CR_8`.

`getCR()` function retrieves the CR value from the module and stores it in the `_codingRate` variable. `isCR(uint8_t cod)` function can be used to check if CR has a valid value, returning returning '`true`' means CR has a valid value or '`false`' in case the CR value does not exist or has an invalid value.

int8_t setSF(uint8_t spr)

`setSF()` function sets the indicated Spreading Factor in the module.

The function returns an integer that determines if there has been any error

`state = 2` The command has not been executed
`state = 1` There has been an error while executing the command
`state = 0` The command has been executed with no errors

`setSF()` function has one parameter which is `spr`. `spr` is the spreading factor value to set in LoRa modem configuration: `SF_6`, `SF_7`, `SF_8`, `SF_9`, `SF_10`, `SF_11` and `SF_12`.

`getSF()` function gets the SF value within the configured module and stores it in the `_spreadingFactor` variable. `isSF(uint8_t spr)` function can be used anytime to check if SF has a valid value. It returns '`true`' if the SF value exists and '`false`' if SF does not exist.

`uint8_t setCRC_ON()`

`setCRC_ON()` function sets the module with CRC on. This function is called to activate the CRC at the transmitter side only. It takes no parameter.

Similarly another function with the name of `setCRC_OFF()`, as it name suggests, deactivates the CRC. `getCRC()` function is used to know if CRC is ON or OFF (i.e. activated or deactivated in a specific module).

The function returns an integer that determines if there has been any error.

`state = 2` The command has not been executed
`state = 1` There has been an error while executing the command
`state = 0` The command has been executed with no errors

`uint16_t getToA(uint8_t pl)`

`getToA()` function is used to get the airtime of a transmission. It takes '`pl`' as a sole parameter. '`pl`' stands for payload size. This function returns current time on air (current ToA) value in ms.

`long limitToA()`

`limitToA()` function is used to limit the radio activity time to 1% per hour (i.e. 36s) according to ETSI Short Range Device regulations. Once the function is called, there will be a verification for each packet transmission and if the limit has been reached, the packet will not be transmitted. This function returns the maximum radio activity time available for the current period, i.e. 36000ms. The current design makes that once ToA limitation has been set, it is not possible to set it back to false. This function is useful when deploying operational devices that must comply with regulations.

int8_t setSyncWord(uint8_t sw)

`setSyncWord()` function sets the sync word in the LoRa module.

The function returns an integer that determines if there has been any error.

`state = 2` indicates the command has not been executed.
`state = 1` indicates there has been an error while executing the command.
`state = 0` indicates the command has been executed with no errors.
`state = -1` indicates a forbidden command for the protocol.

The only parameter this function uses is `sw`. `sw` is the sync word value to set in LoRa modem configuration. The default value is 0x12 which means "not LoRaWAN networks". LoRaWAN networks should use 0x34 as the sync word.

`getSyncWord()` function can be used to get the sync word in the module. It returns the same states as `setSyncWord()`.

int8_t setSleepMode()

`setSleepMode()` sets the LoRa radio module in sleep mode to save energy. Any call to send or receive packet will set the radio module back to normal mode.

The function returns an integer that determines if there has been any error.

`state = 2` The command has not been executed
`state = 1` There has been an error while executing the command
`state = 0` The command has been executed with no errors

void setCSPin(uint8_t cs)

The `setCSPin()` function set the Chip Select pin of SPI module to the cs value. This function is useful to support a new board where the CS pin of the radio module is not handled by the library. The current library detects most of Arduino boards and will use pin 10. It also detects Adafruit Feather32U4 and FeatherM0 boards to use pin 8. It also detects the ESP8266 ESP01 to use pin 15. For other boards, use this function instead of changing the library.

Additional configuration variables:

There are 4 additional variables to set specific behavior for the communication library.

bool _RSSIonSend

A boolean variable `_RSSIonSend` is used, in case the developer wants to check the received signal strength before sending the data. By default the `_RSSIonSend` is set to '`true`'.

bool _enableCarrierSense

`_enableCarrierSense` is used to enable the carrier sense mechanism when `CarrierSense()` function is called. It is a Boolean variable and by default `_enableCarrierSense` is set to '`false`'.

bool _needPABOOST

Some radio modules use PABOOST line instead of RFO line (or another line) to increase their transmission power, in that case `_needPABOOST` is set to '`true`'. `_needPABOOST` is set to '`false`' by default.

bool _rawFormat

The boolean variable `_rawFormat` indicates to the communication library to NOT handle the 4-byte header. In a normal packet, a 4-byte header is handled at reception. By default `_rawFormat` is set to '`false`', i.e the packet format is normal by default. `_rawFormat` is useful for a gateway to pass raw data (the raw LoRa payload) to higher layer, if any. Be sure to understand the impact of raw format before modifying this variable. A normal device SHOULD NOT set `_rawFormat` to '`true`'.

Basic example:

The `Arduino_LoRa_Simple_temp` sketch can serve as a good basic example for developers. This sketch is used to take a temperature value from the sensor and transmit the temperature value to the gateway. It uses the SX1272 library.

```
#include <SPI.h>
// Include the SX1272
#include "SX1272.h"

#define MAX_DBM 14
const uint32_t DEFAULT_CHANNEL=CH_10_868;

////////////////////// IMPORTANT//////////////////////
// uncomment if your radio is an HopeRF RFM92W, HopeRF RFM95W,
// Modtronix inAir9B, NiceRF1276
// or you know from the circuit diagram that output use the PABOOST
// line instead of the RFO line
```

```
//#define PABOOST
/////////////////////////////////////////////////

// CHANGE HERE THE LORA MODE, NODE ADDRESS
#define LORAMODE  1
#define node_addr 6
#define DEFAULT_DEST_ADDR 1

uint8_t message[100];
int loraMode=LORAMODE;

void setup()
{
  int e;
  delay(3000);
  // Open serial communications and wait for port to open:
  Serial.begin(38400);

  // Print a start message
  Serial.println("Minimum sender device");
 // Power ON the module
  sx1272.ON();

  // Set transmission mode and print the result
  e = sx1272.setMode(loraMode);

  // enable carrier sense
  sx1272._enableCarrierSense=true;

  // Select frequency channel
  e = sx1272.setChannel(DEFAULT_CHANNEL);

  // Select amplifier line; PABOOST or RFO
#ifdef PABOOST
  sx1272._needPABOOST=true;
#endif

  e = sx1272.setPowerDBM((uint8_t)MAX_DBM);

  // Set the node address and print the result
  e = sx1272.setNodeAddress(node_addr);

  Serial.println("SX1272 successfully configured");
  delay(500);
}

void loop(void)
{
  long startSend;
  long endSend;
  int e;

  r_size=sprintf((char*)message, "%s", "Hello World");

  int pl=r_size;

  sx1272.CarrierSense();
```

```
  sx1272.setPacketType(PKT_TYPE_DATA);
  e = sx1272.sendPacketTimeout(DEFAULT_DEST_ADDR, message, pl);

  Serial.println("Switch to power saving mode");

  // this will set the radio module in sleep mode, but not the
  // Arduino board, see the full example to see how the board
  // can be put in deep sleep mode
  e=sx1272.setSleepMode();

  delay(5000);
}
```

2.2. Customization of templates

The templates can easily be customized by changing some define statements and variables that are clearly identified in the template code. We summarize below the main parts that can be customized for adaptation to use cases.

```
//////////////////////////////////////////////////////////////////
// please uncomment only 1 choice
//
#define ETSI_EUROPE_REGULATION
//#define FCC_US_REGULATION
//#define SENEGAL_REGULATION
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
// please uncomment only 1 choice
#define BAND868
//#define BAND900
//#define BAND433
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
//
// uncomment if your radio is an HopeRF RFM92W, HopeRF RFM95W,
// Modtronix inAir9B, NiceRF1276
// or you known from the circuit diagram that output use the PABOOST
// line instead of the RFO
//#define PABOOST
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
// COMMENT OR UNCOMMENT TO CHANGE FEATURES.
// ONLY IF YOU KNOW WHAT YOU ARE DOING!!! OTHERWISE LEAVE AS IT IS
#if not defined _VARIANT_ARDUINO_DUE_X_ && not defined
__SAMD21G18A__
#define WITH_EEPROM
#endif
#define WITH_APPKEY
#define FLOAT_TEMP
#define NEW_DATA_FIELD
```

```
#define LOW_POWER
#define LOW_POWER_HIBERNATE
//#define WITH_ACK
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// CHANGE HERE THE LORA MODE, NODE ADDRESS
#define LORAMODE  1
#define node_addr 10
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// CHANGE HERE THE TIME IN MINUTES BETWEEN 2 READING & TRANSMISSION
unsigned int idlePeriodInMin = 10;
/////////////////////////////////////////////////////////////////
```

More information are detailed in:

[Doc] Low-cost LoRa IoT platform part list
Latest version can be downloaded from:
https://github.com/CongducPham/tutorials/blob/master/low-cost-iot-hardware-parts.pdf

[Doc] "Low-cost LoRa IoT device leaflet"
Latest version can be downloaded from:
https://github.com/CongducPham/tutorials/blob/master/Low-cost-LoRa-device-leaflet.pdf

[Slides] "Low-cost LoRa IoT device: a step-by-step tutorial"
Latest version can be downloaded from:
https://github.com/CongducPham/tutorials/blob/master/Low-cost-LoRa-IoT-step-by-step.pdf

[Slides] "Building IoT device for outdoor usage: a step-by-step tutorial"
Latest version can be downloaded from:
https://github.com/CongducPham/tutorials/blob/master/Low-cost-LoRa-IoT-outdoor-step-by-step.pdf

[Slides] "Low-cost LoRa IoT device: supported physical sensors"
Latest version can be downloaded from:
https://github.com/CongducPham/tutorials/blob/master/Low-cost-LoRa-IoT-supported-sensors.pdf

[Video] Build your low-cost, long-range IoT device with WAZIUP
https://www.youtube.com/watch?v=YsKbJeeav_M

[Video] Extreme low-cost & low-power LoRa IoT for real-world deployment
https://www.youtube.com/watch?v=2_VQpcCwdd8

3. UNDERSTANDING THE LOW-COST GATEWAY

3.1.1. Gateway hardware and architecture

Our LoRa low-cost gateway can be qualified as "single connection" as it is built around an SX1272/76, much like an end-device would be. The low-cost gateway is based on a Raspberry PI (1A+/1B/1B+/2B/3B) which is both a low-cost and a reliable embedded Linux platform. To install the Raspberry, our pre-installed SD card image can be downloaded (on http://cpham.perso.univ-pau.fr/LORA/WAZIUP/raspberrypi-jessie-WAZIUP-demo.dmg.zip). Complete instructions to install from scratch with the Raspbian OS is also provided. When all the software components are installed, a radio module can be connected. Figure 6 shows the various steps for building our low-cost LoRa gateway.

The gateway software architecture has been designed for maximum flexibility and easy third-party appropriation and customization. In our gateway architecture we clearly want to decouple the specific lower level radio bridge program from the higher-level data post-processing stage that must be easily customized by third parties. The data post-processing block (orange block in Figure 7) is written in high-level language (e.g. Python) for simplicity and maximum customization possibilities by third-parties.

Figure 6 – Low-cost single channel LoRa gateway

As can be seen in the right part of Figure 7, the WAZIUP project will provide most of the gateway software logic, with the top layer being highly customizable for specific application's needs.

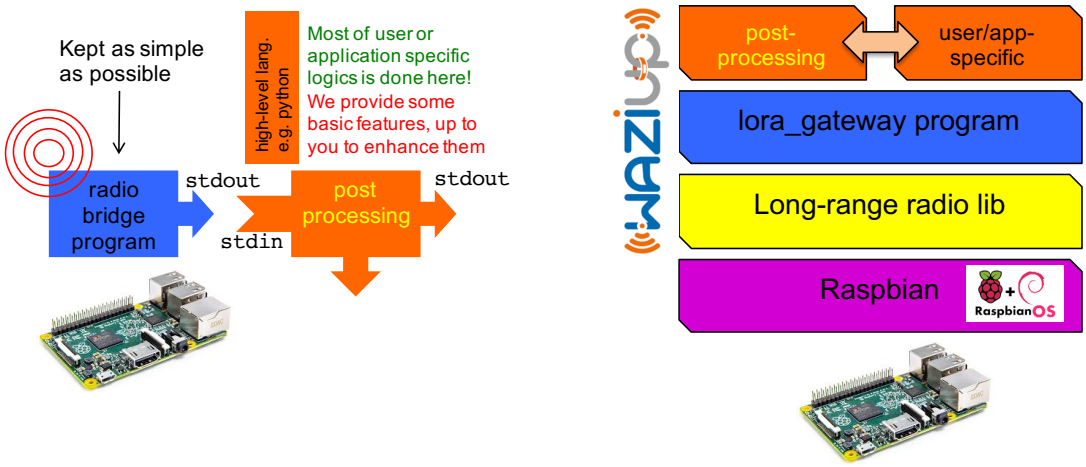


Figure 7 – Gateway architecture

As can be seen, the post-processing block is the core of all incoming packet data processing tasks. We provide a Python template for the post-processing block (post_processing_gw.py) and Figure 8 shows a detailed view of the proposed post-processing template components and features. **The main component is the « incoming data parsing block » that calls user/app-specific cloud scripts**. Our post-processing template also provides additional features such as gateway temperature monitoring (with a DHT22 sensor connected to the gateway), AES encryption and decryption, management of downlink data requests, management of data from other radio interfaces, definition of user-specific periodic tasks and simple LoRaWAN interoperability.

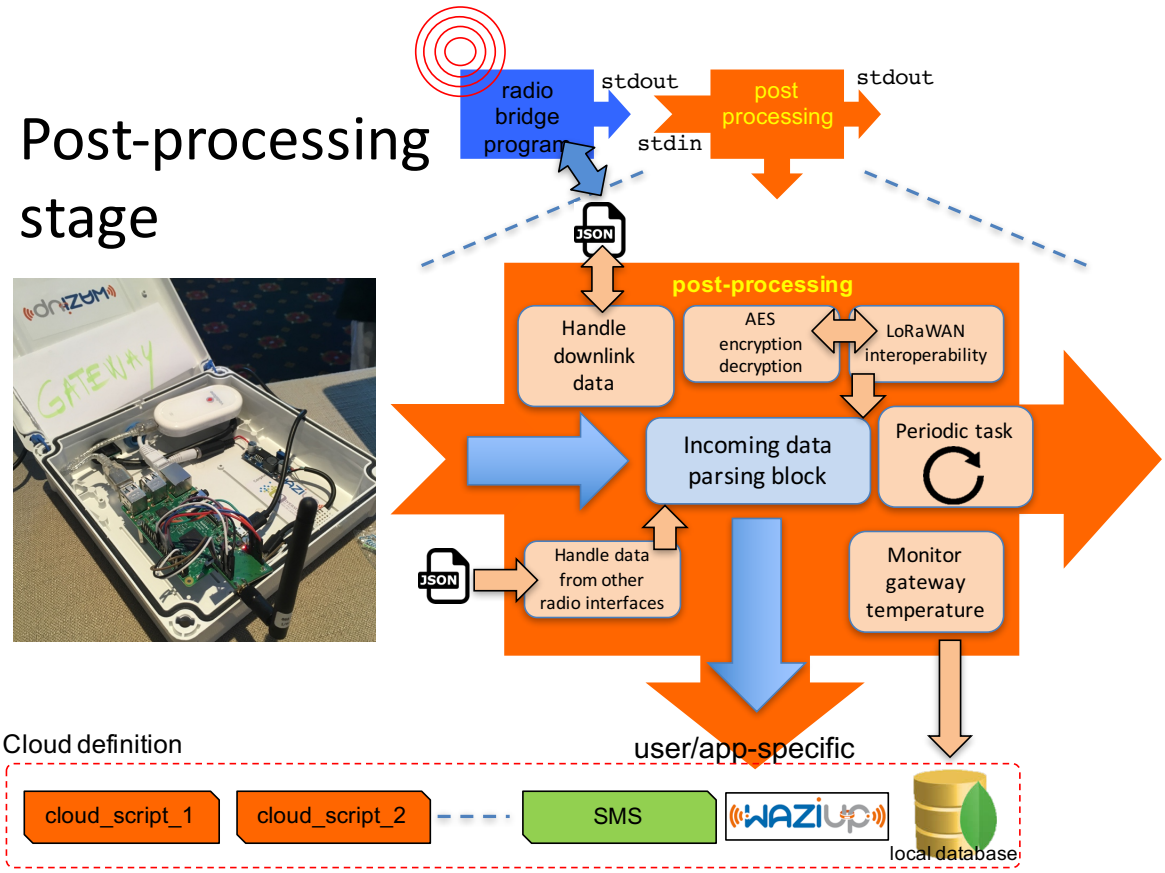


Figure 8 – post-processing block template

3.1.2. Lower level radio bridge and post-processing block interaction

As shown in Figure 8, the lower level radio bridge write formatted received data to `stdout` and the post-processing block will read from `stdin`.

For each incoming LoRa packet, the lower level radio bridge will write the 5 following formatted strings:

1. `^p`. Indicates a packet info string formatted as follows :
   `^pdst(%d),ptype(%d),src(%d),seq(%d),len(%d),SNR(%d),RSSI(%d)`
   e.g. `^p1,16,3,0,10,8,-45`
2. `^r`. Indicates a radio info string formatted as follows : `^rbw(%d),cr(%d),sf(%d)`
   e.g. `^r125,5,12`
3. `^t`. Indicate a timestamp info string formatted as follows :
   `^ttime(%s)`
   e.g. `^t2016-12-25T01:15:11.264700`
4. `\xFF\xFE`. 2-bytes prefix to indicate incoming packet payload
5. received packet payload

For instance if "hello" from sensor 6 is received at the lower level radio bridge, the following strings will be written to `stdout`, assuming LoRa mode 1 (see Figure 1 for a list of LoRa modes):

```
^p1,16,6,0,5,8,-45
^r125,5,12
^t2016-12-25T01:51:11.058
\xFF\xFEhello
```

The post-processing block will read these strings to get packet info, radio info, timestamp info and finally packet payload content to take further action such as parsing the payload content and eventually upload payload content to IoT clouds. The current post-processing template would show the following output when receiving the previous strings:

```
2016-12-25T00:51:11.059762
rcv ctrl pkt info (^p): 1,16,6,0,5,8,-45
splitted in:  [1, 16, 6, 0, 5, 8, -45]
(dst=1 type=0x10(DATA) src=6 seq=0 len=5 SNR=8 RSSI=-45)
rcv ctrl radio info (^r): 125,5,12
splitted in:  [125, 5, 12]
(BW=500 CR=5 SF=12)
rcv timestamp (^t): 2016-12-25T01:51:11.058
got first framing byte
--> got data prefix
hello
```

3.1.3. What clouds for low-cost IoT?

If the gateway is connected to the Internet as shown in Figure 9, data received on the gateway are usually pushed/uploaded to some Internet/cloud servers such as the WAZIUP Cloud Platform (see WAZIUP D3.1). These tasks are handled by the post-processing stage as previously illustrated in Figure 8.

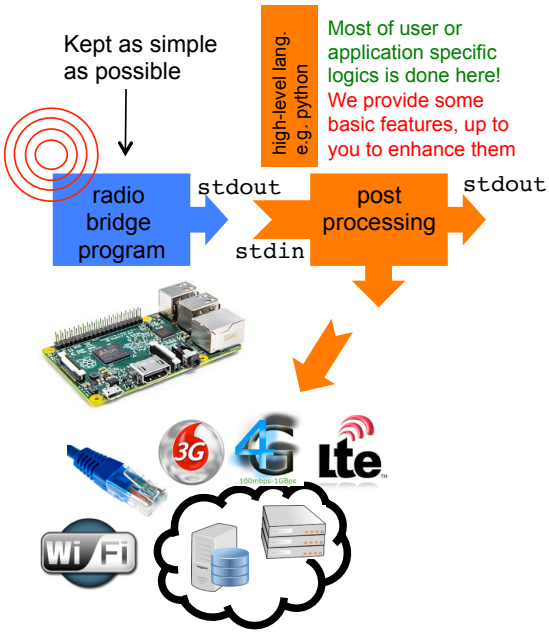


Figure 9 – Post-processing stage with Internet connectivity.

Additionally, it is important in the context of developing countries to be able to use a wide range of infrastructures and, if possible, at the lowest cost. Fortunately, along with the global IoT uptake, there is also a tremendous availability of sophisticated and public IoT clouds platforms and tools, offering an unprecedented level of diversity which contributes to limit dependency to proprietary infrastructures. Many of these platforms offer free accounts with limited features but that can already satisfy the needs of most agriculture/micro and small farm/village business models. It is therefore desirable to highly decouple the low-level stage gateway functionalities from the high-level stage with data post-processing features, privileging high-level languages for the latter stage (e.g. Python) so that customizing data management tasks can be done in minutes, using standard tools, simple REST API interfaces and available public clouds as depicted in Figure 10 with publicly available IoT clouds such as Firebase, ThingSpeak or GroveStreams.

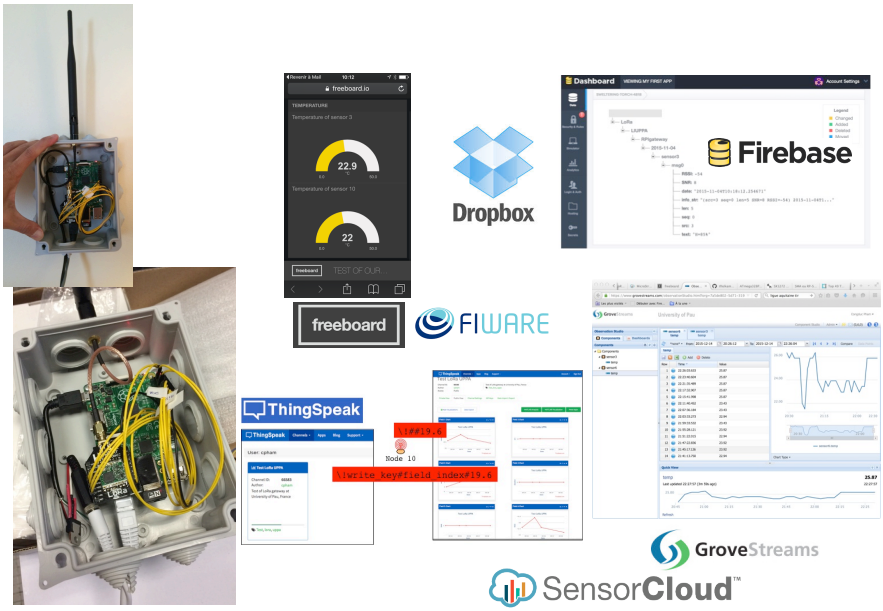


Figure 10 – From gateway to IoT clouds

3.1.4. Uploading to clouds

To indicate that a payload content should be uploaded to IoT clouds, the current post-processing block uses the « \ ! » prefix. For each IoT cloud that is enabled at the gateway, a cloud script should be provided. The cloud script will typically dissect the payload to extract relevant information and will upload the relevant information to the corresponding cloud platform using adequate commands.

The post-processing block will parse a `clouds.json` file that contains a list of clouds definitions for data to be uploaded. Here is an example with 3 clouds: local MongoDB, ThingSpeak and Grovestreams.

```
{
	"clouds" : [
		{
			"notice":"do not remove"
			"name":"Local gateway MongoDB",
			"script":"python CloudMongoDB.py",
			"type":"database",
			"max_months_to_store":2,
			"enabled":false
		},
		{
			"name":"ThingSpeak cloud",
			"script":"python CloudThingSpeak.py",
			"type":"iotcloud",
			"write_key":"",
			"enabled":true
		},
		{
			"name":"GroveStreams cloud",
			"script":"python CloudGroveStreams.py",
			"type":"iotcloud",
			"write_key":"",
			"enabled":true
		}
}
```

For each cloud declaration, there are only 2 relevant fields: "script" and "enabled". "script" is used to provide the name of a script. The launcher that will be used must also be indicated. Actually, "script" is more a command line than a file name. In this way, several script languages can be used (including shell scripts and binary executables provided that they read parameters that are passed by their command line). For instance, if the script is a python script, "script" should contain "`python my_script_filename`".

"enabled", when set to true, indicates that this cloud is active so that the post-processing block will call the associated script to perform upload of the received data. All the other fields are not relevant for the post-processing block but can be used by the associated script to get additional information that the user may want to provide through the `clouds.json` file.

After parsing `clouds.json`, the post-processing block has the list of enabled clouds as shown in the following output from the post-processing block :

```
Parsing cloud declarations
[u'python CloudThingSpeak.py']
[u'python CloudThingSpeak.py', u'python CloudGroveStreams.py']
Parsed all cloud declarations
post_processing_gw.py got cloud list:
[u'python CloudThingSpeak.py', u'python CloudGroveStreams.py']
```

Then, assuming that _enabled_clouds contains:

```
['python CloudThingSpeak.py', 'python CloudGroveStreams.py']
```

the main data upload processing loop in the post-processing block is now very simple and looks as follows:

```
#ldata will contain the payload
ldata = getAllLine()
print "number of enabled clouds is %d" % len(_enabled_clouds)

#loop over all enabled clouds to upload data
#it is up to the corresponding cloud script to handle the data format
#
for cloud_index in range(0,len(_enabled_clouds)):
      print "--> cloud[%d]" % cloud_index
      cloud_script=_enabled_clouds[cloud_index]
      print "uploading with "+cloud_script
      cmd_arg=cloud_script+" \""+ldata+"\""+" \""+pdata+"\""+
            " \""+rdata+"\""+" \""+tdata+"\""+" \""+_gwid+"\""
      os.system(cmd_arg)
print "--> cloud end"
```

For instance, we provide the CloudThingSpeak.py template script that accepts the following formatted content: « write_key#field_index#TC/18.5 ». If write_key and field_index are not specified, then the default write key and field index will be used. Therefore, an IoT device sending « \ !##TC/18.5 » as illustrated previously in Figure 5 will trigger at the gateway the execution of the CloudThingSpeak.py script that will upload « 18.5 » to the default ThingSpeak channel at the default field index. With the previous example, the execution of the main data upload processing loop will look as follows, with text in red printed by the cloud script while text in black are those printed by the main processing loop:

```
number of enabled clouds is 2
--> cloud[0]
uploading with python CloudThingSpeak.py
ThingSpeak: uploading
rcv msg to log (\!) on ThingSpeak ( default , 4 ): 18.5
ThingSpeak: will issue curl cmd
curl -s -k -X POST --data field4=18.5 https://api.thingspeak.com/[…]
ThingSpeak: returned code from server is 156
--> cloud[1]
uploading with python CloudGroveStreams.py
GroveStreams: uploading
Grovestreams: Uploading feed to: /api/feed?compId=node_6&TC=18.5
--> cloud end
```

More information are detailed in:

[Doc] Low-cost LoRa IoT platform part list
Latest version can be downloaded from:
https://github.com/CongducPham/tutorials/blob/master/low-cost-iot-hardware-parts.pdf

[Doc] "Low-cost LoRa gateway leaflet"
Latest version can be downloaded from:
https://github.com/CongducPham/tutorials/blob/master/Low-cost-LoRa-GW-leaflet.pdf

[Slides] "Low-cost LoRa gateway: a step-by-step tutorial"
Latest version can be downloaded from:
https://github.com/CongducPham/tutorials/blob/master/Low-cost-LoRa-GW-step-by-step.pdf

[Slides] "Low-cost LoRa IoT antenna tutorial for gateway"
Latest version can be downloaded from:
https://github.com/CongducPham/tutorials/blob/master/Low-cost-LoRa-IoT-antennaCable.pdf

[Slides] "Low-cost LoRa IoT: using the WAZIUP demo kit"
Latest version can be downloaded from:
https://github.com/CongducPham/tutorials/blob/master/Low-cost-LoRa-IoT-using-demo-kit.pdf

[Video] Build your low-cost LoRa gateway with WAZIUP
https://www.youtube.com/watch?v=mj8ItKA14PY

3.2. Software and tutorial materials available on the github

All the software for both IoT devices and gateway are distributed through a github repository: https://github.com/CongducPham/LowCostLoRaGw with extensive REAME files for explanations.

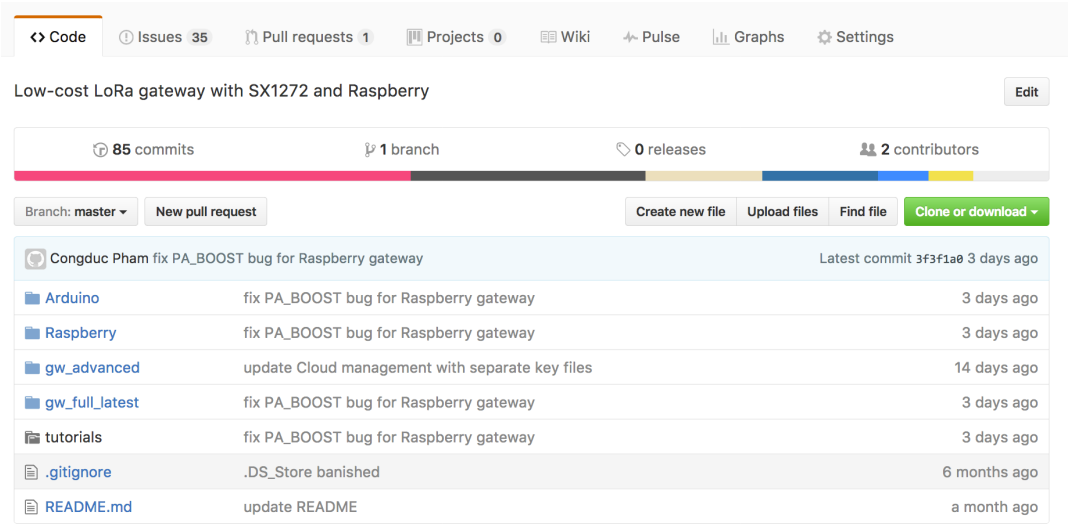


Figure 11 – github repository for IoT devices and gateway

ACRONYMS LIST

Acronym	Explanation
ACK	Acknowledgement packet
AES	Advanced Encryption Standard
BW	Bandwidth
CR	Coding Rate
CRC	Cyclic Redundancy Check
ETSI	European Telecommunications Standards Institute
IDE	Integrated Development Environment
IoT	Internet-of-Thing
LPWAN	Low Power Wide Area Networks
PHY layer	Physical Layer
REST API	REpresentational State Transfer API
RSSI	Received Signal Strength Indicator
ToA	Time-on-Air
SF	Spreading Factor
SNR	Signal to Noise Ratio
SRD	Shord Range Device

PROJECT CO-ORDINATOR CONTACT

Dr. Abdur Rahim
CREATE-NET
Via alla Cascata 56/D
Povo- 38123 Trento, Italy
Tel: (+39) 0461 408400
Fax: (+39) 0461421157
Email: abdur.rahim@create-net.org

ACKNOWLEDGEMENT

This document has been produced in the context of the H2020 WAZIUP project. The WAZIUP project consortium would like to acknowledge that the research leading to these results has received funding from the European Union's H2020 Research and Innovation Program under the Grant Agreement H2020-ICT-687607.