Part 46
-
LoRa & LoRaWAN

Version: 2020-03-27

Introduction

LoRa (Long Range) is a low-power wide-area network wireless technology. Originally developed and patented by SemTech.

LoRa and LoRaWAN permit long-range connectivity for Internet of Things (IoT) devices in different types of industries.

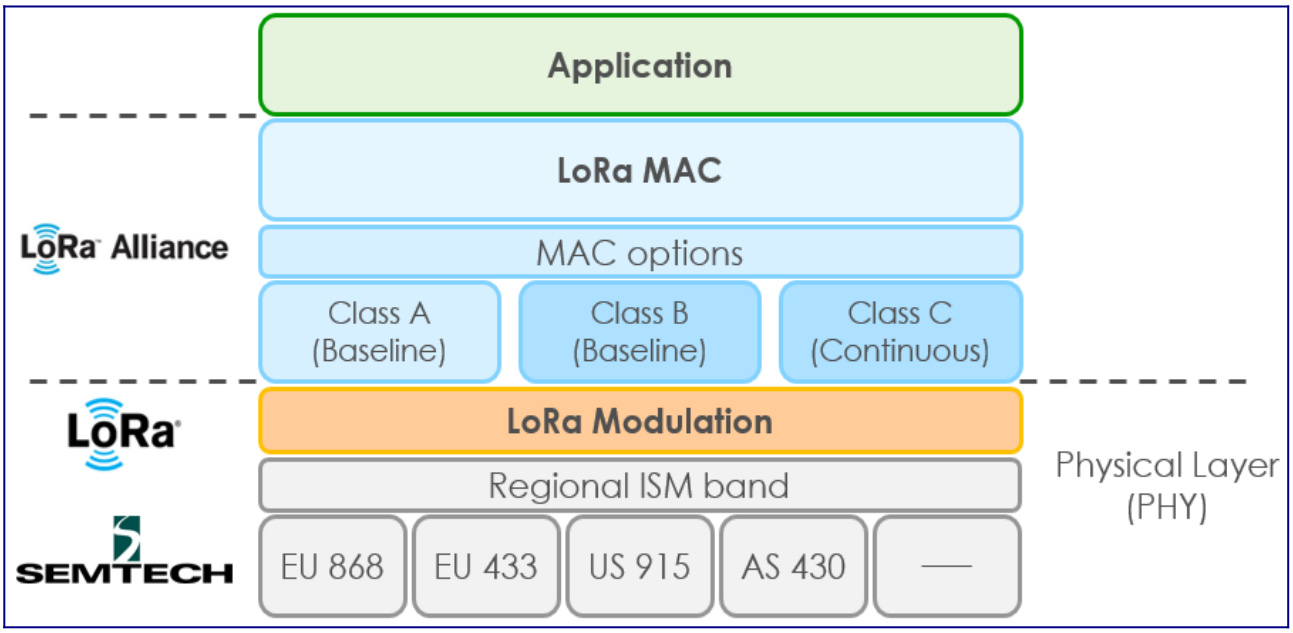
LoRaWAN defines the communication protocol and system architecture for the network, while the LoRa physical layer enables the long-range communication link.

LoRa enables sending small amounts of information between objects with very low power. LoRa receivers are very sensitive due to spread spectrum, allowing the sender to send with very low power even when noise is high and distance are far. LoRa enables long-range transmissions of more than 10 km in rural areas with low power consumption. LoRa uses license-free sub-gigahertz radio frequency bands like 433 MHz, 868 MHz (Europe), 915 MHz (Australia and North America) and 923 MHz (Asia).
LoRa devices have geolocation capabilities used for triangulating positions of devices via timestamps from gateways

LoRa Wide-Area-Network (LoRaWAN) uses LoRa modulation to build a network of a few gateways (cell-towers) to communicate with many devices. Basically it is a set of rules on how to communicate and what protocols to use. LoRaWAN is designed to enable many devices to send uplink messages (from a device to the network). LoRaWAN also enables replying with a downlink message (from server to device), although the number of downlink messages is limited.

LoRaWAN is a cloud-based medium access control (MAC) layer protocol but acts mainly as a network layer protocol for managing communication between LPWAN gateways and end-node devices as a routing protocol, maintained by the LoRa Alliance.
LoRaWAN is also responsible for managing the communication frequencies, data rate, and power for all devices. Devices in the network are asynchronous and transmit when they have data available to send. Data transmitted by an end-node device is received by multiple gateways, which forward the data packets to a centralized network server. The network server filters duplicate packets, performs security checks, and manages the network. Data is then forwarded to application servers. The technology shows high reliability for the moderate load, however, it has some performance issues related to sending acknowledgements.

The below figure gives a graphical representation of the complete stack.



The LoRa Alliance is an association created in 2015 to support LoRaWAN protocol as well as ensure interoperability of all LoRaWAN products and technologies. This open, nonprofit association has over 500 members. Some members of the LoRa Alliance are IBM, Actility, MicroChip, Orange, Cisco, KPN, Swisscom, Semtech, Bouygues Telecom, Singtel, Proximus and Cavagna Group. In 2018, the LoRa Alliance had over 100 LoRaWAN network operators in over 100 countries.

While in most countries, LoRaWAN network are managed by regular companies and will request you to pay a monthly fee to use their LoRaWAN network, there is a open-source free organisation, The Things Network, or short TTN that allows you to hookup your own LoRaWAN gateway, devices and applications to get your data across.

A bit more technical

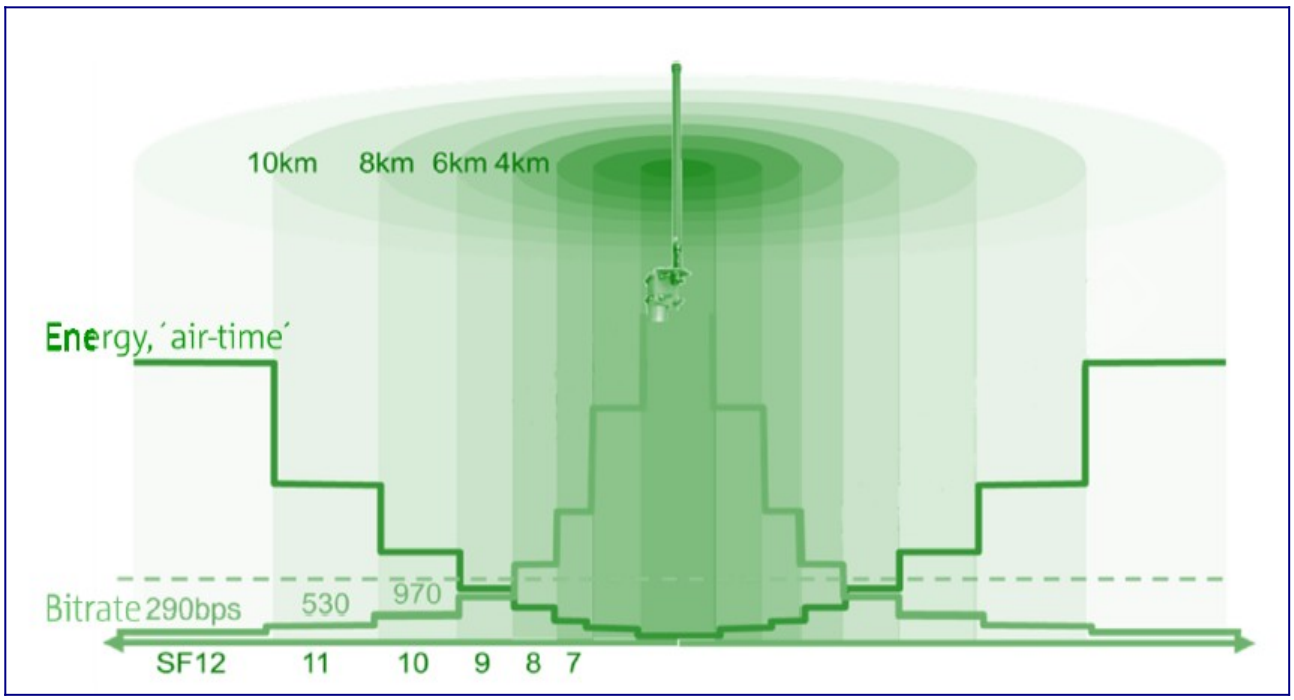
Data Rate
Communication between end-devices and gateways is spread out over different frequency channels and data rates. The selection of the data rate is a trade-off between communication range and message duration. Within the selected channel the LoRa protocol, which is a chirp spread spectrum modulation technique, determines how many bits are required to code the data (coding rate) which results in a maximum data rate.

Spreading Factor
The basic principle of spread spectrum is that each bit of information is encoded as multiple chirps. Within the given bandwidth the relationship between the bit and chirp rate for LoRa modulation may differ between spreading factor (SF) 7 to 12. The end-device may transmit on any channel available at any time, using any available data rate, as long as the following rules are respected:

- The end-device changes channel in a pseudo-random fashion for every transmission. The resulting frequency diversity makes the system more robust to interferences.
- The end-device respects the maximum transmit duty cycle relative to the sub-band used and local regulations.
- The end-device respects the maximum transmit duration (or dwell time) relative to the sub-band used and local regulations.

Time on Air
An important consequence of using a higher spreading factor for LoRa is a longer time on air (ToA). The LoRa Radio module needs more time to send the same amount of data. This means that power consumption increases with increasing Spreading Factor. To optimize spreading factor, Adaptive Data Rate (ADR) should be used.
Note: LoRaWAN specifies that each time a message is send in one ISM subband, the device must wait the remaining time of the duty cycle in that band before resending (For time on air of 0.5s and 1% duty-cycle, this means waiting 49,5s).

Adaptive Data Rate (ADR)
LoRa data rates range from 0.3 kbps to 50 kbps. Depending on the environmental conditions between the communication device and the gateway the network will determine the best spreading factor (SF) to work on. To maximize both battery life, range and overall network capacity, the LoRa network infrastructure can manage the data rate and output power used for the communication for each end-device individually by means of an adaptive data rate (ADR) scheme. Meaning the better the coverage the lower the SF will be (see the figure below). Whether the ADR functionality will be used is requested by the device, not by the network. Switching ADR OFF is only advised for continually moving objects.



The default channels in Europe are

Channel	Frequency	Spreading factor	Bandwidth
0	868,1 MHz	SF7 to SF12	125 KHz
1	868,3 MHz	SF7 to SF12	125 KHz
2	868,5 MHz	SF7 to SF12	125 KHz

Note: Semtech Chip SX1276/77/78/79

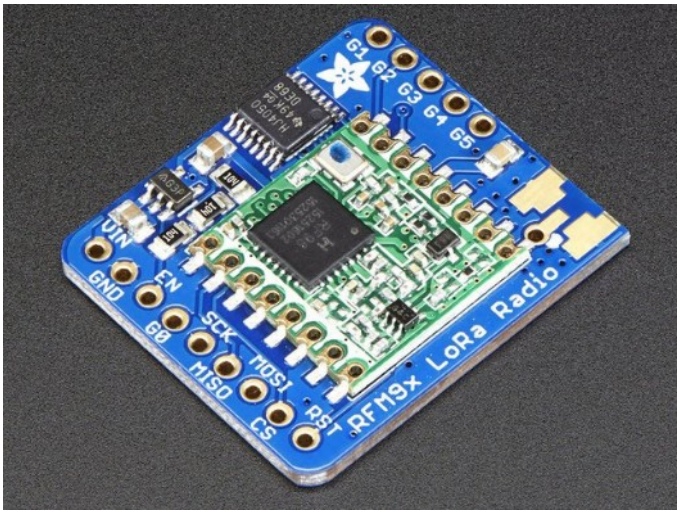
Part Number	Frequency Range	Spreading Factor	Bandwidth	Effective Bitrate	Est. Sensitivity
SX1276	137-1020 MHz	6-12	7.8-500 kHz	0.018-37.5 kbps	-111 to -148 dBm
SX1277	137-1020 MHz	6-9	7.8-500 kHz	0.110-37.5 kbps	-111 to -139 dBm
SX1278	137-525 MHz	6-12	7.8-500 kHz	0.018-37.5 kbps	-111 to -148 dBm
SX1279	137-960MHz	6-12	7.8-500 kHz	0.018-37.5 kbps	-111 to -148 dBm

Let's use LoRa

There is no need to use LoRaWAN if you want to have devices to talk between eachother over long distance (we talk about kilometers). You only have to use 2 LoRa radio modules, each hooked up to a device (Eg. Arduino, Raspberry Pi) and have some software so they can talk to eachother.

Adafruit has such a module, the RFM9x LoRa Radio. But you can also use the Dragino LoRa/GPS HAT. In the code I added the changes that are required to use the HAT.

Important: when buying modules, make sure you buy the right frequency set for you region. See above. For Europe, this is 868 MHz.



Note:

- You will need 2 modules (and 2 Pi's) to run this test/demo.
- Do not forget to solder at least a wire as antenna onto the board (see https://learn.adafruit.com/adafruit-rfm69hcw-and-rfm96-rfm95-rfm98-lora-packet-padio-breakouts/assembly)

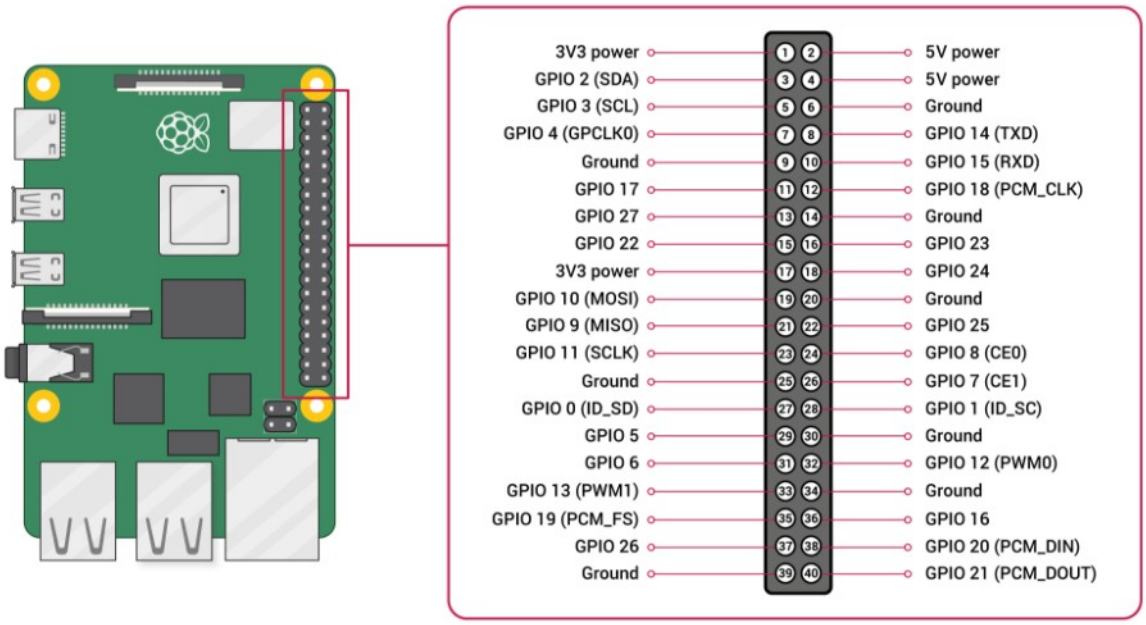
The module use SPI (as the chip does), so configure your Pi to activate SPI. You can do this using the `raspi-config` tool

```
sudo raspi-config
```

or follow my document `Raspberry Pi - Part 11 - Activating SPI.pdf`.

The wiring to use

RFM9x LoRa	Raspberry Pi GPIO	Raspberry Pi Connector
VIN	3.3V	1
GND	GND	6
G0	GPIO 5	29
RST	GPIO 25	22
CS	GPIO 7 /CE 1	26
MISO	GPIO 13 / MISO	21
MOSI	GPIO 12 / MOSI	19
SCK	GPIO 14 / SCLK	23



We're going to use CircuitPython (created by AdaFruit) on the Raspberry Pi to be able to use the AdaFruit library for the RFM9x module.

Install CircuitPython
So first, install the required CircuitPython libraries

```
pip3 install adafruit-blinka
```

Test if all is installed correctly

```
nano testCP.py

import board
import digitalio
import busio

# Try to great a Digital input
pin = digitalio.DigitalInOut(board.D4)
print("Digital IO ok!")

# Try to create an SPI device
spi = busio.SPI(board.SCLK, board.MOSI, board.MISO)
print("SPI ok!")

print("done!")
```

Save and run it

```
python3 testCP.py
```

You should get this as result

```
Digital IO ok!
SPI ok!
done!
```

Test the RFM9x Connection (or Dragino LoRa/GPS Hat)
Let's first install the RFM9x module that allows us to easily write Python code that sends and receives packets of data with the radio. To install the library for the RFM9x Module, enter the following

```
pip3 install adafruit-circuitpython-rfm9x
```

Note: This module expects to find a SX1276 chip.

With the following code we can check if the RFM9x radio is hooked up the right way

```
nano rfm9x_check.py

import time
import busio

from digitalio import DigitalInOut, Direction, Pull
import board
# Import the RFM9x radio module.
import adafruit_rfm9x

# Configuration RFM9x LoRa Radio - GPIO/BCM numbering
CS = DigitalInOut(board.CE1)
RESET = DigitalInOut(board.D25)
# Configuration Dragino LoRa/GPS HAT
# CS = DigitalInOut(board.D25)
# RESET = DigitalInOut(board.D17)

spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)

try:
    rfm9x = adafruit_rfm9x.RFM9x(spi, CS, RESET, 868.1)
    print('RFM9x: Detected')
    print('Frequency         : ' + str(rfm9x.frequency_mhz))
    print('High Power        : ' + str(rfm9x.high_power))
    print('Transmit Power    : ' + str(rfm9x.tx_power))
    print('Coding Rate       : ' + str(rfm9x.coding_rate))
    print('Spreading Factor  : ' + str(rfm9x.spreading_factor))
    print('Bandwidth         : ' + str(rfm9x.signal_bandwidth))

except RuntimeError as error:
    # Thrown on version mismatch
    print('RFM9x Error: ', error)
    time.sleep(5)
```

To use the code, enter the following in your terminal:

```
python3 rfm9x_check.py
```

If the wiring of the radio module is incorrect, the error will be printed. Check over your wiring and re-run the test.

All The Way
Next, we want to use the radio to transmit and receive data. It's easy when using the RFM9x LoRa radio with CircuitPython and the Adafruit CircuitPython RFM9x module.
Below is an example of using the RFM9x to transmit to, or receive from, another RFM9x radio. To have this working, set-up 2 Pi's with each a RFM9x module and install CircuitPython, RFM9x package on it. Create on both this application

```
nano rfm9x_radio.py

# Import Python System Libraries
import time
from datetime import datetime
import random
import busio
import board
from digitalio import DigitalInOut, Direction, Pull
import adafruit_rfm9x

# Configure RFM9x LoRa Radio
CS = DigitalInOut(board.CE1)
RESET = DigitalInOut(board.D25)
# Configure Dragino LoRa/GPS HAT
# CS = DigitalInOut(board.D25)
# RESET = DigitalInOut(board.D17)
spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)

rfm9x = adafruit_rfm9x.RFM9x(spi, CS, RESET, 868.1)
rfm9x.tx_power = 23  # range from 5 to 23 if high_power is True
rfm9x.coding_rate = 5  # range from 5 to 8
rfm9x.spreading_factor = 7 # range from 6 tot 12
rfm9x.signal_bandwidth = 125000
# valid values 7800, 10400, 15600, 20800, 31250, 41700, 62500, 125000, 250000
prev_packet = None

while True:
    packet = None

    # check for packet rx
    packet = rfm9x.receive()
    if packet is None:
        print('Waiting for data ...')

    else:
        prev_packet = packet
        packet_text = str(prev_packet, "utf-8")
        print('RX: ' + packet_text)
        time.sleep(1)

    time.sleep(0.1)

    if random.randint(0,100) < 10:
        text = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        data = bytes( text + "\r\n","utf-8")
        rfm9x.send(data)
        print('TX : ' + text)
```

To run the test/demo, enter the following into the terminal:

```
python3 rfm9x_radio.py
```

Both of the radios will listen for a new incoming packet. When they receive a new packet, they'll print the text from the packet.

For detailed information on the settings of the radio parameter I refer to https://circuitpython.readthedocs.io/projects/rfm9x/en/latest/api.html

Same but not using the AdaFruit software

There is also a pyLoRa package on PyPi that can be used for client-server set-up.

Before starting make sure these packages are installed and are the latest version

```
sudo apt -y install python3-dev
sudo apt -y install python3-pip
sudo apt -y install python3-rpi.gpio
sudo apt -y install python3-spidev
sudo apt -y install git
pip3 install RPi.GPIO
```

Get the Pyhthon LoRa library

```
pip3 install pyLoRa
```

For encrypted versions (see later in examples) **only,** it is necessary to add also

```
pip3 install pycryptodome
pip3 install pycrypto
```

Basics on how to use the pyLoRa module
First import the modules

```
from SX127x.LoRa import *
from SX127x.board_config import BOARD
```

then set up the board GPIOs

```
BOARD.setup()
```

The LoRa object is instantiated and put into the standby mode

```
lora = LoRa()
lora.set_mode(MODE.STDBY)
```

Registers are queried like so:

```
print lora.get_version()    # this prints the sx127x chip version
print lora.get_freq()       # this prints the frequency setting
```

and setting registers is easy, too

```
lora.set_freq(868.1)        # Set the frequency to 868.1 MHz
```

In applications the LoRa class should be subclassed while overriding one or more of the callback functions that are invoked on successful RX or TX operations, for example.

```
class MyLoRa(LoRa):

  def __init__(self, verbose=False):
    super(MyLoRa, self).__init__(verbose)
    # setup registers etc.

  def on_rx_done(self):
    payload = self.read_payload(nocheck=True)
    # etc.
```

In the end the resources should be freed properly

```
BOARD.teardown()
```

Examples are here
Download the examples

```
sudo git clone https://github.com/rpsreal/pySX127x
```

First edit the file board_config.py in de SX127x folder and change the pin numbers to match your set-up.

RFM9x LoRa	board_config	Raspberry Pi
VIN		3.3V
GND		GND
G0	DIO0	GPIO 5
RST	RST	GPIO 25
CS	SPI_CS	GPIO 7 /CE 1
MISO		GPIO 13 / MISO
MOSI		GPIO 12 / MOSI
SCK		GPIO 14 / SCLK

So in this case

```
DIO0 = 5    # RasPi GPIO 5
RST  = 25   # RasPi GPIO 25

# The spi object is kept here
spi     = None
SPI_BUS = 0
SPI_CS  = 1
```

Note: we do not use the LED, so you can ignore that one

How to Use
View the sample files. If you downloaded the library and sample files, now you can start LORA_SERVER or LORA_CLIENT (encrypted or non-encripted). To work, there must be another LORA_SERVER or LORA_CLIENT running on another Raspberry Pi

On the server, you start

```
cd pySX127x
python3 ./LORA_SERVER.py
```

On the client you start

```
cd pySX127x
python3 ./LORA_CLIENT.py
```

If you look into the coding, you will see that you can change some radio parameters

```
lora.set_pa_config(pa_select=1, max_power=21, output_power=15)
lora.set_bw(BW.BW125)
lora.set_coding_rate(CODING_RATE.CR4_8)
lora.set_spreading_factor(12)
lora.set_rx_crc(True)
```

but you will not find the frequency. Default the frequency is on 868 (lucky us!) but if you want to change it, you can specify it this way,

```
lora = mylora(verbose=False, calibration_freq=868)
```

Note: if you install the pyLoRapackage and do not install the sample files from Git, you need to change your board settings in the pyLoRa package.

Do a search for board_config.py, and edit the file

```
find . -name "board_config.py"
```

In my case

```
/usr/local/lib/python3.7/dist-packages/SX127x/board_config.py

sudo nano /usr/local/lib/python3.7/dist-packages/SX127x/board_config.py
```

Good To Know

Class Reference
The interface to the SX127x LoRa modem is implemented in the class SX127x.LoRa.LoRa. The most important modem configuration parameters are:

Function	**Description**
set_mode	Change OpMode, use the constants.MODE class
set_freq	Set the frequency
set_bw	Set the bandwidth 7.8kHz ... 500kHz
set_coding_rate	Set the coding rate 4/5, 4/6, 4/7, 4/8

Most set_* functions have a mirror get_* function, but beware that the getter return types do not necessarily match the setter input types.

Other included scripts

Continuous receiver rx_cont.py

The SX127x is put in RXCONT mode and continuously waits for transmissions. Upon a successful read the payload and the irq flags are printed to screen.

```
usage: rx_cont.py [-h] [--ocp OCP] [--sf SF] [--freq FREQ] [--bw BW]
                  [--cr CODING_RATE] [--preamble PREAMBLE]

Continous LoRa receiver

optional arguments:
  -h, --help            show this help message and exit
  --ocp OCP, -c OCP     Over current protection in mA (45 .. 240 mA)
  --sf SF, -s SF        Spreading factor (6...12). Default is 7.
  --freq FREQ, -f FREQ  Frequency
  --bw BW, -b BW        Bandwidth (one of BW7_8 BW10_4 BW15_6 BW20_8 BW31_25
                        BW41_7 BW62_5 BW125 BW250 BW500). Default is BW125.
  --cr CODING_RATE, -r CODING_RATE
                        Coding rate (one of CR4_5 CR4_6 CR4_7 CR4_8). Default
                        is CR4_5.
  --preamble PREAMBLE, -p PREAMBLE
                        Preamble length. Default is 8.
```

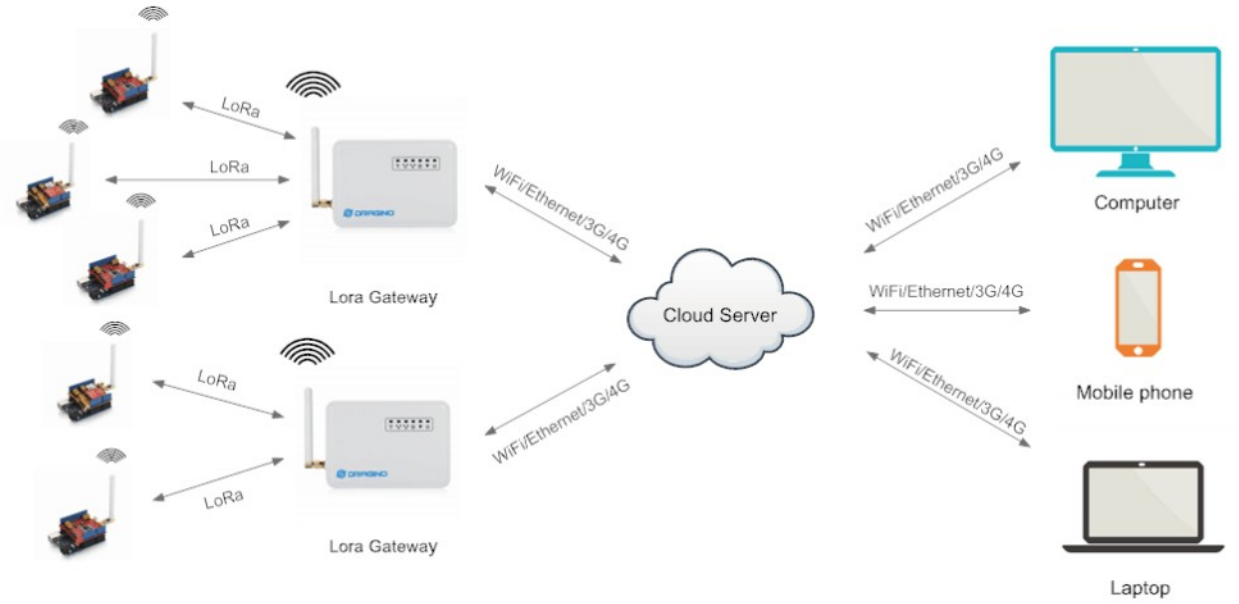
Simple LoRa beacon tx_beacon.py

A small payload is transmitted in regular intervals.

```
usage: tx_beacon.py [-h] [--ocp OCP] [--sf SF] [--freq FREQ] [--bw BW]
                    [--cr CODING_RATE] [--preamble PREAMBLE] [--single]
                    [--wait WAIT]

A simple LoRa beacon

optional arguments:
  -h, --help            show this help message and exit
  --ocp OCP, -c OCP     Over current protection in mA (45 .. 240 mA)
  --sf SF, -s SF        Spreading factor (6...12). Default is 7.
  --freq FREQ, -f FREQ  Frequency
  --bw BW, -b BW        Bandwidth (one of BW7_8 BW10_4 BW15_6 BW20_8 BW31_25
                        BW41_7 BW62_5 BW125 BW250 BW500). Default is BW125.
  --cr CODING_RATE, -r CODING_RATE
                        Coding rate (one of CR4_5 CR4_6 CR4_7 CR4_8). Default
                        is CR4_5.
  --preamble PREAMBLE, -p PREAMBLE
                        Preamble length. Default is 8.
  --single, -S          Single transmission
  --wait WAIT, -w WAIT  Waiting time between transmissions (default is 0s)
```

Let's use LoRaWAN

If you want to get get your data across the world (beyond 10 kilometers), you need to connect to a LoRaWAN. Hence need a LoRaWAN gateway in order to get the data of your IoT sensor across.

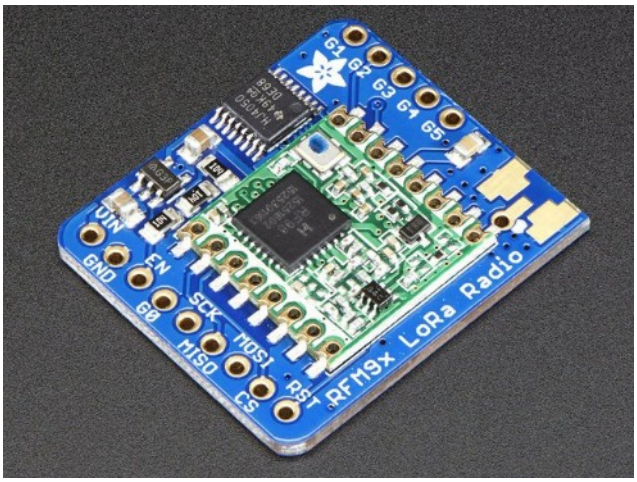


This gateway can be a commercial one, managed by companies like Telenet, Proximus. You will have to get a contract with one of them and pay monthly fees in order to be able to use it. These gateways have 8 channels and can handle hunderds of devices.
If you only have a few IoT devices, also called nodes, that need to transfer data and they are located or moved around in an area of less then a few kilometers around the gateway, you can use an existing one or set-up your own gateway.

Using an existing one is teid to a LoRaWAN network. As we are going to use The Things Network, you can check on their site if there is one in your neighbourhood.
Check out https://www.thethingsnetwork.org/map.

If none is in your neighbourhood, you can add a gateway of your own. If you go for a 8 channel gateway, you will need to spend several hunders of Euro's but a single channel gateway, that can handle between 1 and 50 IoT devices, is already available for less than 70 Euro, eg. Dragino LG01. And even an outdoor unit single channel for less than 80 Euro eg Dragino 0LG01.

Another option is to use a Raspberry Pi with a LoRa Hat to make a single channel gateway. Dragino has such a hat for about 35 Euro (which includes also a GPS module) while AdaFruit has a breakout module RFM9x LoRa that can be used for gateway or node. This costs about 20 Euro.



Most of these boards use the chipset SX127x of SemTech. The LoRa/GPS HAT is based on the SX127x transceiver

In this document we will cover the set-up of Dragino LG01, Raspberry Pi with Dragino hat as LoRa gateway and Raspberry Pi with AdaFruit RFM9X LoRa Radio as node.

Note: It is also possible to make a gateway with Raspberry Pi with AdaFruit RFM9X LoRa Radio using the same set-up but just make sure the right pins are defined in the gateway software.

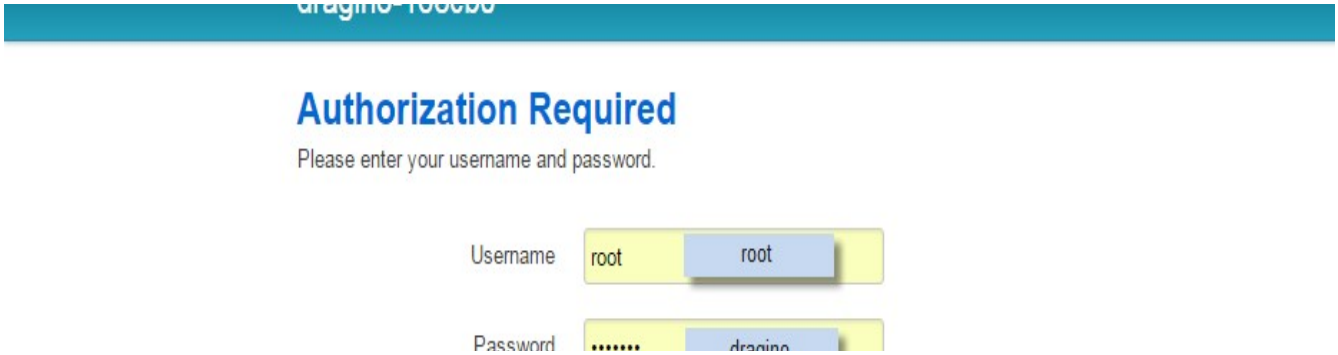
Setting up Dargino LG01 and connecting to The Things Network TTN

First we need to connect Dragino to our netwerk and to internet. This can be achieved in two ways either using an Ethernet cable in the WAN port or configuring the device as WiFi Client.

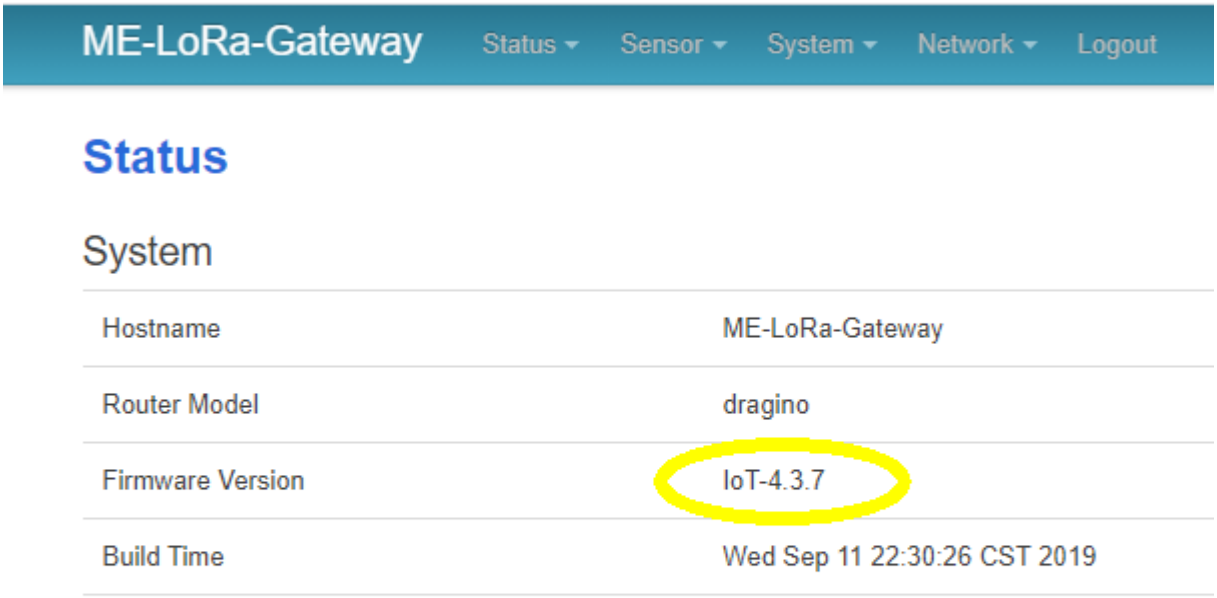
To accomplish both the connections we firstly need to link the Dragino gateway to our PC through another Ethernet cable using the LAN port. This, as the port name suggests, creates a LAN among the two items and gives us the opportunity to access the configuration page just typing in the browser this IP '10.130.1.1'.

or connect via WiFi. The LG01 is configured as a WiFi AP by factory default. User can access and configure the LG01 after connect to its WiFi network. At the first boot of LG01, it will auto generate an unsecure WiFi network call dragino2-xxxxxx. The laptop will get an IP address 10.130.1.xxx and the LG01 has the default IP 10.130.1.1.

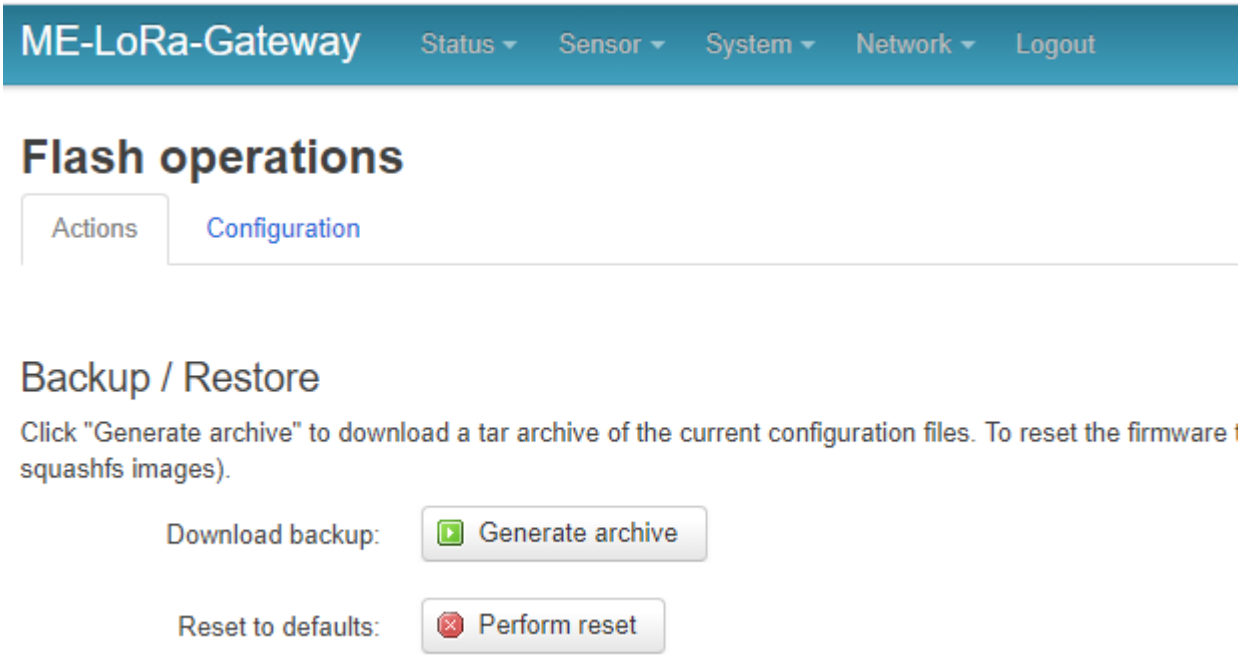
Open a browser and type 10.130.1.1. You will see the login screen. The account for is:
User Name: root
Password: dragino



First check if your LG01 has the latest frimware. Check on the site of Dragino and, if needed download the latest firmware.



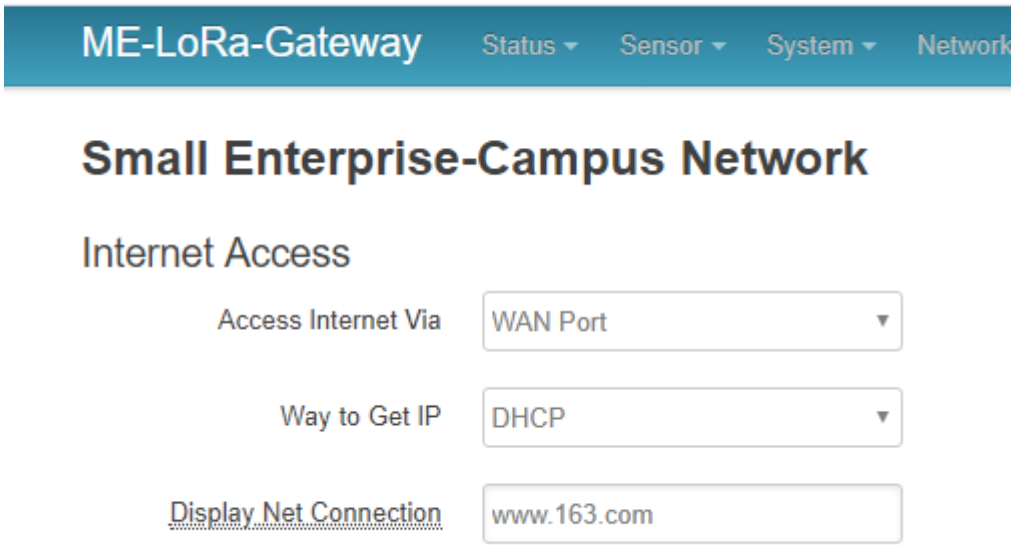
When upgrading the firmware of the LG01 or other Dragino device, you better perform a reset to defaults. I experienced connectivity problems not doing this.



Now that the LG01 is up to date, we can proceed in two different ways both navigating to "*Network -> Internet Access"*:

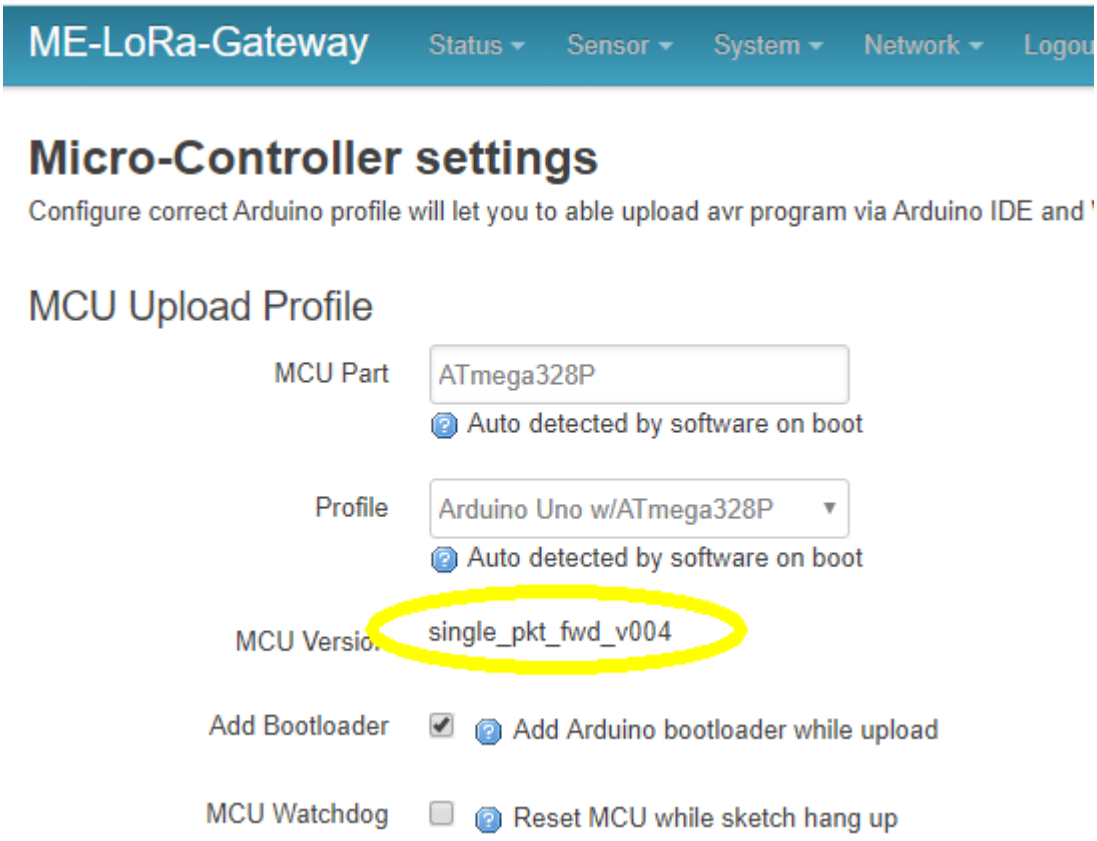
- **Ethernet procedure,** selecting in the "*Access Internet Via*" the option "*WAN port*" and linking the WAN port to the internet router.
- **Wifi procedure,** selecting in "*Access Internet Via*" the option "*WiFi Client*", filling "*SSID*", "*Password*" and "*Encryption*" with the proper information.

I have chosen to hook up via an Ethernet cable a the WAN port of my router (using a switch). My network provider gives me 4 public IP-addresses.

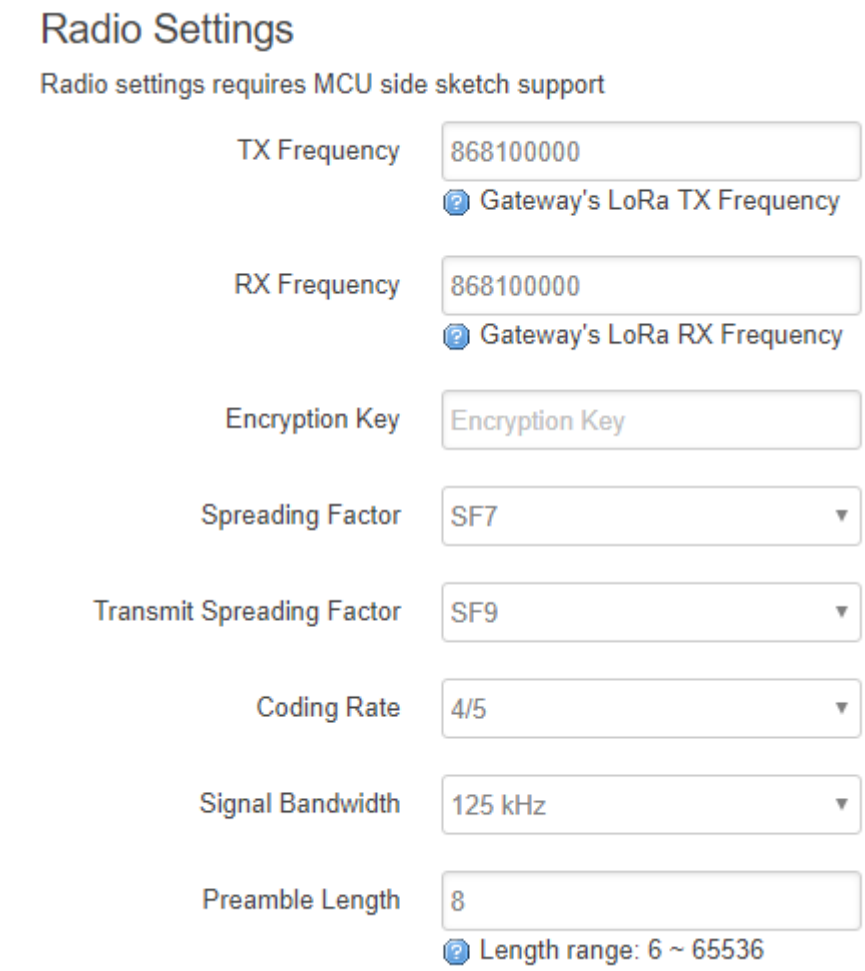


Save & Apply.

Once this is all set up and working, we need to upload the "Single Packet Forwarder" software to LG01. Go to http://www.dragino.com/downloads/index.php?dir=motherboards/lg01/sketch/ and download hex file `single_pkt_fwd_v004.hex` or later.
In LG01, go to page *Sensor --> Flash MCU* , Select the `single_pkt_fwd_v004.hex` file and upload it to LG01. Reboot LG01, after reboot, check page *Sensor --> MicroController*, see if the MCU version option shows the correct version, if yes, it means you have upload the code correctly.



Time to configure LG01 LoRa Radio. Go to *Sensor --> LoRaWAN*, input the correct LoRa radio settings. These settings are used to receive the LoRa radio info from LoRa nodes.
I am using the setting for Europe EU-868 LoRaWAN is 868.1Mhz.
Note: other frequenties to use are 868.3 and 868.5 MHz.
Note down these settings as you will need this info to set the node correctly

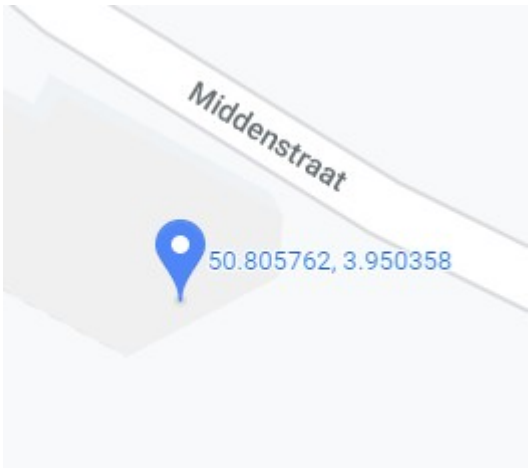


Save & Apply

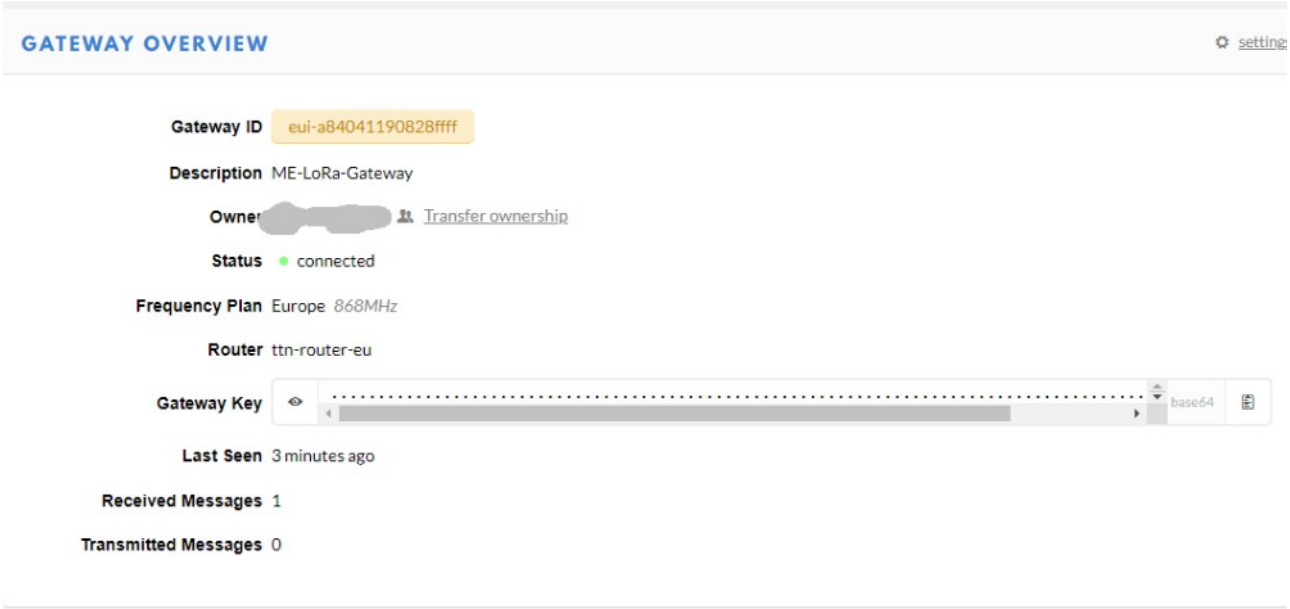
Before we can continue to the last step of the config, we need to set-up our connection in TTN.

Note that a LoRa gateway is used to receive packets from LoRa devices and forward them to the TTN networks or vice-versa.

Note down the MAC address of the WiFi that is mentioned on the bottom of the LG01. You should get something similar to `a8:40:41:19:08:28`.
Also, open Google Maps in your browser and find the coordinates of the place where you will put the gateway

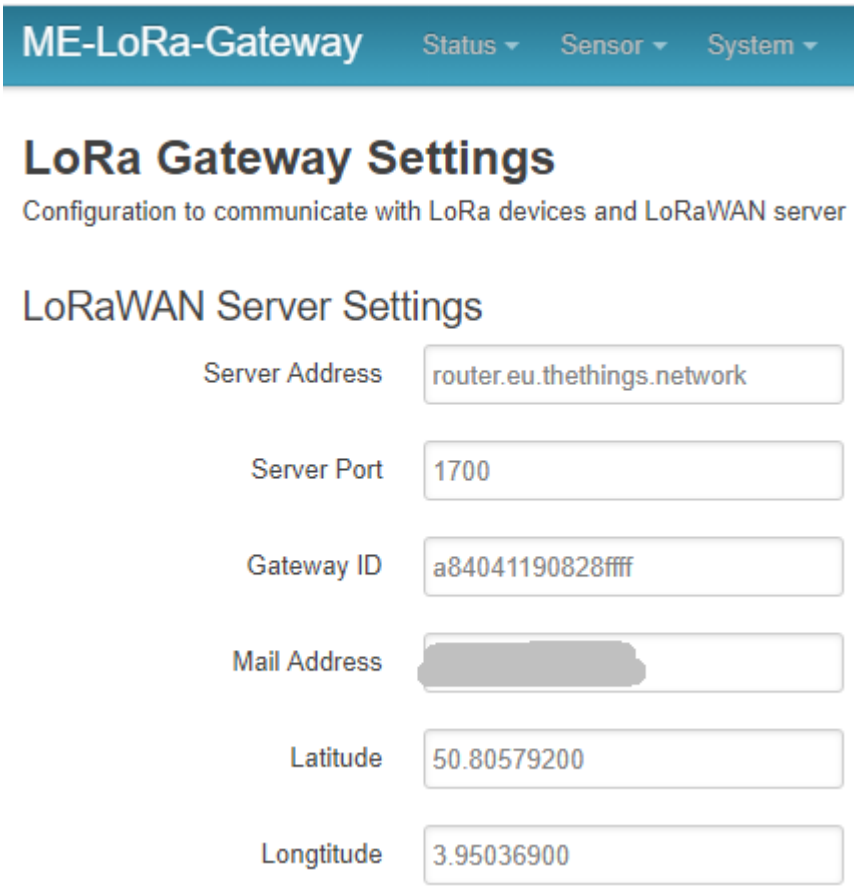


Now we can create a gateway into the TTN platform. Note that Dragino LG01 supports only the single channel gateway to connect to TTN server.
First create an account in TTN. Once this is completed, create a gateway in the TTN Console, making sure to use *"legacy packet forward method"* and as unique gateway ID, use the MAC address of the Dragino plus a suffix eg FFFF. The ID is a unique 8 Byte Hex. Eg. gateway ID: `a84041190828ffff`. When done, go to the overview



Here you see your Gateway ID which should be the same as you entered but preceeded with, in my case, `eui-` . Also, check *Router*, here it states `ttn-router-eu`.
It's up to you to complete the other information parts.

Now, go back to your LG01 and browse to *LG01 Sensor → LoRaWAN* to input the correct Lora Radio settings used to receive data fromthe LoRa nodes. *Save & Apply*

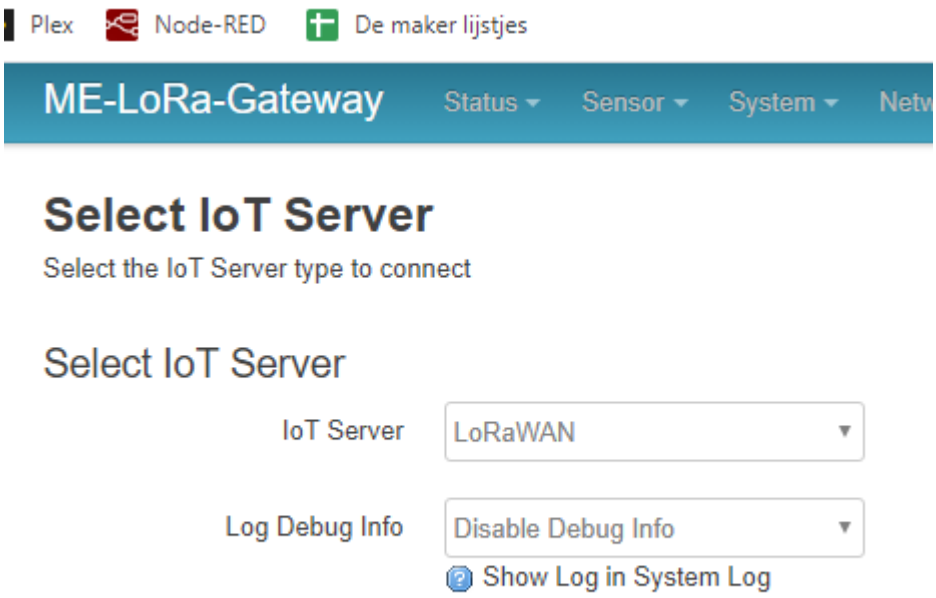


For the server address you need to fill in the right router address based on the info provided in your gateway overvies

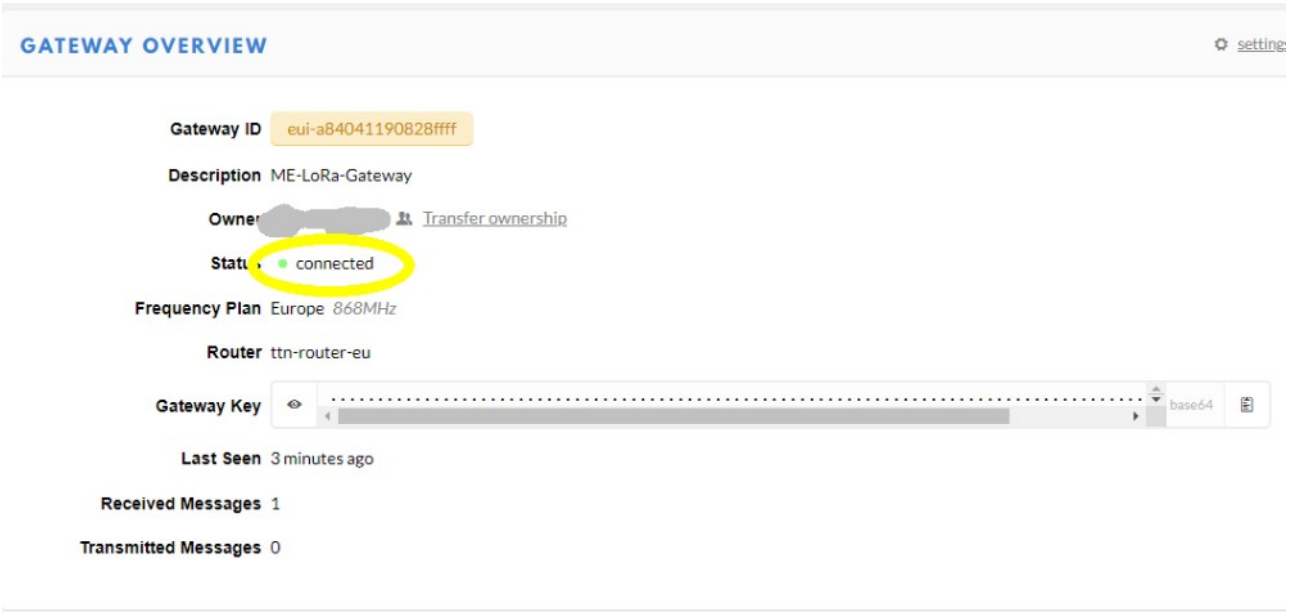
Region	**Router address**
EU 433 and EU 863-870	router.eu.thethings.network
US 902-928	router.us.thethings.network
China 470-510 and 779-787	router.cn.thethings.network
Southeast Asia 923 MHz	router.as.thethings.network
Southeast Asia 920-923 MHz	router.as1.thethings.network
Southeast Asia 923-925 MHz	router.as2.thethings.network
Korea 920-923 MHz	router.kr.thethings.network
Japan 923-925 MHz	router.jp.thethings.network
Australia 915-928 MHz	thethings.meshed.com.au
Australia (Southeast Asia 923MHz)	as923.thethings.meshed.com.au
Switzerland (EU 433 and EU 863-870)	ttn.opennetworkinfrastructure.org

Enter the gateway ID you noted down, your emailaddress, and the coordinates you found in Google Maps.

One more step is needed to get you up and running. Go to *Sensor → IoT Server* and set it to `LoRaWAN`



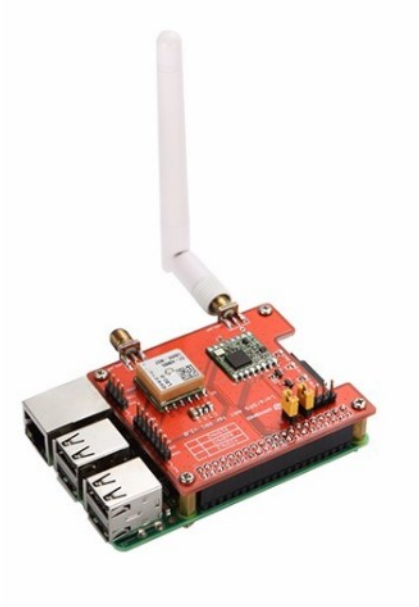
Save & Apply. That's it. Just reboot your LG01 to make sure all configs are taken into account. After the reboot, wait a couple of minutes and check out your gateway in TTN. It should be connected.



For any problem or any troubleshooting issue I refer you to the Dragino Wiki page http://wiki.dragino.com/index.php?title=Connect_to_TTN

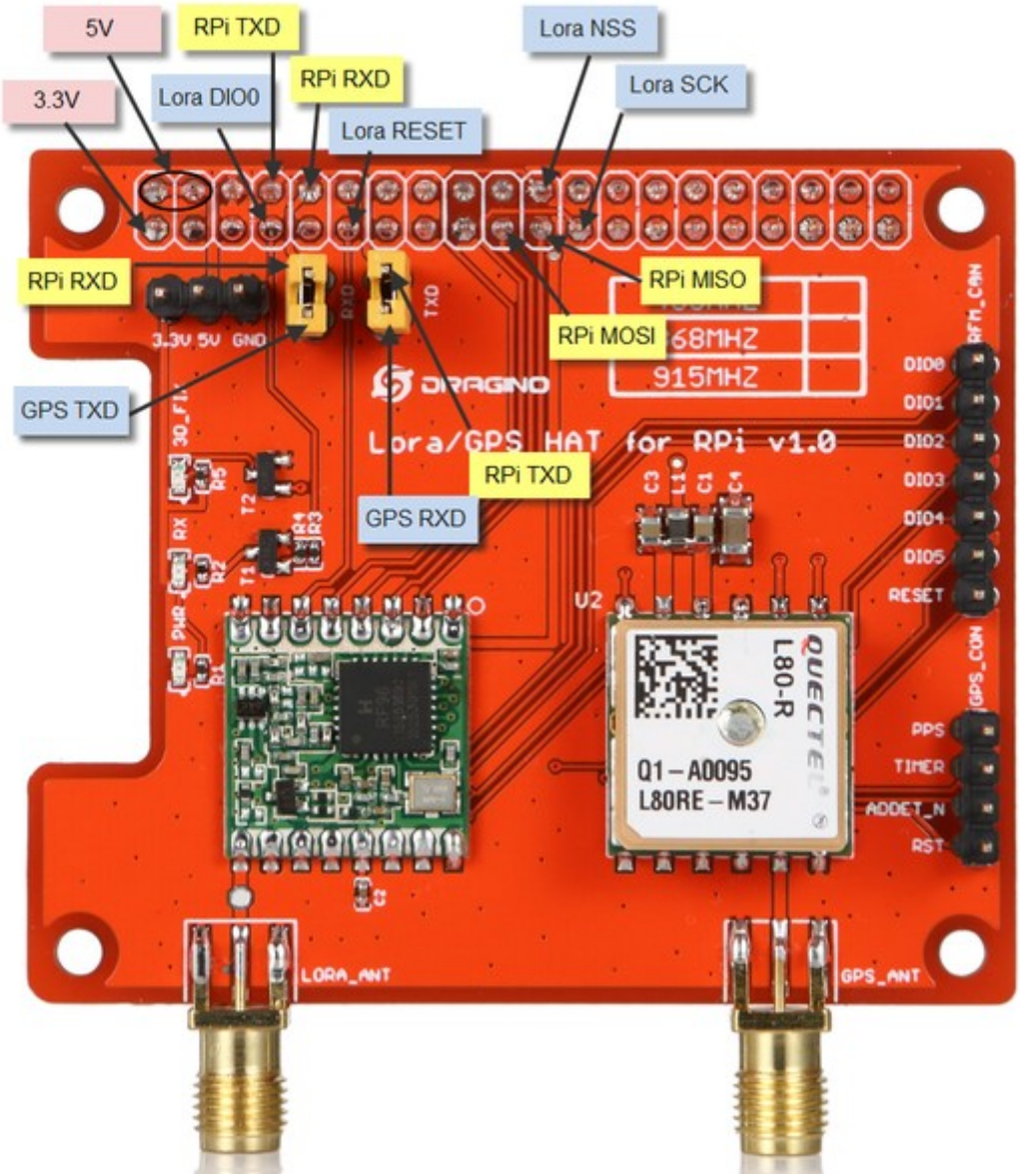
Setting up Raspberry Pi with Dargino HAT as LoRaWAN gateway

Note: It is also possible to make a gateway with Raspberry Pi with AdaFruit RFM9X LoRa Radio, just make sure the right pins are defined in the gateway softwate.



The assembly is easy. The Pin layout of the HAT is as follows

RFM9x LoRa	LoRa/GPS Hat	Raspberry Pi	Pin
VIN	3.3V	3.3V	1
	5V	5V	2
GND	GND	GND	6
G0	DI00	GPIO 7	7
	RX	GPIO 15/TX	8
	TX	GPIO 16/RX	10
RST	RESET	GPIO 0	11
CS	NSS	GPIO 6	22
MISO	MISO	GPIO 13 / MISO	21
MOSI	MOSI	GPIO 12 / MOSI	19
SCK	SCK	GPIO 14 / SCLK	23



The SX127X use SPI as interface, while DI00 is used as interrupt to indicate that the radio module received data and the data is ready to be read by your Pi. RX and TX are used for the GPS module on the HAT and not need at this moment. There are extra pins on the HAT, DIO1 to DI05, these are just extra pins in the SX127x module which we will not use.

Note: you find similar connections on the AdaFruit breakout module. Check out https://learn.adafruit.com/adafruit-rfm69hcw-and-rfm96-rfm95-rfm98-lora-packet-padio-breakouts/pinouts.

Once assembled, make an SD card with the latest Raspbian Lite on it, activate SSH, and if needed, you can also activate WiFi on your Pi. Set-up your Pi as you usually do.

Once your Pi is ready, start prepping for converting it to a LoRaWAN gateway.
As mentiond above, the modules use SPI, so configure your Pi to activate SPI. You can do this using the `raspi-config` tool

```
sudo raspi-config
```

or follow my document `Raspberry Pi - Part 11 - Activating SPI.pdf`.

Next, the software we will install uses a library/tool called `wiringpi`. So install this

```
sudo apt -y install wiringpi
```

Once install, check it using

```
gpio -v
```

You should get smething like this

```
root@MeRasPi3B-Test:/home/pi# gpio -v
gpio version: 2.50
Copyright (c) 2012-2018 Gordon Henderson
This is free software with ABSOLUTELY NO WARRANTY.
For details type: gpio -warranty

Raspberry Pi Details:
  Type: Pi 3B+, Revision: 03, Memory: 1024MB, Maker: Sony
  * Device tree is enabled.
  *--> Raspberry Pi 3 Model B Plus Rev 1.3
* This Raspberry Pi supports user-level GPIO access.
```

Then issue the command

```
gpio readall
```

```
+-----+-----+---------+------+---+---Pi 3B+-+---+------+---------+-----+-----+
| BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
+-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
|     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
|   2 |   8 |   SDA.1 |   IN | 1 |  3 || 4  |   |      | 5v      |     |     |
|   3 |   9 |   SCL.1 |   IN | 1 |  5 || 6  |   |      | 0v      |     |     |
|   4 |   7 | GPIO. 7 |   IN | 0 |  7 || 8  | 1 | ALT0 | TxD     | 15  | 14  |
|     |     |      0v |      |   |  9 || 10 | 1 | ALT0 | RxD     | 16  | 15  |
|  17 |   0 | GPIO. 0 |   IN | 1 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
|  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
|  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
|     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
|  10 |  12 |    MOSI | ALT0 | 0 | 19 || 20 |   |      | 0v      |     |     |
|   9 |  13 |    MISO | ALT0 | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
|  11 |  14 |    SCLK | ALT0 | 0 | 23 || 24 | 1 | OUT  | CE0     | 10  | 8   |
|     |     |      0v |      |   | 25 || 26 | 1 | OUT  | CE1     | 11  | 7   |
|   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
|   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
|   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
|  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
|  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
|  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
|     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
+-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
| BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
+-----+-----+---------+------+---+---Pi 3B+-+---+------+---------+-----+-----+
```

Next we are going to install the single packet forwarder software

```
git clone https://github.com/hallard/single_chan_pkt_fwd.git
```

Before running this gateway software, we need to check/modify some settings in the software. So

```
cd single_chan_pkt_fwd
nano global_conf.json
```

```
{
  "SX127x_conf":
  {
    "freq": 868100000,
    "spread_factor": 7,
    "pin_nss": 6,
    "pin_dio0": 7,
    "pin_rst": 0
  },
  "gateway_conf":
  {
    "ref_latitude": 50.805736,
    "ref_longitude": 3.950306,
    "ref_altitude": 5,

    "name": "Single-Channel DIY Gate",
    "email": "marc@engrie.be",
    "desc": "Dragino Single Channel Gateway on RPI",

    "servers":
    [
      {
        "address": "router.eu.thethings.network",
        "port": 1700,
        "enabled": true
      }
    ]
  }
}
```

First check the assigned pins and frequency

```
    "freq": 868100000,
    "spread_factor": 7,
    "pin_nss": 6,
    "pin_dio0": 7,
    "pin_rst": 0
```

```
Note: pin_led is not needed, so just remove it
```

Set the coordinates

```
    "ref_latitude": 50.805736,
    "ref_longitude": 3.950306,
    "ref_altitude": 5,
```

Set some information data

```
    "name": "Single-Channel DIY Gate",
    "email": "marc@engrie.be",
    "desc": "Dragino Single Channel Gateway on RPI",
```

```
Note: name should not be longer than 23 characters
```

Set the server to reach TTN

```
     {
        "address": "router.eu.thethings.network",
        "port": 1700,
        "enabled": true
      }
```

```
Note: only 1 is need and must be set to enabled: true
```

Once these changes are made, save and exit.

Now compile the program using

```
rm single_chan_pkt_fwd
make

g++ -c -Wall base64.c
g++ -c -Wall main.cpp
g++ main.o base64.o -lwiringPi -o single_chan_pkt_fwd
```

Note: You might see some text going by, no worries. If you run a second time make, they are gone.

When down check the whole

```
ls -l

...
-rwxr-xr-x 1 root root 24120 Feb 29 13:03 single_chan_pkt_fwd
...
```

As you can see we have an executable program single_chan_pkt_fwd.
So run it

```
./single_chan_pkt_fwd

server: .address = router.eu.thethings.network; .port = 1700; .enable = 1
Gateway Configuration
  Single-Channel DIY Gate (marc@engrie.be)
  Dragino Single Channel Gateway on RPI
  Latitude=50.80573654
  Longitude=3.95030594
  Altitude=5
Trying to detect module with NSS=6 DIO0=7 Reset=0 Led1=unused
SX1276 detected, starting.
Gateway ID: b8:27:eb:ff:ff:9a:b4:41
Listening at SF7 on 868.300000 Mhz.
---------------------------------
stat update: 2020-02-29 16:58:15 GMT no packet received yet
```

As long as you keep this program running, your gateway is active and usable. When you stop it, not more gateway.

As you see, when the program starts, it dispalys the Gateway ID you need to create the gateway in TTN. Note it down without the collums
Make sure you have your Google maps coordinates handy (see above).
Create now your account, if not yet done, in TTN (see above) and create the gateway (see above).

Once this is all done, wait a few minutes and you should see your gateway connected.

You can also make this to run as a service. Run

```
sudo make install
```

To start and stop the service or get the status of the service

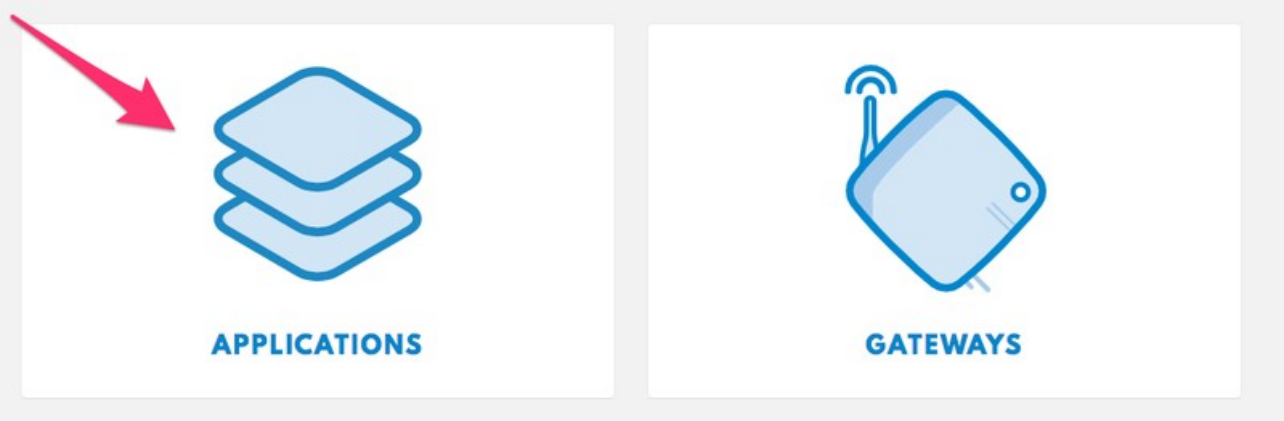
```
systemctl start single_chan_pkt_fwd
systemctl stop single_chan_pkt_fwd
systemctl status single_chan_pkt_fwd
```

To see gateway log in real time

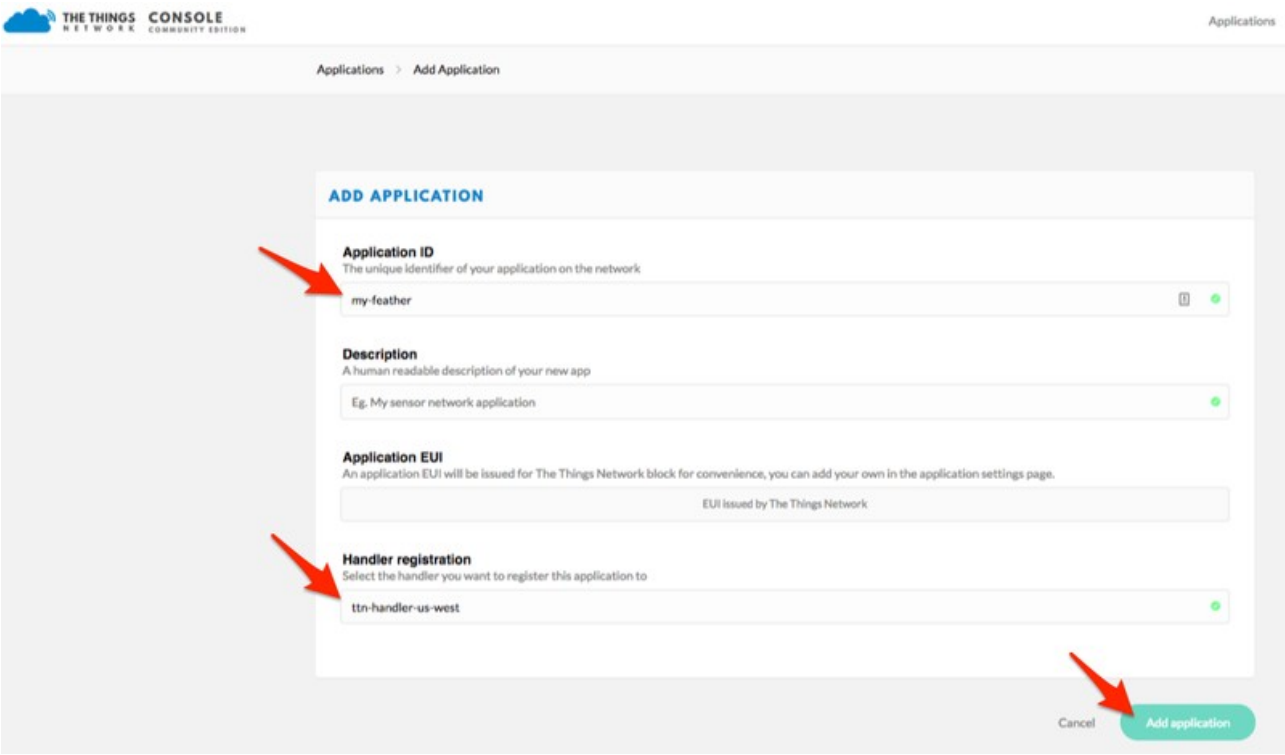
```
journalctl -f -u single_chan_pkt_fwd
```

Making a Raspberry Pi LoRa Node in Python

First setup TTN
Before you can communicate with The Things Network, you'll need to create an application. Log in and open the console. This page is where you can register applications. Click Applications

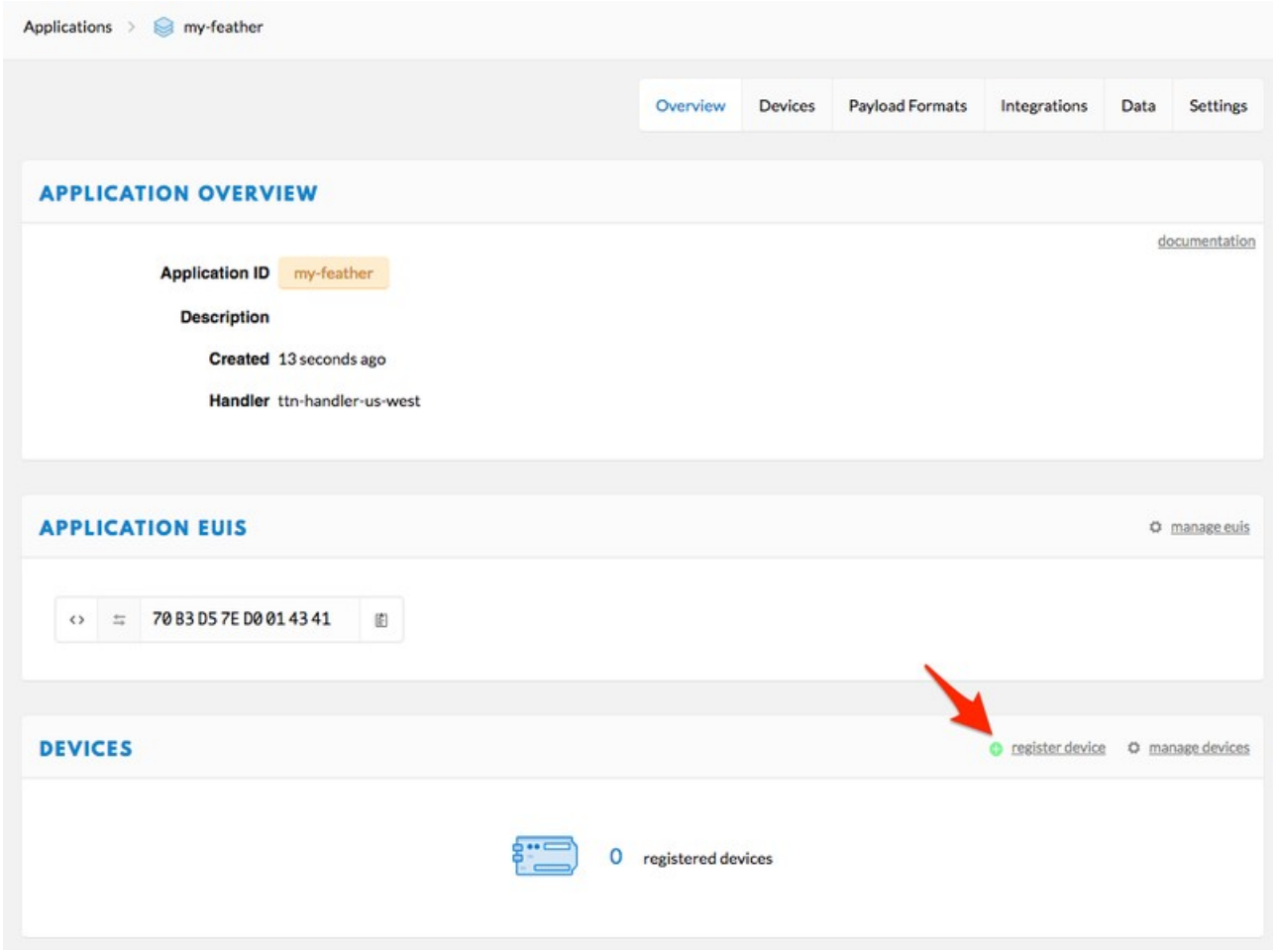


Register an application
Click Add Application.
Fill out an Application ID to identify the application, and a description of what the application is. We set our Handler Registration to match our region, *us-west*. If you're not located in the U.S., TTN provides multiple regions for handler registration.



Once created, you'll be directed to the Application Overview. From here, you can add devices, view data coming into (and out of) the application, add integrations for external services, and more. We'll come back to this section later in the guide.

Register a device
Click Register Device



On the Register Device Page, The Device ID should be a unique string to identify the device.

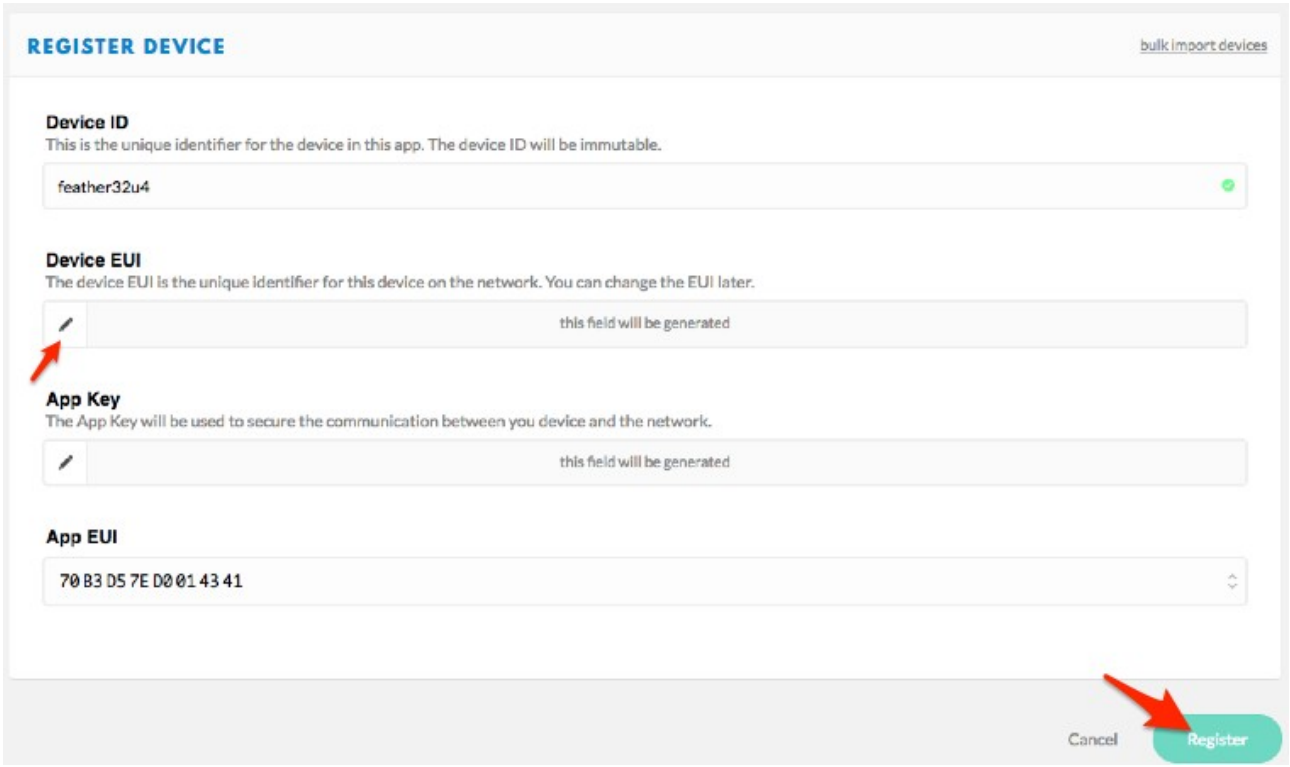
The Device EUI is the unique identifier. You can use the one of the MAC address of your PI and pad the middle of the string with four zeroes.

```
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.43  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::3fa6:c35e:a22b:e7f7  prefixlen 64  scopeid 0x20<link>
        ether b8:27:eb:9a:b4:41  txqueuelen 1000  (Ethernet)
        RX packets 203599  bytes 63940840 (60.9 MiB)
        RX errors 0  dropped 1  overruns 0  frame 0
        TX packets 52814  bytes 9234414 (8.8 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlan0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether b8:27:eb:cf:e1:14  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

So, eg., `b8 27 eb 00 00 9a b4 41`
Lost your key? No worries - you can also click the "mix" icon to switch it to auto-generate.

The App Key will be randomly generated for you by TTN.

Select the App EUI (used to identify the application) from the list.



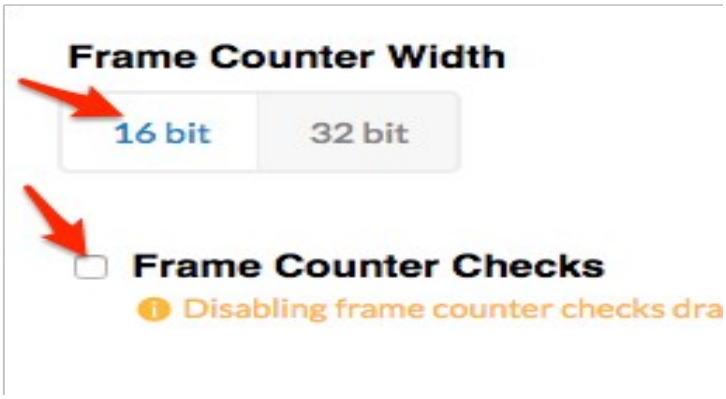
Now Register the device.

Fine tuning
Next, we're going to switch the device settings from Over-the-Air-Activation to Activation-by-Personalization. From the Device Overview, click Settings

On the settings screen, change the Activation Method from OTAA to ABP.

Then, switch the Frame Counter Width from 32bit to 16bit and disable frame counter checks. TTN will display a warning, ignore it, and click Save.

Make sure you have disabled Frame Counter Checks



Note: Why disabling Frame Counter Checks if The Things Network Console doesn't recommend it?
Disabling frame counter checks allows you to transmit data to The Things Network without requiring a match between your device's frame counter and the console's frame counter.
If you're making a project and doing a lot of prototyping/iteration to the code, disabling these checks is *okay* as it'll let you reset the device and not re-register it to the application (it'll also prevent counter overflows).
If you're deploying a project, you'll want to re-activate the frame counter for security purposes. With the frame counter disabled, one could re-transmit the messages sent to TTN using a replay attack.
The Things Network's documentation page has a full explanation about the role of the Frame Counter. (https://www.thethingsnetwork.org/docs/lorawan/security.html#frame-counters)

Making your Raspberry Pi LoRaWAN node
Now that we've set up our Things Network application and device, we're going to move on to installing TinyLoRa onto our Raspberry Pi. To do this, enter the following into your terminal to install the library system-wide:

```
pip3 install adafruit-circuitpython-tinylora
```

Unlike sending data to another device, we're going to be sending data from our Pi to the gateway into TTN network. While we don't have any sensors hooked up to our radio, we'll send the Raspberry Pi's CPU utilization to The Things Network.

Below is an example of using TinyLoRa to send data to The Things Network. Save this as `radio_lorawan.py`.

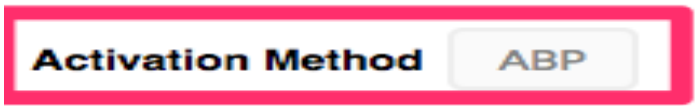
```
nano  radio_lorawan.py
```

```
import time
import subprocess
import busio
import board
from digitalio import DigitalInOut, Direction, Pull

# Import Adafruit TinyLoRa
from adafruit_tinylora.adafruit_tinylora import TTN, TinyLoRa

# TinyLoRa Configuration
spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
cs = DigitalInOut(board.D25)
irq = DigitalInOut(board.D4)
rst = DigitalInOut(board.D17)

# TTN Device Address, 4 Bytes, MSB
devaddr = bytearray([0x00, 0x00, 0x00, 0x00])
# TTN Network Key, 16 Bytes, MSB
nwkey = bytearray([0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00])
# TTN Application Key, 16 Bytess, MSB
app = bytearray([0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00])

# Initialize ThingsNetwork configuration
ttn_config = TTN(devaddr, nwkey, app, country='EU')

# Initialize lora object
# keep radio on channel 0 = 868.1 MHz = same as the single-channel gateway
lora = TinyLoRa(spi, cs, irq, rst, ttn_config, channel=0)
# you can switchchannel during the run of your code
# lora.set_channel(0)
# you might change the datarate
# possible settings are
#  SF7BW125, SF7BW250, SF8BW125, SF9BW125, SF10BW125, SF11BW125, SF12BW125
#lora.set_datarate(lora.SF7BW125)
lora.frame_counter = int(time.perf_counter())

# time to delay periodic packet sends (in seconds)
data_pkt_delay = 30.0

def tobytearray(str):
    bstr=str.encode()
    result = []
    for i in range(0, len(str)):
        result.append(bstr[i])
    # result.reverse()
    return bytearray(result)

def send_pi_data(data):
    # make sure we have a string
    text = str(data)
```

```
        # convert to bytearray
        datapkt = tobytearray(text)
        # Send data packet
        lora.send_data(datapkt, len(datapkt), lora.frame_counter)
        print('Sent Data to TTN : ' + text + " - Counter: " + str(lora.frame_counter))
        lora.frame_counter += 1
        time.sleep(0.5)

    print('RasPi LoRaWAN')
    while True:
        packet = None

        # read the raspberry pi cpu temp
        cmd = "vcgencmd measure_temp | sed -e " + '"s/^temp=//"'
        CPU = subprocess.check_output(cmd, shell = True )
        CPU = CPU[:-5]
        CPU = int(CPU)

        # Display CPU Load
        print('CPU Temp : ' + str(CPU) + '°C')
        # Send Packet
        send_pi_data(str(CPU))

        print('Sleeping 60 sec')
        time.sleep(60)
```

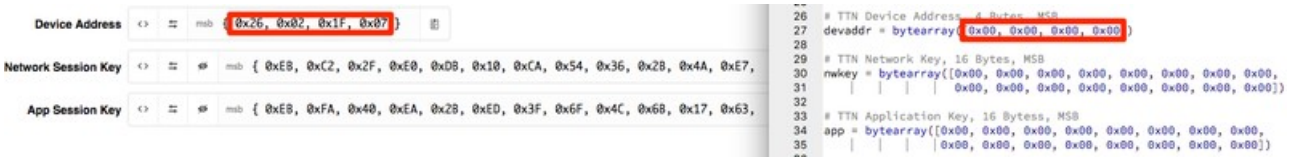
Setting up the code for The Things Network
While we can send data to The Things Network, and our gateway might notice it, it isn't registered to a device yet. To register your device with The Things Network using ABP, you'll need to set three unique identifiers in radio_lorawan.py: the Network Session Key, the Device Address, and the Application Session Key.

Navigate to TTN to the *Device Overview* page for your Raspberry Pi device. Make sure the Activation Method is set to ABP.



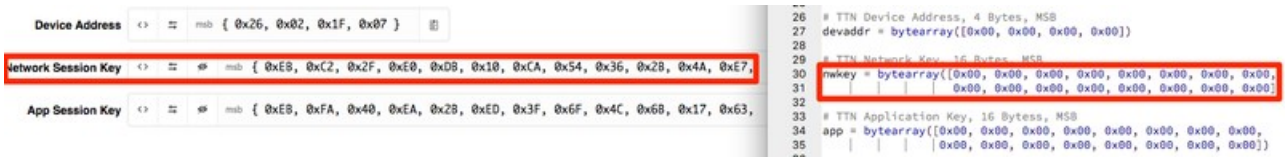
Before adding the unique identifiers to our code, we'll need to first expand them by clicking the <> icon.
These are your keys. We're going to enter them into our code, but we need to be careful - the keys on the Things Network console use parentheses or curly braces *{ }* instead of brackets *[ ].*

First, copy the *Device Address* from the TTN console to the `devaddr` variable in the code.

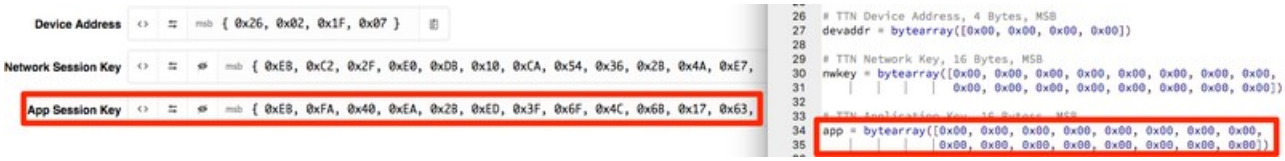


Then, remove the braces *{ }* from the device address.
A device address copied from The Things Network console would look like: `{ 0x26, 0x02, 0x1F, 0x07 }`. In the code, it'd look like: `devaddr = bytearray([0x26, 0x02, 0x1F, 0x07])`.

Then, copy the *Network Session Key* from the TTN console to the `nwkey` variable in the code. Make sure to modify the code to remove the parentheses/curly braces { }.



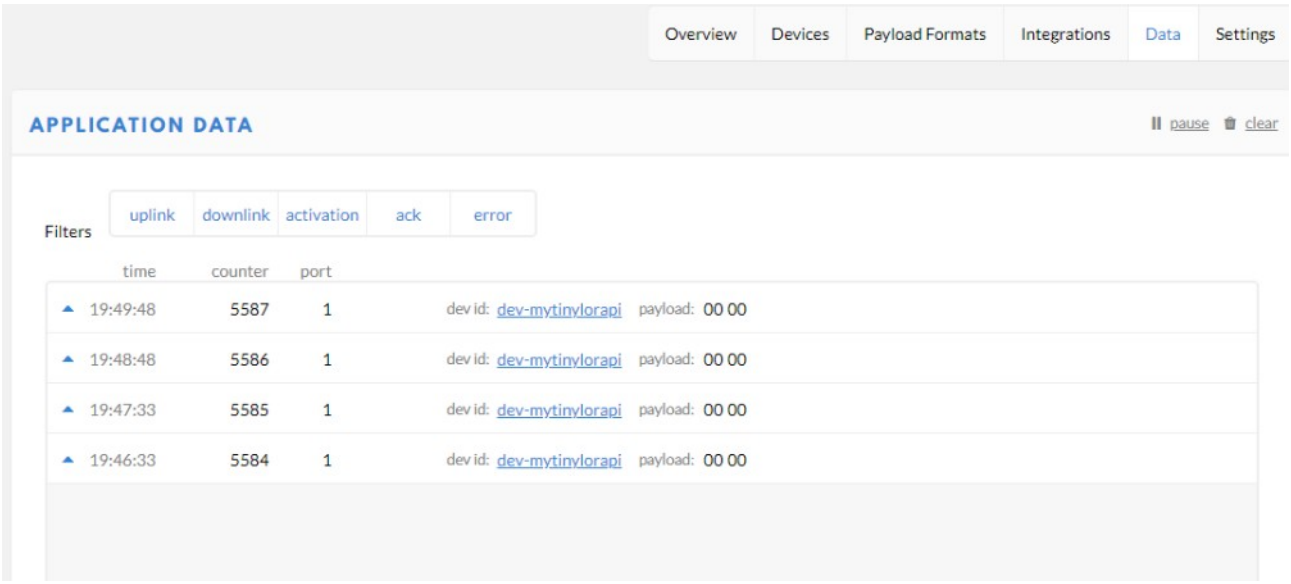
Finally, copy the *Application Session Key* from the TTN console to the `app` variable in the code. Make sure to modify the code to remove the parentheses/curly braces { }.



That's all for now - we're ready to run our code!
Enter the following into the terminal and press enter:

```
python3 radio_lorawan.py
```

Now check in TTN to see if your data is being received. Open the *Applications* panel, select your application and hit *Data*.
Note: you will only see the data being sent during the period you have the panel open. So, also if you do a browser refresh, data records you see in there will be gone. So leave the panel open and you will see your data arriving, with a little delay.



TinyLoRa CircuitPython FAQ

I'm located in a region other than the United States
The country defaults to the **US**, but you can change it to any of the four supported regions:

- **US** - USA, Canada, and South America
- **EU** - Europe
- **AS** - Asia
- **AU** - Australia

If you wanted to switch to an European frequency plan, you'd configure the ttn_config object with the following parameters:

```
ttn_config = TTN(devaddr, nwkey, app, country = 'EU')
```

A subset of the major frequency plans is embedded in the initial release of TinyLoRa for CircuitPython.

If you'd like to use a specific channel...
By default, TinyLoRa for CircuitPython broadcasts on multiple channels, randomly hopping between them. Since most gateways only listen for a transmission on a few channels at a time, this is a way to increase the probability of the gateway *"seeing"* the packet.
You can specify a channel by feeding the LoRa object constructor a channel as a keyword argument. For example, if you'd like to send data on channel 3, the object would be created like the following:

```
lora = TinyLoRa(spi, cs, irq, ttn_config, channel=0)
```

Want to change channels during the event loop, between sending different data packets? We included a set_channel function which will set up the radio to use the specified channel

```
lora.set_channel(0)
```

If you'd like to use a specific data rate...
By default, TinyLoRa uses the datarate SF7BW125. That is, a spreading factor of 7 and a bandwidth of 125kHz. Similar to setting a channel, TinyLoRa can set the datarate given a specific bandwidth and spreading factor. If you'd like to set the transmission datarate to a spreading factor of 10 and a bandwidth of 125kHz, call set_datarate() like the following

```
lora.set_datarate(SF10BW125)
```

TinyLoRa has the following datarates available for use by this method:
SF7BW125, SF7BW250, SF8BW125, SF9BW125, SF10BW125, SF11BW125, SF12BW125

My packet never successfully sends
If you're hitting an error, RuntimeError: Timeout during packet send , it's likely a gateway is not in range, or connected properly. Ensure that there are gateways near you and in-range. If you're a Things Network Gateway operator, check from the Things Network Console if the gateway has been seen recently. If the *Last Seen* value is greater than a few minutes, reset your gateway.

My project has disconnected/been reset and I don't see any data in my application
If you haven't disabled frame counter checks from within your device, you'll encounter this issue when the device is reset/loses power.
However, we can still reset both the node and the frame counter on the application.

To do this, from your Things Network Console, navigate to your device *Applications->Application->Device*

Click the *reset frame counters* button next to the *Frames up* label. This will reset the frame counter of the device (on The Things Network's Console) to zero.
We'll also need to reset the frame counter on our device to zero.

What's the deal with the frame counter? Do I need it in my code?
Disabling frame counter checks allows you to transmit data to The Things Network without requiring a match between your device's frame counter and the console's frame counter.
If you're making a project and doing a lot of prototyping/iteration to the code, disabling these checks is *okay* as it'll let you reset the device and not re-register it to the application (it'll also prevent counter overflows).
If you're deploying a project, you'll want to re-activate the frame counter for security purposes. With the frame counter disabled, one could re-transmit the messages sent to TTN using a replay attack.
The Things Network's documentation page has a full explanation about the role of the Frame Counter. https://www.thethingsnetwork.org/docs/lorawan/security.html#frame-counters

Getting your data out of TTN using a Raspberry Pi

There are several way to get the data sent by your device out of TTN. All methods are listed here https://www.thethingsnetwork.org/docs/applications/options.html

As all my documents are based on Python, so I am going to use a Python piece of code to get the data out of TTN.

Now, you might still find a Python module ttn but this has been discontinued since Dec 2019, so we need to fall back on MQTT. If you wonder what MQTT is, look here http://mqtt.org/faq

You can see also Node-Red + InfluxDB and Grafana combine into a strong 'team' to make the data of your device visible in a nice way.

For the more C-driven people among you, HTTP API is the way to go, I think. Look at this video https://www.youtube.com/watch?v=Uebcq7xmI1M&index=2&list=PLM8eOeiKY7JVwrBYRHxsf9p0VM_dVapXl

I am going to use a Raspberry Pi to run the code but you should be able to run this also an other Linux box or a Windows box with Python3 on it.

Install or update these modules first

```
pip3 install pyjson
pip3 install paho-mqtt
pip3 install pybase64
```

Enter this code

```
nano ttn_mqtt.py
```

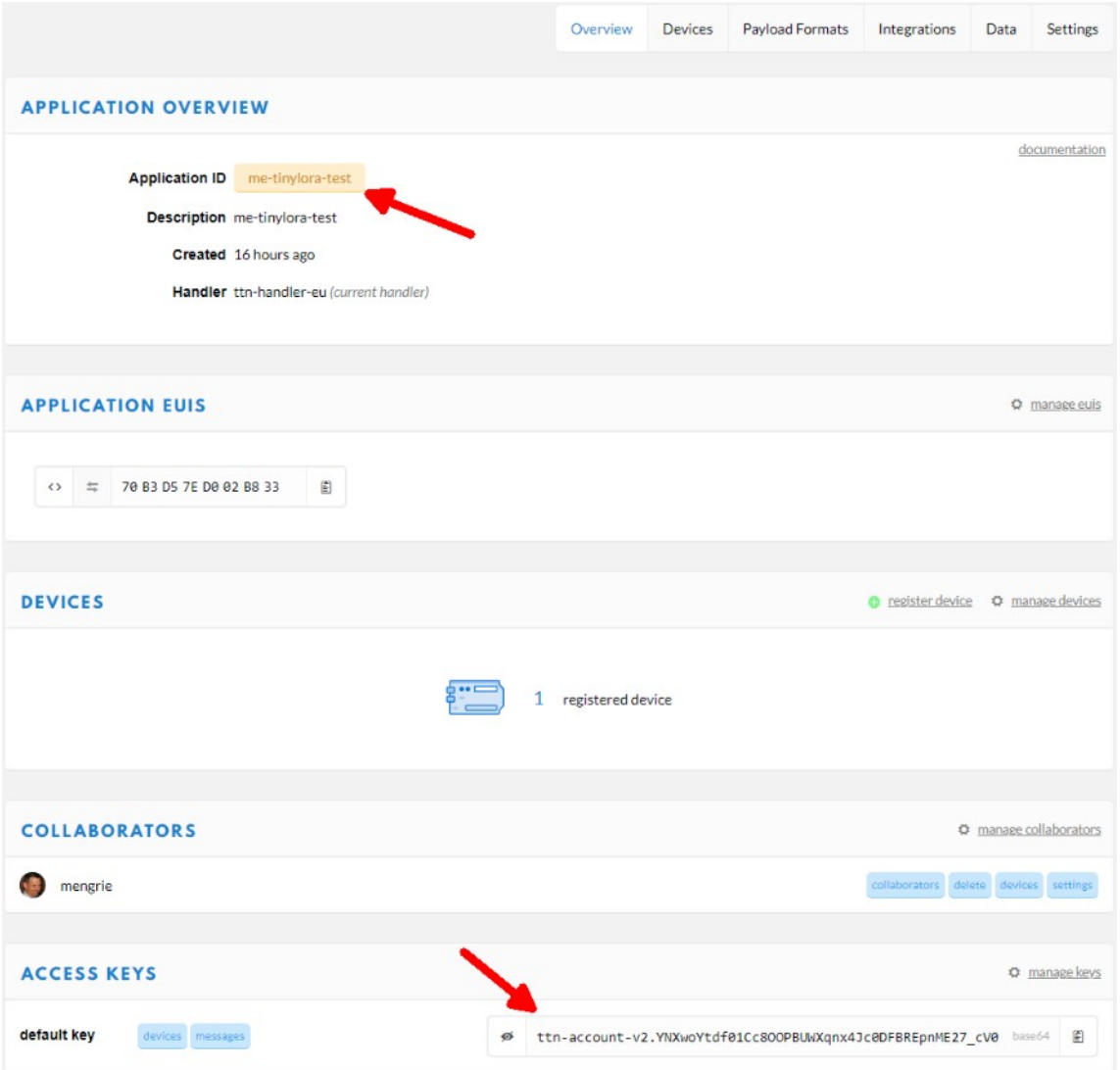
```
import paho.mqtt.client as mqtt
import json
import base64

APPID     = "yourappid"
ACCESSKEY = 'ttn-account-v2.00000000000000000000000000000000000'

#Call back functions

# gives connection message
def on_connect(mqttc, mosq, obj,rc):
    print("Connected with result code:"+str(rc))
    # subscribe for all devices of user
    mqttc.subscribe('+/devices/+/up')

# gives message from device
def on_message(mqttc,obj,msg):
    try:
        # print("payload : " + msg.payload.decode('utf-8'))
        x = json.loads(msg.payload.decode('utf-8'))
        device = x["dev_id"]
        # print(device)
        counter = x["counter"]
        # print(counter)
        payload_raw = x["payload_raw"]
        # convert base64 payload to text
        pl64_bytes = payload_raw.encode('ascii')
        pl_bytes   = base64.b64decode(pl64_bytes)
        data       = pl_bytes.decode('ascii')
        # payload_fields = x["payload_fields"]
        datetime = x["metadata"]["time"]
        print(datetime[:-11] + " : Device = " + device + " - Frame Counter = " + str(counter))
        print("  Payload (Raw) =  " + str(payload_raw) + " - Data = " + data)

    except Exception as e:
        print(e)
        pass
```

```
def on_publish(mosq, obj, mid):
    print("mid: " + str(mid))

def on_subscribe(mosq, obj, mid, granted_qos):
    print("Subscribed: " + str(mid) + " " + str(granted_qos))

def on_log(mqttc,obj,level,buf):
    print("message:" + str(buf))
    print("userdata:" + str(obj))

mqttc= mqtt.Client()
# Assign event callbacks
mqttc.on_connect=on_connect
mqttc.on_message=on_message

mqttc.username_pw_set(APPID, ACCESSKEY)
mqttc.connect("eu.thethings.network",1883,60)

# and listen to server
run = True
while run:
    mqttc.loop()
```

First make sure you are using the right server. Correct if needed

```
mqttc.connect("eu.thethings.network",1883,60)
```

Now go to TTN and open the Application panel. Select the application you want to talk to and go to the overview. Note down and/or copy the Application ID and its Access Key



Copy these into your code.

Application ID → APPID
ACCESS KEYS → ACCESSKEY

Now run the programs

```
python3 ttn_mqtt.py
```

Make sure your Pi that sends its temperature is running its code as well and check in TTN if data is coming in. After a few minutes you should see

```
Connected with result code:0
2020-03-02T11:14:05.566394941Z: Device = dev-mytinylorapi: Frame Counter = 43434
  Payload (Raw) =  Q1BVVGVtcDo0Mw== - Data = CPUTemp:43
2020-03-02T11:15:05.484489413Z: Device = dev-mytinylorapi: Frame Counter = 43435
  Payload (Raw) =  Q1BVVGVtcDo0Mw== - Data = CPUTemp:43
2020-03-02T11:16:05.27135934Z: Device = dev-mytinylorapi: Frame Counter = 43436
  Payload (Raw) =  Q1BVVGVtcDo0Mg== - Data = CPUTemp:42
```

Et voilà, we end-to-end communication over LoRaWAN/TTN.

From here on it is up to you to extend, change, or adapt for your needs.

Getting your data out of TTN with Telegraf

As Telegraf will only retreive numerical data from the json string sent by TTN MQTT, we need to make some modifications to our sender program. We will need numpy for controlling the integers. I have also a DS18B20 connected to the Pi and will read the temp outside as well. I am not going to layout all of the code but the most important parts

```
import numpy as np

…

def ttn_send_data(data):

    # Send data packet
    rfm95.send_data(data, len(data), rfm95.frame_counter)
    framecounter += 1
    rfm95.frame_counter = int(framecounter)
    time.sleep(0.5)


# CPU Temp in integer format
line = os.popen('vcgencmd measure_temp').readline()
tempCPU = int(float(line.replace("temp=","").replace("'C\n","")))

# DS18B20
# check temperature in Celsius in integet format
tempOUT = int(sensorTempOUT.read_temp())

…

u16cpu   = np.uint16(tempCPU)
u16temp  = np.uint16(tempOUT)
data = []
data.append(u16cpu >> 8)
data.append(u16cpu & 0xFF)
data.append(u16temp >> 8)
data.append(u16temp & 0xFF)
badata  = bytearray(data)
ttn_send_data(badata)
```

So, we read the temperatures into Python general integer format. Next we convert them to fixed unsigned integer 16 bit. We shift these into a bytearray and send these out.

But as we saw above, this data is sent out as byte64 encoded into the payload_raw field. A field Telegraf does not take into account as it is not numeric.
The solution to this is the Payload Formats tab in TTN. This allows you to decode, encode, convert data that arrives and do something with it. We are going to use the decoder option and enter this Javascript piece of code

```
function Decoder(bytes, port)
{
  var decoded = {};

  var cpu = (bytes[0] << 8) | bytes[1]; // temperature cpu
  var out = (bytes[2] << 8) | bytes[3]; // temperature outside

  // Decode integer to float
  decoded.tempcpu = cpu;
  decoded.tempout = out;

  return decoded;
}
```

So we unraffle the bytesarray into variables and add them to the dictionary '`decoded`'.

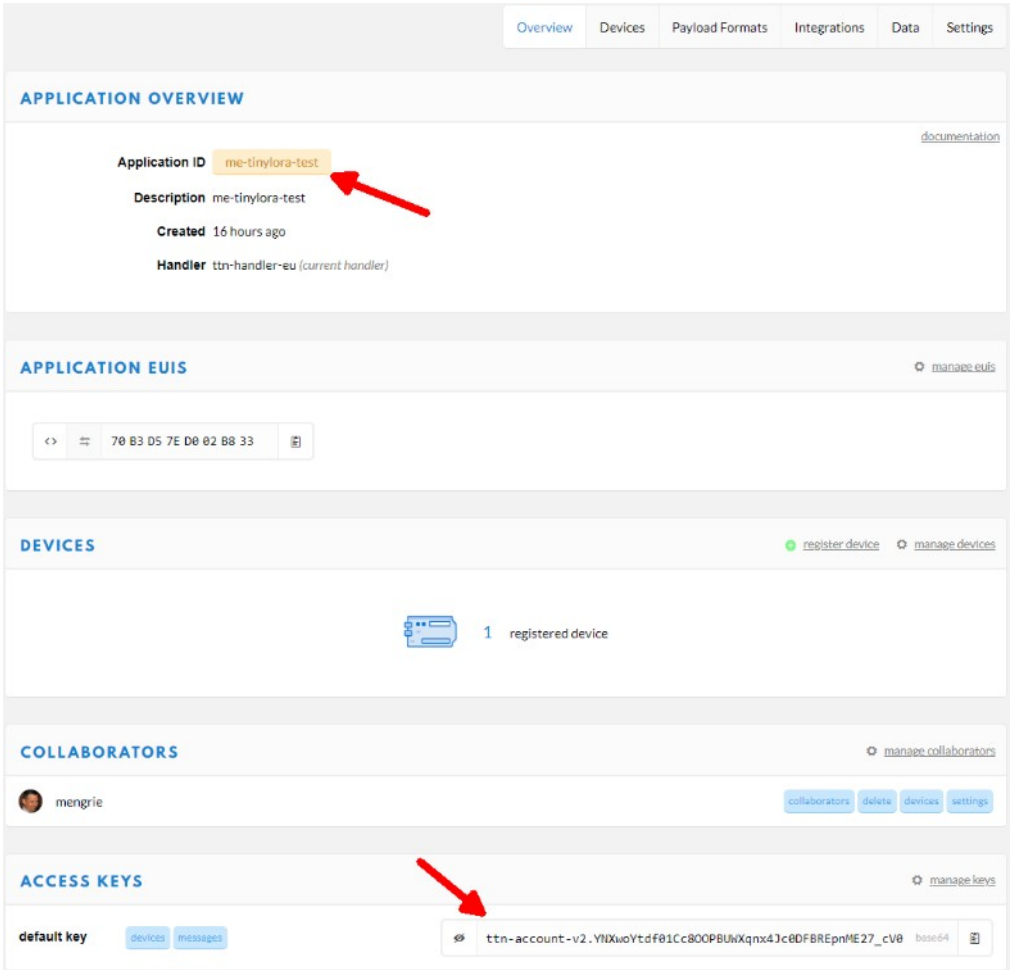
Now if we look at what's coming out of our MQTT broker, we will see:

```
{ "app_id":"my-temp-app",
  "dev_id":"my-device",
  "hardware_serial":"009E9BA30C869232",
  "port":1,
  "counter":28,
  "payload_raw":"CeEVJg6JAQAW7A==",
  "payload_fields":{
      "tempcpu":58,
      "tempout":18
  },
}
```

Which is a proper JSON object, with our data in a usable form! Now for the final bit: getting it all into InfluxDB using Telegraf

```
[[inputs.mqtt_consumer]]
  servers = ["tcp://eu.thethings.network:1883"]
  qos = 0
  connection_timeout = "30s"
  topics = [ "+/devices/+/up" ]
  client_id = "ttn"
  username = "APP_NAME"
  password = "APPKEY"
  data_format = "json"
```

with APP_NAME being the Application ID and password the ACCESS KEY



Then (re)start Telegraf, and, like magic, you should be getting a lot data into InfluxDB into the database you defined into the mqtt_consumer measruement. Besides all numeric data in the TTN message, you will also find the fields `payload_fields_tempcpu` and `payload_fields_tempout` which contain the data you sent out with the Pi.

Appendix: Using GPS module on the Dragino Lora/GPS HAT

The add-on L80 GPS, base on MTK MT3339, is designed for applications that use a GPS connected via the serial ports to the Raspberry Pi such as timing applications or general applications that require GPS information.
This GPS module can calculate and predict orbits automatically using the ephemeris data (up to 3 days) stored in internal flash memory, so the HAT can fix position quickly even at indoor signal levels with low power consumption. With AlwaysLocate™ technology, the GPS can adaptively adjust the on/off time to achieve balance between positioning accuracy and power consumption according to the environmental and motion conditions. The GPS also supports automatic antenna switching function. It can achieve the switching between internal patch antenna and external active antenna. Moreover, it keeps positioning during the switching process.

The GPS module has a built in 15x15x4 patch Antenna, make sure the GPS antenna points to the sky. if the module must be placed in indoor enviroment where GPS antenna is poor, you can use external GPS antenna. The GPS module will auto switch between the patch antenna and external antenna. The RF part of GPS module is sensitive to temperature, please keep them away from heat-emitting circuit.

Getting GPS to work on a Raspberry Pi 3 Model B
Set-up the serial port on your Pi using raspi-config or my document `Raspberry Pi - Part 10 - Activating Serial.pdf`. Make sure to disable the console.

In the recent models of Raspberry Pi (3 and above) the mini-uart is now routed to GPIO14/15 and the PL011 UART is now used for bluetooth communications. The mini-uart doesn't have a separate clock divisor and uses the core clock frequency. Changes in core clock (e.g. through throttling or idle/load frequency changes) will result in arbitrary modification of the effective baud rate.

So first we need to make sure the hardware UART is used for the serial port on the GPIO connector and the mini-uart for Bluetooth, or even better, disable bluetooth all te way.
I refer to my document `Raspberry Pi - Part 10 - Activating Serial.pdf` where I explain how to swap the serial ports.

Check it out using

```
$ ls -l /dev | grep serial
```

If you see

```
lrwxrwxrwx 1 root root           7 Feb 29 12:28 serial0 -> ttyS0
lrwxrwxrwx 1 root root           5 Feb 29 12:28 serial1 -> ttyAMA0
```

you need to swap. You should see

```
lrwxrwxrwx 1 root root           7 Feb 29 12:28 serial0 -> ttyS0
lrwxrwxrwx 1 root root           5 Feb 29 12:28 serial1 -> ttyAMA0
```

Make sure getty is not active on ttyAMA0

```
systemctl list-units –type=service | grep getty

  getty@tty1.service                 loaded active running Getty on tty1
● serial-getty@ttyAMA0.service       loaded failed failed  Serial Getty on ttyAMA0
```

If you just want to do a quick check to see what data is coming out of the GPS, you can enter the following command

```
sudo cat /dev/ttyAMA0

6,26,146,34,14,17,319,,17,15,037,16*7B
$GPVTG,160.32,T,,M,0.00,N,0.00,K,D*3E
0238,E,0.00,160.32,290220,,,D*6D
6,26,146,34,14,17,319,,17,15,037,16*7B
$GPGGA,132418.000,5048.3455,N,00357.0238,E,2,7,1.21,10.1,M,47.3,M,,*61
6,26,146,34,14,17,319,,17,15,037,16*7B
$GPGSA,A,3,19,17,12,15,06,24,25,,,,,,1.45,1.21,0.81*07
1,M,47.3,M,,*61
6,26,146,34,14,17,319,,17,15,037,16*7B
$GPRMC,132422.000,A,5048.3455,N,00357.0238,E,0.00,160.32,290220,,,D*64

$GPVTG,160.32,T,,M,0.00,N,0.00,K,D*3E

$GPGGA,132422.000,5048.3455,N,00357.0238,E,2,7,1.21,10.1,M,47.3,M,,*68

$GPGSA,A,3,19,17,12,15,06,24,25,,,,,,1.45,1.21,0.81*07

$GPGSV,4,1,13,12,82,254,30,24,65,126,38,25,42,249,15,32,36,302,21*70

$GPGSV,4,2,13,19,32,051,25,36,26,146,34,14,17,319,,17,15,037,16*7B

$GPGSV,4,3,13,06,14,086,16,02,10,117,18,10,07,260,16,15,06,173,32*78

$GPGSV,4,4,13,29,01,194,*4D

$GPGLL,5048.3455,N,00357.0238,E,132422.000,A,D*5E

$GPTXT,01,01,02,ANTSTATUS=OK*3B

$GPGSV,4,1,13,12,82,253,30,24,65,126,38,25,42,249,15,32,36,302,21*77

^C
```

Hit CTRL+C to quit

NOT TESTED YET

Install GPSD
You can always just read that raw data, but its much nicer if you can have some Linux software prettify it. We'll try out `gpsd` which is a GPS-handling Daemon (background-helper). To install `gpsd`, make sure your Pi has an Internet connection and run the following commands from the console:

```
sudo apt-get -y install gpsd gpsd-clients python-gps
```

Note: if you're using the Raspbian Jessie or later release you'll need to disable a `systemd` service that `gpsd` installs. This service has `systemd` listen on a local socket and run `gpsd` when clients connect to it, however it will also interfere with other `gpsd` instances that are manually run (like in this guide). You will need to disable the `gpsd systemd` service by running the following commands:

```
sudo systemctl stop gpsd.socket
sudo systemctl disable gpsd.socket
```

Should you ever want to enable the default `gpsd systemd` service you can run these commands to restore it (but remember the rest of the steps in this guide won't work!):

```
sudo systemctl enable gpsd.socket
sudo systemctl start gpsd.socket
```

After disabling the `gpsd systemd` service above you're ready to try running gpsd manually. GPSD needs to be started up, using the following command:

```
sudo gpsd /dev/ttyAMA0 -F /var/run/gpsd.sock
```

Next, check and, if needed, change defaults

```
 sudo nano /etc/default/gpsd
```

change it to look like this

```
# Default settings for gpsd.
# Please do not edit this file directly - use `dpkg-reconfigure gpsd' to
# change the options.
START_DAEMON="true"
GPSD_OPTIONS="-n"
DEVICES="/dev/ttyAMA0"
USBAUTO="false"
GPSD_SOCKET="/var/run/gpsd.sock"
```

Save and reboot

Now GPS doesn’t work indoors with the build-in atenna as it needs a clear view of the sky. So for this place the PI on the window sill. If you have attached an external gain GPS antenna, you should be able to get signal. If not, move the antenna and/or the Pi to a place nearer to a window.

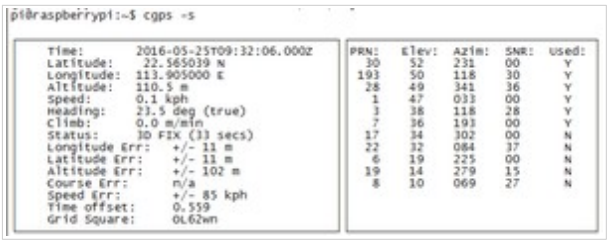
Before being able to use the gpsd tools,

```
sudo killall gpsd
sudo gpsd /dev/ttyAMA0 -F /var/run/gpsd.sock
```

There is a simple GPS client which you can run to test everything is working:

```
cgps -s
```

The -s flag is there to tell the command not to write raw data to the screen as well as the processed data.
It may take a few seconds for data to come through, but you should see a screen like this



If you have any problems and cgps always displays 'NO FIX' under status and then aborts after a few seconds, you may need to restart the gpsd service. You can do that via the following commands:

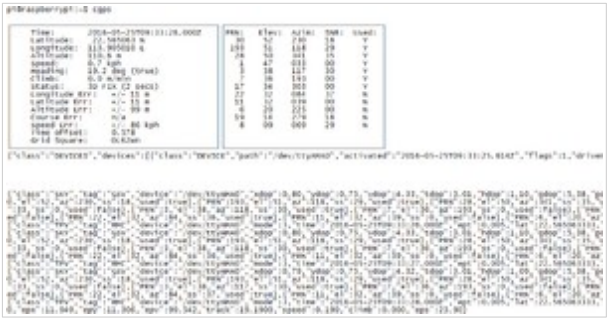
```
sudo killall gpsd
sudo gpsd /dev/ttyAMA0 -F /var/run/gpsd.sock
```

Note: If the GPS receiver is new, or has not been used for some time, it may need several minutes to receive a current almanac. You need 3 GPS satellites for a 2D fix (no height) or 4 satellites for a 3D fix. Once fixed,the LED '3D_FIX' will blink.

You can also try to use the following command:

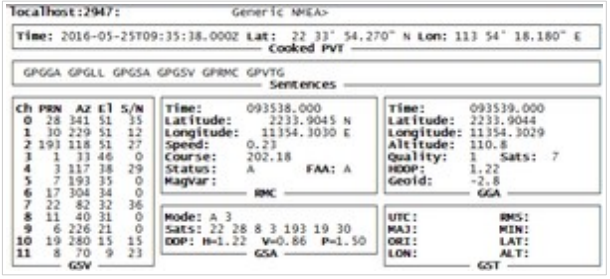
```
cgps
```

cgps and gpspipe should both just show curated data in the same way as your cat command did.



Try running gpsmon to get a live-streaming update of GPS data.

```
gpsmon
```



You can more information about gpsd on http://www.catb.org/gpsd/