Degree's Final Project

Bachelor's Degree in Industrial Engineering

LoRa sniffer using python and one MPSSE cable

Report

Author: David Fructuoso Keller
Supervisor: Manuel Moreno Eguilaz
Call: June 2018

ETSEIB

School of Industrial Engineering of Barcelona

UPC

ETSEIB

Summary

The main objective of this project is to develop the necessary software to create a LoRa Sniffer for PC, capable of receiving data packets from several transmitters by using one LoRa board from Semtech.

In addition to creating the necessary library for the operation of the LoRa chip, a graphic user interface (GUI) will also be developed so that no programming knowledge is required for its use. An MPSSE cable will be used to connect the LoRa board to the computer.

First of all, a library that permits to link the LoRa board with the MPSEE cable will be developed. Next, another library will be created, and this will contain the functions to allow to modify the configurations and the modes of the LoRa chip and finally, another library will be developed referred to the GUI interface.

The used programming language in this project is python because it is the language that I am most familiarized and is the one that I have learned at the university, as well as being a widely extended and a high level programming language.

Table of Contents

SUMMARY 2

TABLE OF CONTENTS 3

1. **GLOSSARY** 5

2. INTRODUCTION 7

2.1. Objectives 7

2.2. Lora features 7

3. HARDWARE 10

3.1. SX1272 RF-LoRa module 10

3.1.1. The SX1272 board pin connections 11

3.2. USB Hi-Speed – MPSSE Cable (C232HM-DDHSL-0) 12

3.2.1. The MPSSE Cable (C232HM-DDHSL-0) connections 13

3.3. Connections between the MPSSE cable and the SX1272 module 15

4. SOFTWARE 17

4.1. The MPSSE cable library (ft.py) 17

4.1.1. Installation of the library (for windows and python 2.7.x) 17

4.1.2. The GPIO usage 17

4.1.3. Description of the SPI protocol 18

4.1.4. Usage of the SPI protocol 20

4.2. The SX1272 RF-LoRa library (sx1272.py) 23

4.2.1. The main configurable parameters for reception and transmission 25

4.2.2. Configuration functions of the class SX1272() 28

4.2.3. Reception functions of the class SX1272() 29

4.2.4. Transmition functions of the class SX1272() 33

4.3. The GUI interface library (gui_frames_3.0.py) 35

4.3.1. Frames 36

4.3.2. Elements 37

4.3.3. Graphics 38

4.3.4. Button functions 39

4.3.5. The print redirector 40

4.3.6. Problems betwen Tkinter and threads 41

5. TESTS 42

5.1. Reception test with CRC ON and variable length messages 42

ETSEIB

5.2. Reception test between a Lora node and a gateway..... 47
5.3. Emission test 48

6. BUDGET AND COSTS 50

7. ENVIRONMENTAL IMPACT 51

8. CONCLUSIONS 52

9. BIBLIOGRAPHY 53

1. Glossary

DI/DO: Data In/ Data Out

C232HM-DDHSL-0: USB Hi-Speed – MPSSE Cable's reference

CS: Chip Select (active low)

GND: Ground

GPIO: General Purpose Input/Output

GUI: Graphic User Interface

MISO: Master In Slave Out

MOSI: Master Out Slave In

MPSSE: Multi-Protocol Synchronous Serial Engine

RF: Radio Frequency

SCK/SK: Serial Clock

SPI: Serial Peripheral Interface

VCC: Collector to Collector Voltage

CRC: Circuit Redundancy Check

FSK: Frequency-Shift Keying

OOK: On-Off Keying

IOT: Internet of Things

TX: Transmitter

RX: Receiver

BW: Bandwidth

SF: Spreading Factor

SNR: Signal Noise Ratio

RSSI: Received Signal Strength Indicator

SAR: Specific Absorption Rate

2. Introduction

This project was proposed by the Department of Electronic Engineering (UPC) for the development of software for the sx1272 radio receiver/transmitter based on LoRa technology [1]. This module, in addition to the LoRa modulation, has got FSK and OOK modulations, but in this project we will only focus on the LoRa modem.

2.1. Objectives

Although different devices will be used during the project, the characteristics of these will not be discussed in detail, but basic specifications will be given for the understanding of the project.

Therefore, the objectives of the project will be:

- Develop a python library to communicate the computer with the sx1272 chip using the MPSSE C232HM cable. The communications protocol used by the chip is SPI.
- Develop a python library that allows access to the module registers to modify the configurations and send or receive data.
- Develop a graphical user interface with the python module Tkinter so that anyone can use the sniffer in a very user-friendly way.

2.2. Lora features

Long Range, abbreviated by LoRa, is a type of modulation derived from the LPWAN (low power wide area network) [1], specially designed for the internet of things. The internet of things can be used from sensors to smart cities. The fundamental characteristics of LoRa are:

- **Spread spectrum:** It consists of a modulation technique that takes a signal generated in a specific bandwidth and propagates it in the frequency domain in order to increase the bandwidth, which provides a higher quality signal and presents a

greater resistance to noise and interference.

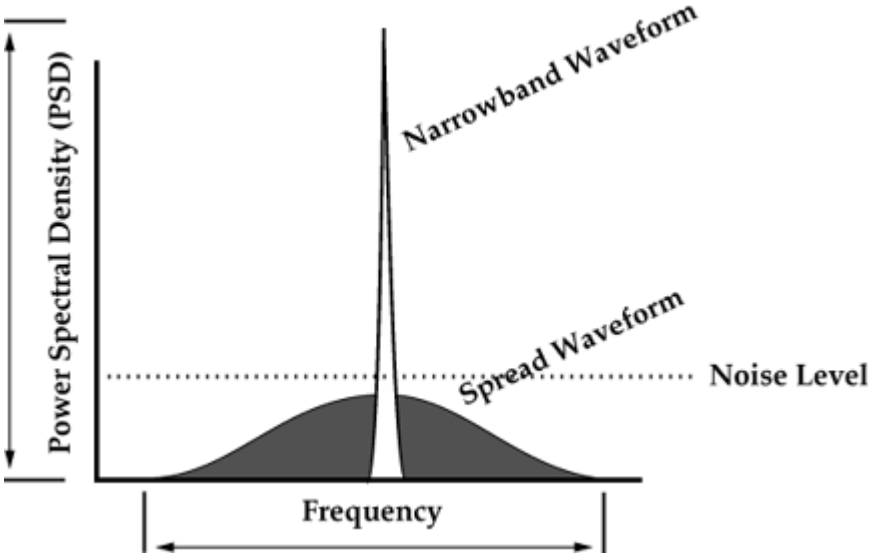


Fig.1: Comparison of a narrowband signal with a Direct Sequence Spread Spectrum signal. The narrowband signal is suppressed when transmitting spread spectrum. Source: www.tapr.org, 25 of February 2018

- **ADR (Adaptive Data Rate):** A technology that allows LoRa modulation to have a great performance, regardless of the distance between the Gateway and the node. The radio frequency settings are automatically changed for successful communication at all times.

- **AES128 end-to-end encryption:** Ensures that data is always protected.

- **Bi-directionality:** Allows sending data by offer or demand. By offer are the data that is sent by the meter, which has been pre-programmed, while by demand refers to the fact that the meter can be asked at any time for information about its status.

- **High capacity:** Supports millions of messages per base station, ideal for public network operators serving many customers.

- **Long range:** Single base station provides deep penetration in dense urban/indoor regions, plus connects rural areas up to 30 miles away.

- **Low power:** Protocol designed specifically for low power consumption extending battery lifetime up to 20 years, depending on the quantity of messages.

- **Standardized:** Improved global interoperability speeds adoption and roll out of LoRaWAN-based networks and IoT applications.

- **Low cost:** Reduces costs in three ways: infrastructure investment, operating expenses and end-node sensors.

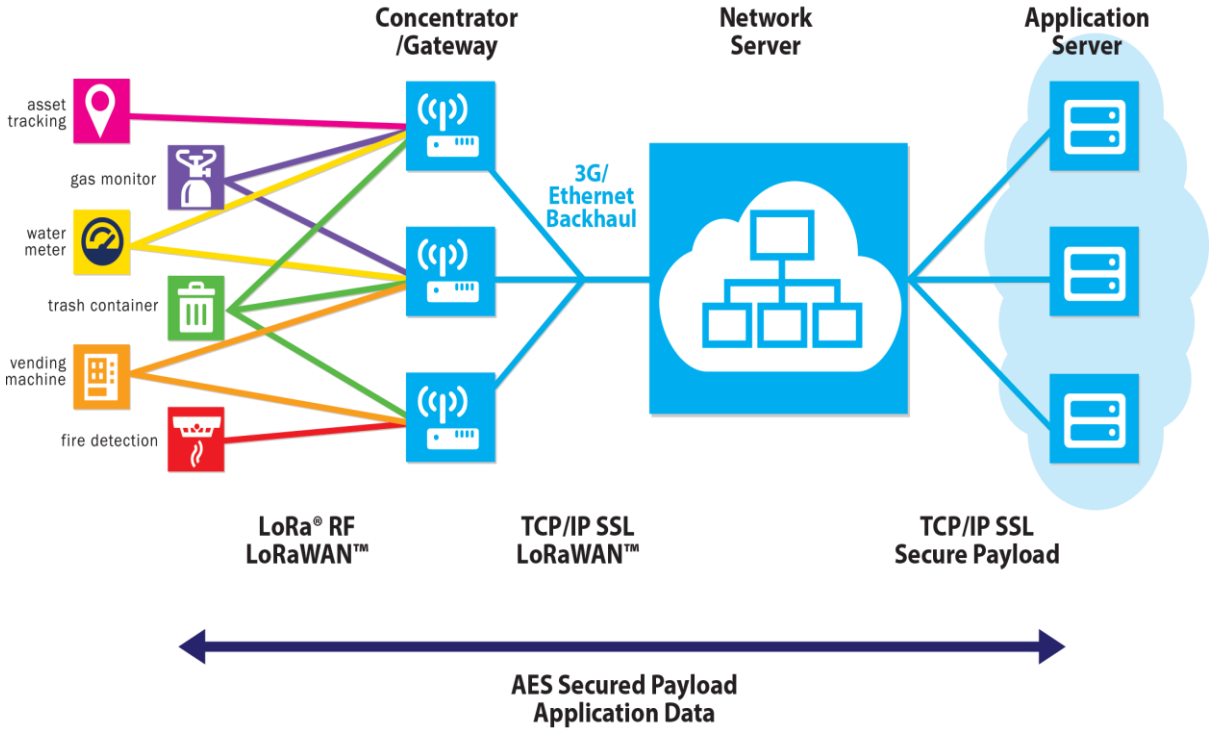


Fig. 2: Example of a network diagram. Source: https://www.semtech.com, 25 of February 2018

One example on the use of Lora network is shown in Fig. 2. At the example several sensors send data via Lora RF to the gateways. The objective of these gateways is to translate the information received in Lora RF protocol to TCP/IP SSL protocol used in the net. Once the information has reached the network server, it can be viewed by the aplication server.

3. Hardware

On this project 3 devices will be used to reach the objective of developing a Lora sniffer. These are the followings:

- **The SX1272 RF-LoRa module [2] and [3].**
- **A USB Hi-Speed – MPSSE Cable (C232HM-DDHSL-0) [4] and [5].**
- **A personal computer (PC).**

3.1. SX1272 RF-LoRa module

The SX1272 [2], as shown in Fig. 3, is a chip which incorporates the LoRa spread spectrum modem, capable of achieving significantly longer range than existing systems based on FSK or OOK modulation. For maximum flexibility the user may decide on the spread spectrum modulation bandwidth (BW), spreading factor (SF) and error correction rate (CR). Another benefit of the spread modulation is that each spreading factor is orthogonal - thus multiple transmitted signals can occupy the same channel without interfering. This also permits simple coexistence with existing FSK based systems. For more information please read the corresponding datasheet [2].

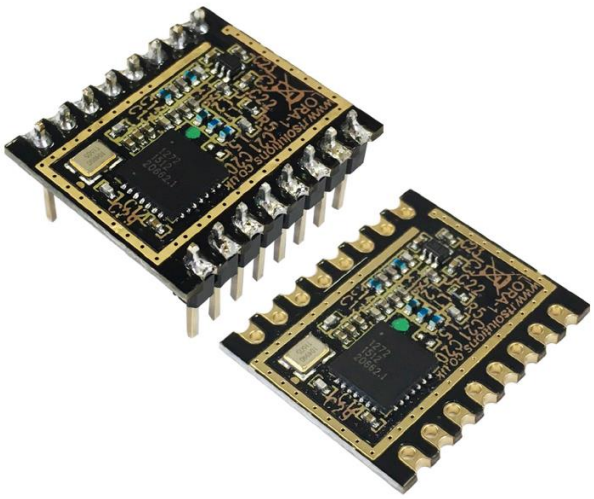


Fig. 3. Sx1272 module. Source: [3]

3.1.1. The SX1272 board pin connections

The pin connections referred to the used LoRa board are shown in Fig 4:

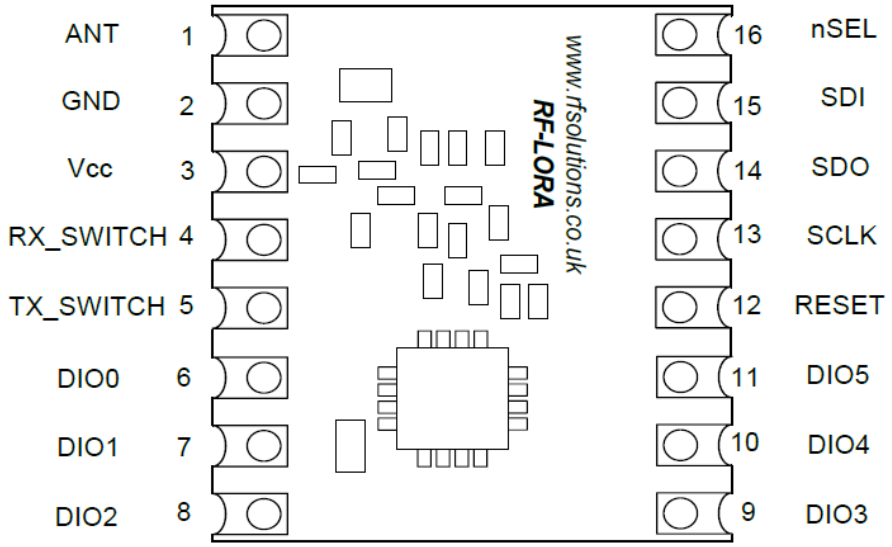


Fig. 4: Pin connection. Source: [3].

The pin description of the used pins is shown in Table 1.

PIN	DEFINITION	DIRECTION	FUNCTION
1	ANT	In/Out	Antenna pin connection
2	GND	-	Ground connection
3	VCC	In	Power connection
4	RX	In	Reception connection
5	TX	In	Transmission connection
12	RESET	In	Reset Trigger Input
13	SCLK	In	Clock
14	SDO	Out	MISO
15	SDI	In	MOSI
16	nSEL	In	Slave select

Table 1: Pin description of sx1272. Source: [3].

3.2. USB Hi-Speed – MPSSE Cable (C232HM-DDHSL-0)

This USB cable, as shown in Fig. 5, permits to establish the connection between a personal computer and the SX1272 module. The USB 2.0 Hi-Speed-MPSSE cable contains a small internal electronic circuit board, based on the FTDI FT232H, which is encapsulated into the USB connector end of the cable, and handles all the USB signaling and protocols. The cable provides a fast, simple way to connect devices with 3.3 Volt digital interfaces to USB. For more information read datasheets [4] and [5]. In our case the protocol that will be used will be the SPI protocol, because is the one that is supported by the SX1272 module.

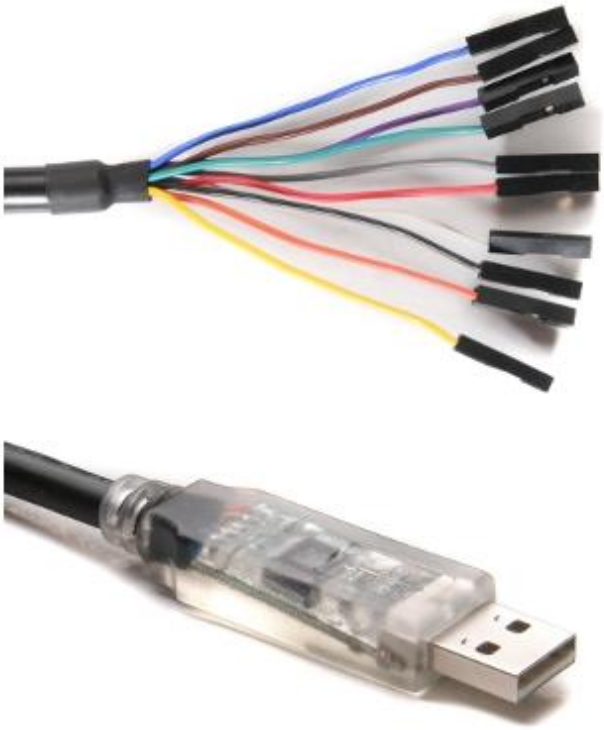


Fig. 5: MPSSE Cable (C232HM-DDHSL-0). Source: http://www.ftdichip.com, 7 of March 2018

3.2.1. The MPSSE Cable (C232HM-DDHSL-0) connections

The pin connection of the MPSSE cable is shown in Fig. 6:

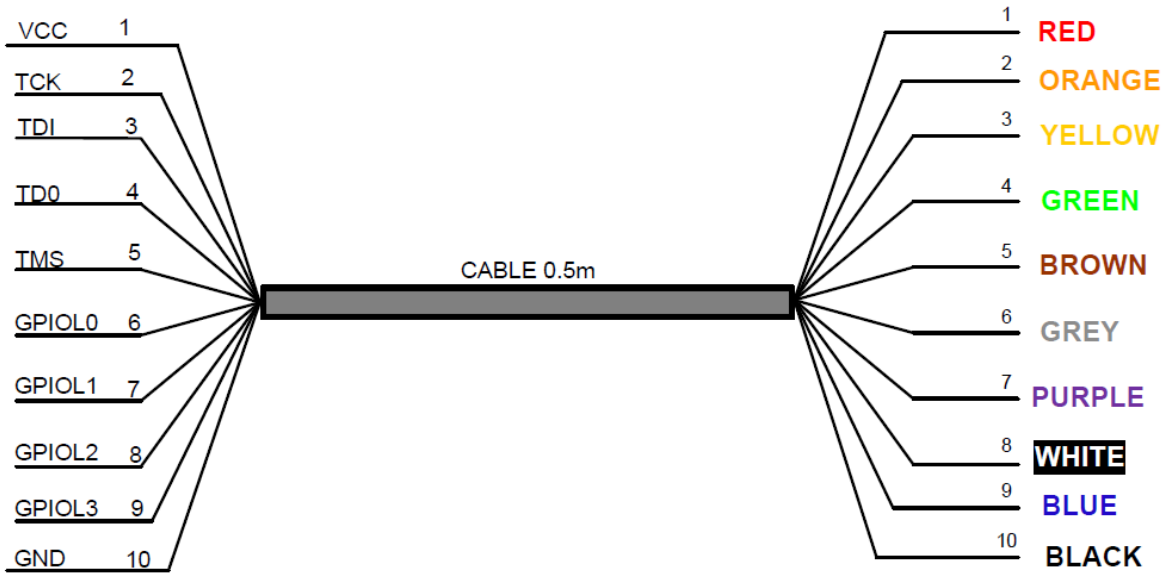


Fig. 6: Pin connection of MPSSE Cable. Source: [4].

The common cable signal description is shown in Table 2:

Colour	Pin Number	Name	Type	Description
Red	1	VCC	Output	Power Supply Output to target board.
Gray	6	GPIOL0	Input/Output	General Purpose input/output.
Purple	7	GPIOL1	Input/Output	General Purpose input/output.
White	8	GPIOL2	Input/Output	General Purpose input/output.
Blue	9	GPIOL3	Input/Output	General Purpose input/output.
Black	10	GND	GND	Device ground supply pin.

Table 2: Common cable signal description. Source: [4].

For the SPI option, the cable connection is given in Table 3:

Colour	Pin Number	Name	Type	Description
Orange	2	SK	Output	Serial Clock
Yellow	3	DO	Output	Serial data output
Green	4	DI	Input	Serial Data Input
Brown	5	CS	Output	Serial Chip Select

Table 3: MPSSE Option SPI. Source: [4].

3.3. Connections between the MPSSE cable and the SX1272 module

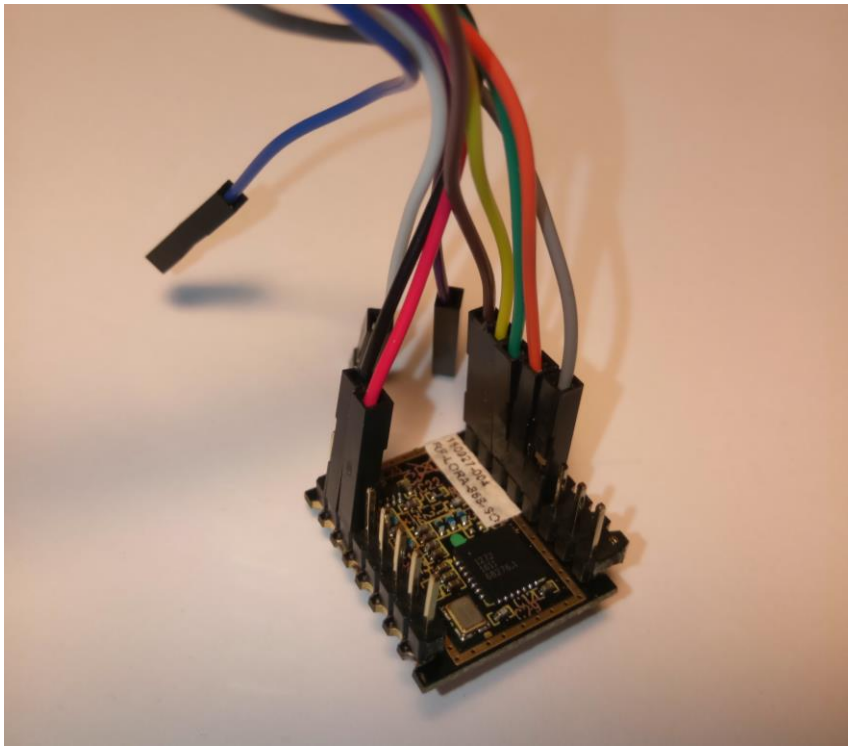


Fig. 7: Cable connections with SX1272. Own source.

Table 4 represents the connections that will be used over the whole project:

SX1272	MPSSE CABLE
GND	GND - Black
VCC	VCC - Red
RESET	GPIOL0 - Grey
SCLK	SK - Orange
SDO	DI - Green
SDI	DO - Yellow

nSEL	CS - Brown
RX	GPIOL1-Purple
TX	GPIOL2-White

Table 4: Connections between the MPSSE cable and the LoRa board. Own source.

4. Software

In this section, the methodology used to develop this project will be explained. The first thing to do was to search a library to link the USB MPSSE cable with the computer. After searching the net, it was found a library that was created for the FT232H chip by Adafruit [6] [13]. This library was named as ft.py. From this library the used classes were the FT232() and the SPI(). Once this library was successfully installed and operative, a new class was developed in the called sx1272.py library, containing all the functions to change configurations or modes of the LoRa module. In this library, the previous library was imported in order to use all the relevant methods. For this library there was not so much information on the internet and if there was, it was in C language. Finally, a GUI was developed inside the file gui.py, using the Tkinter tool provided by python. In this module, the previous libraries sx1272.py and ft.py were included.

4.1. The MPSSE cable library (ft.py)

4.1.1. Installation of the library (for windows and python 2.7.x)

Following the instructions given by the Adafruit website [7], it is very easy to install it. The main steps are:

1. Install the VCP drivers given by ftdichip [8], so that MPSEE cable would appear as a COM port and the computer would recognize it.
2. Installation of libusB driver replacing VCP drivers with zadig executable [9].
3. Installation of libftdi library needed to install the adafruit library [10].
4. Installation of the python adafruit GPIO library [11].

4.1.2. The GPIO usage

This library, in addition to the SPI and I2C communication protocols, offers the possibility to configure some pins as GPIO. In our MPSSE cable these pins are the ones that go from 6 to 9, both included. These can be configured as digital inputs or outputs. For testing it the sketch **led_test.py (see Sketch 1)** was developed. The objective was to test a led to turn on and off for a certain period of time.

After importing the necessary libraries the first thing to do is to create an object of the FT232H() class. If the MPSSE cable is not connected, it will cause an error. Next, pin 4 is declared as digital output that corresponds to pin 6 of the cable using the setup function. This change in the numbering is due to the fact that this library follows the numbering of the FT232H chip. In the loop, the output method is called which turns the led on or off for a while.

```
import time

# Import GPIO and FT232H modules.
import Adafruit_GPIO as GPIO
import Adafruit_GPIO.FT232H as FT232H

# Temporarily disable the built-in FTDI serial driver on Mac
& Linux platforms.
FT232H.use_FT232H()

# Create an FT232H object that grabs the first available
FT232H device found.
ft232h = FT232H.FT232H()

ft232h.setup(4, GPIO.OUT)  # Make pin GPIOL0 a digital
output.

while True:
     # Set pin GPIOL0 to a high level so the LED turns on.
     ft232h.output(4, GPIO.HIGH)
     print ('encendido')
     # Sleep for 0.1 second.
     time.sleep(0.1)
     # Set pin GPIOL0 to a low level so the LED turns off.
     ft232h.output(4, GPIO.LOW)
     print('apagado')
     # Sleep for 0.1 second.
     time.sleep(0.1)
```

Sketch 1: led_test.py. Source: own.

4.1.3. Description of the SPI protocol

The Serial Peripheral Interface or SPI is a 4-wire synchronous communication protocol developed by Motorola. It is a transmission protocol that allows to reach very high speeds, it was designed to communicate a microcontroller with different peripherals in a full duplex way. A full duplex communication allows to communicate from TX to RX and from RX to TX simultaneously.

The SPI bus uses usually 4 signals:

- **SCK**: Bus clock signal. This signal governs the speed at which each bit is transmitted.
- **MISO:** It is the input signal to the master, allowing the data reception from several slaves.
- **MOSI:** It is the output signal of the master, allowing the data transmission to one or several slaves.
- **CS:** It enables usually one of the slaves to which the data is sent to. This signal is optional and in some cases is not used.

The SPI bus could have more than four signals in case that we had several slaves, so per each slave we need an extra pin (in general, one chip select CS per slave).

The operation for sending information from the master to the slave is:

- The slave is enabled to receive information through the CS. Normally, the slave is enabled when the CS goes from high to low and stays in low during the whole transmission.
- The output buffer loads the byte to send.
- The clock line begins to generate the square signal where normally for each falling edge a bit is put in the MOSI line.
- The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it.

There are four possible modes that can be used in an SPI protocol:

1. **Mode 0**: CPOL = 0 and CPHA = 0. Mode in which the clock status remains in the logical low state and the information is sent in each transition from low to high (see Fig. 7). This is the mode that we are going to use following the datasheet of the sx1272 [2].

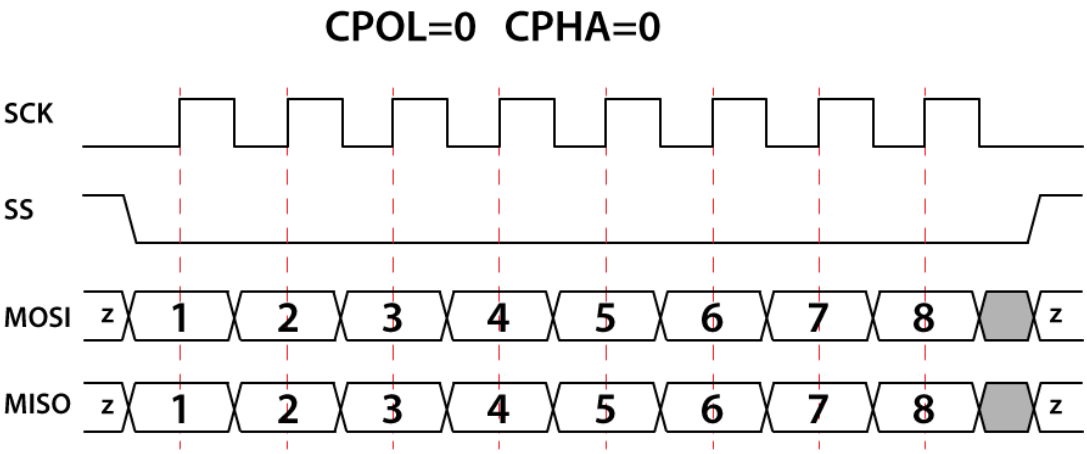


Fig. 8: SPI mode with CPOL=0 and CPHA=0. Source: http://dlnware.com, 10 of April 2018

2. **Mode 1**: CPOL = 0 and CPHA = 1. Mode in which the clock status remains in the logical low state and the information is sent in each transition from high to low.

3. **Mode 2**: CPOL = 1 and CPHA = 0. Mode in which the clock status remains in high logic status and the information is sent in each transition from low to high.

4. **Mode 3**: CPOL = 1 and CPHA = 1. Mode in which the clock status remains in high logic status and the information is sent in each transition from high to low.

4.1.4. Usage of the SPI protocol

Once the Adafruit library was installed the first test to do was to test if the LoRa worked properly by reading several registers. At the same page from Adafruit [12] there are many examples to test the reading of registers. Looking at these examples it is noted that there are two ways for reading registers and it is unknown which one is the correct one. Looking to the datasheet of the sx1272 it is found that the SPI mode is 0 and that the Most Significant Bit is the first to be clocked out.

The first way to read registers is to write first the address of the register using the write() method from the SPI() class, and then use the read() method from the SPI() class in order to read the MISO line. The second approach is to use the transfer() method from the SPI() class. But reading the description of this method and looking the datasheet it is noted that this way cannot work properly because it is a Full-Duplex communication in which an array of bytes will be clocked out the MOSI line, while simultaneously bytes will be read from the MISO line. For testing the reading of registers it is used the **test.py** sketch with the write() and read()

methods, but at first it does not work. Some delay is added after the write() method but it is still not working. Finally, reading the sketch FT232H.py from the Adafruit library [10] in the SPI() class it is realized that in the write() method the CS goes from high to low, then data is sent and after that the CS goes from low to high, and here is the problem. The CS has to stay on low level during the transmission and reception of bytes. This fact can be seen in Fig. 9.

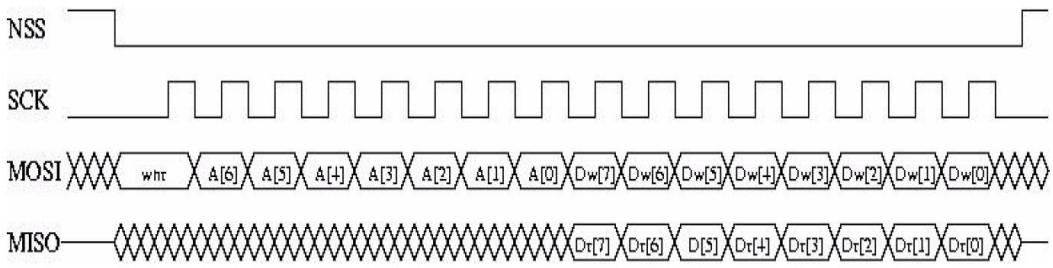


Fig. 9: Example of reading or writing a register. Source: [2].

Therefore, for solving this problem it is developed a new method called writeR() that is exactly the same to the write() method with the only difference that once the byte (this byte corresponds to the address of a register) is sent, the CS stayed in low. In the read() method it is eliminated the command that put the CS from high to low (self._assert_cs()) because the CS must be already in a low state.

Summarizing, to read registers it will be used first the writeR() method sending the address of the register, and afterwards the read() method for reading the value of this register that will be clocked out in the MISO line, whereas for writing a value in a register the default write() method of the library will be used. But in this case two bytes will be sent, the first one corresponds to the address and the second one to the value.

Reading the sx1272 datasheet [2] we have to keep in mind that the first byte that corresponds to the address byte comprises:

- A wnr bit, which is 1 for write access and 0 for read access.
- Then, 7 bits of address, MSB first.

```
import time
import Adafruit_GPIO as GPIO
import ft

# Temporarily disable FTDI serial drivers.
ft.use_FT232H()

# Find the first FT232H device.
ft232h= ft.FT232H()

# Create a SPI interface from the FT232H using pin 3 (D3) as
chip select.
# Use a clock speed of 3mhz, SPI mode 0, and most significant
bit first.
spi = ft.SPI(ft232h, cs=3, max_speed_hz=3000000, mode=0,
bitorder=ft.MSBFIRST)

# Configure digital inputs and outputs using the setup
function.
# Note that pin numbers 0 to 15 map to pins D0 to D7 then C0
to C7 on the board.
ft232h.setup(4, GPIO.OUT)    # Make pin D4 a digital input.



# Write one byte (0x01) out using the SPI protocol.
spi.writeR([0x01])
response = spi.read(1)

spi.write([0x81,0x00]) #wnr set to 1 for writing in register,
so register 0x01 now is 0x81

spi.writeR([0x01])
response1 = spi.read(1)

spi.write([0x81,0x80])
spi.write([0x81,0x81])

spi.writeR([0x01])
response2 = spi.read(1)
print 'response:',
print(hex(response[0]))
print 'response1:',
print(hex(response1[0]))
print 'response2:',
print(hex(response2[0]))
```

Sketch 2: test.py. Own source

Running the **test.py** sketch the following values of the register 0x01 in the LoRa board are obtained, as shown in Fig. 10.

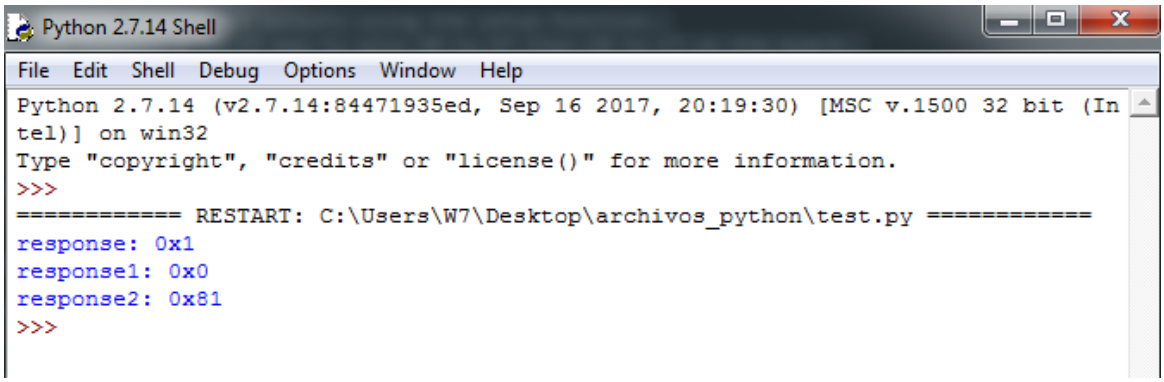


Fig. 10: Execution of test.py. Source: own.

4.2. The SX1272 RF-LoRa library (sx1272.py)

In this library the modules time.py, the Adafruit_GPIO.py, the ft.py and ctypes.py are imported. The first thing to do is to declare the registers names as global variables and then declare some configuration variables. A lot of useful code written in C++ is found in [14].

An interesting idea found in the net for changing bits is to use the ctypes.py module [15]. An example of usage can be found in sketch 3. At this example, the bits, following the datasheet and its meaning, are defined. Intel or Little Endian Structure is used, that means that the first declared bits or bytes are the least significant ones (LSB). So, in sketch 3 the declared bits are: bits 0, 1 and 2 for changing the Mode, bits 3, 4 and 5 are unused, bit 6 correspond to the AccesSharedReg and bit 7 correspond to the LongRangeMode. When c.octet is used, we are accessing to the entire byte and when we use c.bits.(name of the desired configuration) we are accessing to the defined bits for this configuration.

```
import ctypes

# Operation mode only for Lora
class OpModeBits(ctypes.LittleEndianStructure):
    _fields_ = [
        ("Mode",              ctypes.c_uint8, 3),
        ("Unused",            ctypes.c_uint8, 3),
        ("AccesSharedReg",    ctypes.c_uint8, 1),
        ("LongRangeMode",     ctypes.c_uint8, 1),
    ]


class RegOpMode(ctypes.Union):
    _fields_ = [
        ("bits", OpModeBits),
        ("octet", ctypes.c_uint8)
    ]


if __name__ == "__main__":
    c=RegOpMode()
    c.octet=0x00
    c.bits.LongRangeMode=0x01
    print(hex(c.octet))

    c.octet=0x01
    print(hex(c.octet))
```

Sketch 3: ctypes_example.py. Own source

Running sketch 3 we obtained the correct declared values:

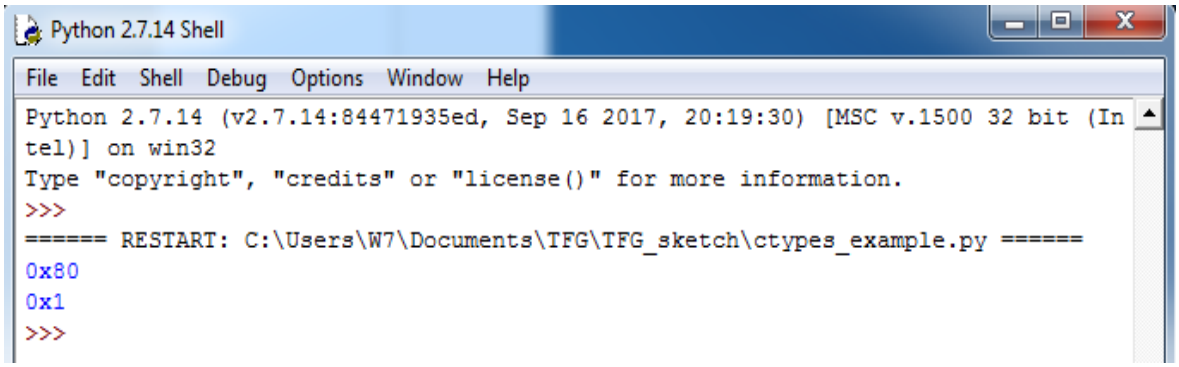


Fig. 11: Execution of ctypes_example.py. Source: own.

4.2.1. The main configurable parameters for reception and transmission

The most important parameters for LoRa reception and transmission are:

- Bandwidth: it is the frequency range occupied by a modulated carrier signal.
- Code Rate: it is the proportion of the data-stream that is useful (non-redundant). That is, if the code rate is *k/n*, for every *k* bits of useful information, the coder generates a total of *n* bits of data, of which *n-k* are redundant.
- Spreading factor: it is the number of bits encoded into each symbol (chip). A symbol is an instantaneous change in frequency. So, in an unique variation of frequency, it is associated not only one bit, but more (e.g. in SF7 -> 7 bits).

Using the formula given in Fig. 12, the bps is calculated and making all the possible combinations Fig. 13 is obtained.

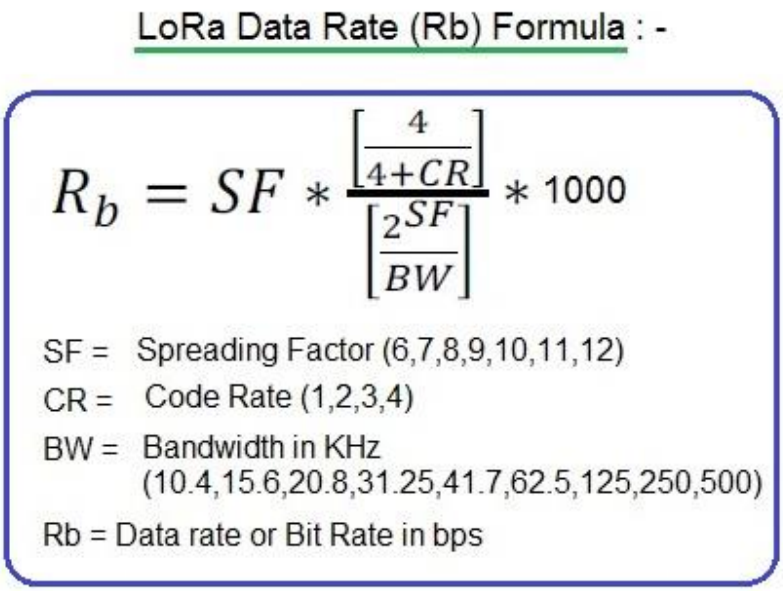


Figure 12: Data rate formula.

Source: http://www.rfwireless-world.com/calculators/LoRa-Data-Rate-Calculator.html, 22 of April 2018

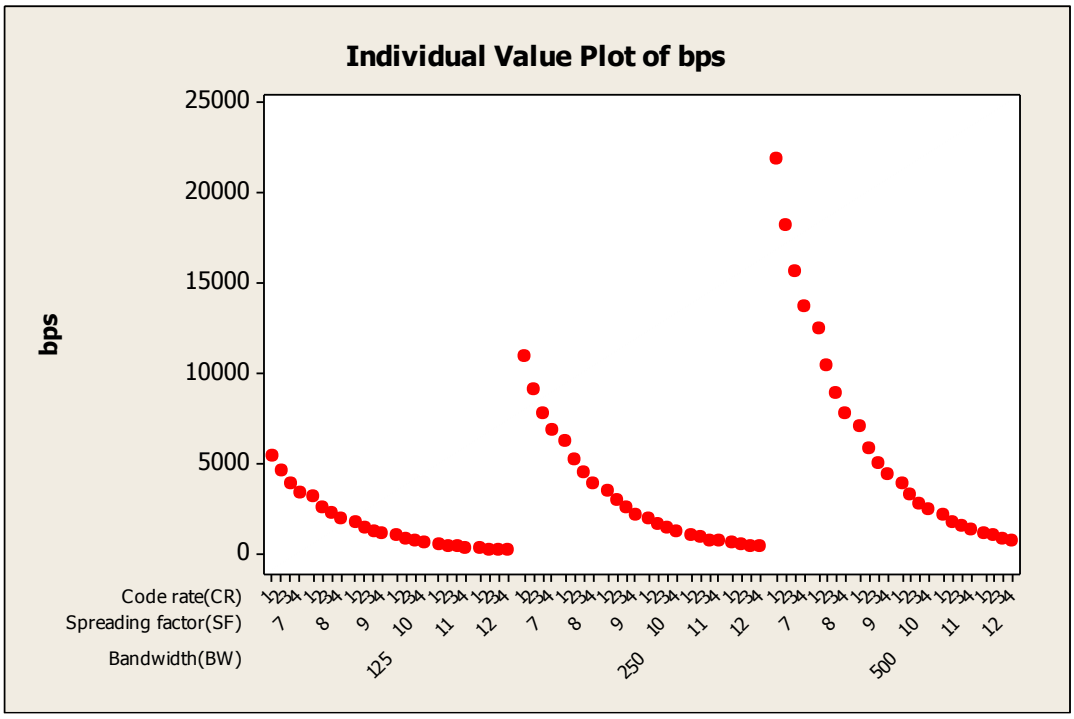


Figure 13: Data rate Lora. Own source

For making the graphic more comprehensible, the data rate is encoded as:

- Code rate 1 is an error coding rate of 4/5.
- Code rate 2 is an error coding rate of 4/6.
- Code rate 3 is an error coding rate of 4/7.
- Code rate 4 is an error coding rate of 4/8.

Moreover, the sensitivity of the module for the different possible configurations using formula given in figure 14 can also be calculated.

$$S = -174 + 10 \log_{10} BW + NF + SNR$$

Fig 14: Sensitivity calculation.

Source: http://www.techplayon.com/lora-link-budget-sensitivity-calculations-example-explained/, 22 of April 2018

Where:

- **BW** is the bandwidth in Hz.

- **NF** is the noise figure. In our case, looking at the sx1272 datasheet [2], for a gain G1 (this value will not be changed over the whole project) the noise figure is 7 dB.

- **SNR** is the ratio of signal power to the noise power and the limits for each spreading factor are given in Table 5. At this table the bit rate is calculated for a 125 KHz bandwidth.

SF (Spreading Factor)	Chips/Symbol	SNR limit	Time on Air for 10 byte packet (ms)	Bit Rate (bps)
7	128	-7.5	56	5470
8	256	-10	103	3125
9	512	-12.5	205	1758
10	1024	-15	371	977
11	2048	-17.5	741	537
12	4096	-20	1483	293

Table 5: SNR limits.

Source: http://www.techplayon.com/lora-link-budget-sensitivity-calculations-example-explained/, 23 of April 2018

- **S** is the sensivity in dbm

So, the sensitivity for all the possible configurations is given in Fig. 15.

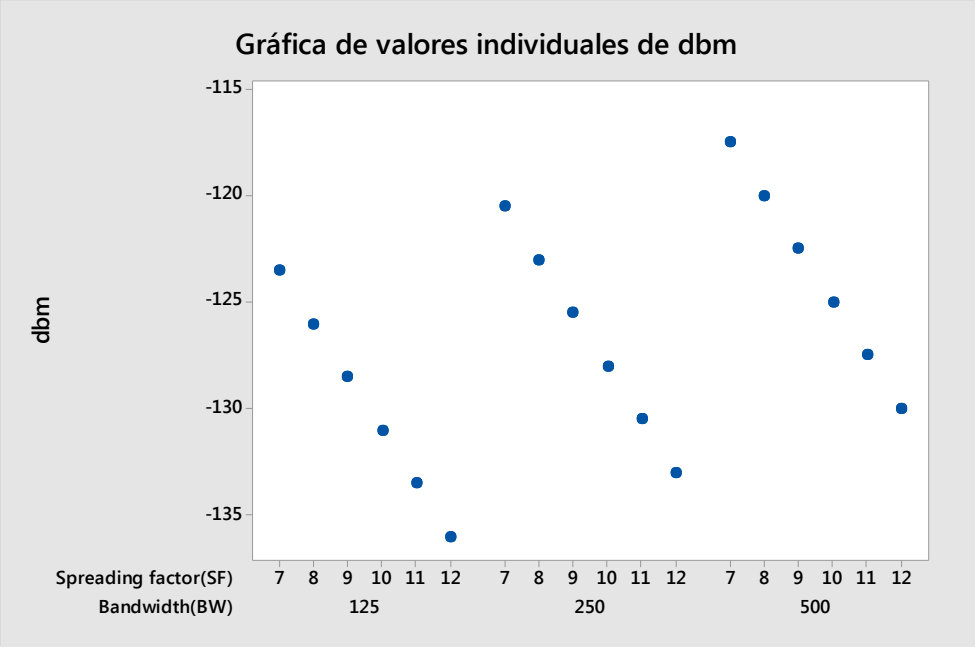


Figure 15: Sensitivity in dbm. Own source.

As conclusion, we can see that if we are near the sender, we should use lower spreading factors and high bandwidths in order to have higher speeds communications. In addition, if we are far away the sender, we should use higher spreading factors and low bandwidths in order to have more range. It is like a balance. If we want higher speeds, we are going to lose range and if we want more range, we are going to lose speed.

4.2.2. Configuration functions of the class SX1272()

In this section, we are going to describe the configuration methods of the LoRa module.

The first thing to do is to create a readRegister() and a writeRegister() methods for reading and writing registers.

Then, it is defined the ON() method, where a reset pulse is done in order to have default values of the registers. After that, the SPI configuration is done.

The value of the register RegOpMode indicates that the default value of the mode is FSK after a power on reset. Therefore, it is necessary to create a new method called setLORA() for changing from FSK to LORA mode.

The other methods (*setMode(), getMode(), setCodingrate(), setSpreadingFactor(), setBandwidth(), setLowDataRateOptimize(), getHeader(), setHeaderON(), setHeaderOFF(), getCRC(), setCRC_ON(), setCRC_OFF(), getPreambleLength(), setPreambleLength(), getChannel(), setChannel(), getPower(), setPower(), setPowerNum()*) are quite similar, basically they consisted of modifying values of the registers to configure the LoRa module.

In the methods that begin with **set,** the Mode needs to be stored because for writing registers the Mode must be Sleep or standby (the registers can only be written in Sleep or Standby mode). In this way, once the value wanted to modify of the register is changed, the previous Mode is restored. So, for example, if we are in Rx mode and we want to change the spreading factor, what the method does is to store first the mode, then changes to Standby mode for modifying the register and finally, puts the mode in Rx Mode again.

The methods that begin with **get** only need to read registers, and that can be done in all Modes.

As an example of configuration sketch 4 is shown.

```
from sx1272 import *

#Set configuration
c=SX1272()
c.ON()
c.setLORA()
c.setBandwidth(125)
c.setCodingRate(5)
c.setSpreadingFactor(7)
c.setHeaderON()
c.setCRC_OFF()
c.setChannel(387.3984375)

#Checking configuration
c.getMode()
c.getHeader()
c.getPreambleLength()
c.getChannel()
c.getPower()
c.getCRC()
```

Sketch 4: Setting a random configuration. Own source.

If sketch 4 is run, the following values are obtained:

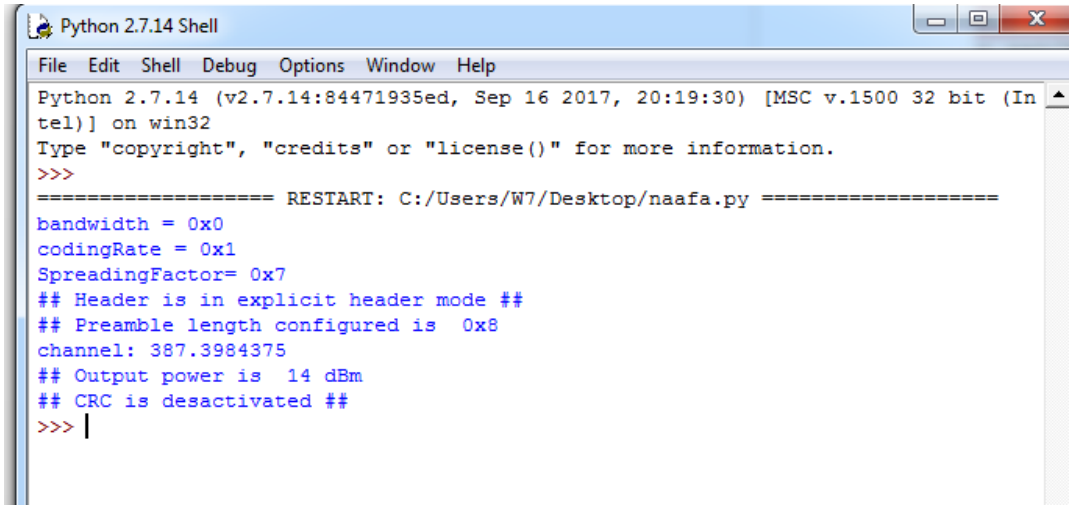


Fig. 16. Execution of sketch 4. Source: own.

4.2.3. Reception functions of the class SX1272()

The first thing to do is to create a function that initializes all the reception parameters, and this method has been called receive(). Taking a look to the datasheet [1] it is observed that for putting the modem in reception mode, the pin 4 (this pin is the Rx pin) must be triggered to High, whereas the pin 5 (this is the Tx pin) must be in a low state. For this reason two GPIO

outputs are used. After that, the Fifo address pointer is put to the Rx base address, so the received packets will be stored in the position that is defined at the Rx base address. Finally, the module is set in Rx continuous mode.

The functions getpacketSnr() and getpacketRssi() are related to the strength of the received signal of the last received packet. In the function getpacketSnr() the received data is returned in two's compliment, so it is necessary to create a new function for transforming this value to decimal. This function is called a2_to_dec().

Moreover, the validpacketsCnt() and the validheadersCnt() functions are created. These functions are in charge of counting the valid received packets and the valid received headers, respectively since the module is switched to Rx mode.

Finally, the function that reads the message from the Fifo and stored all the information in a queue is developed.

The structure of the received packet is the following (explained with an example):

The received packet looks like this:

[1, 20, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19], 'CRC_OFF', 7, -34, -26]

- The first element of the list is the packet number.
- The second element is the payload length in bytes.
- The third element is a list that contains the payload, each element of this list is a byte, so the numbers that we could represent goes from 0 to 255, both included.
- The fourth element indicate the state of the CRC (payload verification).When the CRC is disabled the string 'CRC_OFF' is generated. In case that the CRC is enabled (this parameter can only be configured in the Tx side) the given string is CRC_KO, if the payload data is corrupted or CRC_OK, if the payload data is correct.
- The fifth element is the SNR packet.
- The sixth element is the RSSI packet.
- The last element is the channel RSSI.

The function that builds and receives the packet is called getonePacket() and it works in the following way:

First of all, the variable ‘fl’, that will contain the flags information, is declared. Then, it is checked if a packet is received by reading the flags Rxdone and ValidHeader. If yes, it is checked if the CRC is enabled or not by reading the register REG_HOP_CHANNEL and if it is enabled, if it is correct or not. After that, the register REG_FIFO_RX_CURRENT_ADDR is read for knowing the memory position where the packet is located. Next, the payload is stored by reading the register FIFO n times (the value of n is given by reading the register REG_RX_NB_BYTES). Finally, the packet is built as explained. Finally we append the packet to a queue.

It is important to note that in the whole project only the explicit mode of operation is going to be used, which is the one that contains header. This mode is much more versatile that the implicit mode, because with explicit mode messages with variable length can be send whereas in implicit it is not allowed and you need to know the message length it in advance.

Therefore, for receiving packets it is necessary to run a loop calling the function getonePacket(), but if we use a simply while for that we will not be able to do more things at the same time. Therefore, for solving this, concurrent programming based on threads will be used by importing the library **PeriodicThread.py.** The way of using threads is first to declare the thread indicating the function and the time that will indicate how often the function will be executed. After that, for starting the thread to run, the method start() must be used and for killing the thread, the method cancel().

An example of reception using threads is given by runnig sketch 5.

```
from sx1272 import *
#from PeriodicThread import *

if __name__ == "__main__":
     # Set configuration
     c=SX1272()
     c.ON()
     c.setLORA()
     c.setBandwidth(125)
     c.setCodingRate(5)
     c.setSpreadingFactor(7)
     c.setHeaderON()
     c.setCRC_OFF()
     c.setChannel(387.3984375)
     c.receive() # Initialize reception parameters

     # Checking configuration
     c.getMode()
     c.getHeader()
     c.getPreambleLength()
     c.getChannel()
     c.getPower()
     c.getCRC()

     # Start threading
     c.startRx()
     try:
          while True:
               if len(c.queuerx)>0:
                    print (c.queuerx.pop())

     except KeyboardInterrupt:
          c.stopRx()
```

Sketch 5: Reception test. Own source.

Running sketch 5 and having a sender with the same configuration but set in tx mode, the following results are obtained:

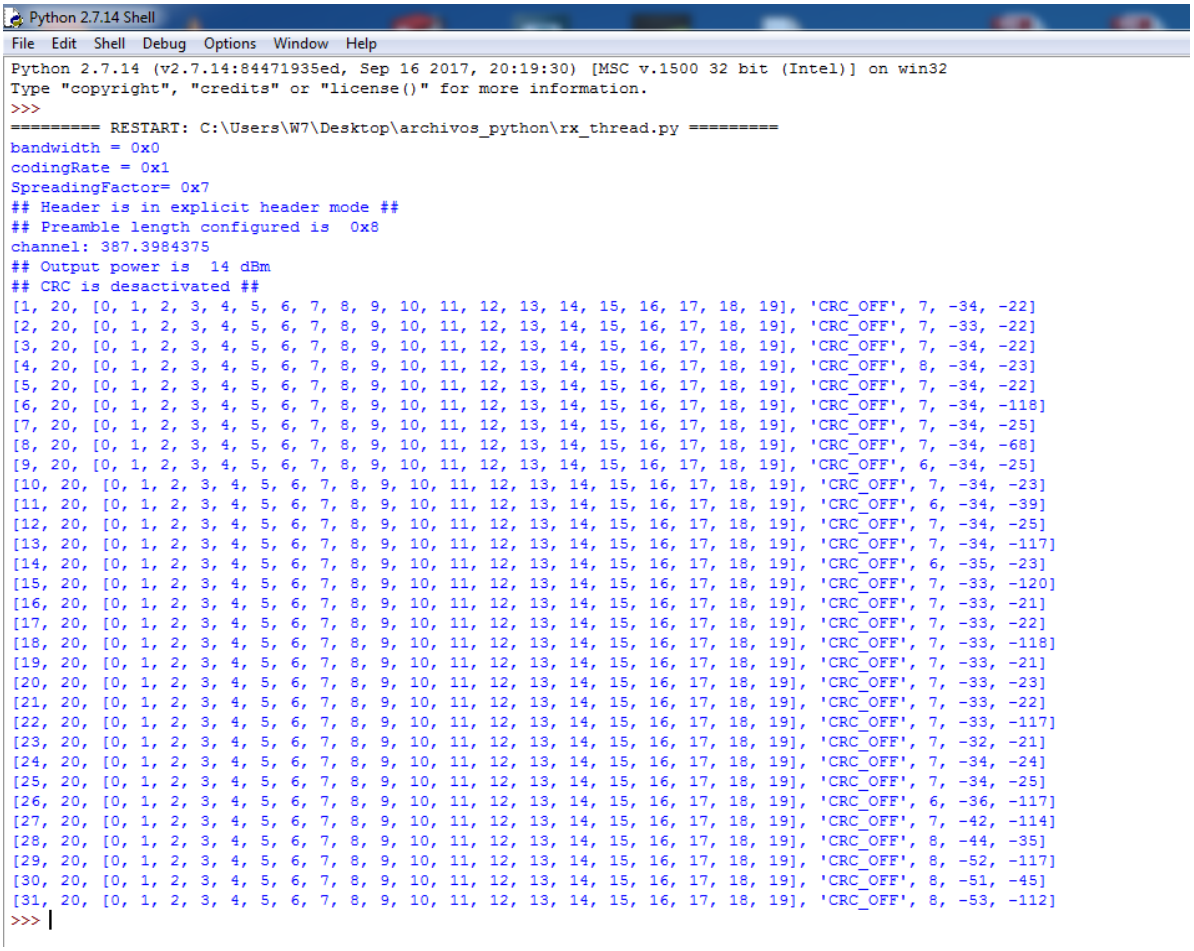
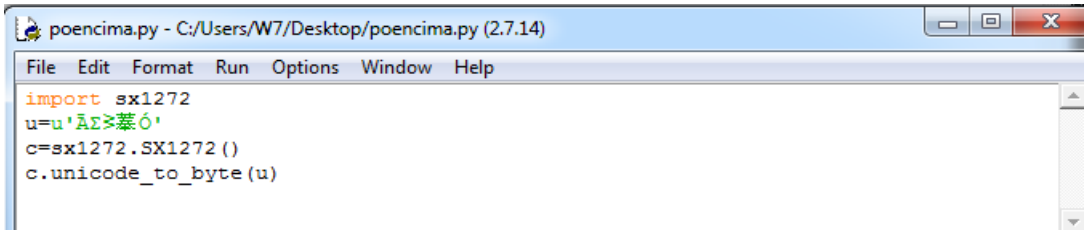


Fig. 17. Execution of sketch 5. Source: own.

4.2.4. Transmition functions of the class SX1272()

For transmitting a packet it is first necessary to do an upload of the buffer to the Fifo memory so a function that made this called upload_buffer() is written.

What the function upload_buffer(packet) does is to put the Lora modem in Stanby mode for writing the registers and next, to codify the input packet that must be a string in unicode format. Unicode codification has been chosen because is the one we get when we read a text box of the graphic user interface. With Unicode we will be able to send 65536 different characters (from 0x0000 to 0xFFFF). Due to the large number of characters that can be represented each character will take 2 bytes of memory, but the Lora modem is only able to save 1 byte in each position memory of the Fifo. Therefore, the proposed way for encoding the characters is to create a list where each character occupies 2 position of the list. The function that encodes the characters is called unicode_to_byte(). For example, if we want to represent the characters ĀΣ⩾幕Ó, the sketch 6 is created.



Sketch 6: Encoding characters ĀΣ≥搴Ó. Own source.

And the following results are obtained:

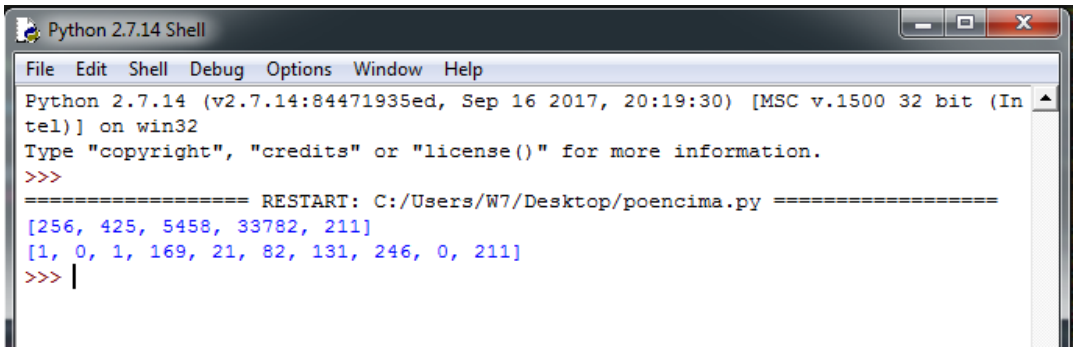


Fig. 18. Execution of sketch 6. Source: own.

The first list represents the values that takes each unicode character and the second list is uploaded to the Fifo. The length of the second list is always double the first one. Therefore, for example, the fourth character (搴) has the value 33782 in decimal, i.e. 0x83F6 in hexadecimal format. The way of creating the second list is to cut the hexadecimal number in two numbers, the first (0x83) and the second (0xF6) and placing the numbers in this way: [⋯, 0x83, 0xF6,⋯], and doing that for all the characters. If it is necessary to reconvert the number in its original state, sketch 7 can be applied.

```
Def from_bytes_to_unicode(a):
     l=[]
     for c in range(0,len(a)-1,2):
          v1=format(a[c],'#04x')
          v2=format(a[c+1],'#04x')
          l=l+[int(v1[2:]+v2[2:],16)]
     return l

u = [1, 0, 1, 169, 21, 82, 131, 246, 0, 211]
a = from_bytes_to_unicode(u)
print a
for i in range(len(a)):
     print unichr(a[i])
```

Sketch 7: Reconverting the codificated number. Own source.

The results obtained by using skech 7 are:

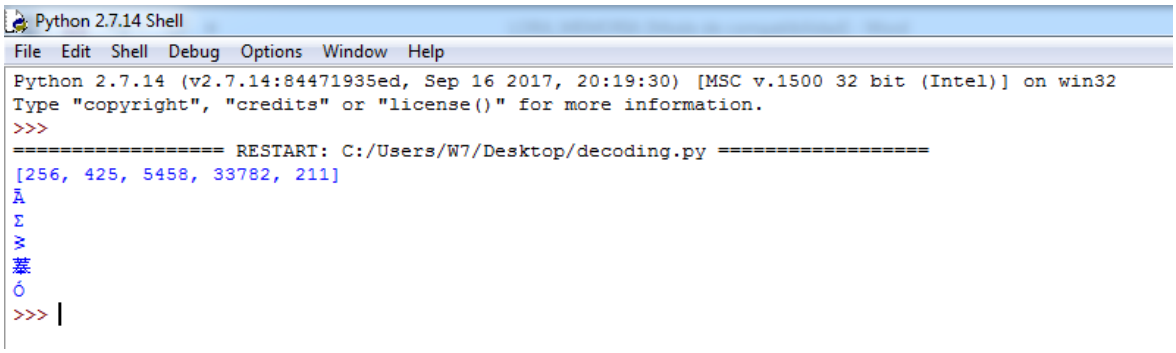


Fig. 19. Execution of sketch 7. Source: own.

Apart from this way of encoding the function hex_to_byte() is created to be used in the method upload_buffer(). It is much easier that the other one. With this method it is possible to enter a string that represents an hexadecimal number for example ('EF') and the function returns in this case a list with the value in decimal [239].

Therefore, the way of encoding can be chosen, by changing line number 1030 of the sx1272.py,i.e., the variable self.buffer=self.unicode_to_byte() or self.buffer=self.hex_to_byte().

After uploading the buffer to the Fifo, the modem has to be put in Tx single mode and wait for a certain period of time to do the transmission. Once the tranmission is done, the flag Txdone will be triggered. The function that do that is called transmit().

4.3. The GUI interface library (gui_frames_3.0.py)

The module Tkinter included in the standard python installation has been used for the development of the GUI. The Tkinter library permits to create quite complex GUI in an easy way. Basically, it consists of creating objects such as text boxes, buttons, labels… and placing them in a window.

Basically, the developed GUI has three buttons, one to put the modem in reception mode and begin to receive packets, another to transmit messages and the last button for clearing the text box. The GUI has also a panel to set the desired configuration that will be used on transmission and reception. A part from that, also two graphics are added to show the strength of each received packet in real time.

4.3.1. Frames

In our case, due to the large number of used objects, several frames are designed for positioning all of them. Each frame is a box where an object or several objects are placed. Finally, the pack() method is used for positioning each frame in the main window. Using frames it is only necessary to indicate if we want the frame left, right, at the top or at the bottom of the main window for creating all the GUI.

The frames used in our GUI are shown in Fig. 20.

```
#Frames
        self.Frame1 = tk.Frame(self.raiz,bd=2,relief=tk.RAISED)
        self.Frame11 = tk.Frame(self.Frame1,bd=2,relief=tk.RAISED)
        self.Frame12 = tk.Frame(self.Frame1,bd=2,relief=tk.RAISED)
        self.Frame13 = tk.Frame(self.Frame1,bd=2,relief=tk.RAISED)
        self.Frame14 = tk.Frame(self.Frame1,bd=2,relief=tk.RAISED)
        self.Frame15 = tk.Frame(self.Frame1,bd=2,relief=tk.RAISED)
        self.Frame16 = tk.Frame(self.Frame1,bd=2,relief=tk.RAISED)
        self.Frame2 = tk.Frame(self.raiz,bd=2)
        self.Frame21 = tk.Frame(self.Frame2,bd=2,relief=tk.RAISED)
        self.Frame22 = tk.Frame(self.Frame2,bd=2,relief=tk.RAISED)
        self.Frame23 = tk.Frame(self.Frame2,bd=2,relief=tk.RAISED)
        self.Frame3 = tk.Frame(self.raiz,bd=2,relief=tk.RAISED)
        self.Frame4 = tk.Frame(self.raiz,bd=2,relief=tk.RAISED)
        self.Frame5=tk.Frame(self.raiz,bd=2,relief=tk.RAISED,height=15)
```

Fig. 20: Used frames. Own source.

The position of each frame in the main window (self.raiz) is shown in Fig. 21.

```
#Positioning Frames
        self.Frame1.pack(side = tk.TOP)
        self.Frame11.pack(side = tk.LEFT)
        self.Frame12.pack(side = tk.LEFT)
        self.Frame13.pack(side = tk.LEFT)
        self.Frame14.pack(side = tk.LEFT)
        self.Frame15.pack(side = tk.LEFT)
        self.Frame16.pack(side = tk.LEFT)
        self.Frame2.pack(side=tk.TOP)
        self.Frame21.pack(side = tk.LEFT)
        self.Frame22.pack(side = tk.LEFT)
        self.Frame23.pack(side = tk.LEFT)
        self.Frame3.pack(side = tk.TOP,fill=tk.BOTH)#expand=tk.YES)
        self.Frame4.pack(side = tk.TOP,fill=tk.BOTH,expand=tk.YES)
        self.Frame5.pack(side=tk.TOP,fill=tk.BOTH)#expand=tk.YES)
```

Fig. 21: Frames position: Own source.

Therefore, we have 5 frames, named 1, 2, 3, 4, 5, that are positioned one above the other where frame 1 is on the top and frame 5 in the bottom. Then, we have frame 11, 12, 13, 14, 15 and 16, that are positioned inside frame 1 and positioned one to left of the other. We have

the same with frames 21, 22 and 23, that are inside frame 2.

4.3.2. Elements

The used elements in the GUI are:

- Labels: Allows to set text in a window.
- Combobox: Allows you to select an option from those available.
- Text: Allows you to write or show text in a defined box.
- Entry: Allows you to write and the get the written data.
- Button: Allows you to execute a command once you click on it.
- Scrollbar: Allows you to scroll a content in a predetermined direction.
- Canvas: This is a highly versatile widget which can be used to draw graphs and plots, create graphics editors, and implement various kinds of custom widgets. It is used to create a circle.

And the elements of the designed GUI are shown in Fig. 22.

```
#Main window
self.raiz = tk.Tk(mt_check_period=1,mt_debug=0)
self.raiz.geometry('1625x976+0+0')
self.raiz.title("Sniffer")

#Textbox
self.text box = tk.Text(self.Frame4, state=tk.DISABLED,height=0,width=0)

#Scrollbar
self.scrollb = tk.Scrollbar(self.Frame4, command=self.text box.yview,orient=tk.VERTICAL)
self.text_box['yscrollcommand']=self.scrollb.set

#Labels
self.etiq1=tk.Label(self.Frame11,text='Coding rate:')
self.etiq2=tk.Label(self.Frame11,text='Spreading factor:')
self.etiq3=tk.Label(self.Frame11,text='Bandwidth:')
self.etiq4=tk.Label(self.Frame13,text='Preamble lenght:')
self.etiq5=tk.Label(self.Frame13,text='Channel:')
self.etiq6=tk.Label(self.Frame13,text='CRC:')
self.etiq7=tk.Label(self.Frame15,text='Set power:')
self.etiq8=tk.Label(self.Frame5,text='Receiving:')


#Inputs
self.codingrate=ttk.Combobox(self.Frame12,textvariable=self.codingraten,width=10)
self.codingrate['values']=(5,6,7,8)

self.spredingfactor=ttk.Combobox(self.Frame12,textvariable=self.spredingfactorn,width=10)
self.spredingfactor['values']=(7,8,9,10,11,12)
self.bw=ttk.Combobox(self.Frame12,textvariable=self.bwn,width=10)
self.bw['values']=(125,250,500)

self.preamblelenght=tk.Entry(self.Frame14,textvariable=self.preamblelenghtn,width=13)
self.setchannel=tk.Entry(self.Frame14,textvariable=self.setchanneln,width=13)
self.crc=ttk.Combobox(self.Frame14,textvariable=self.crcn,width=10)
```

```
self.crc['values']=('ON','OFF')
self.setpower=tk.Entry(self.Frame16,textvariable=self.setpowern,width=13)
self.st = tk.Button(self.Frame21, text="Start/Stop", command=self.start,width=15)
self.data = tk.Button(self.Frame23, text="Clear window", command=self.clearwindow,width=15)
self.tx = tk.Button(self.Frame22, text="TX", command=self.TX,width=15)
#Canvas
self.circle=tk.Canvas(self.Frame5,width=15, height=15, bg='#F0F0ED')
self.item=self.circle.create_oval(2, 2, 15, 15, width=1, fill='')
```

Figure 22: Elements. Own source.

After creating the elements, everything is positioned in the frames (Fig. 23).

```
self.etiq1.pack(side=tk.TOP)
self.etiq2.pack(side=tk.TOP)
self.etiq3.pack(side=tk.TOP)
self.codingrate.pack(side=tk.TOP)
self.spredingfactor.pack(side=tk.TOP)
self.bw.pack(side=tk.TOP)
self.etiq4.pack(side=tk.TOP)
self.etiq5.pack(side=tk.TOP)
self.etiq6.pack(side=tk.TOP)
self.preamblelenght.pack(side=tk.TOP)
self.setchannel.pack(side=tk.TOP)
self.crc.pack(side=tk.TOP)
self.etiq7.pack(side=tk.TOP)
self.setpower.pack(side=tk.TOP)
self.st.pack()
self.data.pack()
self.tx.pack()
self.text_box.pack(side=tk.LEFT,expand=tk.YES,fill=tk.BOTH)
self.scrollb.pack(side=tk.LEFT,fill=tk.Y)
self.circle.pack(side=tk.RIGHT)
self.etiq8.pack(side=tk.RIGHT)
```

Figure 23: Positioning elements. Own source.

4.3.3. Graphics

For adding the graphics the library matplotlib 2.1.2 is used. A part from the real-time graphic, a toolbar is added to allow the user to save the graphics, maximize the graphs …

The part of the code that initialize the graphs is as follows: (Fig. 24).

```
#Matplot

self.f = Figure( figsize=(20, 6), dpi=80 )
self.ax=self.f.add_subplot(211)

self.ax.set_xlim(0,500)
self.ax.set_ylim(-80,-20)



self.ax.set_ylabel( 'RSSI (dbm)' )


self.ax1=self.f.add_subplot(212)
self.ax1.set_xlim(0,500)
self.ax1.set_ylim(-20,20)
self.ax1.set_xlabel( 'Packet Number' )
self.ax1.set_ylabel( 'SNR (db)' )
```

```
self.canvas = FigureCanvasTkAgg(self.f, master=self.Frame3)
self.canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=1)


#Toolbar
self.toolbar = NavigationToolbar2TkAgg(self.canvas, self.Frame3)
self.canvas._tkcanvas.pack(side=tk.TOP, fill=tk.BOTH, expand=True)
```

Figure 24: Graph elements. Own source.

Putting it all together, the front-end will have the following aspect (Fig. 25).

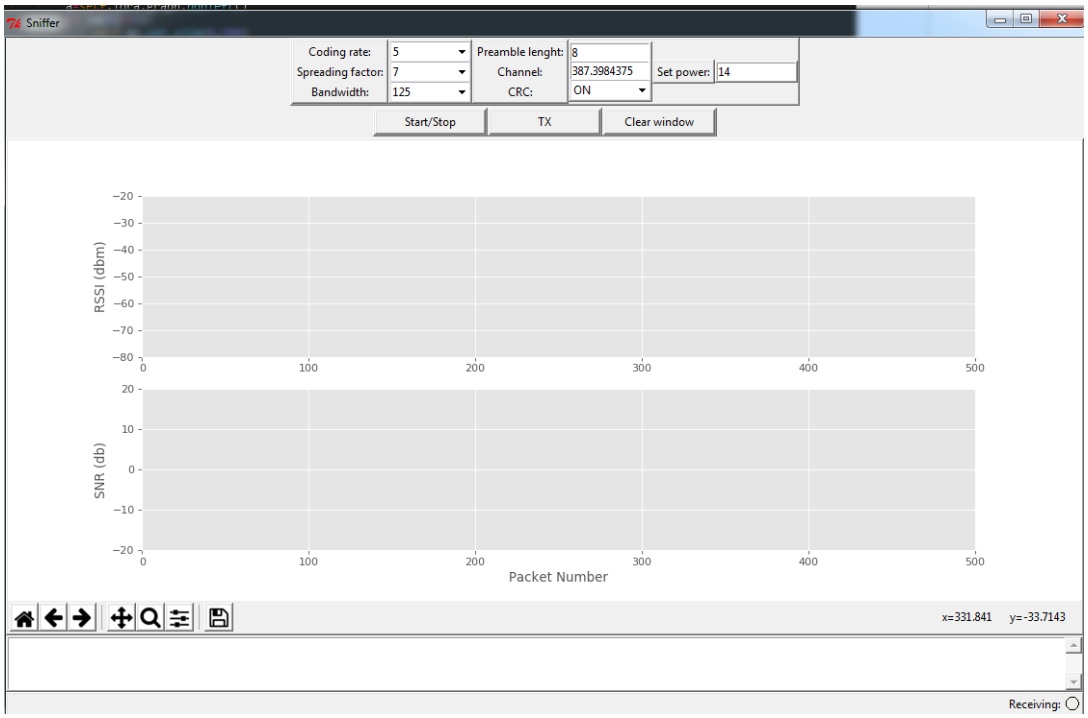


Fig. 25. Front-end view of the developed GUI for the LoRa sniffer. Own source.

4.3.4. Button functions

- Start/Stop button: Once this button is clicked, the program will ask for a folder to save all the received data into a txt file. If this is cancelled, data will not be saved. Moreover, all the received data will be displayed in the main text box. For each received packet the strength signals SNR and RSSI will be graphed. Moreover, a green light will light up. While receiving, if the same button is clicked, the program will stop receiving and the green light will be turned off. The start() function for threads and the update() function for refreshing the graphs are called.

- TX button: When it is pressed, a new window will appear, where we can write the message to send. The function TX() is called.

- Clear window button: It clears all the content of the text box. The clearwindow() function is called.

When we stop receiving or when we want to kill the program we have always to stop the threads, so we stop the threads if we are receiving and we press to the start/stop button or when we close the program.

4.3.5. The print redirector

For writing all the content in the text box an internet search has been done to see if there is any way to redirect the prints that appears normally in the python idle to the text box. Searching, reference [16] was found and one example is shown in sketch 8:

```
import Tkinter
import sys

class StdRedirector():
    def __init__(self, text_widget):
        self.text_space = text_widget

    def write(self, string):
        self.text_space.config(state=Tkinter.NORMAL)
        self.text_space.insert("end", string)
        self.text_space.see("end")
        self.text_space.config(state=Tkinter.DISABLED)

class CoreGUI():
    def __init__(self, parent):
        text_box = Tkinter.Text(parent, state=Tkinter.DISABLED)
        text_box.pack()

        sys.stdout = StdRedirector(text_box)
        sys.stderr = StdRedirector(text_box)

        output_button = Tkinter.Button(parent, text="Output", command=self.main)
        output_button.pack()

    def main(self):
        print "Std Output"
        raise ValueError("Std Error")

root = Tkinter.Tk()
CoreGUI(root)
root.mainloop()
```

Sketch 8: The print redirector. Source [12].

If output is clicked, all the prints and errors will appear now in the text box.

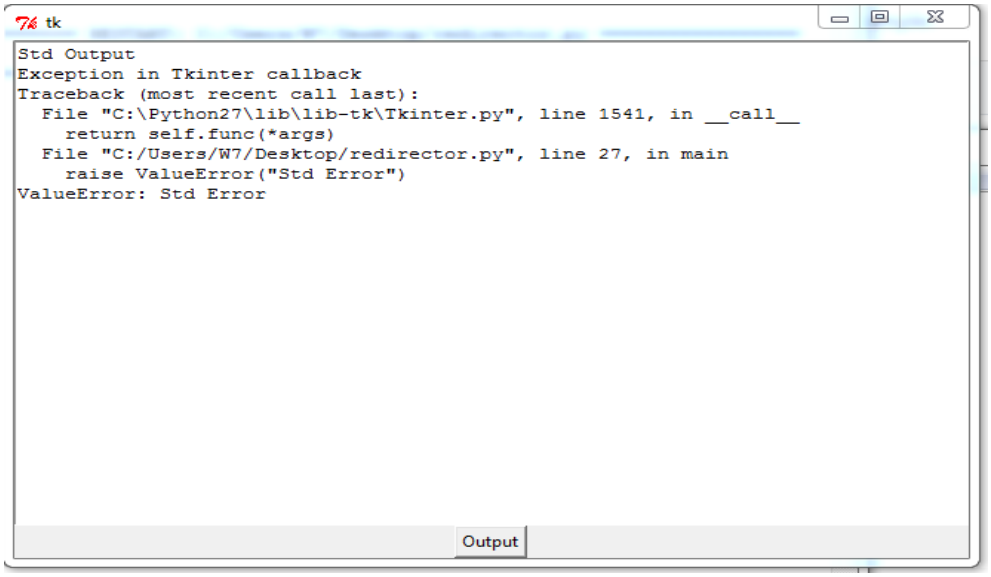


Fig. 26. Execution of sketch 8. Source: own.

4.3.6. Problems betwen Tkinter and threads

Once the GUI has been finished, it is time to test if everything is running fine and as expected. But the first version of the developed application crashes while it is receiving packets and the scrollbar is hold. Searching on the net about this problem many people talks about this in many forums. The problem is that Tkinter is not thread-safe. Finally, a library called mttkinter [17] is found and that solves this incompatibility with threads. The good thing about this library is that the only thing you have to do for using it is to import the library in your code and then you will have all the tkinter methods and now your tkinter will be thread-safe.

5. Tests

In this section the results of several tests done with the developed LoRa sniffer are shown.

5.1. Reception test with CRC ON and variable length messages

In this test one of the boards developed in the project Experiments with LoRa Communications by Alberto Molinari is used. One of the Molinari's board is used as a sender and the developed Lora sniffer as receiver. The used configuration in both boards is:

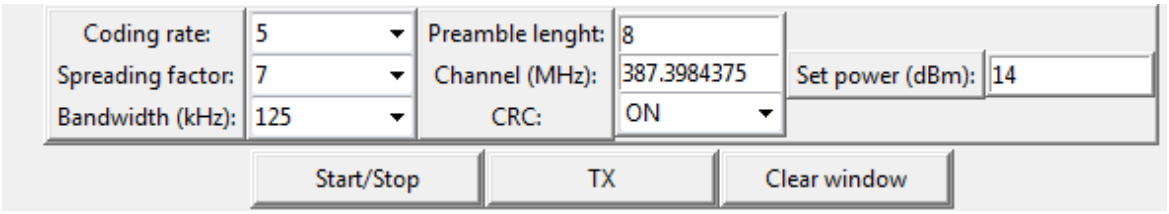
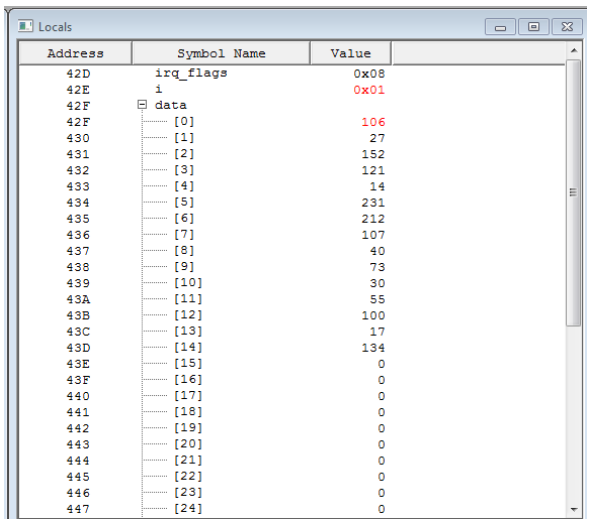
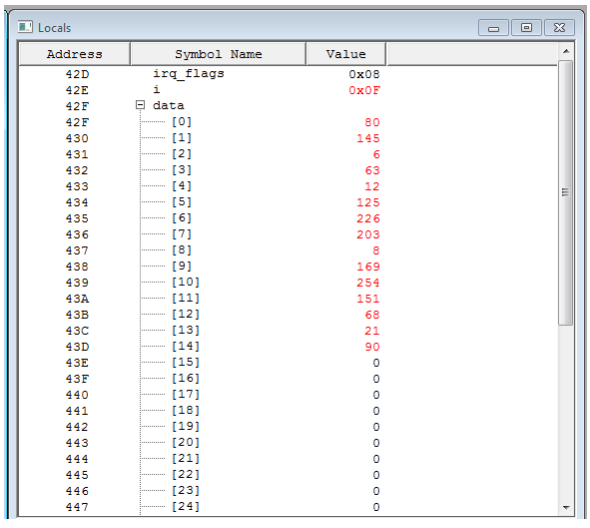
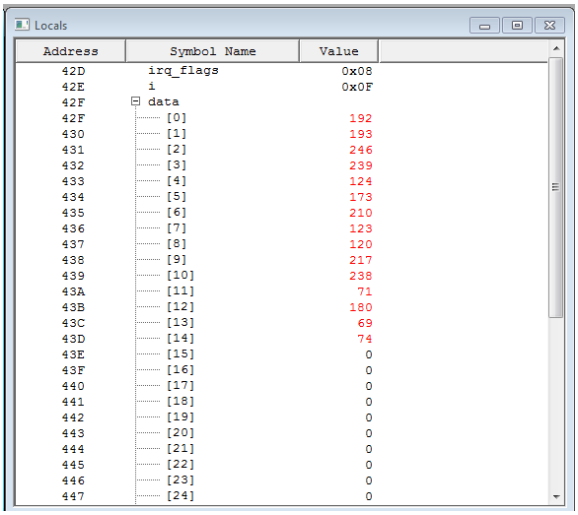


Figure 27: Test configuration. Own source.

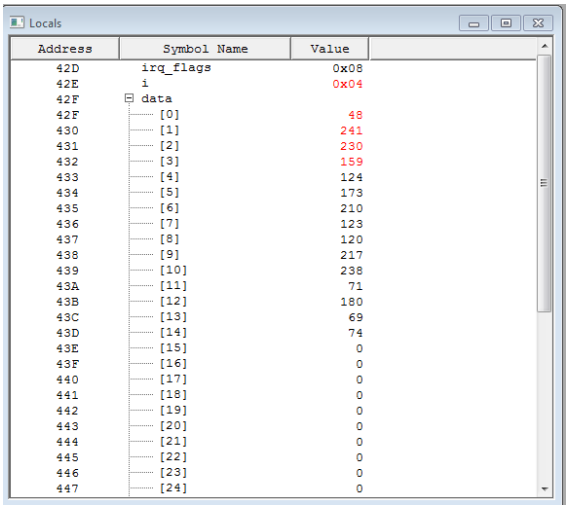
The sender board is programmed via Mplab to send random data with random length. The following 15 packets expressed in decimal codification are sent by LoRa (see table 6). The sent values are those that are shown in red:

Packet 1	Packet 2
	

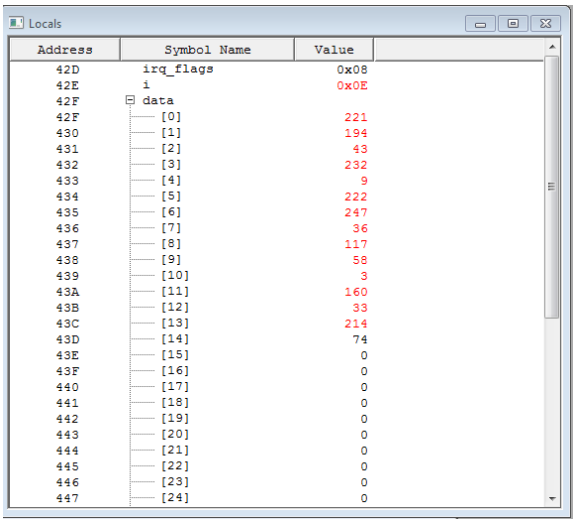
Packet 3



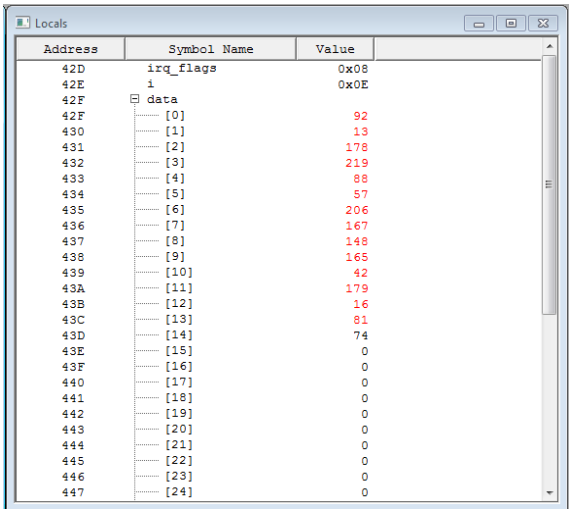
Packet 4



Packet 5

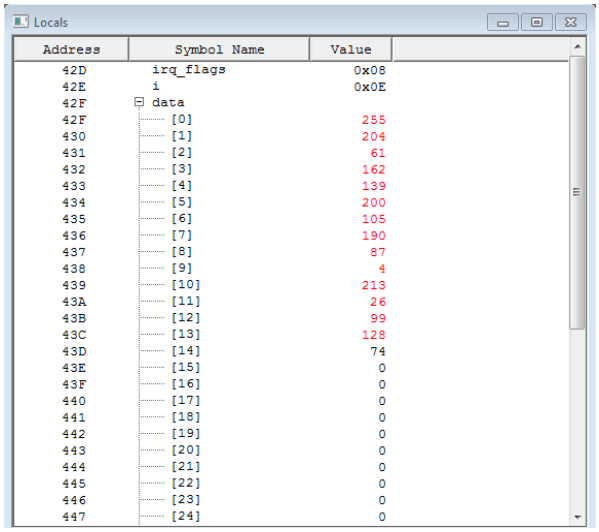


Packet 6



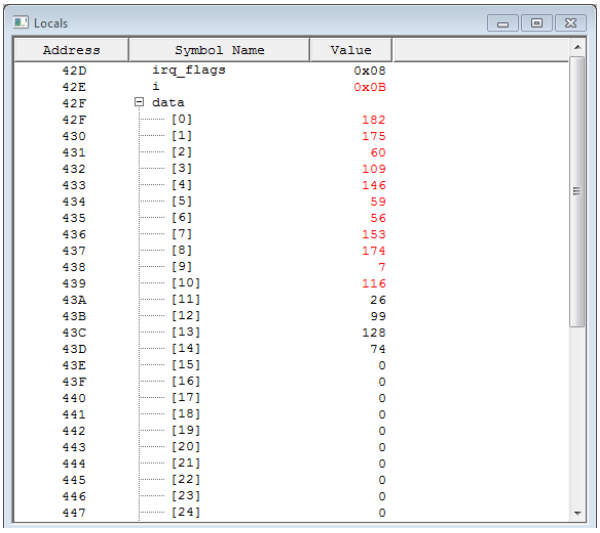
Packet 7

Packet 8


Locals

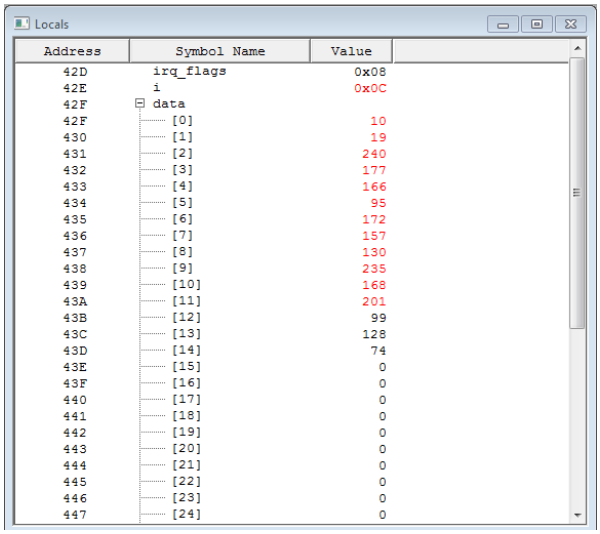
Address	Symbol Name	Value
42D	irq_flags	0x08
42E	i	0x0E
42F	data	
42F	[0]	255
430	[1]	204
431	[2]	61
432	[3]	162
433	[4]	139
434	[5]	200
435	[6]	105
436	[7]	190
437	[8]	87
438	[9]	4
439	[10]	213
43A	[11]	26
43B	[12]	99
43C	[13]	128
43D	[14]	74
43E	[15]	0
43F	[16]	0
440	[17]	0
441	[18]	0
442	[19]	0
443	[20]	0
444	[21]	0
445	[22]	0
446	[23]	0
447	[24]	0

Packet 9


Locals

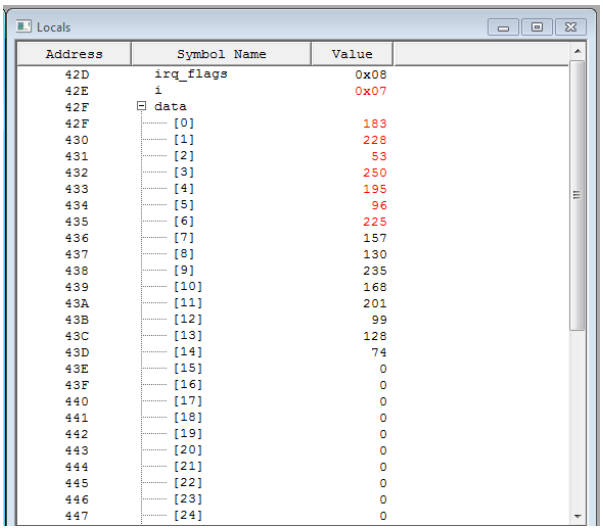
Address	Symbol Name	Value
42D	irq_flags	0x08
42E	i	0x0B
42F	data	
42F	[0]	182
430	[1]	175
431	[2]	60
432	[3]	109
433	[4]	146
434	[5]	59
435	[6]	56
436	[7]	153
437	[8]	174
438	[9]	7
439	[10]	116
43A	[11]	26
43B	[12]	99
43C	[13]	128
43D	[14]	74
43E	[15]	0
43F	[16]	0
440	[17]	0
441	[18]	0
442	[19]	0
443	[20]	0
444	[21]	0
445	[22]	0
446	[23]	0
447	[24]	0

Packet 10


Locals

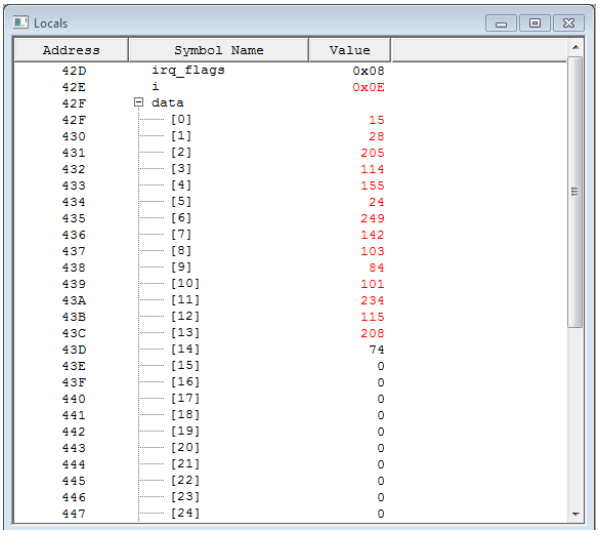
Address	Symbol Name	Value
42D	irq_flags	0x08
42E	i	0x0C
42F	data	
42F	[0]	10
430	[1]	19
431	[2]	240
432	[3]	177
433	[4]	166
434	[5]	95
435	[6]	172
436	[7]	157
437	[8]	130
438	[9]	235
439	[10]	168
43A	[11]	201
43B	[12]	99
43C	[13]	128
43D	[14]	74
43E	[15]	0
43F	[16]	0
440	[17]	0
441	[18]	0
442	[19]	0
443	[20]	0
444	[21]	0
445	[22]	0
446	[23]	0
447	[24]	0

Packet 11


Locals

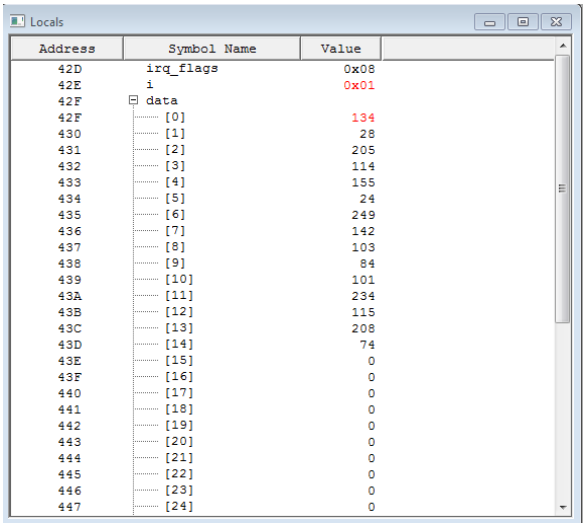
Address	Symbol Name	Value
42D	irq_flags	0x08
42E	i	0x07
42F	data	
42F	[0]	183
430	[1]	228
431	[2]	53
432	[3]	250
433	[4]	195
434	[5]	96
435	[6]	225
436	[7]	157
437	[8]	130
438	[9]	235
439	[10]	168
43A	[11]	201
43B	[12]	99
43C	[13]	128
43D	[14]	74
43E	[15]	0
43F	[16]	0
440	[17]	0
441	[18]	0
442	[19]	0
443	[20]	0
444	[21]	0
445	[22]	0
446	[23]	0
447	[24]	0

Packet 12


Locals

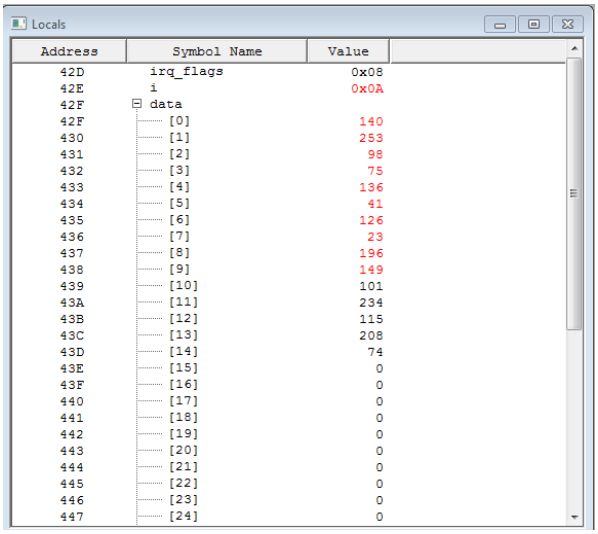
Address	Symbol Name	Value
42D	irq_flags	0x08
42E	i	0x0E
42F	data	
42F	[0]	15
430	[1]	28
431	[2]	205
432	[3]	114
433	[4]	155
434	[5]	24
435	[6]	249
436	[7]	142
437	[8]	103
438	[9]	84
439	[10]	101
43A	[11]	234
43B	[12]	115
43C	[13]	208
43D	[14]	74
43E	[15]	0
43F	[16]	0
440	[17]	0
441	[18]	0
442	[19]	0
443	[20]	0
444	[21]	0
445	[22]	0
446	[23]	0
447	[24]	0

Packet 13


Locals

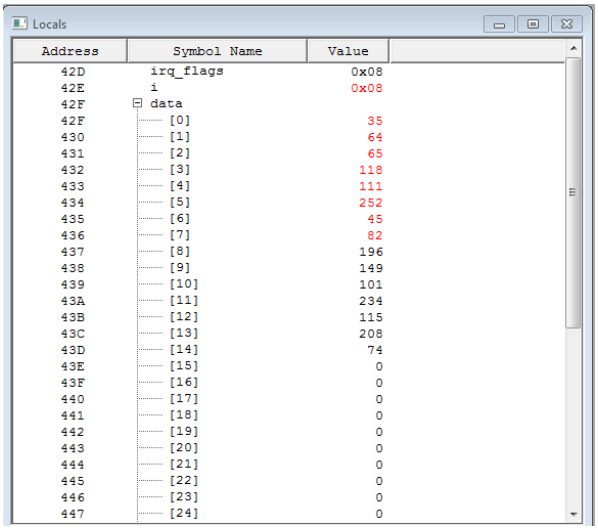
Address	Symbol Name	Value
42D	irq_flags	0x08
42E	i	0x01
42F	data	
42F	[0]	134
430	[1]	28
431	[2]	205
432	[3]	114
433	[4]	155
434	[5]	24
435	[6]	249
436	[7]	142
437	[8]	103
438	[9]	84
439	[10]	101
43A	[11]	234
43B	[12]	115
43C	[13]	208
43D	[14]	74
43E	[15]	0
43F	[16]	0
440	[17]	0
441	[18]	0
442	[19]	0
443	[20]	0
444	[21]	0
445	[22]	0
446	[23]	0
447	[24]	0

Packet 14


Locals

Address	Symbol Name	Value
42D	irq_flags	0x08
42E	i	0x0A
42F	data	
42F	[0]	140
430	[1]	253
431	[2]	98
432	[3]	75
433	[4]	136
434	[5]	41
435	[6]	126
436	[7]	23
437	[8]	196
438	[9]	149
439	[10]	101
43A	[11]	234
43B	[12]	115
43C	[13]	208
43D	[14]	74
43E	[15]	0
43F	[16]	0
440	[17]	0
441	[18]	0
442	[19]	0
443	[20]	0
444	[21]	0
445	[22]	0
446	[23]	0
447	[24]	0

Packet 15


Locals

Address	Symbol Name	Value
42D	irq_flags	0x08
42E	i	0x08
42F	data	
42F	[0]	35
430	[1]	64
431	[2]	65
432	[3]	118
433	[4]	111
434	[5]	252
435	[6]	45
436	[7]	82
437	[8]	196
438	[9]	149
439	[10]	101
43A	[11]	234
43B	[12]	115
43C	[13]	208
43D	[14]	74
43E	[15]	0
43F	[16]	0
440	[17]	0
441	[18]	0
442	[19]	0
443	[20]	0
444	[21]	0
445	[22]	0
446	[23]	0
447	[24]	0

ETSEIB

Locals

Address	Symbol Name	Value
42D	irq_flags	0x08
42E	i	0x07
42F	data	
42F	[0]	248
430	[1]	89
431	[2]	110
432	[3]	199
433	[4]	52
434	[5]	197
435	[6]	202
436	[7]	82
437	[8]	196
438	[9]	149
439	[10]	101
43A	[11]	234
43B	[12]	115
43C	[13]	208
43D	[14]	74
43E	[15]	0
43F	[16]	0
440	[17]	0
441	[18]	0
442	[19]	0
443	[20]	0
444	[21]	0
445	[22]	0
446	[23]	0
447	[24]	0

Table 6: Sent packets using Mplab software. Own source.

The received data is saved in a txt file as:

hello: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
Receiving..........



Header indicates CRC on
[1, 1, [106], 'CRC_OK', 8, -69, -112]
Header indicates CRC on
[2, 15, [80, 145, 6, 63, 12, 125, 226, 203, 8, 169, 254, 151, 68, 21, 90], 'CRC_OK', 7, -52, -111]
Header indicates CRC on
[3, 15, [192, 193, 246, 239, 124, 173, 210, 123, 120, 217, 238, 71, 180, 69, 74], 'CRC_OK', 6, -49, -112]
Header indicates CRC on
PayloadCrcError from packet: 4
[4, 4, [17, 241, 230, 159], 'CRC_KO', 5, -58, -112]
Header indicates CRC on
[5, 14, [221, 194, 43, 232, 9, 222, 247, 36, 117, 58, 3, 160, 33, 214], 'CRC_OK', 7, -50, -111]
Header indicates CRC on
[6, 14, [92, 13, 178, 219, 88, 57, 206, 167, 148, 165, 42, 179, 16, 81], 'CRC_OK', 7, -57, -111]
Header indicates CRC on
[7, 14, [255, 204, 61, 162, 139, 200, 105, 190, 87, 4, 213, 26, 99, 128], 'CRC_OK', 7, -55, -109]
Header indicates CRC on
[8, 11, [182, 175, 60, 109, 146, 59, 56, 153, 174, 7, 116], 'CRC_OK', 8, -56, -110]
Header indicates CRC on
[9, 12, [10, 19, 240, 177, 166, 95, 172, 157, 130, 235, 168, 201], 'CRC_OK', 7, -55, -109]
Header indicates CRC on
[10, 7, [183, 228, 53, 250, 195, 96, 225], 'CRC_OK', 7, -57, -112]
Header indicates CRC on
[11, 14, [15, 28, 205, 114, 155, 24, 249, 142, 103, 84, 101, 234, 115, 208], 'CRC_OK', 7, -56, -110]
Header indicates CRC on
[12, 1, [134], 'CRC_OK', 5, -67, -107]
Header indicates CRC on
[13, 10, [140, 253, 98, 75, 136, 41, 126, 23, 196, 149], 'CRC_OK', 7, -58, -112]
Header indicates CRC on
[14, 8, [35, 64, 65, 118, 111, 252, 45, 82], 'CRC_OK', 7, -58, -111]
Header indicates CRC on
[15, 7, [248, 89, 110, 199, 52, 197, 202], 'CRC_OK', 7, -57, -114]
Stop receiving...
Receiving stopped
```

Fig. 28: Received packets. Own source.

As it can be shown, that package number 4 is not the same as the one transmitted, but thanks to the CRC it is detected that the packet is corrupted.

5.2. Reception test between a Lora node and a gateway

In this test, it is wished to use our Lora sniffer in a real situation, where the messages that send a Lora node to a gateway want to be monitored. In order to do this we need the collaboration of Juan Manuel Galán Serrano, a master student doing his final master project related to LoRa and LoRaWan technologies. The project he is developing is entitled Development of 4.0 Industry Technologies based on LoRa and LoRaWAN. Basically, the test consisted of putting Juanma's boards to emit LoRa messages (one acting as a node and the other as a gateway) and seeing if the sniffer could display the messages. His board was emitting a message every 30 seconds with the configuration given below and the received messages are:

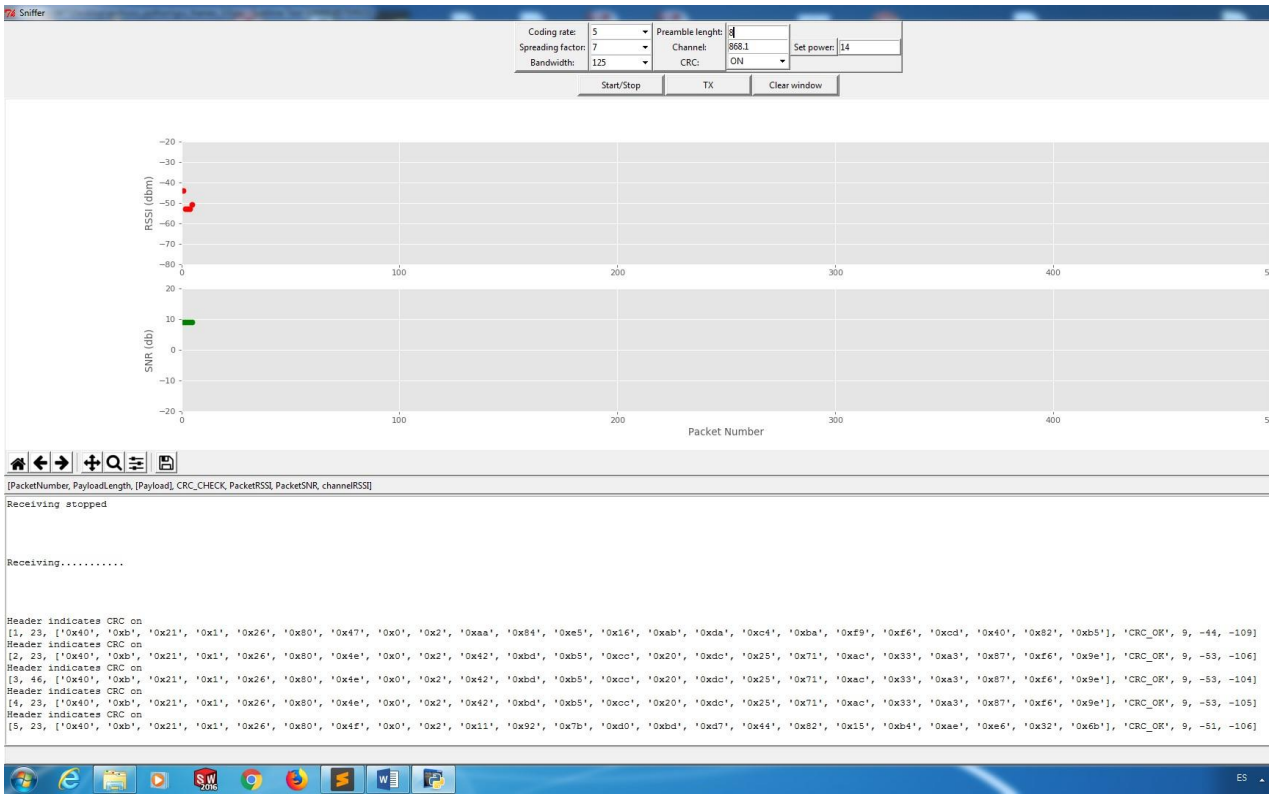


Figure 29: Received messages transmitted by the LoRa node and read by the LoRa sniffer. Own source.

In the gateway side, the results are:

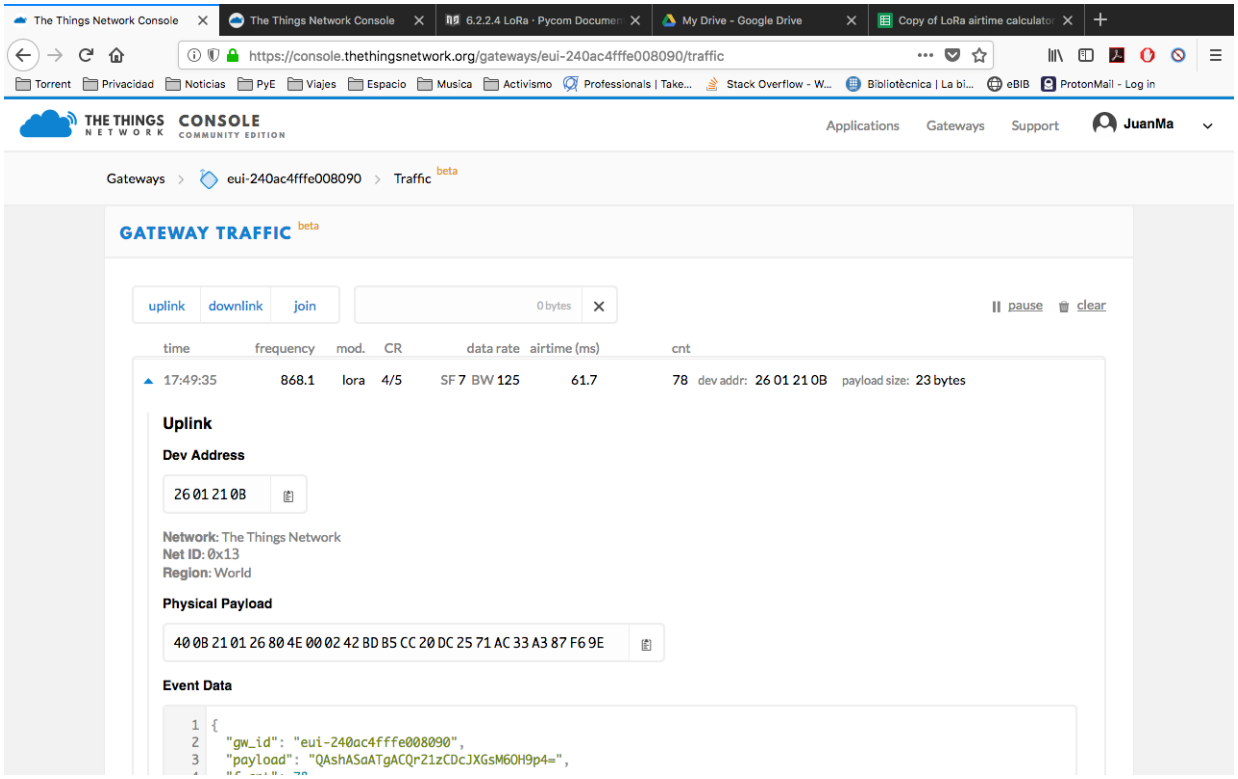


Fig. 30: Received message in the gateway. Source: Juan Manuel Galán Serrano .

Juan Manuel Galán Serrano explaines that the seventh payload number is the packet number, in this case 4E and looking to the sniffer we see one packet with the same packet number and the same payload. However, we got unexpectedly a packet of 46 bytes with the same packet number.

5.3. Emission test

For this test the developed LoRa sniffer is used as sender and one of Molinari's board as receiver. The test consists of transmit a LoRa message and see if the sender can get the message. The used configuration in the two boards is:

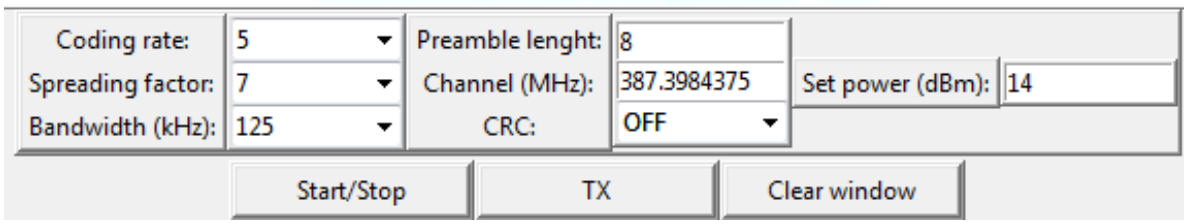


Fig. 31: Emission test parameters. Own source.

The message to transmit is:

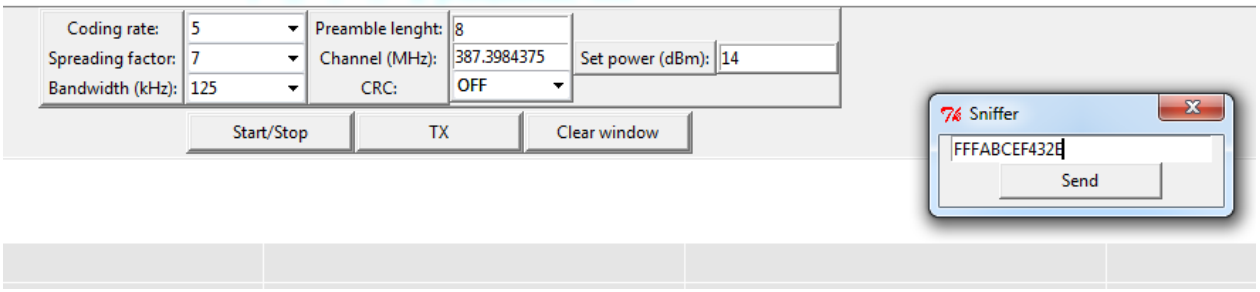


Fig. 32: Message. Own source.

In the receiver side and using Mplab, the following same results are obtained:

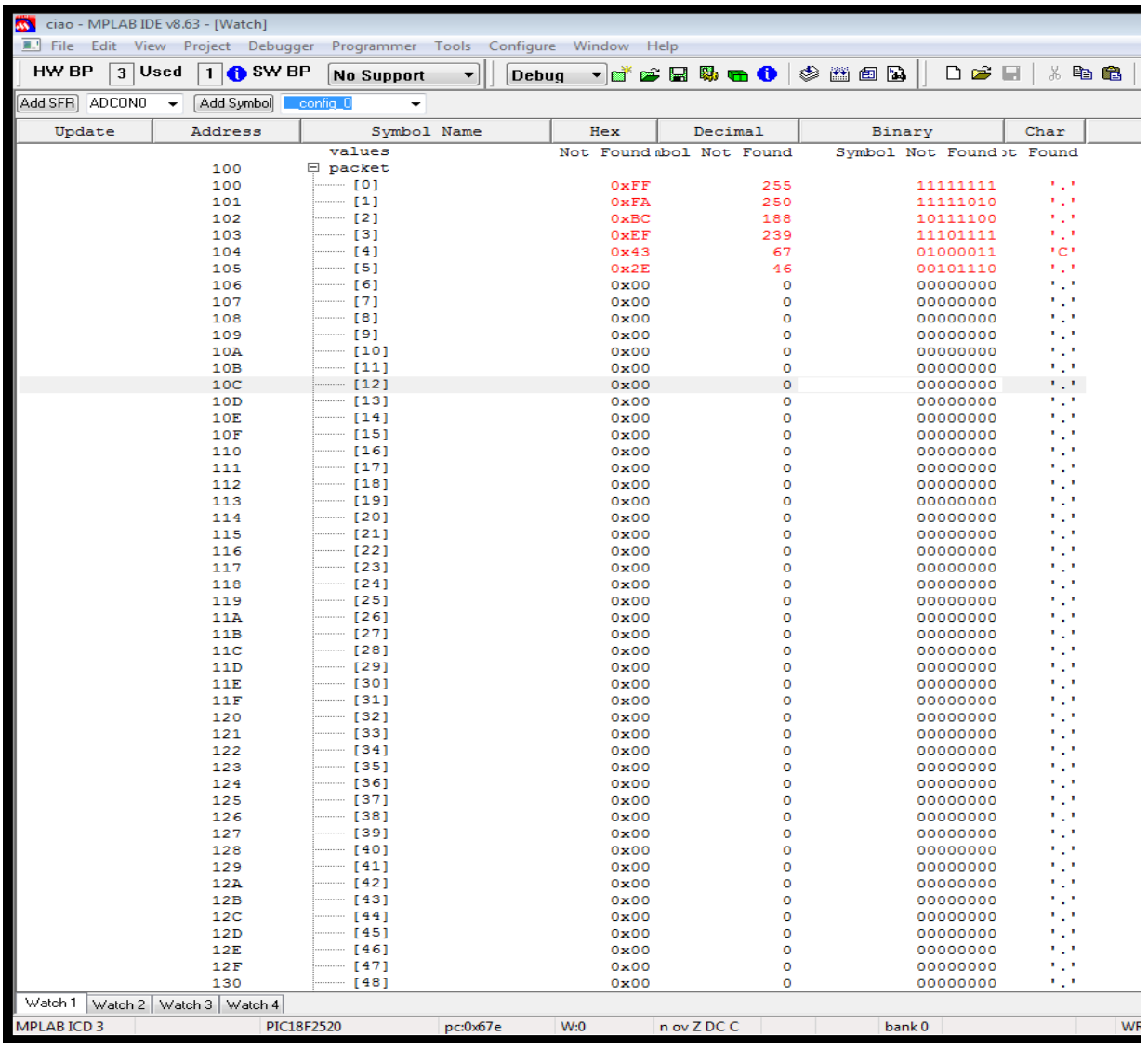


Figure 33: Reception in Molinari's board. Own source.

6. Budget and costs

The hardware costs are:

Component	Cost per unit (€)	Quantity	Total (€)	Source
SX 1272 RF-LoRa module	18,64	1	18,64	www.farnell.es
MPSEE Cable (C232HM-DDHSL-0)	23,65	1	23,65	www.ftdichip.com

Table 7: Hardware costs.

Moreover, we have to consider the cost in hours of the invested time to develop the project:

Activities	Time (hours)
Installation of python and needed libraries	2
SPI study	10
SX1272 datasheet study	30
Development of the sx1272 library	200
Development of the GUI library	100
Reception and emission tests	4
Document redaction	100
Total project time	446

Table 8: Time costs.

Total hardware costs	42,29 €
Total time costs (50€/h)	22300 €
Total	22342,29 €

Table 9: Total costs.

7. Environmental impact

In this project there are two points to take into consideration:

- Manufacturing and recycling of the chips: Reading the publication 'Tiny chips have big environmental impact' [18], it is realized that, for example, a 32-bit DRAM memory chip need at least 1272 grams of fossil fuel and chemicals to be produced and another 400 grams of fossil fuel is required to produce the electricity required to operate over its lifetime. Nowadays, semiconductors are part of our lives, mobile phones, computers, tablets… use them. So, we have to throw electronic devices as less as possible and reuse them as much as possible. When the electronic device is broken, we should recycle it. The consumer can hand the electronic device to a called 'Collection Point', that are distributed over the whole country.

- Impact of RF to humans: According to the Real Decretory 1066/2001 [19] it is observed that the max SAR in the band of 100 kHz-10 MHz in which our device operates is 0,08 W/Kg. Assuming that the maximum power that can emit our device is 17 dB and all the waves impact to a human body of 70 Kg, the SAR in this case is about 0,00072 W/Kg. Therefore, we conclude that our device is safe and works inside the defined SAR range by the Real Decreto 1066/2001 [19].

8. Conclusions

All the main objectives of this project have been successfully achieved, although many problems have appeared:

- To read a register properly by using the FT232H.py library, the Chip Select signal has to have been modified according to the LoRa chip specifications.
- In the first version of the sniffer, no packets are received from the Molinari's board.
- Problems for sending messages are found in the initial tests.
- The GUI application crashes in an unusual way while receiving packets.

The solutions for all this problems have been (in the same order):

- The ft.py library is created, including the writerR() method and the modified read method to control properly the chip select in low during the sending of the register address and the reception of the address value.
- In the receive() method of the sx1272.py library two GPIO outputs have been added, pin 4 high and pin 5 low of the LoRa module [1] for putting the device in RX mode.
- The register RegPayloadLength (0x22) must be defined to store the payload length in bytes.
- It is found that Tkinter is not thread safe and to solve this problem the module mttkinter, with all the functionalities of tkinter in addition of being thread safe, is used.

Finally, the LoRa sniffer based on python is working properly and it is a valuable tool for future projects based on LoRa communications.

9. Bibliography

[1] https://www.semtech.com/technology/lora/what-is-lora, 25 of February 2018

[2] https://www.semtech.com/uploads/documents/sx1272.pdf, 2 of March 2018

[3] https://www.rfsolutions.co.uk/downloads/1464959121DS-RFLORA.pdf, 2 of March 2018

[4] http://www.ftdichip.com/Documents/DataSheets/Cables/DS_C232HM_MPSSE_CABLE.pdf, 7 of March 2018

[5] http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232H.pdf, 7 of March 2018

[6] https://learn.adafruit.com/adafruit-ft232h-breakout/overview, 20 of March 2018

[7] https://learn.adafruit.com/adafruit-ft232h-breakout/windows-setup, 20 of March 2018

[8] http://www.ftdichip.com/Drivers/VCP.htm, 20 of March 2018

[9] http://zadig.akeo.ie/, 20 of March 2018

[10] http://downloads.sourceforge.net/project/picusb/libftdi1-1.0_devkit_mingw32_17Feb2013.zip, 25 of March 2018

[11] https://github.com/adafruit/Adafruit_Python_GPIO/archive/master.zip, 24 of March 2018

[12] https://learn.adafruit.com/adafruit-ft232h-breakout/spi, 10 of April 2018

[13] https://github.com/adafruit/Adafruit_Python_GPIO/blob/master/Adafruit_GPIO/FT232H.py, 20 of March 2018

[14] https://github.com/CongducPham/LowCostLoRaGw/tree/master/Arduino/libraries/SX1272, 16 of April 2018

[15] https://docs.python.org/2/library/ctypes.html, 20 of April 2018

[16] https://gist.github.com/RascalTwo/55ea5480af4a7b031e49, 27 of May 2018

[17] https://pythonhosted.org/mttkinter/, 30 of May 2018

[18] https://pubs.acs.org/doi/abs/10.1021/es032335x, 4 of June 2018

[19] https://www.boe.es/diario_boe/txt.php?id=BOE-A-2001-18256, 11 of June 2018