Degree's Final Project

Bachelor's Degree in Industrial Engineering

LoRa sniffer using python and one MPSSE cable

Annex

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

School of Industrial Engineering of Barcelona

1. SPI library (ft.py)

```
# Copyright (c) 2014 Adafruit Industries
# Author: Tony DiCola
#
# Permission is hereby granted, free of charge, to any person
obtaining a copy
# of this software and associated documentation files (the
"Software"), to deal
# in the Software without restriction, including without
limitation the rights
# to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell
# copies of the Software, and to permit persons to whom the
Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall
be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN
# THE SOFTWARE.

import atexit
import logging
import math
import os
import subprocess
import sys
import time

import ftdi1 as ftdi

import Adafruit_GPIO.GPIO as GPIO


logger = logging.getLogger(__name__)
```

```
FT232H_VID = 0x0403   # Default FTDI FT232H vendor ID
FT232H_PID = 0x6014   # Default FTDI FT232H product ID

MSBFIRST = 0
LSBFIRST = 1

_REPEAT_DELAY = 4


def _check_running_as_root():
    # NOTE: Checking for root with user ID 0 isn't very
portable, perhaps
    # there's a better alternative?
    if os.geteuid() != 0:
        raise RuntimeError('Expected to be run by root user!
Try running with sudo.')

def disable_FTDI_driver():
    """Disable the FTDI drivers for the current platform.
This is necessary
    because they will conflict with libftdi and accessing the
FT232H.  Note you
    can enable the FTDI drivers again by calling
enable_FTDI_driver.
    """
    logger.debug('Disabling FTDI driver.')
    if sys.platform == 'darwin':
        logger.debug('Detected Mac OSX')
        # Mac OS commands to disable FTDI driver.
        _check_running_as_root()
        subprocess.call('kextunload -b
com.apple.driver.AppleUSBFTDI', shell=True)
        subprocess.call('kextunload
/System/Library/Extensions/FTDIUSBSerialDriver.kext',
shell=True)
    elif sys.platform.startswith('linux'):
        logger.debug('Detected Linux')
        # Linux commands to disable FTDI driver.
        _check_running_as_root()
        subprocess.call('modprobe -r -q ftdi_sio',
shell=True)
        subprocess.call('modprobe -r -q usbserial',
shell=True)
    # Note there is no need to disable FTDI drivers on
Windows!

def enable_FTDI_driver():
    """Re-enable the FTDI drivers for the current
platform."""
```

```
    logger.debug('Enabling FTDI driver.')
    if sys.platform == 'darwin':
        logger.debug('Detected Mac OSX')
        # Mac OS commands to enable FTDI driver.
        _check_running_as_root()
        subprocess.check_call('kextload -b
com.apple.driver.AppleUSBFTDI', shell=True)
        subprocess.check_call('kextload
/System/Library/Extensions/FTDIUSBSerialDriver.kext',
shell=True)
    elif sys.platform.startswith('linux'):
        logger.debug('Detected Linux')
        # Linux commands to enable FTDI driver.
        _check_running_as_root()
        subprocess.check_call('modprobe -q ftdi_sio',
shell=True)
        subprocess.check_call('modprobe -q usbserial',
shell=True)

def use_FT232H():
    """Disable any built in FTDI drivers which will conflict
and cause problems
    with libftdi (which is used to communicate with the
FT232H).  Will register
    an exit function so the drivers are re-enabled on program
exit.
    """
    disable_FTDI_driver()
    atexit.register(enable_FTDI_driver)

def enumerate_device_serials(vid=FT232H_VID, pid=FT232H_PID):
    """Return a list of all FT232H device serial numbers
connected to the
    machine.  You can use these serial numbers to open a
specific FT232H device
    by passing it to the FT232H initializer's serial
parameter.
    """
    try:
        # Create a libftdi context.
        ctx = None
        ctx = ftdi.new()
        # Enumerate FTDI devices.
        device_list = None
        count, device_list = ftdi.usb_find_all(ctx, vid, pid)
        if count < 0:
            raise RuntimeError('ftdi_usb_find_all returned
error {0}: {1}'.format(count,
ftdi.get_error_string(self._ctx)))
```

```
        # Walk through list of devices and assemble list of
serial numbers.
        devices = []
        while device_list is not None:
            # Get USB device strings and add serial to list
of devices.
            ret, manufacturer, description, serial =
ftdi.usb_get_strings(ctx, device_list.dev, 256, 256, 256)
            if serial is not None:
                devices.append(serial)
            device_list = device_list.next
        return devices
    finally:
        # Make sure to clean up list and context when done.
        if device_list is not None:
            ftdi.list_free(device_list)
        if ctx is not None:
            ftdi.free(ctx)


class FT232H(GPIO.BaseGPIO):
    # Make GPIO constants that match main GPIO class for
compatibility.
    HIGH = GPIO.HIGH
    LOW  = GPIO.LOW
    IN   = GPIO.IN
    OUT  = GPIO.OUT

    def __init__(self, vid=FT232H_VID, pid=FT232H_PID,
serial=None):
        """Create a FT232H object.  Will search for the first
available FT232H
        device with the specified USB vendor ID and product
ID (defaults to
        FT232H default VID & PID).  Can also specify an
optional serial number
        string to open an explicit FT232H device given its
serial number.  See
        the FT232H.enumerate_device_serials() function to see
how to list all
        connected device serial numbers.
        """
        # Initialize FTDI device connection.
        self._ctx = ftdi.new()
        if self._ctx == 0:
            raise RuntimeError('ftdi_new failed! Is libftdi1
installed?')
        # Register handler to close and cleanup FTDI context
on program exit.
        atexit.register(self.close)
```

```
        if serial is None:
            # Open USB connection for specified VID and PID
if no serial is specified.
            self._check(ftdi.usb_open, vid, pid)
        else:
            # Open USB connection for VID, PID, serial.
            self._check(ftdi.usb_open_string,
's:{0}:{1}:{2}'.format(vid, pid, serial))
        # Reset device.
        self._check(ftdi.usb_reset)
        # Disable flow control. Commented out because it is
unclear if this is necessary.
        #self._check(ftdi.setflowctrl,
ftdi.SIO_DISABLE_FLOW_CTRL)
        # Change read & write buffers to maximum size, 65535
bytes.
        self._check(ftdi.read_data_set_chunksize, 65535)
        self._check(ftdi.write_data_set_chunksize, 65535)
        # Clear pending read data & write buffers.
        self._check(ftdi.usb_purge_buffers)
        # Enable MPSSE and syncronize communication with
device.
        self._mpsse_enable()
        self._mpsse_sync()
        # Initialize all GPIO as inputs.
        self._write('\x80\x00\x00\x82\x00\x00')
        self._direction = 0x0000
        self._level = 0x0000

    def close(self):
        """Close the FTDI device.  Will be automatically
called when the program ends."""
        if self._ctx is not None:
            ftdi.free(self._ctx)
        self._ctx = None

    def _write(self, string):
        """Helper function to call write_data on the provided
FTDI device and
        verify it succeeds.
        """
        # Get modem status. Useful to enable for debugging.
        #ret, status = ftdi.poll_modem_status(self._ctx)
        #if ret == 0:
        # logger.debug('Modem status {0:02X}'.format(status))
        #else:
        # logger.debug('Modem status error {0}'.format(ret))
        length = len(string)
        ret = ftdi.write_data(self._ctx, string, length)
```

```
        # Log the string that was written in a python hex
string format using a very
        # ugly one-liner list comprehension for brevity.
        #logger.debug('Wrote
{0}'.format(''.join(['\\x{0:02X}'.format(ord(x)) for x in
string])))
        if ret < 0:
            raise RuntimeError('ftdi_write_data failed with
error {0}: {1}'.format(ret,
ftdi.get_error_string(self._ctx)))
        if ret != length:
            raise RuntimeError('ftdi_write_data expected to
write {0} bytes but actually wrote {1}!'.format(length, ret))

    def _check(self, command, *args):
        """Helper function to call the provided command on
the FTDI device and
        verify the response matches the expected value.
        """
        ret = command(self._ctx, *args)
        logger.debug('Called ftdi_{0} and got response
{1}.'.format(command.__name__, ret))
        if ret != 0:
            raise RuntimeError('ftdi_{0} failed with error
{1}: {2}'.format(command.__name__, ret,
ftdi.get_error_string(self._ctx)))

    def _poll_read(self, expected, timeout_s=5.0):
        """Helper function to continuously poll reads on the
FTDI device until an
        expected number of bytes are returned.  Will throw a
timeout error if no
        data is received within the specified number of
timeout seconds.  Returns
        the read data as a string if successful, otherwise
raises an execption.
        """
        start = time.time()
        # Start with an empty response buffer.
        response = bytearray(expected)
        index = 0
        # Loop calling read until the response buffer is full
or a timeout occurs.
        while time.time() - start <= timeout_s:
            ret, data = ftdi.read_data(self._ctx, expected -
index)
            # Fail if there was an error reading data.
            if ret < 0:
                raise RuntimeError('ftdi_read_data failed
with error code {0}.'.format(ret))
```

```
            # Add returned data to the buffer.
            response[index:index+ret] = data[:ret]
            index += ret
            # Buffer is full, return the result data.
            if index >= expected:
                return str(response)
            time.sleep(0.01)
        raise RuntimeError('Timeout while polling
ftdi_read_data for {0} bytes!'.format(expected))

    def _mpsse_enable(self):
        """Enable MPSSE mode on the FTDI device."""
        # Reset MPSSE by sending mask = 0 and mode = 0
        self._check(ftdi.set_bitmode, 0, 0)
        # Enable MPSSE by sending mask = 0 and mode = 2
        self._check(ftdi.set_bitmode, 0, 2)

    def _mpsse_sync(self, max_retries=10):
        """Synchronize buffers with MPSSE by sending bad
opcode and reading expected
        error response.  Should be called once after enabling
MPSSE."""
        # Send a bad/unknown command (0xAB), then read buffer
until bad command
        # response is found.
        self._write('\xAB')
        # Keep reading until bad command response (0xFA 0xAB)
is returned.
        # Fail if too many read attempts are made to prevent
sticking in a loop.
        tries = 0
        sync = False
        while not sync:
            data = self._poll_read(2)
            if data == '\xFA\xAB':
                sync = True
            tries += 1
            if tries >= max_retries:
                raise RuntimeError('Could not synchronize
with FT232H!')

    def mpsse_set_clock(self, clock_hz, adaptive=False,
three_phase=False):
        """Set the clock speed of the MPSSE engine.  Can be
any value from 450hz
        to 30mhz and will pick that speed or the closest
speed below it.
        """
        # Disable clock divisor by 5 to enable faster speeds
on FT232H.
```

```
        self._write('\x8A')
        # Turn on/off adaptive clocking.
        if adaptive:
            self._write('\x96')
        else:
            self._write('\x97')
        # Turn on/off three phase clock (needed for I2C).
        # Also adjust the frequency for three-phase clocking
as specified in section 2.2.4
        # of this document:
        #
http://www.ftdichip.com/Support/Documents/AppNotes/AN_255_USB
%20to%20I2C%20Example%20using%20the%20FT232H%20and%20FT201X%2
0devices.pdf
        if three_phase:
            self._write('\x8C')
        else:
            self._write('\x8D')
        # Compute divisor for requested clock.
        # Use equation from section 3.8.1 of:
        #
http://www.ftdichip.com/Support/Documents/AppNotes/AN_108_Com
mand_Processor_for_MPSSE_and_MCU_Host_Bus_Emulation_Modes.pdf
        # Note equation is using 60mhz master clock instead
of 12mhz.
        divisor = int(math.ceil((30000000.0-
float(clock_hz))/float(clock_hz))) & 0xFFFF
        if three_phase:
            divisor = int(divisor*(2.0/3.0))
        logger.debug('Setting clockspeed with divisor value
{0}'.format(divisor))
        # Send command to set divisor from low and high byte
values.
        self._write(str(bytearray((0x86, divisor & 0xFF,
(divisor >> 8) & 0xFF))))

    def mpsse_read_gpio(self):
        """Read both GPIO bus states and return a 16 bit
value with their state.
        D0-D7 are the lower 8 bits and C0-C7 are the upper 8
bits.
        """
        # Send command to read low byte and high byte.
        self._write('\x81\x83')
        # Wait for 2 byte response.
        data = self._poll_read(2)
        # Assemble response into 16 bit value.
        low_byte = ord(data[0])
        high_byte = ord(data[1])
```

```
        logger.debug('Read MPSSE GPIO low byte = {0:02X} and
high byte = {1:02X}'.format(low_byte, high_byte))
        return (high_byte << 8) | low_byte

    def mpsse_gpio(self):
        """Return command to update the MPSSE GPIO state to
the current direction
        and level.
        """
        level_low  = chr(self._level & 0xFF)
        level_high = chr((self._level >> 8) & 0xFF)
        dir_low  = chr(self._direction & 0xFF)
        dir_high = chr((self._direction >> 8) & 0xFF)
        return str(bytearray((0x80, level_low, dir_low, 0x82,
level_high, dir_high)))

    def mpsse_write_gpio(self):
        """Write the current MPSSE GPIO state to the FT232H
chip."""
        self._write(self.mpsse_gpio())

    def get_i2c_device(self, address, **kwargs):
        """Return an I2CDevice instance using this FT232H
object and the provided
        I2C address.  Meant to be passed as the i2c_provider
parameter to objects
        which use the Adafruit_Python_GPIO library for I2C.
        """
        return I2CDevice(self, address, **kwargs)

    # GPIO functions below:

    def _setup_pin(self, pin, mode):
        if pin < 0 or pin > 15:
            raise ValueError('Pin must be between 0 and 15
(inclusive).')
        if mode not in (GPIO.IN, GPIO.OUT):
            raise ValueError('Mode must be GPIO.IN or
GPIO.OUT.')
        if mode == GPIO.IN:
            # Set the direction and level of the pin to 0.
            self._direction &= ~(1 << pin) & 0xFFFF
            self._level     &= ~(1 << pin) & 0xFFFF
        else:
            # Set the direction of the pin to 1.
            self._direction |= (1 << pin) & 0xFFFF

    def setup(self, pin, mode):
        """Set the input or output mode for a specified pin.
 Mode should be
```

```
        either OUT or IN."""
        self._setup_pin(pin, mode)
        self.mpsse_write_gpio()

    def setup_pins(self, pins, values={}, write=True):
        """Setup multiple pins as inputs or outputs at once.
 Pins should be a
        dict of pin name to pin mode (IN or OUT).  Optional
starting values of
        pins can be provided in the values dict (with pin
name to pin value).
        """
        # General implementation that can be improved by
subclasses.
        for pin, mode in iter(pins.items()):
            self._setup_pin(pin, mode)
        for pin, value in iter(values.items()):
            self._output_pin(pin, value)
        if write:
            self.mpsse_write_gpio()

    def _output_pin(self, pin, value):
        if value:
            self._level |= (1 << pin) & 0xFFFF
        else:
            self._level &= ~(1 << pin) & 0xFFFF

    def output(self, pin, value):
        """Set the specified pin the provided high/low value.
 Value should be
        either HIGH/LOW or a boolean (true = high)."""
        if pin < 0 or pin > 15:
            raise ValueError('Pin must be between 0 and 15
(inclusive).')
        self._output_pin(pin, value)
        self.mpsse_write_gpio()

    def output_pins(self, pins, write=True):
        """Set multiple pins high or low at once.  Pins
should be a dict of pin
        name to pin value (HIGH/True for 1, LOW/False for 0).
 All provided pins
        will be set to the given values.
        """
        for pin, value in iter(pins.items()):
            self._output_pin(pin, value)
        if write:
            self.mpsse_write_gpio()

    def input(self, pin):
```

```
        """Read the specified pin and return HIGH/true if the
pin is pulled high,
        or LOW/false if pulled low."""
        return self.input_pins([pin])[0]

    def input_pins(self, pins):
        """Read multiple pins specified in the given list and
return list of pin values
        GPIO.HIGH/True if the pin is pulled high, or
GPIO.LOW/False if pulled low."""
        if [pin for pin in pins if pin < 0 or pin > 15]:
            raise ValueError('Pin must be between 0 and 15
(inclusive).')
        _pins = self.mpsse_read_gpio()
        return [((_pins >> pin) & 0x0001) == 1 for pin in
pins]


class SPI(object):
    def __init__(self, ft232h, cs=None, max_speed_hz=1000000,
mode=0, bitorder=MSBFIRST):
        self._ft232h = ft232h
        # Initialize chip select pin if provided to output
high.
        if cs is not None:
            ft232h.setup(cs, GPIO.OUT)
            ft232h.set_high(cs)
        self._cs = cs
        # Initialize clock, mode, and bit order.
        self.set_clock_hz(max_speed_hz)
        self.set_mode(mode)
        self.set_bit_order(bitorder)

    def _assert_cs(self):
        if self._cs is not None:
            self._ft232h.set_low(self._cs)

    def _deassert_cs(self):
        if self._cs is not None:
            self._ft232h.set_high(self._cs)

    def set_clock_hz(self, hz):
        """Set the speed of the SPI clock in hertz.  Note
that not all speeds
        are supported and a lower speed might be chosen by
the hardware.
        """
        self._ft232h.mpsse_set_clock(hz)

    def set_mode(self, mode):
```

```
        """Set SPI mode which controls clock polarity and
phase.  Should be a
        numeric value 0, 1, 2, or 3.  See wikipedia page for
details on meaning:

http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
        """
        if mode < 0 or mode > 3:
            raise ValueError('Mode must be a value 0, 1, 2,
or 3.')
        if mode == 0:
            # Mode 0 captures on rising clock, propagates on
falling clock
            self.write_clock_ve = 1
            self.read_clock_ve  = 0
            # Clock base is low.
            clock_base = GPIO.LOW
        elif mode == 1:
            # Mode 1 capture of falling edge, propagate on
rising clock
            self.write_clock_ve = 0
            self.read_clock_ve  = 1
            # Clock base is low.
            clock_base = GPIO.LOW
        elif mode == 2:
            # Mode 2 capture on rising clock, propagate on
falling clock
            self.write_clock_ve = 1
            self.read_clock_ve  = 0
            # Clock base is high.
            clock_base = GPIO.HIGH
        elif mode == 3:
            # Mode 3 capture on falling edge, propagage on
rising clock
            self.write_clock_ve = 0
            self.read_clock_ve  = 1
            # Clock base is high.
            clock_base = GPIO.HIGH
        # Set clock and DO as output, DI as input.  Also
start clock at its base value.
        self._ft232h.setup_pins({0: GPIO.OUT, 1: GPIO.OUT, 2:
GPIO.IN}, {0: clock_base})

    def set_bit_order(self, order):
        """Set order of bits to be read/written over serial
lines.  Should be
        either MSBFIRST for most-significant first, or
LSBFIRST for
        least-signifcant first.
        """
```

```
        if order == MSBFIRST:
            self.lsbfirst = 0
        elif order == LSBFIRST:
            self.lsbfirst = 1
        else:
            raise ValueError('Order must be MSBFIRST or
LSBFIRST.')

    def write(self, data):
        """Half-duplex SPI write.  The specified array of
bytes will be clocked
        out the MOSI line.
        """
        # Build command to write SPI data.
        command = 0x10 | (self.lsbfirst << 3) |
self.write_clock_ve
        logger.debug('SPI write with command
{0:2X}.'.format(command))
        # Compute length low and high bytes.
        # NOTE: Must actually send length minus one because
the MPSSE engine
        # considers 0 a length of 1 and FFFF a length of
65536
        length = len(data)-1
        len_low  = length & 0xFF
        len_high = (length >> 8) & 0xFF
        self._assert_cs()
        # Send command and length.
        self._ft232h._write(str(bytearray((command, len_low,
len_high))))
        # Send data.
        self._ft232h._write(str(bytearray(data)))
        self._deassert_cs()

    def writeR(self, data):
        """Half-duplex SPI write.  The specified array of
bytes will be clocked
        out the MOSI line.
        """
        # Build command to write SPI data.
        command = 0x10 | (self.lsbfirst << 3) |
self.write_clock_ve
        logger.debug('SPI write with command
{0:2X}.'.format(command))
        # Compute length low and high bytes.
        # NOTE: Must actually send length minus one because
the MPSSE engine
        # considers 0 a length of 1 and FFFF a length of
65536
        length = len(data)-1
```

```
        len_low  = length & 0xFF
        len_high = (length >> 8) & 0xFF
        self._assert_cs()
        # Send command and length.
        self._ft232h._write(str(bytearray((command, len_low,
len_high))))
        # Send data.
        self._ft232h._write(str(bytearray(data)))
        #self._deassert_cs()

    def read(self, length):
        """Half-duplex SPI read.  The specified length of
bytes will be clocked
        in the MISO line and returned as a bytearray object.
        """
        # Build command to read SPI data.
        command = 0x20 | (self.lsbfirst << 3) |
(self.read_clock_ve << 2)
        logger.debug('SPI read with command
{0:2X}.'.format(command))
        # Compute length low and high bytes.
        # NOTE: Must actually send length minus one because
the MPSSE engine
        # considers 0 a length of 1 and FFFF a length of
65536
        len_low  = (length-1) & 0xFF
        len_high = ((length-1) >> 8) & 0xFF
        #self._assert_cs()
        # Send command and length.
        self._ft232h._write(str(bytearray((command, len_low,
len_high, 0x87))))
        a=bytearray(self._ft232h._poll_read(length))
        self._deassert_cs()
        # Read response bytes.
        return a

    def transfer(self, data):
        """Full-duplex SPI read and write.  The specified
array of bytes will be
        clocked out the MOSI line, while simultaneously bytes
will be read from
        the MISO line.  Read bytes will be returned as a
bytearray object.
        """
        # Build command to read and write SPI data.
        command = 0x30 | (self.lsbfirst << 3) |
(self.read_clock_ve << 2) | self.write_clock_ve
        logger.debug('SPI transfer with command
{0:2X}.'.format(command))
        # Compute length low and high bytes.
```

```
        # NOTE: Must actually send length minus one because
the MPSSE engine
        # considers 0 a length of 1 and FFFF a length of
65536
        length = len(data)
        len_low  = (length-1) & 0xFF
        len_high = ((length-1) >> 8) & 0xFF
        # Send command and length.
        self._assert_cs()
        self._ft232h._write(str(bytearray((command, len_low,
len_high))))
        self._ft232h._write(str(bytearray(data)))
        self._ft232h._write('\x87')
        self._deassert_cs()
        # Read response bytes.
        return bytearray(self._ft232h._poll_read(length))


class I2CDevice(object):
    """Class for communicating with an I2C device using the
smbus library.
    Allows reading and writing 8-bit, 16-bit, and byte array
values to registers
    on the device."""
    # Note that most of the functions in this code are
adapted from this app note:
    #
http://www.ftdichip.com/Support/Documents/AppNotes/AN_255_USB
%20to%20I2C%20Example%20using%20the%20FT232H%20and%20FT201X%2
0devices.pdf
    def __init__(self, ft232h, address, clock_hz=100000):
        """Create an instance of the I2C device at the
specified address on the
        specified I2C bus number."""
        self._address = address
        self._ft232h = ft232h
        # Enable clock with three phases for I2C.
        self._ft232h.mpsse_set_clock(clock_hz,
three_phase=True)
        # Enable drive-zero mode to drive outputs low on 0
and tri-state on 1.
        # This matches the protocol for I2C communication so
multiple devices can
        # share the I2C bus.
        self._ft232h._write('\x9E\x07\x00')
        self._idle()

    def _idle(self):
        """Put I2C lines into idle state."""
```

```
        # Put the I2C lines into an idle state with SCL and
SDA high.
        self._ft232h.setup_pins({0: GPIO.OUT, 1: GPIO.OUT, 2:
GPIO.IN},
                                {0: GPIO.HIGH, 1: GPIO.HIGH})

    def _transaction_start(self):
        """Start I2C transaction."""
        # Clear command buffer and expected response bytes.
        self._command = []
        self._expected = 0

    def _transaction_end(self):
        """End I2C transaction and get response bytes,
including ACKs."""
        # Ask to return response bytes immediately.
        self._command.append('\x87')
        # Send the entire command to the MPSSE.
        self._ft232h._write(''.join(self._command))
        # Read response bytes and return them.
        return
bytearray(self._ft232h._poll_read(self._expected))

    def _i2c_start(self):
        """Send I2C start signal. Must be called within a
transaction start/end.
        """
        # Set SCL high and SDA low, repeat 4 times to stay in
this state for a
        # short period of time.
        self._ft232h.output_pins({0: GPIO.HIGH, 1: GPIO.LOW},
write=False)
        self._command.append(self._ft232h.mpsse_gpio() *
_REPEAT_DELAY)
        # Now drop SCL to low (again repeat 4 times for short
delay).
        self._ft232h.output_pins({0: GPIO.LOW, 1: GPIO.LOW},
write=False)
        self._command.append(self._ft232h.mpsse_gpio() *
_REPEAT_DELAY)

    def _i2c_idle(self):
        """Set I2C signals to idle state with SCL and SDA at
a high value. Must
        be called within a transaction start/end.
        """
        self._ft232h.output_pins({0: GPIO.HIGH, 1:
GPIO.HIGH}, write=False)
        self._command.append(self._ft232h.mpsse_gpio() *
_REPEAT_DELAY)
```

```
    def _i2c_stop(self):
        """Send I2C stop signal. Must be called within a
transaction start/end.
        """
        # Set SCL low and SDA low for a short period.
        self._ft232h.output_pins({0: GPIO.LOW, 1: GPIO.LOW},
write=False)
        self._command.append(self._ft232h.mpsse_gpio() *
_REPEAT_DELAY)
        # Set SCL high and SDA low for a short period.
        self._ft232h.output_pins({0: GPIO.HIGH, 1: GPIO.LOW},
write=False)
        self._command.append(self._ft232h.mpsse_gpio() *
_REPEAT_DELAY)
        # Finally set SCL high and SDA high for a short
period.
        self._ft232h.output_pins({0: GPIO.HIGH, 1:
GPIO.HIGH}, write=False)
        self._command.append(self._ft232h.mpsse_gpio() *
_REPEAT_DELAY)

    def _i2c_read_bytes(self, length=1):
        """Read the specified number of bytes from the I2C
bus.  Length is the
        number of bytes to read (must be 1 or more).
        """
        for i in range(length-1):
            # Read a byte and send ACK.
            self._command.append('\x20\x00\x00\x13\x00\x00')
            # Make sure pins are back in idle state with
clock low and data high.
            self._ft232h.output_pins({0: GPIO.LOW, 1:
GPIO.HIGH}, write=False)
            self._command.append(self._ft232h.mpsse_gpio())
        # Read last byte and send NAK.
        self._command.append('\x20\x00\x00\x13\x00\xFF')
        # Make sure pins are back in idle state with clock
low and data high.
        self._ft232h.output_pins({0: GPIO.LOW, 1: GPIO.HIGH},
write=False)
        self._command.append(self._ft232h.mpsse_gpio())
        # Increase expected number of bytes.
        self._expected += length

    def _i2c_write_bytes(self, data):
        """Write the specified number of bytes to the
chip."""
        for byte in data:
            # Write byte.
```

```
            self._command.append(str(bytearray((0x11, 0x00,
0x00, byte))))
            # Make sure pins are back in idle state with
clock low and data high.
            self._ft232h.output_pins({0: GPIO.LOW, 1:
GPIO.HIGH}, write=False)
            self._command.append(self._ft232h.mpsse_gpio() *
_REPEAT_DELAY)
            # Read bit for ACK/NAK.
            self._command.append('\x22\x00')
        # Increase expected response bytes.
        self._expected += len(data)

    def _address_byte(self, read=True):
        """Return the address byte with the specified R/W bit
set.  If read is
        True the R/W bit will be 1, otherwise the R/W bit
will be 0.
        """
        if read:
            return (self._address << 1) | 0x01
        else:
            return self._address << 1

    def _verify_acks(self, response):
        """Check all the specified bytes have the ACK bit
set.  Throws a
        RuntimeError exception if not all the ACKs are set.
        """
        for byte in response:
            if byte & 0x01 != 0x00:
                raise RuntimeError('Failed to find expected
I2C ACK!')

    def ping(self):
        """Attempt to detect if a device at this address is
present on the I2C
        bus.  Will send out the device's address for writing
and verify an ACK
        is received.  Returns true if the ACK is received,
and false if not.
        """
        self._idle()
        self._transaction_start()
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(False)])
        self._i2c_stop()
        response = self._transaction_end()
        if len(response) != 1:
```

```
            raise RuntimeError('Expected 1 response byte but
received {0} byte(s).'.format(len(response)))
        return ((response[0] & 0x01) == 0x00)

    def writeRaw8(self, value):
        """Write an 8-bit value on the bus (without
register)."""
        value = value & 0xFF
        self._idle()
        self._transaction_start()
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(False),
value])
        self._i2c_stop()
        response = self._transaction_end()
        self._verify_acks(response)

    def write8(self, register, value):
        """Write an 8-bit value to the specified register."""
        value = value & 0xFF
        self._idle()
        self._transaction_start()
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(False),
register, value])
        self._i2c_stop()
        response = self._transaction_end()
        self._verify_acks(response)

    def write16(self, register, value, little_endian=True):
        """Write a 16-bit value to the specified register."""
        value = value & 0xFFFF
        value_low  = value & 0xFF
        value_high = (value >> 8) & 0xFF
        if not little_endian:
            value_low, value_high = value_high, value_low
        self._idle()
        self._transaction_start()
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(False),
register, value_low,
                               value_high])
        self._i2c_stop()
        response = self._transaction_end()
        self._verify_acks(response)

    def writeList(self, register, data):
        """Write bytes to the specified register."""
        self._idle()
        self._transaction_start()
```

```
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(False),
register] + data)
        self._i2c_stop()
        response = self._transaction_end()
        self._verify_acks(response)

    def readList(self, register, length):
        """Read a length number of bytes from the specified
register.  Results
        will be returned as a bytearray."""
        if length <= 0:
            raise ValueError("Length must be at least 1
byte.")
        self._idle()
        self._transaction_start()
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(True),
register])
        self._i2c_stop()
        self._i2c_idle()
        self._i2c_start()
        self._i2c_read_bytes(length)
        self._i2c_stop()
        response = self._transaction_end()
        self._verify_acks(response[:-length])
        return response[-length:]

    def readRaw8(self):
        """Read an 8-bit value on the bus (without
register)."""
        self._idle()
        self._transaction_start()
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(False)])
        self._i2c_stop()
        self._i2c_idle()
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(True)])
        self._i2c_read_bytes(1)
        self._i2c_stop()
        response = self._transaction_end()
        self._verify_acks(response[:-1])
        return response[-1]

    def readU8(self, register):
        """Read an unsigned byte from the specified
register."""
        self._idle()
        self._transaction_start()
```

```
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(False),
register])
        self._i2c_stop()
        self._i2c_idle()
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(True)])
        self._i2c_read_bytes(1)
        self._i2c_stop()
        response = self._transaction_end()
        self._verify_acks(response[:-1])
        return response[-1]

    def readS8(self, register):
        """Read a signed byte from the specified register."""
        result = self.readU8(register)
        if result > 127:
            result -= 256
        return result

    def readU16(self, register, little_endian=True):
        """Read an unsigned 16-bit value from the specified
register, with the
        specified endianness (default little endian, or least
significant byte
        first)."""
        self._idle()
        self._transaction_start()
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(False),
register])
        self._i2c_stop()
        self._i2c_idle()
        self._i2c_start()
        self._i2c_write_bytes([self._address_byte(True)])
        self._i2c_read_bytes(2)
        self._i2c_stop()
        response = self._transaction_end()
        self._verify_acks(response[:-2])
        if little_endian:
            return (response[-1] << 8) | response[-2]
        else:
            return (response[-2] << 8) | response[-1]

    def readS16(self, register, little_endian=True):
        """Read a signed 16-bit value from the specified
register, with the
        specified endianness (default little endian, or least
significant byte
        first)."""
```

```
        result = self.readU16(register, little_endian)
        if result > 32767:
            result -= 65536
        return result

    def readU16LE(self, register):
        """Read an unsigned 16-bit value from the specified
register, in little
        endian byte order."""
        return self.readU16(register, little_endian=True)

    def readU16BE(self, register):
        """Read an unsigned 16-bit value from the specified
register, in big
        endian byte order."""
        return self.readU16(register, little_endian=False)

    def readS16LE(self, register):
        """Read a signed 16-bit value from the specified
register, in little
        endian byte order."""
        return self.readS16(register, little_endian=True)

    def readS16BE(self, register):
        """Read a signed 16-bit value from the specified
register, in big
        endian byte order."""
        return self.readS16(register, little_endian=False)
```

2. LoRa module library (sx1272.py)

```
import time
import Adafruit_GPIO as GPIO
import ft
import ctypes
import PeriodicThread
import collections

#REGISTERS
REG_FIFO                          =    0x00
REG_OP_MODE                       =    0x01
#REG_BITRATE_MSB                  =    0x02
#REG_BITRATE_LSB                  =    0x03
#REG_FDEV_MSB                     =    0x04
#REG_FDEV_LSB                     =    0x05
REG_FRF_MSB                            =    0x06
REG_FRF_MID                            =    0x07
REG_FRF_LSB                            =    0x08
REG_PA_CONFIG                     =    0x09
REG_PA_RAMP                            =    0x0A
REG_OCP                                =    0x0B
REG_LNA                                =    0x0C
#REG_RX_CONFIG                    =    0x0D
REG_FIFO_ADDR_PTR                 =    0x0D
#REG_RSSI_CONFIG                  =    0x0E
REG_FIFO_TX_BASE_ADDR             =    0x0E
#REG_RSSI_COLLISION               =    0x0F
REG_FIFO_RX_BASE_ADDR            =    0x0F
#REG_RSSI_THRESH                  =    0x10
REG_FIFO_RX_CURRENT_ADDR     =  0x10
#REG_RSSI_VALUE_FSK               =    0x11
REG_IRQ_FLAGS_MASK                =    0x11
#REG_RX_BW                        =    0x12
REG_IRQ_FLAGS                =    0x12
#REG_AFC_BW                       =    0x13
REG_RX_NB_BYTES                   =    0x13
#REG_OOK_PEAK                =    0x14
REG_RX_HEADER_CNT_VALUE_MSB  =    0x14
#REG_OOK_FIX                  =   0x15
REG_RX_HEADER_CNT_VALUE_LSB  =    0x15
#REG_OOK_AVG                      =   0x16
REG_RX_PACKET_CNT_VALUE_MSB  =    0x16
REG_RX_PACKET_CNT_VALUE_LSB  =    0x17
REG_MODEM_STAT                    =   0x18
REG_PKT_SNR_VALUE                 =   0x19
#REG_AFC_FEI                      =   0x1A
REG_PKT_RSSI_VALUE                =   0x1A
#REG_AFC_MSB                      =   0x1B
```

```
REG_RSSI_VALUE_LORA             =   0x1B
#REG_AFC_LSB                    =   0x1C
REG_HOP_CHANNEL                  =   0x1C
#REG_FEI_MSB                    =   0x1D
REG_MODEM_CONFIG1               =   0x1D
#REG_FEI_LSB                    =   0x1E
REG_MODEM_CONFIG2               =   0x1E
#REG_PREAMBLE_DETECT            =   0x1F
REG_SYMB_TIMEOUT_LSB            =   0x1F
#REG_RX_TIMEOUT1                =   0x20
REG_PREAMBLE_MSB_LORA           =   0x20
REG_PREAMBLE_LSB_LORA           =   0x21
#REG_RX_TIMEOUT2                =   0x21
#REG_RX_TIMEOUT3                =   0x22
REG_PAYLOAD_LENGTH_LORA         =   0x22
#REG_RX_DELAY                   =   0x23
REG_MAX_PAYLOAD_LENGTH          =   0x23
#REG_OSC                        =   0x24
REG_HOP_PERIOD                  =   0x24
#REG_PREAMBLE_MSB_FSK           =   0x25
REG_FIFO_RX_BYTE_ADDR           =   0x25
#REG_PREAMBLE_LSB_FSK           =   0x26
#REG_SYNC_CONFIG                =   0x27
#REG_SYNC_VALUE1                =   0x28
REG_FEI_MSB                   =  0X28
REG_FEI_MID                      =   0X29
#REG_SYNC_VALUE2                =   0x29
#REG_SYNC_VALUE3                =   0x2A
REG_FEI_LSB                      =   0X2A
#REG_SYNC_VALUE4                =   0x2B
REG_SYNC_VALUE5                  =   0x2C
#REG_SYNC_VALUE6                =   0x2D
#REG_SYNC_VALUE7                =   0x2E
#REG_SYNC_VALUE8                =   0x2F
#REG_PACKET_CONFIG1             =   0x30
#REG_PACKET_CONFIG2             =   0x31
REG_DETECT_OPTIMIZE           =  0x31
#REG_PAYLOAD_LENGTH_FSK         =   0x32
#REG_NODE_ADRS                  =   0x33
REG_INVERT_IQ                   =   0X33
#REG_BROADCAST_ADRS             =   0x34
#REG_FIFO_THRESH                =   0x35
#REG_SEQ_CONFIG1                =   0x36
#REG_SEQ_CONFIG2                =   0x37
REG_DETECTION_THRESHOLD       =  0x37
#REG_TIMER_RESOL                =   0x38
#REG_TIMER1_COEF                =   0x39
REG_SYNC_WORD                   =   0X39
#REG_TIMER2_COEF                =   0x3A
#REG_IMAGE_CAL                  =   0x3B
```

```
#REG_TEMP                       =   0x3C
#REG_LOW_BAT                    =   0x3D
#REG_IRQ_FLAGS1                     =   0x3E
#REG_IRQ_FLAGS2                     =   0x3F
REG_DIO_MAPPING1                =   0x40
REG_DIO_MAPPING2                =   0x41
REG_VERSION                         =   0x42
REG_AGC_REF                         =   0x43
REG_AGC_THRESH1                     =   0x44
REG_AGC_THRESH2                     =   0x45
REG_AGC_THRESH3                     =   0x46
REG_PLL_HOP                         =   0x4B
REG_TCXO                        =   0x58
REG_PA_DAC                          =   0x5A
REG_PLL                             =   0x5C
REG_PLL_LOW_PN                  =   0x5E
REG_FORMER_TEMP                     =   0x6C
REG_BIT_RATE_FRAC               =   0x70

#FREQUENCY CHANNELS:
CH_10_868 = 0xD84CCC # channel 10, central freq = 865.20MHz
CH_11_868 = 0xD86000 # channel 11, central freq = 865.50MHz
CH_12_868 = 0xD87333 # channel 12, central freq = 865.80MHz
CH_13_868 = 0xD88666 # channel 13, central freq = 866.10MHz
CH_14_868 = 0xD89999 # channel 14, central freq = 866.40MHz
CH_15_868 = 0xD8ACCC # channel 15, central freq = 866.70MHz
CH_16_868 = 0xD8C000 # channel 16, central freq = 867.00MHz
CH_17_868 = 0xD90000 # channel 16, central freq = 868.00MHz
CH_00_900 = 0xE1C51E # channel 00, central freq = 903.08MHz
CH_01_900 = 0xE24F5C # channel 01, central freq = 905.24MHz
CH_02_900 = 0xE2D999 # channel 02, central freq = 907.40MHz
CH_03_900 = 0xE363D7 # channel 03, central freq = 909.56MHz
CH_04_900 = 0xE3EE14 # channel 04, central freq = 911.72MHz
CH_05_900 = 0xE47851 # channel 05, central freq = 913.88MHz
CH_06_900 = 0xE5028F # channel 06, central freq = 916.04MHz
CH_07_900 = 0xE58CCC # channel 07, central freq = 918.20MHz
CH_08_900 = 0xE6170A # channel 08, central freq = 920.36MHz
CH_09_900 = 0xE6A147 # channel 09, central freq = 922.52MHz
CH_10_900 = 0xE72B85 # channel 10, central freq = 924.68MHz
CH_11_900 = 0xE7B5C2 # channel 11, central freq = 926.84MHz
CH_12_900 = 0xE4C000 # default channel 915MHz, the module is
configured with it


#LORA BANDWIDTH:
BW_125 = 0x00
BW_250 = 0x01
BW_500 = 0x02
```

```
#LORA CODING RATE:
CR_5 = 0x01
CR_6 = 0x02
CR_7 = 0x03
CR_8 = 0x04

#LORA SPREADING FACTOR:
SF_6 = 0x06
SF_7 = 0x07
SF_8 = 0x08
SF_9 = 0x09
SF_10 = 0x0A
SF_11 = 0x0B
SF_12 = 0x0C

#LORA MODES:
LORA_SLEEP_MODE = 0x80
LORA_STANDBY_MODE = 0x81
LORA_TX_MODE = 0x83
LORA_RX_MODE = 0x85
LORA_STANDBY_FSK_REGS_MODE = 0xC1

#FSK MODES:
FSK_SLEEP_MODE = 0x00
#FSK_STANDBY_MODE = 0x01
#FSK_TX_MODE = 0x03
#FSK_RX_MODE = 0x05

#OTHER CONSTANTS:
HEADER_ON = 0
HEADER_OFF = 1
CRC_ON = 1
CRC_OFF = 0
LORA = 1
FSK = 0
BROADCAST_0 = 0x00
MAX_LENGTH = 255
MAX_PAYLOAD = 251
ACK_LENGTH = 5
OFFSET_PAYLOADLENGTH = 5
OFFSET_RSSI = 137
NOISE_FIGURE = 6.0
NOISE_ABSOLUTE_ZERO = 174.0
MAX_TIMEOUT = 8000
MAX_WAIT = 12000
MAX_RETRIES = 5
CORRECT_PACKET = 0
INCORRECT_PACKET = 1

# 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)
     ]

class RegModemConfig1Bits(ctypes.LittleEndianStructure):
     _fields_ = [
          ("LowDataRateOptimize",  ctypes.c_uint8,1),
          ("RxPayloadCrcOn",  ctypes.c_uint8,1),
          ("ImplicitHeaderModeOn", ctypes.c_uint8,1),
          ("CodingRate", ctypes.c_uint8,3),
          ("Bw",    ctypes.c_uint8,2),
     ]

class RegModemConfig1(ctypes.Union):
     _fields_ = [
          ("bits", RegModemConfig1Bits),
          ("octet", ctypes.c_uint8)
     ]


class RegModemConfig2Bits(ctypes.LittleEndianStructure):
     _fields_ = [
          ("SymbTimeout(9:8)",     ctypes.c_uint8,2),
          ("AgcAutoOn",  ctypes.c_uint8,1),
          ("TxContinuousMode",     ctypes.c_uint8,1),
          ("SpreadingFactor", ctypes.c_uint8,4),
     ]

class RegModemConfig2(ctypes.Union):
     _fields_ = [
          ("bits", RegModemConfig2Bits),
          ("octet", ctypes.c_uint8)
     ]

class adressBits(ctypes.LittleEndianStructure):
     _fields_ = [
          ("others",     ctypes.c_uint8, 7),
          ("wnr",   ctypes.c_uint8, 1),      # wnr=0 for read
and wnr=1 for write
```

```
        ]

class adress1(ctypes.Union):
        _fields_ = [
                ("bits",  adressBits),
                ("octet", ctypes.c_uint8)
        ]

class preambleBits(ctypes.LittleEndianStructure):
        _fields_ = [
                ("LSB",   ctypes.c_uint16, 8),
                ("MSB",   ctypes.c_uint16, 8),
        ]

class preamble(ctypes.Union):
        _fields_ = [
                ("bits",  preambleBits),
                ("octet", ctypes.c_uint16)
       ]

class frfBits(ctypes.LittleEndianStructure):
        _fields_ = [
                ("LSB",   ctypes.c_uint8, 8),
                ("MIB",   ctypes.c_uint8, 8),
                ("MSB",   ctypes.c_uint8, 8),

        ]

class frf(ctypes.Union):
        _fields_ = [
                ("bits",  frfBits),
                ("octet", ctypes.c_uint)
        ]

class powerBits(ctypes.LittleEndianStructure):
        _fields_ = [
                ("OutputPower",     ctypes.c_uint8, 4),
                ("Unused",     ctypes.c_uint8, 3),
                ("PaSelect",   ctypes.c_uint8, 1),

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

class flagsBits(ctypes.LittleEndianStructure):
        _fields_ = [
                ("CadDetected",     ctypes.c_uint8, 1),
```

```
            ("FhssChangeChannel",    ctypes.c_uint8, 1),
            ("CadDone",    ctypes.c_uint8, 1),
            ("TxDone",     ctypes.c_uint8, 1),
            ("ValidHeader",     ctypes.c_uint8, 1),
            ("PayloadCrcError", ctypes.c_uint8, 1),
            ("RxDone",     ctypes.c_uint8, 1),
            ("RxTimeout",  ctypes.c_uint8, 1),

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


class HopChannelBits(ctypes.LittleEndianStructure):
      _fields_ = [
            ("FhssPresentChannel",   ctypes.c_uint8, 6),
            ("CrcOnPayload",    ctypes.c_uint8, 1),
            ("PllTimeout", ctypes.c_uint8, 1),

      ]

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

class ValidpacketsRXBits(ctypes.LittleEndianStructure):
      _fields_ = [
            ("ValidPacketCntLsb",    ctypes.c_uint8, 8),
            ("ValidPacketCntMsb",    ctypes.c_uint8, 8),

      ]

class ValidpacketsRX(ctypes.Union):
      _fields_ = [
            ("bits",  ValidpacketsRXBits),
            ("octet", ctypes.c_uint16)
      ]

class ValidheadersRXBits(ctypes.LittleEndianStructure):
      _fields_ = [
            ("ValidHeaderCntLsb",    ctypes.c_uint8, 8),
            ("ValidHeaderCntMsb",    ctypes.c_uint8, 8),

      ]
```

```
class ValidheadersRX(ctypes.Union):
     _fields_ = [
          ("bits",  ValidheadersRXBits),
          ("octet", ctypes.c_uint16)
     ]



class SX1272:
     def __init__(self):
          #Initialize class variables
          self.bandwidth = BW_125
          self.codingRate = CR_5
          self.spreadingFactor = SF_7
          self.channel = CH_12_900
          self.header = HEADER_ON
          self.CRC = CRC_OFF
          self.modem = FSK
          self.power = 15
          self.packetNumber = 0
          #self.reception = CORRECT_PACKET
          #self.retries = 0
          #self.maxRetries = 3
          #self.packet_sent_retry = 0
          self.queuerx=collections.deque()

     self.threadrx=PeriodicThread.PeriodicThread(self.getoneP
acket,0.01)
          self.rx=False


     def ON(self):
          # Temporarily disable FTDI serial drivers.
          ft.use_FT232H()
          # Find the first FT232H device.
          self.ft232h = ft.FT232H()
          # Make pin D4 a digital output, conected to pin 6
on Lora module.
          self.ft232h.setup(4, GPIO.OUT)


          # reset pulse for LoRa module initialization
          self.ft232h.output(4, GPIO.HIGH)
          time.sleep(0.1)
          self.ft232h.output(4, GPIO.LOW)
          time.sleep(0.1)
          self.bandwidth = BW_125
          self.codingRate = CR_5
          self.spreadingFactor = SF_7
          self.channel = CH_12_900
```

```
          self.header = HEADER_ON
          self.CRC = CRC_OFF
          self.modem = FSK
          self.power = 15
          self.packetNumber = 0
          #self.reception = CORRECT_PACKET
          #self.retries = 0
          #self.maxRetries = 3
          #self.packet_sent_retry = 0
          # 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.
          self.spi = ft.SPI(self.ft232h, cs=3,
max_speed_hz=3000000, mode=0, bitorder=ft.MSBFIRST)
          self.ko=0 #Variable para controlar si configuracion
es correcta


     def readRegister(self,adress):
          a=adress1()
          a.octet=adress
          a.bits.wnr=0
          self.spi.writeR([a.octet]) # Nose si usar funcion
spi.transfer()
          response=self.spi.read(1)
          return response[0]

     def writeRegister(self,adress,data):
          a=adress1()
          a.octet=adress
          a.bits.wnr=1
          self.spi.write([a.octet,data])

     def clearFlags(self):
          if(self.modem==LORA):
               st0 = self.readRegister(REG_OP_MODE)
     #Save the previous status
               self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE)  # Stdby mode to write in registers
               self.writeRegister(REG_IRQ_FLAGS, 0xFF) # LoRa
mode flags register
               self.writeRegister(REG_OP_MODE, st0)         #
Getting back to previous status
          else:
               print('Not in LORA mode')

     def setLORA(self):
          self.writeRegister(REG_OP_MODE, FSK_SLEEP_MODE)
# Sleep mode (mandatory to set LoRa mode)
```

```
          self.writeRegister(REG_OP_MODE, LORA_SLEEP_MODE)
 # LoRa sleep mode
          self.writeRegister(REG_OP_MODE, LORA_STANDBY_MODE)
     # LoRa standby mode


     self.writeRegister(REG_MAX_PAYLOAD_LENGTH,MAX_LENGTH)

          time.sleep(0.1)

          st0 = self.readRegister(REG_OP_MODE)     # Reading
config mode

          if(st0==LORA_STANDBY_MODE):
               self.modem=LORA
               #print('## LoRa set with success')
          else:
               print('** There has been an error while
setting LoRa **')
               self.ko=1

     def getMode(self):
          if(self.modem==LORA):
               value = self.readRegister(REG_MODEM_CONFIG1)
               a=RegModemConfig1()
               a.octet=value
               self.bandwidth=a.bits.Bw
               print 'bandwidth =' ,
               print hex(self.bandwidth)
               self.codingRate=a.bits.CodingRate
               print 'codingRate =' ,
               print hex(self.codingRate)
               value = self.readRegister(REG_MODEM_CONFIG2)
               a=RegModemConfig2()
               a.octet=value
               self.spreadingFactor=a.bits.SpreadingFactor
               print 'SpreadingFactor=' ,
               print hex(self.spreadingFactor)
          else:
               print('Not in LORA mode')

     def setMode(self,mode):
          if(self.modem==LORA):
               st0 = self.readRegister(REG_OP_MODE)
     #Save the previous status

               self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE)  #LoRa standby mode
               c1=RegModemConfig1()
               c2=RegModemConfig2()
```

```
                value1=self.readRegister(REG_MODEM_CONFIG1)
                value2=self.readRegister(REG_MODEM_CONFIG2)
                c1.octet=value1
                c2.octet=value2

                if(mode==1):   #mode 1 (better reach, medium
time on air)
                     c1.bits.CodingRate=CR_5
                     c2.bits.SpreadingFactor=SF_12
                     c1.bits.Bw=BW_125


                elif(mode==2): #mode 2 (medium reach, less
time on air)
                     c1.bits.CodingRate=CR_5
                     c2.bits.SpreadingFactor=SF_12
                     c1.bits.Bw=BW_250

                elif(mode==3): #mode 3 (worst reach, less time
on air)
                     c1.bits.CodingRate=CR_5
                     c2.bits.SpreadingFactor=SF_10
                     c1.bits.Bw=BW_125

                elif(mode==4): #mode 4 (better reach, low time
on air)
                     c1.bits.CodingRate=CR_5
                     c2.bits.SpreadingFactor=SF_12
                     c1.bits.Bw=BW_500

                elif(mode==5): #mode 5 (better reach, medium
time on air)
                     c1.bits.CodingRate=CR_5
                     c2.bits.SpreadingFactor=SF_10
                     c1.bits.Bw=BW_250

                elif(mode==6): #mode 6 (better reach, worst
time-on-air)
                     c1.bits.CodingRate=CR_5
                     c2.bits.SpreadingFactor=SF_11
                     c1.bits.Bw=BW_500

                elif(mode==7): #mode 7 (medium-high reach,
medium-low time-on-air)
                     c1.bits.CodingRate=CR_5
                     c2.bits.SpreadingFactor=SF_9
                     c1.bits.Bw=BW_250

                elif(mode==8): #mode 8 (medium reach, medium
time-on-air)
```

```
                    c1.bits.CodingRate=CR_5
                    c2.bits.SpreadingFactor=SF_9
                    c1.bits.Bw=BW_500

               elif(mode==9): #mode 9 (medium-low reach,
medium-high time-on-air)
                    c1.bits.CodingRate=CR_5
                    c2.bits.SpreadingFactor=SF_8
                    c1.bits.Bw=BW_500

               elif(mode==10):     #mode 10 (worst reach,
less time_on_air)
                    c1.bits.CodingRate=CR_5
                    c2.bits.SpreadingFactor=SF_7
                    c1.bits.Bw=BW_500
               else:
                    print("** The indicated mode doesn't
exist, ")
                    print("please select from 1 to 10 **")
                    self.ko=1

               self.bandwidth = c1.bits.Bw
               self.codingRate = c1.bits.CodingRate
               self.spreadingFactor = c2.bits.SpreadingFactor
               self.writeRegister(REG_MODEM_CONFIG1,c1.octet)
               self.writeRegister(REG_MODEM_CONFIG2,c2.octet)
               self.writeRegister(REG_OP_MODE, st0)
          else:
               print('Not in LORA mode')

     def setCodingRate(self,codingrate):
          if(self.modem==LORA):
               st0 = self.readRegister(REG_OP_MODE)
     #Save the previous status
               self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE)  #LoRa standby mode
               c=RegModemConfig1()
               value=self.readRegister(REG_MODEM_CONFIG1)
               c.octet=value
               if (codingrate==5):
                    c.bits.CodingRate=CR_5
               elif (codingrate==6):
                    c.bits.CodingRate=CR_6
               elif (codingrate==7):
                    c.bits.CodingRate=CR_7
               elif (codingrate==8):
                    c.bits.CodingRate=CR_8
               else:
                    print('Invalid CodingRate')
                    self.ko=1
```

```
                self.codingRate = c.bits.CodingRate
                self.writeRegister(REG_MODEM_CONFIG1,c.octet)
                self.writeRegister(REG_OP_MODE, st0)
           else:
                print('Not in LORA mode')


     def setSpreadingFactor(self,spreading):
          if(self.modem==LORA):
                st0 = self.readRegister(REG_OP_MODE)
     #Save the previous status
                self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE)  #LoRa standby mode
                c=RegModemConfig2()
                value=self.readRegister(REG_MODEM_CONFIG2)
                c.octet=value
                if (spreading==6):
                     c.bits.SpreadingFactor=SF_6
                elif (spreading==7):
                     c.bits.SpreadingFactor=SF_7
                elif (spreading==8):
                     c.bits.SpreadingFactor=SF_8
                elif (spreading==9):
                     c.bits.SpreadingFactor=SF_9
                elif (spreading==10):
                     c.bits.SpreadingFactor=SF_10
                elif (spreading==11):
                     c.bits.SpreadingFactor=SF_11
                elif (spreading==12):
                     c.bits.SpreadingFactor=SF_12
                else:
                     print('Invalid SpreadingFactor')
                     self.ko=1
                self.spreadingFactor = c.bits.SpreadingFactor
                self.writeRegister(REG_MODEM_CONFIG2,c.octet)
                self.writeRegister(REG_OP_MODE, st0)
           else:
                print('Not in LORA mode')

     def setBandwidth(self,bandwidth):
          if(self.modem==LORA):
                st0 = self.readRegister(REG_OP_MODE)
     #Save the previous status
                self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE)  #LoRa standby mode
                c=RegModemConfig1()
                value=self.readRegister(REG_MODEM_CONFIG1)
                c.octet=value
                if (bandwidth==125):
                     c.bits.Bw=BW_125
```

```
                elif (bandwidth==250):
                     c.bits.Bw=BW_250
                elif (bandwidth==500):
                     c.bits.Bw=BW_500
                else:
                     print('Invalid Bandwidth')
                     self.ko=1
                self.bandwidth = c.bits.Bw
                self.writeRegister(REG_MODEM_CONFIG1,c.octet)
                self.writeRegister(REG_OP_MODE, st0)
           else:
                print('Not in LORA mode')

     def setLowDataRateOptimize(self):
           if(self.modem==LORA):
                st0 = self.readRegister(REG_OP_MODE)
     #Save the previous status
                self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE)  #LoRa standby mode
                c=RegModemConfig1()
                value=self.readRegister(REG_MODEM_CONFIG1)
                c.octet=value
                if (self.spreadingFactor==SF_11 and
self.bandwidth==BW_125):
                     c.bits.LowDataRateOptimize=1
                elif (self.spreadingFactor==SF_12 and
self.bandwidth==BW_125):
                     c.bits.LowDataRateOptimize=1
                else:
                     c.bits.LowDataRateOptimize=0
                self.writeRegister(REG_MODEM_CONFIG1,c.octet)
                self.writeRegister(REG_OP_MODE, st0)
           else:
                print('Not in LORA mode')



     def getHeader(self):
           c1=RegModemConfig1()
           value1=self.readRegister(REG_MODEM_CONFIG1)
           c1.octet=value1
           if(self.modem==FSK):
                print("## Notice that FSK mode packets hasn't
header ##")
           else:
                if (c1.bits.ImplicitHeaderModeOn==0):
                     self.header=HEADER_ON
                     print("## Header is in explicit header
mode ##")
                else:
```

```
                    self.header=HEADER_OFF
                    print("## Header is in implicit header
mode ##")


     def setHeaderON(self): #Explicit mode
          if(self.modem==FSK):
               print("## FSK mode packets hasn't header ##")
               self.ko=1
          else:
               if(self.spreadingFactor==SF_6):
                    print("## Mandatory implicit header mode
with spreading factor = 6 ##")
                    self.ko=1
               else:
                    c1=RegModemConfig1()

     value1=self.readRegister(REG_MODEM_CONFIG1)
                    c1.octet=value1
                    c1.bits.ImplicitHeaderModeOn=0

     self.writeRegister(REG_MODEM_CONFIG1,c1.octet)
                    self.header=HEADER_ON
                    #print("## Header has been activated ##")

     def setHeaderOFF(self): #Implicit mode
          if(self.modem==FSK):
               print("## FSK mode packets hasn't header ##")
               self.ko=1
          else:
               c1=RegModemConfig1()
               value1=self.readRegister(REG_MODEM_CONFIG1)
               c1.octet=value1
               c1.bits.ImplicitHeaderModeOn=1
               self.writeRegister(REG_MODEM_CONFIG1,c1.octet)
               self.header=HEADER_OFF
               #print("## Header has been desactivated ##")

     def getCRC(self):
          if(self.modem==LORA):
               c1=RegModemConfig1()
               value1=self.readRegister(REG_MODEM_CONFIG1)
               c1.octet=value1
               if (c1.bits.RxPayloadCrcOn==CRC_ON):
                    self.CRC=CRC_ON
                    print("## CRC is activated ##")
               else:
                    self.CRC=CRC_OFF
                    print("## CRC is desactivated ##")
          else:
```

```
            print("Not in LORA mode")

    def setCRC_ON(self):
        if(self.modem==LORA):
            c1=RegModemConfig1()
            value1=self.readRegister(REG_MODEM_CONFIG1)
            c1.octet=value1
            c1.bits.RxPayloadCrcOn=1
            self.CRC=CRC_ON
            self.writeRegister(REG_MODEM_CONFIG1,c1.octet)
            #print("## CRC has been activated ##")

        else:
            print("Not in LORA mode")

    def setCRC_OFF(self):
        if(self.modem==LORA):
            c1=RegModemConfig1()
            value1=self.readRegister(REG_MODEM_CONFIG1)
            c1.octet=value1
            c1.bits.RxPayloadCrcOn=0
            self.CRC=CRC_OFF
            self.writeRegister(REG_MODEM_CONFIG1,c1.octet)
            #print("## CRC has been desactivated ##")

        else:
            print("Not in LORA mode")


    def getPreambleLength(self):
        if(self.modem==LORA):
            pre=preamble()

    pre.bits.LSB=self.readRegister(REG_PREAMBLE_LSB_LORA)

    pre.bits.MSB=self.readRegister(REG_PREAMBLE_MSB_LORA)
            self.preamblelength=pre.octet
            print "## Preamble length configured is " ,
            print(hex(self.preamblelength))
        else:
            print("Not in LORA mode")

    def setPreambleLength(self,l):
        if(self.modem==LORA):
            if (l>=6 and l<=65535):
                st0 = self.readRegister(REG_OP_MODE)
                pre=preamble()
                pre.octet=l
                self.preamblelenght=l
```

```
                    self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE) #Set Standby mode to write in registers
                    self.writeRegister(REG_PREAMBLE_MSB_LORA,
pre.bits.MSB)
                    self.writeRegister(REG_PREAMBLE_LSB_LORA,
pre.bits.LSB)
                    self.writeRegister(REG_OP_MODE, st0)
     #Getting back to previous status
                else:
                    print('l out of range 6<=l<=65535')
                    self.ko=1
          else:
               print("Not in LORA mode")

     def getChannel(self):
          if(self.modem==LORA):
               fr=frf()
               fr.bits.MSB=self.readRegister(REG_FRF_MSB)
               fr.bits.MIB=self.readRegister(REG_FRF_MID)
               fr.bits.LSB=self.readRegister(REG_FRF_LSB)
               self.channel=fr.octet
               freq=float(32*(fr.octet))/(2**19)
               print 'channel:',
               print(freq)
          else:
               print("Not in LORA mode")

     def setChannel(self,channel): #channel in Mhz
          if(self.modem==LORA):
               if(channel>0 and channel<1024):
                    st0 = self.readRegister(REG_OP_MODE)
     #Save the previous status
                    Frf=int((channel*(2**19))/32)
                    a=frf()
                    a.octet=Frf
                    self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE)  #LoRa Stdby mode in order to write in
registers
                    self.writeRegister(REG_FRF_MSB,
a.bits.MSB )
                    self.writeRegister(REG_FRF_MID,
a.bits.MIB)
                    self.writeRegister(REG_FRF_LSB,
a.bits.LSB)
                    self.writeRegister(REG_OP_MODE, st0)
                    self.channel=Frf
               else:
                    print('Channel out of range')
                    self.ko=1
          else:
```

```
                print("Not in LORA mode")

     def getPower(self):
          if(self.modem==LORA):
                pow=power()
                pow.octet=self.readRegister(REG_PA_CONFIG)
                print "## Output power is ",
                if pow.bits.PaSelect==1:
                     print 2+pow.bits.OutputPower,
                else:
                     print -1+pow.bits.OutputPower,
                print 'dBm'
          else:
                print("Not in LORA mode")

     def setPower(self,p):     # p= M(max),L(low) or H(high)
          if(self.modem==LORA):
                st0 = self.readRegister(REG_OP_MODE)
#Save the previous status
                self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE)  #LoRa Stdby mode to write in registers
                pow=power()
                pow.octet=self.readRegister(REG_PA_CONFIG)
                if (p=='M'):
                     self.power=0x0F
                elif (p=='L'):
                     self.power=0x00
                elif (p=='H'):
                     self.power=0x07
                else:
                     print'Not M(max),L(low) or H(high)
selected'
                pow.bits.OutputPower=self.power
                self.writeRegister(REG_PA_CONFIG,pow.octet)
                self.writeRegister(REG_OP_MODE, st0)          #
Getting back to previous status
          else:
                print("Not in LORA mode")

     def setPowerNum(self,pow):     #pow = Pout (-1 to 17
dBm), RFIO enabled
          if(self.modem==LORA):
                if (pow>=-1 and pow<=14):
                     st0 = self.readRegister(REG_OP_MODE)
  #Save the previous status
                     self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE)  #LoRa Stdby mode to write in registers
                     self.power=pow+1
                     p=power()
                     p.octet=self.readRegister(REG_PA_CONFIG)
```

```
                    p.bits.OutputPower=self.power
                    self.writeRegister(REG_PA_CONFIG,p.octet)
                    self.writeRegister(REG_OP_MODE, st0)
     # Getting back to previous status

                elif (pow>14 and pow<=17): #PA_BOOST enabled
                    st0 = self.readRegister(REG_OP_MODE)
  #Save the previous status
                    self.writeRegister(REG_OP_MODE,
LORA_STANDBY_MODE)  #LoRa Stdby mode to write in registers
                    self.power=pow-2
                    p=power()
                    p.octet=self.readRegister(REG_PA_CONFIG)
                    p.bits.OutputPower=self.power
                    p.bits.PaSelect=1
                    self.writeRegister(REG_PA_CONFIG,p.octet)
                    self.writeRegister(REG_OP_MODE, st0)
     # Getting back to previous status

                else:
                    print('Power out of range')
                    self.ko=1

           else:
                print("Not in LORA mode")



     def receive(self):
          if(self.modem==LORA):
               self.ft232h.setup(5, GPIO.OUT) #Pin d5 Rx
(Purple)
               self.ft232h.setup(6, GPIO.OUT) #Pin d6 Tx
(White)


               self.ft232h.output(5, GPIO.HIGH)
               self.ft232h.output(6, GPIO.LOW)



     self.writeRegister(REG_OP_MODE,LORA_STANDBY_MODE)

     self.writeRegister(REG_FIFO_ADDR_PTR,REG_FIFO_RX_BASE_AD
DR)
               self.writeRegister(REG_OP_MODE,LORA_RX_MODE)
               self.payload=[]
               self.d=[]
               self.crcreceived=[]
               self.snrpacket=[]
```

```
                self.rssipacket=[]
                self.rssi=[]
                self.graph=collections.deque()
                self.packetNumber=0
           else:
                print("Not in LORA mode")

     def getpacketSnr(self):
           if(self.modem==LORA):
                a=self.readRegister(REG_PKT_SNR_VALUE)
                #print a
                return (self.a2_to_dec(a)/4)
           else:
                print("Not in LORA mode")

     def getpacketRssi(self):
           if(self.modem==LORA):
                return self.readRegister(REG_PKT_RSSI_VALUE)
           else:
                print("Not in LORA mode")

     def getRssi(self):
           if(self.modem==LORA):
                return self.readRegister(REG_RSSI_VALUE_LORA)
           else:
                print("Not in LORA mode")


     def validpacketsCnt(self):
           if(self.modem==LORA):
                a=ValidpacketsRX()

     a.bits.ValidPacketCntLsb=self.readRegister(REG_RX_PACKET
_CNT_VALUE_LSB)
                print self.readRegister(0x17)

     a.bits.ValidPacketCntMsb=self.readRegister(REG_RX_PACKET
_CNT_VALUE_MSB)
                return a.octet
           else:
                print("Not in LORA mode")

     def validheadersCnt(self):
           if(self.modem==LORA):
                a=ValidheadersRX()

     a.bits.ValidHeaderCntLsb=self.readRegister(REG_RX_HEADER
_CNT_VALUE_LSB)
```

```
     a.bits.ValidHeaderCntMsb=self.readRegister(REG_RX_HEADER
_CNT_VALUE_MSB)
                return a.octet
          else:
                print("Not in LORA mode")



     def getonePacket(self):
          if (self.readRegister(REG_OP_MODE)==LORA_RX_MODE):
                self.payload=[]
                fl=flags()
                fl.octet=self.readRegister(REG_IRQ_FLAGS)
                #print(hex(fl.octet))
                if (fl.bits.RxDone==1):
                     #print(self.validheadersCnt())
                     #print 'RxDone'
                     if (fl.bits.ValidHeader==1):

     self.packetNumber=self.packetNumber+1
                          #print (self.validheadersCnt())
                          #print 'ValidHeader received from
packet:',
                          #print(self.packetNumber)
                          hop=HopChannel()

     hop.octet=self.readRegister(REG_HOP_CHANNEL)
                          if (hop.bits.CrcOnPayload==1):
                               print('Header indicates CRC
on')
                               if
(fl.bits.PayloadCrcError==1):
                                    print'PayloadCrcError from
packet:',
                                    print(self.packetNumber)

     self.crcreceived=['CRC_KO']
                               else:

     self.crcreceived=['CRC_OK']
                          else:
                               self.crcreceived=['CRC_OFF']

     self.writeRegister(REG_OP_MODE,LORA_STANDBY_MODE)

     currentaddr=self.readRegister(REG_FIFO_RX_CURRENT_ADDR)
                          #print hex(currentaddr)

     self.writeRegister(REG_FIFO_ADDR_PTR,currentaddr)
```

```
                         for i in
range(self.readRegister(REG_RX_NB_BYTES)):

     self.payload.append(hex(self.readRegister(REG_FIFO)))

     self.writeRegister(REG_FIFO_ADDR_PTR,self.readRegister(R
EG_FIFO_RX_BASE_ADDR))
                         self.snrpacket=[self.getpacketSnr()]
                         if (self.getpacketSnr>=0):
                              self.rssipacket=[-
139+self.getpacketRssi()]
                         else:
                              self.rssipacket=[-
139+self.getpacketRssi()+0.25*self.getpacketSnr()]

     self.graph.append([self.packetNumber]+self.rssipacket+se
lf.snrpacket) #Utilizar para grafico
                         self.rssi=[-139+self.getRssi()]

     self.d=[self.packetNumber]+[self.readRegister(REG_RX_NB_
BYTES)]+[self.payload]+self.crcreceived+self.snrpacket+self.r
ssipacket+self.rssi
                         #print self.d
                         self.queuerx.append(self.d)
                         self.clearFlags()

     self.writeRegister(REG_OP_MODE,LORA_RX_MODE)

                         #print(self.l)


          else:
               pass
               #print('Call function receive for putting
modem in LORA_RX_MODE')




     def getPackets(self):
          if (self.readRegister(REG_OP_MODE)==LORA_RX_MODE):
               self.l=[]
               fl=flags()
               fl.octet=self.readRegister(REG_IRQ_FLAGS)
               #print(hex(fl.octet))
               if (fl.bits.RxDone==1):
                    print 'RxDone'
                    if (fl.bits.ValidHeader==1):
```

```
     self.packetNumber=self.packetNumber+1
                         print 'ValidHeader received from
packet:',
                         print(self.packetNumber)
                         hop=HopChannel()

     hop.octet=self.readRegister(REG_HOP_CHANNEL)
                         if (hop.bits.CrcOnPayload==1):
                              print('Header indicates CRC
on')
                              if
(fl.bits.PayloadCrcError==1):
                                   print'PayloadCrcError from
packet:',
                                   print(self.packetNumber)
                                   self.l=self.l+[1]
                              else:
                                   self.l=self.l+[0]
                         else:
                              self.l=self.l+['CRC_OFF']

     self.writeRegister(REG_OP_MODE,LORA_STANDBY_MODE)

     currentaddr=self.readRegister(REG_FIFO_RX_CURRENT_ADDR)

     self.writeRegister(REG_FIFO_ADDR_PTR,currentaddr)
                         for i in
range(self.readRegister(REG_RX_NB_BYTES)):

     self.l=self.l+[self.readRegister(REG_FIFO)]
                         self.queuerx.append(self.l)

     self.writeRegister(REG_FIFO_ADDR_PTR,self.readRegister(R
EG_FIFO_RX_BASE_ADDR))
                         self.clearFlags()

     self.writeRegister(REG_OP_MODE,LORA_RX_MODE)
                         self.l=[]


          else:
               print('Call function receive for putting modem
in LORA_RX_MODE')

     def startRx(self):
          self.threadrx.start()

     def stopRx(self):
          self.threadrx.cancel()
```

```
     #Transmission.......................


     def upload_buffer(self,packet):    #packet must be
string
          if(self.modem==LORA):
               st0 = self.readRegister(REG_OP_MODE)
#Save the previous status

     self.writeRegister(REG_OP_MODE,LORA_STANDBY_MODE)
               self.buffer=self.hex_to_byte(packet)
               print self.buffer
               self.writeRegister(REG_FIFO_TX_BASE_ADDR,0x00)

     self.writeRegister(REG_FIFO_ADDR_PTR,0x00)#REG_FIFO_TX_B
ASE_ADDR)

               self.ft232h.setup(5, GPIO.OUT) #Pin d5 Rx
(Purple)
               self.ft232h.setup(6, GPIO.OUT) #Pin d6 Tx
(White)


               self.ft232h.output(5, GPIO.LOW)
               self.ft232h.output(6, GPIO.HIGH)

               for i in range(len(self.buffer)):

     self.writeRegister(REG_FIFO,self.buffer[i])
                    #print self.buffer[i]

     self.writeRegister(REG_PAYLOAD_LENGTH_LORA,len(self.buff
er))
               self.writeRegister(REG_OP_MODE, st0)          #
Getting back to previous status
          else:
               print("Not in LORA mode")


     def transmit(self):
          if(self.modem==LORA):
               if (len(self.buffer)>0):
                    #self.ft232h.setup(5, GPIO.OUT) #Pin d5
Rx (Purple)
                    #self.ft232h.setup(6, GPIO.OUT) #Pin d6
Tx (White)
```

```
                    #self.ft232h.output(5, GPIO.LOW)
                    #self.ft232h.output(6, GPIO.HIGH)


     self.writeRegister(REG_OP_MODE,LORA_TX_MODE)
                    start_time=time.time()
                    fl=flags()
                    c=0
                    while (time.time()-start_time<500):
     #500 secons for transmission if more it will print an
error

     fl.octet=self.readRegister(REG_IRQ_FLAGS)
                         if (fl.bits.TxDone==1):
                              print 'Transmission done'
                              c=1
                              break
                    if (c==0):
                         print 'Error during transmission'

                    self.buffer=[]
               else:
                    print('buffer is empty')
               self.clearFlags()
               self.ft232h.output(6, GPIO.LOW)
          else:
               print("Not in LORA mode")



     def
setConfiguration(self,bw,codingrate,spredingfactor,preamblele
nght,setchannel,setpower,crc):
          self.ON()
          self.setLORA()
          self.setBandwidth(bw)
          self.setCodingRate(codingrate)
          self.setSpreadingFactor(spredingfactor)
          self.setPreambleLength(preamblelenght)
          self.setHeaderON()
          self.setLowDataRateOptimize()
          if crc=='ON':
               self.setCRC_ON()
          if crc=='OFF':
               self.setCRC_OFF()
          self.setChannel(setchannel)
          self.setPowerNum(setpower)

          # Checking configuration
```

```
          self.getMode()
          self.getHeader()
          self.getPreambleLength()
          self.getChannel()
          self.getPower()
          self.getCRC()

          print hex(self.readRegister(REG_FRF_MSB))
          print hex(self.readRegister(REG_FRF_MID))
          print hex(self.readRegister(REG_FRF_LSB))









     #Tools

     def a2_to_dec(self,value):    #convert from a2
complement to decimal with 8 bit
          i=2
          a2=format(value,'#010b')
          #print a2
          s=''
          if a2[2]=='1':
               while i<len(a2):
                    if a2[i]=='0':
                         s=s+'1'
                    else:
                         s=s+'0'
                    i=i+1
               v=(int(s,2)+1)*-1
               return v
          else:
               return value


     def unicode_to_byte(self,unicode): #returns a list of
bytes encoded in unicode
          a=map(ord,unicode)
          print a
          l=[]
          for c in range(len(a)):
               a1=format(a[c],'#06x')
               v1=a1[2:4]
```

```
                v2=a1[4:]
                l=l+[int(v1,16)]+[int(v2,16)]
          print l
          return l


    def hex_to_byte(self,data):
          l=[]
          if len(data)%2==1:
                a='0'+data
          else:
                a=data
          for c in range(0,len(a)-1,2):
                l=l+[int(a[c:c+2],16)]
          #print l
          return l
```

3. GUI library (gui_frames_3.0.py)

```
from mttkinter import mtTkinter as tk
import ttk
import sys
import sx1272 as sx


import PeriodicThread
import matplotlib
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg, NavigationToolbar2TkAgg

from matplotlib.figure import Figure
#import matplotlib.animation as animation
from matplotlib import style
import time
import tkFileDialog as fd

style.use("ggplot")



#c=sx.SX1272()

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

    def write(self, string):
        self.text_space.config(state=tk.NORMAL)
        self.text_space.insert("end", string)
        if self.up.state==1:
            self.up.file.write(string)
        self.text_space.see("end")
        self.text_space.config(state=tk.DISABLED)

class Aplicacion():
    def __init__(self):
        self.raiz = tk.Tk(mt_check_period=1,mt_debug=0)
        self.raiz.geometry('1625x976+0+0')
        self.raiz.title("Sniffer")
        self.raiz.protocol("WM_DELETE_WINDOW",
self.on_closing)

self.thr=PeriodicThread.PeriodicThread(self.update,0.01)
        self.lora=sx.SX1272()
        self.state=0
```

```
        # Declara variables de control


        self.codingraten=tk.IntVar(value=5)
        self.spredingfactorn=tk.IntVar(value=7)
        self.bwn=tk.IntVar(value=125)
        self.preamblelenghtn=tk.IntVar(value=8)
        self.setchanneln=tk.DoubleVar(value=387.3984375)
        self.crcn=tk.StringVar(value='ON')
        self.setpowern=tk.IntVar(value=14)



        #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=1
5)

        self.Frame6=tk.Frame(self.raiz,bd=2,relief=tk.RAISED)


        #Matplot

        self.f = Figure( figsize=(20, 6), dpi=80 )
```

```
        self.ax=self.f.add_subplot(211)
        #self.ax0 = self.f.add_axes( (0.05, .05, .50, .50),
axisbg=(.75,.75,.75), frameon=False)
        self.ax.set_xlim(0,500)
        self.ax.set_ylim(-80,-20)


        #self.line,=self.ax.plot([], [], lw=2)
        #self.line.set_data([1,2],[1,2])
        #self.ax.plot([1,2],[20,20],marker='o',linestyle='-
',color='r')
        #self.ax.plot(3,20,marker='o',linestyle='--
',color='r')

        #self.x=0
        #self.y=0


        #self.ax.set_xlabel( 'Packet Number' )
        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)
        #self.canvas._tkcanvas.pack(side=tk.TOP,
fill=tk.BOTH, expand=True)
        #self.canvas.draw()

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

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

        sys.stdout = StdRedirector(self.text_box,self)
        sys.stderr = StdRedirector(self.text_box,self)
```

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

        #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='')



        #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
(kHz):')
        self.etiq4=tk.Label(self.Frame13,text='Preamble
lenght:')
        self.etiq5=tk.Label(self.Frame13,text='Channel
(MHz):')
        self.etiq6=tk.Label(self.Frame13,text='CRC:')
        self.etiq7=tk.Label(self.Frame15,text='Set power
(dBm):')
        self.etiq8=tk.Label(self.Frame5,text='Receiving:')

        self.etiq9=tk.Label(self.Frame6,text='[PacketNumber,
PayloadLength, [Payload], CRC_CHECK, PacketRSSI, PacketSNR,
channelRSSI]')



        #Inputs

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

self.spredingfactor=ttk.Combobox(self.Frame12,textvariable=se
lf.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.p
reamblelenghtn,width=13)
```

```
self.setchannel=tk.Entry(self.Frame14,textvariable=self.setch
anneln,width=13)

self.crc=ttk.Combobox(self.Frame14,textvariable=self.crcn,wid
th=10)
        self.crc['values']=('ON','OFF')

self.setpower=tk.Entry(self.Frame16,textvariable=self.setpowe
rn,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)


        #Positionig labels and inputs


        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)

        self.etiq9.pack(side=tk.LEFT)
```

```
        #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.Frame6.pack(side = tk.TOP,fill=tk.BOTH)
        self.Frame4.pack(side =
tk.TOP,fill=tk.BOTH,expand=tk.YES)

self.Frame5.pack(side=tk.TOP,fill=tk.BOTH)#expand=tk.YES)









        self.raiz.mainloop()


    def start(self):
        cont=0
        error_dato = False
        try:
            codingrate = int(self.codingraten.get())
            spredingfactor=int(self.spredingfactorn.get())
            bw=int(self.bwn.get())
            preamblelenght= int(self.preamblelenghtn.get())
            setchannel=float(self.setchanneln.get())
            #print setchannel
            crc=self.crcn.get()
            setpower=int(self.setpowern.get())
        except:
            error_dato = True
```

```
        if (self.lora.rx==False):
            if not error_dato:
                self.filename =
fd.asksaveasfilename(filetypes=[("Plain text","*.txt")],
defaultextension = "*.txt")
                if self.filename:
                    self.file= open(self.filename, 'w')
                    self.state=1

self.thr=PeriodicThread.PeriodicThread(self.update,0.01)

self.lora.threadrx=PeriodicThread.PeriodicThread(self.lora.ge
tonePacket,0.01)

                self.ax.lines=[]
                self.ax1.lines=[]
                self.lora.queuerx.clear()



self.lora.setConfiguration(bw,codingrate,spredingfactor,pream
blelenght,setchannel,setpower,crc)
                if self.lora.ko==0:
                    self.lora.receive()



                    # Start threading
                    self.lora.startRx()
                    self.thr.start()

self.circle.itemconfig(self.item,fill='green')
                    print('Receiving..........')
                    print('\n')
                    print('\n')
                    cont=1

        if (self.lora.rx==True):
            #c.ON()
            #print('jiji')
            self.lora.stopRx()
            print('Stop receiving...')

            self.lora.ft232h.output(5, sx.GPIO.LOW)
            #while (len(self.lora.graph)>0) or
(len(self.lora.queuerx)>0):
            #    pass
            l=len(self.lora.graph)
            l1=len(self.lora.queuerx)
            #print l
```

```
            #print l1
            #time.sleep(max(l,l1)*0.02)
            self.thr.cancel()
            for i in range(max(l,l1)):
                self.update()
            #print len(self.lora.graph)
            #print len(self.lora.queuerx)
            self.circle.itemconfig(self.item,fill='')
            print('Receiving stopped')
            print('\n')
            print('\n')
            #for l in range(len(self.ax.lines)):

            if self.state==1:
                self.file.close()
            self.state=0
            self.lora.rx=False

        if (cont==1):
            self.lora.rx=True









    def clearwindow(self):
        #if self.lora.rx==True:
        #    self.start()
        #filename = fd.asksaveasfilename(filetypes=[("Plain
text","*.txt")], defaultextension = "*.txt")
        #print filename
        #if filename:
        #    with open(filename, 'w') as file:
        #        text=self.text_box.get(1.0, tk.END)
        #        file.write(text.encode('utf8'))
                #lines=text.split('\n')
                #for l in range(len(lines)):
                #    file.write(lines[l]+'\n')
        self.text_box.config(state=tk.NORMAL)
        self.text_box.delete(1.0,tk.END)
        self.text_box.config(state=tk.DISABLED)

        self.ax.lines=[]
        self.ax1.lines=[]
```

```
    def TX(self):
        if self.lora.rx==True:
            self.start()
        self.top = tk.Toplevel(self.raiz)
        self.top.geometry("200x50-200+100")
        self.top.transient(self.raiz)
        self.top.grab_set()
        self.appc=Demo(self.top,self)
        #self.dialogo.grab_set()
```

```
    def update(self):
        #self.line.set_data([x,x+1],[y,y+1])
        #print self.text_box.yview
        if (len(self.lora.graph)>0):
            a=self.lora.graph.popleft()
            if (a[0]==1):
                self.ax.set_xlim(0,500)
                self.ax1.set_xlim(0,500)


            #Graph Rssi
            xmin,xmax=self.ax.get_xlim()
            if a[0]>=xmax:
                self.ax.set_xlim(xmin+10,xmax+10)

            ymin,ymax=self.ax.get_ylim()
            if a[1]>=ymax:
                self.ax.set_ylim(ymin,a[1]+5)

            if a[1]<=ymin:
                self.ax.set_ylim(a[1]-5,ymax)

            #Graph Snr
```

```
            xmin,xmax=self.ax1.get_xlim()
            if a[0]>=xmax:
                self.ax1.set_xlim(xmin+10,xmax+10)

            ymin,ymax=self.ax1.get_ylim()
            if a[2]>=ymax:
                self.ax1.set_ylim(ymin,a[2]+5)

            if a[2]<=ymin:
                self.ax1.set_ylim(a[2]-5,ymax)



self.ax.plot([a[0]],[a[1]],marker='o',linestyle='-
',color='r')

self.ax1.plot([a[0]],[a[2]],marker='o',linestyle='-
',color='g')




            self.f.canvas.draw_idle()   #self.f.canvas.draw()
petaaaaaa
            #print self.ax.lines
        #app.ax.plot([x,x+1],[y,y+1],marker='o',linestyle='-
',color='r')
        if (len(self.lora.queuerx)>0):
            print self.lora.queuerx.popleft()







    def on_closing(self):
        self.lora.stopRx()
        self.thr.cancel()
        self.raiz.destroy()



class Demo():
    def __init__(self,master,parent):
        self.master=master
        self.parent=parent
```

```
        self.textn=tk.StringVar(value='')


self.text=tk.Entry(self.master,textvariable=self.textn,width=
30)
        self.send = tk.Button(self.master, text="Send",
command=self.send,width=15)

        self.text.pack(side=tk.TOP)
        self.send.pack(side=tk.TOP)

        #self.master.mainloop()

    def send(self):

        error_dato = False
        try:
            codingrate = int(self.parent.codingraten.get())

spredingfactor=int(self.parent.spredingfactorn.get())
            bw=int(self.parent.bwn.get())
            preamblelenght=
int(self.parent.preamblelenghtn.get())
            setchannel=float(self.parent.setchanneln.get())
            #print setchannel
            crc=self.parent.crcn.get()
            setpower=int(self.parent.setpowern.get())
        except:
            error_dato = True
        #print error_dato
        if not error_dato:

#self.parent.circle1.itemconfig(self.parent.item1,fill='green
')

self.parent.lora.setConfiguration(bw,codingrate,spredingfacto
r,preamblelenght,setchannel,setpower,crc)
            #print type(self.text.get())
            #print self.text.get()
            if self.parent.lora.ko==0:
                #print type(self.textn.get())
                #if len(self.textn.get())%2==1:
                #    a='0'+self.textn.get()
                #else:
                #    a=self.textn.get()

self.parent.lora.upload_buffer(self.textn.get())
                print 'Message to send:',self.textn.get()
                self.parent.lora.transmit()
```

```
app=Aplicacion()
```