# IoT Labs with Pomme-Pi Zero (RISC-V)

### SmartComputerLab

## Table of Contents

# IoT Labs with Pomme-Pi Zero (RISC-V)

**SmartComputerLab**

In this set of **IoT Labs** we are going to study the overall architecture of **IoT infrastructure and** IoT **devices** based on our IoT DevKits with RISC-V architecture.

The introductory section covers the essential features of IoT technologies and the IoT development platform with all supplementary components that are provided by **SmartComputerLab** .

The pedagogical content including codes is available on **`github.com/smartcomputerlab`** server.

## 0.1 Introduction

**IoT Architecture** may be seen as an addition to or an extension of the **Internet Infrastructure**. Basically the **Internet Infrastructure** is built with **communication links** and **routers**.
The **Internet Infrastructure** provides the **communication channels** between the Internet **terminals** such as users-clients and Internet servers. The traditional terminals at the client side are personal computers, laptops, smartphones,.. . The terminals on the server side are processing and data centers (“clouds”).

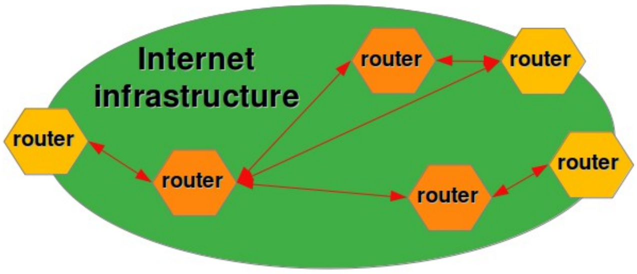



**Fig 0.1** Internet Infrastructure

The Internet Infrastructure provides the routes to send and to receive the **Internet Packets**. Each Internet Packet contains the destination and source address plus the payload (content). The transmission of these packets is controlled by the **Internet Protocol** – **IP**.

The IoT devices are connected (associated) directly or indirectly to the Internet Infrastructure. The IoT devices connected directly to the Internet Infrastructure use IP protocol to carry the data.

Seen from the outside, there are two kinds of entry points to the Internet Infrastructure, **WiFi Access Points** – **AP** and cellular **base stations – BS**. The **IoT servers** in the **cloud** are connected to the Internet Infrastructure via wired/fiber links.

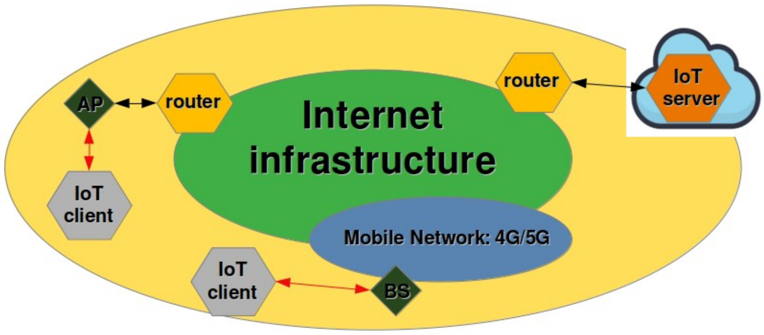



**Fig 0.2** Entry points to Internet Infrastructure (**AP**, **BS**)

The **Things** may be authorized to communicate directly with the access points (**AP**, **BS**). In this case the data from/to sensors/actuators is sent in **IP protocol** packets. We can call these Things **IP-Things**. Another kind of **remote Things** , than we characterize as **NON-IP Things** may communicate with the Internet Infrastructure via the **IoT gateways**. These gateways are devices that combine the IP-based links with **Long Range** radio links such as **LoRa**. The data sent over the LoRa links is simply relayed and sent in IP packets over the links implemented with WiFi or cellular radio. **LoRa is the radio technology** specifically designed for the communication with **IoT terminals**.

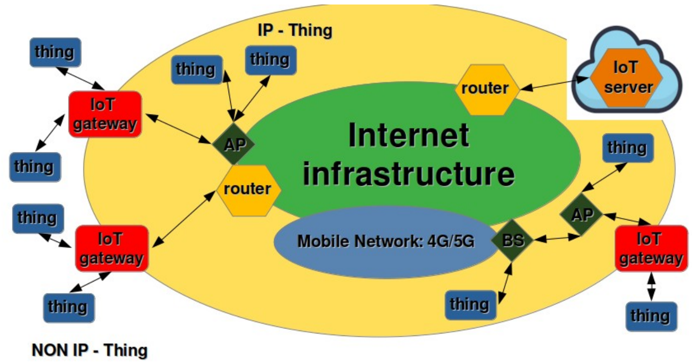



**Fig 0.3** Communication links with IoT devices: **IP Things** and **NON-IP Things**

## 0.2 IoT Devices (IoT cores)

The core part of the IoT devices or Things, combine several kinds of electronic circuits. The **front-end** part of the core has to provide a number of i**nterconnection buses** to accommodate the sensors and the actuators. The central part (**micro-controller**) provides the processing capacity to calculate and coordinate different processing and communication tasks. Finally the **back-end** part of the core must integrate at least one type of **communication modem** such as **BT**/**BLE**, **WiFi**, cellular **4G**/**5G** and/or **LoRa**. Only **BLE** and **LoRa** have the capacity to operate in very **low power** consumption mode.

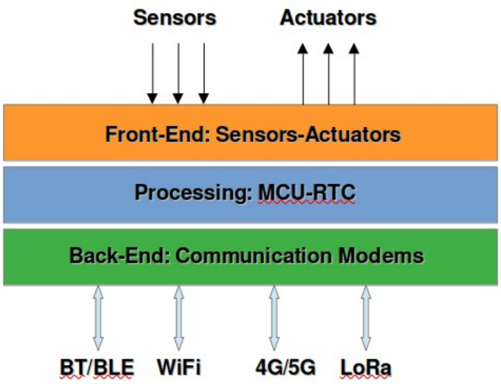



**Fig 0.4** The **SoC** core of an **IoT device**

Modern IoT devices integrate all these circuits in one chip – **System on Chip or SoC**. One of the most popular IoT SoCs is **ESP32**.
ESP32 is a series of **low-cost**, **low-power SoC** with **Wi-Fi** and dual-mode **Bluetooth**. The ESP32 series employs either a Tensilica **Xtensa LX6**/**LX7** dual-core microprocessor(s) or a single-core **RISC-V** microprocessor and includes built-in low power processing unit.
ESP32 is created and developed by **Espressif Systems**, a Shanghai-based Chinese company, and is manufactured by **TSMC** using their **40 nm** process

## 0.3 The processor : RISC-V

**RISC-V** is an **open standard instruction set architecture** (**ISA**) based on established **reduced instruction set computer** (**RISC**) principles. **RISC-V** is descendant of **RISC-I to RISC-IV family developed at Berkeley**. **Unlike** most other **ISA** designs, the **RISC-V ISA** is provided under **open source licenses** that do not require fees to use.

A number of companies including Espressif are offering or have announced **RISC-V hardware**, open source operating systems with RISC-V support are available and the instruction set is supported in several popular software **toolchains**.

Notable features of the **RISC-V ISA** include **a load–store architecture**, bit patterns to simplify the multiplexers in a CPU, **IEEE 754** floating-point, a design that is architecturally neutral, and placing most-significant bits at a fixed location to **speed sign extension**.

The instruction set is designed for a wide range of uses. The base instruction set has a fixed length of **32-bit** naturally aligned instructions, and the **ISA supports variable length extensions** where each instruction could be an any number of 16-bit parcels in length. Subsets support **small embedded systems, personal computers, supercomputers with vector processors**, and **warehouse-scale 19 inch rack-mounted parallel computers**.

The instruction set specification defines **32-bit and 64-bit address space** variants. The specification includes a description of **128-bit flat address** space variant, as an extrapolation of 32 and 64 bit variants, but the 128-bit ISA remains "not frozen" intentionally, because there is yet so little practical experience with such large memory systems.

As of June 2019, version 2.2 of the user-space ISA and version 1.11 of the privileged **ISA** are **frozen,** permitting software and hardware development to proceed.

The user-space ISA, now renamed the **Unprivileged ISA**, was updated, ratified and frozen as version 20191213. A debug specification is available as a draft, version 0.13.2.

## 0.4 ESP32-C3 SoC and RISC-V

ESP32-C3 SoC is build around 32-bit **RISC-V** processor.

ESP32-C3 integrates a 32-bit core RISC-V **RV32IMC** micro-controller with a maximum clock speed of 160 MHz. **RV32IMC** means **base integer** (**I**), **multiplication/division** (**M**) and **compressed** (**C**) standard extensions.

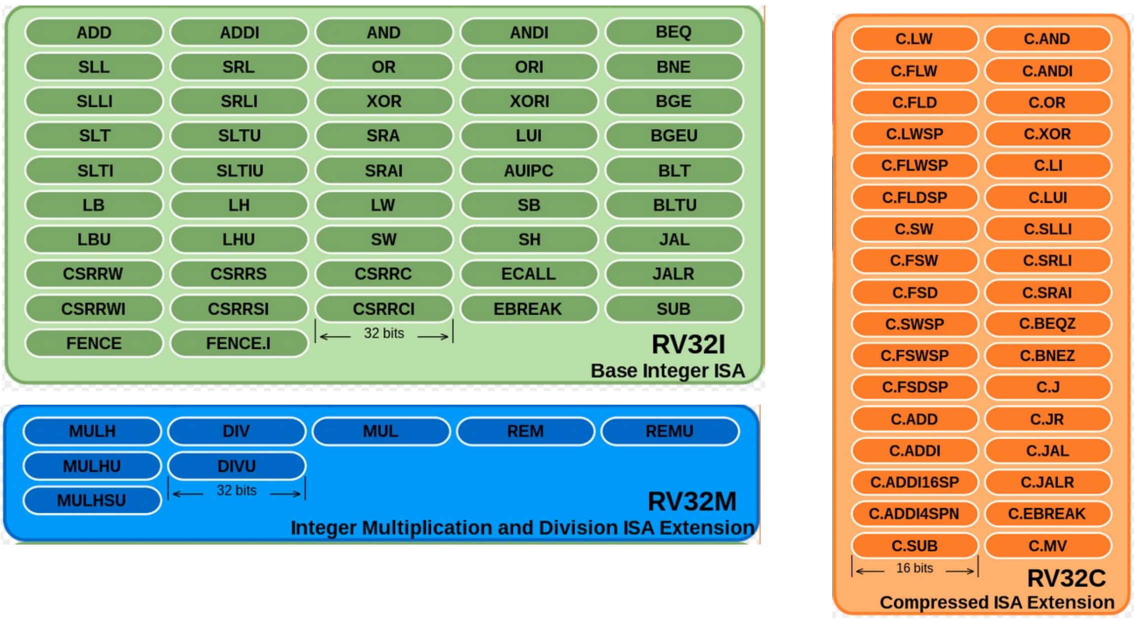



**Fig.05** ESP32-C3 **RV32IMC** **I**nstruction **S**et **A**rchitecture (**ISA**)

**ESP32-C3 SoC** with 22 configurable GPIOs, 400 KB of internal RAM and **low-power-mode** support can facilitate many different use-cases involving connected devices. The SoC comes in multiple variants with integrated and external flash availability. The high-temperature support makes it ideal for industrial and lighting use-cases.

Wi-Fi and **Bluetooth 5** (**BLE**) with long-range (**LR**) support help building devices with great coverage and improved usability. ESP32-C3 continues to support Bluetooth LE SIG Mesh and Espressif Wi-Fi Mesh.
A complete Wi-Fi subsystem that complies with **IEEE 802.11b/g/n** protocol and supports Station mode (**STA**) , **SoftAP** mode, **SoftAP + STA** mode, and **promiscuous** mode.

A Bluetooth LE subsystem that supports features of Bluetooth 5 and Bluetooth mesh State-of-the-art power and RF performance

32-bit RISC-V single-core processor with a four-stage pipeline that operates at up to 160 MHz 400 KB of SRAM (16 KB for cache) and 384 KB of ROM on the chip, and SPI, Dual SPI, Quad SPI, and QPI interfaces that allow connection to external flash

### 0.4.1 Reliable security features

- Cryptographic hardware accelerators that support **AES-128/256**, **Hash**, **RSA**, **HMAC**, digital signature and secure boot
- Random number generator
- Permission control on accessing internal memory, external memory, and peripherals
- External memory encryption and decryption

**Secure Boot**: ESP32-C3 implements the standard RSA-3072-based authentication scheme to ensure that only trusted applications can be used on the platform. This feature protects from executing a malicious application programmed in the flash. We understand that secure boot needs to be efficient, so that instant-on devices (such as light bulbs) can take advantage of this feature. ESP32-C3's secure boot implementation adds less than 100ms overhead in the boot process.

**Flash Encryption**: ESP32-C3 uses the AES-128-XTS-based flash encryption scheme, whereby the application as well as the configuration data can remain encrypted in the flash. The flash controller supports the execution of encrypted application firmware. Not only does this provide the necessary protection for sensitive data stored in the flash, but it also protects from runtime firmware changes that constitute time-of-check-time-of-use attacks.

**Digital Signature and HMAC Peripheral**: ESP32-C3 has a digital signature peripheral that can generate digital signatures, using a private-key that is protected from firmware access. Similarly, the HMAC peripheral can generate a cryptographic digest with a secret that is protected from firmware access. Most of the IoT cloud services use the X.509-certificate-based authentication, and the digital signature peripheral protects the device's private key that defines the device's identity. This provides a strong protection for the device's identity even in case of software vulnerability exploits.

**World Controller:** ESP32-C3 has a new peripheral called world controller. This provides two execution environments fully isolated from each other. Depending on the configuration, this can be used to implement a Trusted Execution Environment (TEE) or a privilege separation scheme. If the application firmware has a task that deals with sensitive security data (such as the DRM service), it can take advantage of the world controller and isolate the execution.

### 0.2.2 Bluetooth 5 (LE) with Long-Range Support

Typically, connected devices use Wi-Fi connectivity to connect to cloud services. However, Wi-Fi-only devices pose some difficulty to the network configuration of the devices, as these devices fail to provide reliable configuration feedback to the provisioner, while at the same time iOS and Android provisioners have additional complexity when connecting to the network. The availability of Bluetooth LE radio in the device makes the provisioning easy. Also, Bluetooth LE provides easy discovery and control in the local environment.
Previous versions of the Bluetooth LE protocol had a smaller range, and that made it not very suitable a protocol for local control in large spaces, e.g. big homes. ESP32-C3 adds support for the Bluetooth 5 (LE) protocol, with coded PHY and extended advertisement features, while it also provides data redundancy to the packets, thus improving the range (typically 100 meters). Furthermore, it supports the Bluetooth LE Mesh protocol. This makes it a strong candidate for controlling devices in a local network, and for communicating with other Bluetooth 5 (LE) sensor devices directly.

### 0.2.3 Memory

With a large variety in the use-cases and their memory requirements, it is tricky to determine the most suitable memory size for the SoC. However, in our experience, it is important to support use-cases with one or, sometimes, two TLS connections to the cloud, which are Bluetooth-LE-active all the time, while also supporting a reasonable application headroom on top of that. ESP32-C3's **400 KB of SRAM** can meet these requirements, while still keeping the chip's cost within the budget target. Also, ESP32-C3 has **dynamic partitioning** for the instruction (**IRAM**) and data (**DRAM**) memory. So, the usable memory is effectively maximized. It is also important to note here that we have optimized the Bluetooth subsystem's memory requirements, in comparison with ESP32.

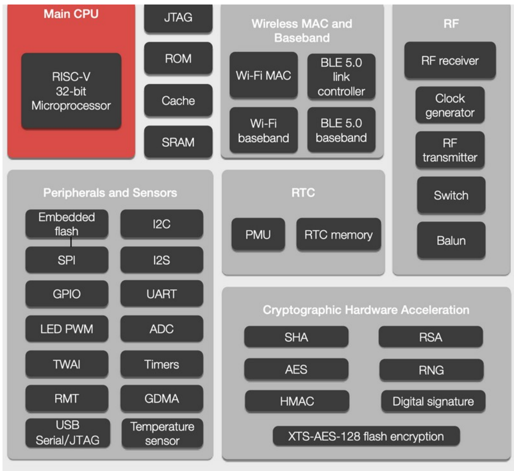



**Fig 0.5 Block scheme** of the internal architecture of **ESP32-C3** SoC with **RISC-V** microprocessor

The implementation of **Pomme-Pi Zero** within the typical **Pi Zero board** requires the use of small MCU component. In our case we use **ESP32-C3 CORE** board.

# 0.3 ESP32-C3 CORE board

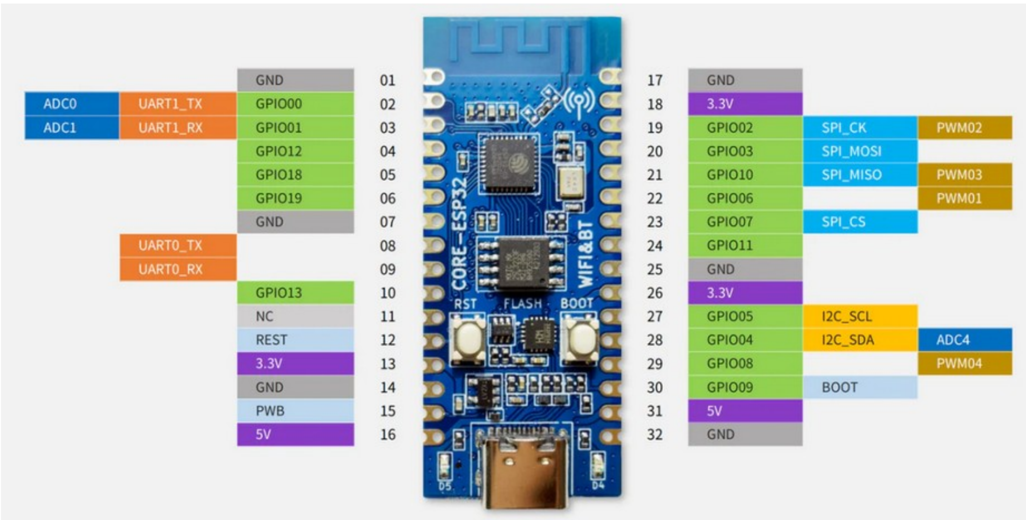



**Figure 0.5 ESP32C3 CORE** board and its pinout

As we can see in the figure above, the board exposes 2x13 pins. These pins carry the **I2C** (**SDA**-4, **SCL**-5), **UART** (**RX1**-1,**TX1**-0), **SPI** (**SCK**-2,**MISO**-10,**MOSI**-3) buses, plus control signals (**NSS**- 7,**RST**-6,**INT**-11,..), integrated **LED** (12,13)

## 0.4 Pomme-Pi ZERO CORE - IoT development platform

Efficient integration of the selected ESP32-C3 Core board into IoT architectures requires the use of a development platform such as **IoT DevKit** provided by **SmartComputerLab**.

**Pomme-Pi CORE** is composed of a base board and a large number of extension boards designed for the efficient use of connection buses and all types of sensors and actuators.

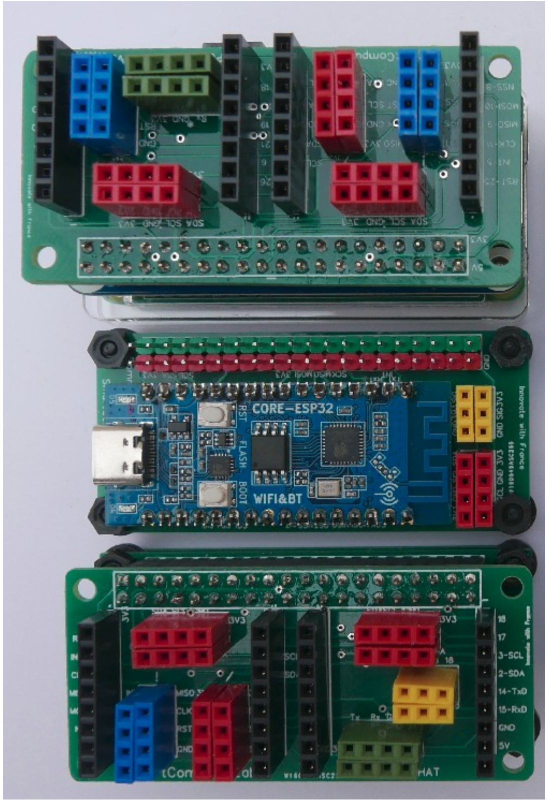



**Figure 0.6 Pomme-Pi CORE** base board with the expansion cards (RPI Zero compatible HAT) :

- **red-I2C**,
- **blue-SPI**,
- **green-UART**,
- **yellow – simple signal** .

The base card can directly accommodate two types of sensors or communication modems. To connect a more complete set of sensors/modems/displays, **expansion card needs to be used**.

Additional expansion boards may be connected to the expansion HAT.

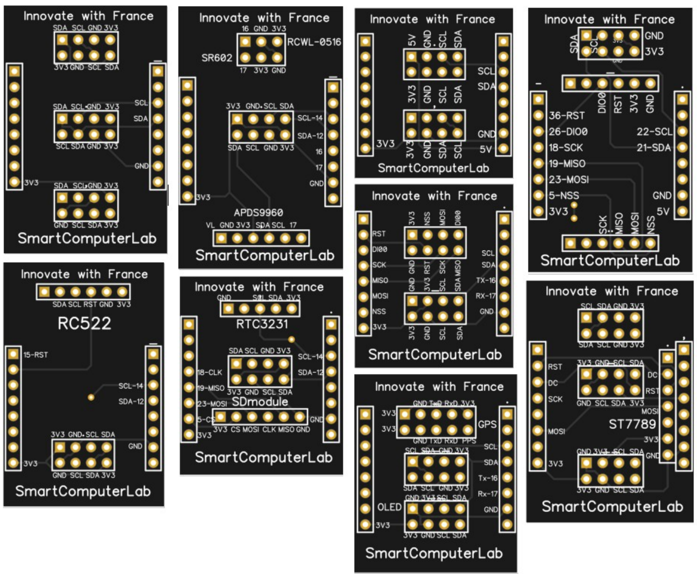



**Figure 0.7** Expansion cards for various IoT components: sensors, displays, modems, ..

# 0.5 Software – Thonny IDE

## 0.5.1 Installing Thonny IDE - `thonny.org`

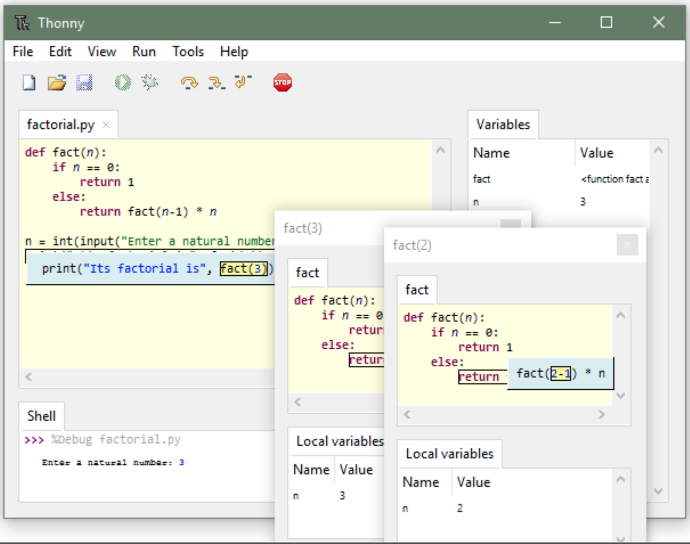
**Thonny** is an **open source IDE** which is used to write and upload **MicroPython** programs to different development boards such as ESP32 and ESP8266. It is an extremely interactive and easy-to-learn IDE, as it is known as the beginner-friendly IDE for new programmers.

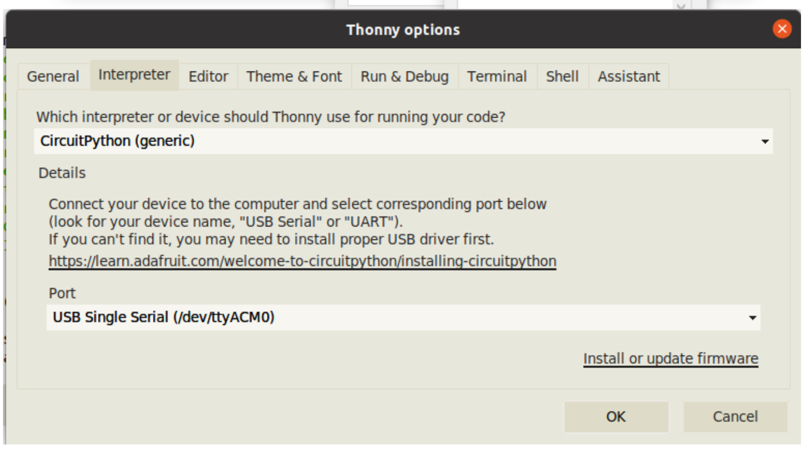
With the help of Thonny, it becomes very easy to code in MicroPython as it has an inbuilt debugger which helps to find any error in the program by debugging the script line by line.

Here is the installation page of the Thonny IDE. You follow the instructions.


The installation of Thonny IDE includes the installation of Python 3.7 (built in).

After this installation, we are therefore ready to program in Python with the Python version 3 interpreter installed on your PC.




The above figure shows the **selection of interpreter** running on your board (ESP32-C3).

## 0.4.2 Preparing the ESP32-C3 Core board

**Thonny IDE** allows you to install the **MicroPython interpreter** corresponding to our card (ESP32-C3).
Go to **Tools→Options** then **Interpreter**.

```
pip3 install esptool
```

```
esptool.py --chip esp32c3 --port /dev/ttyACM0 erase_flash
esptool.py --chip esp32c3 --port /dev/ttyACM0 --baud 460800 write_flash -z 0x0
firmware.esp32c3.all.221023.bin
```

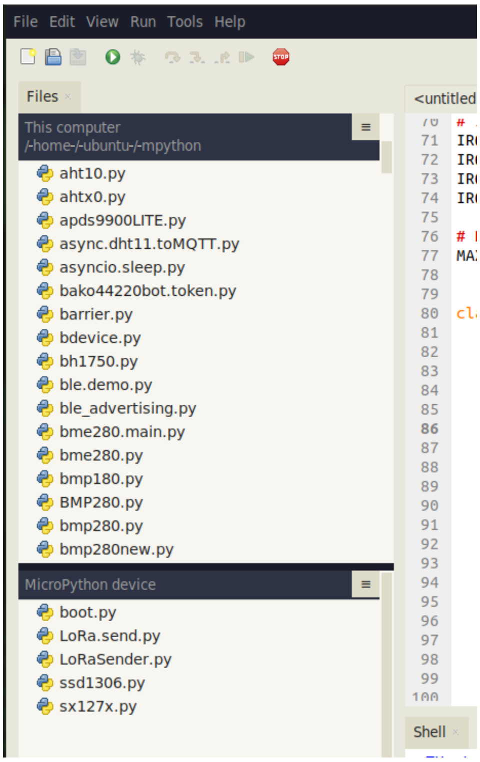
**Attention:**
The name of the firmware may be different depending on the generation date.

After loading the **MicroPython** interpreter on the **ESP32-C3** board we can connect our board with the USB cable to our PC and launch Thonny IDE again.
First let us verify the presence of different modules in our firmware via **`help('modules')`** command.

```
Type "help()" for more information.
>>> help('modules')
CCS811          htu21d          ssd1306         umqtt/robust
VL53L0X         inisetup        st7789          umqtt/simple
__main__        math            sx127x          uos
_boot           max30100        tsl2561         uplatform
_onewire        max30102        uSGP30          upysh
_thread         max30120        uarray          urandom
_uasyncio       max44009        uasyncio/__init__ ure
_webrepl        max7219         uasyncio/core   urequests
apa106          mcp9808         uasyncio/event  uselect
bh1750          mfrc522         uasyncio/funcs  usocket
bmp180          micropyGPS      uasyncio/lock   ussl
bmp280          micropython     uasyncio/stream ustruct
btree           mip             ubinascii       usys
builtins        mpu6050         ubluetooth      utime
cmath           neopixel        ucollections    utimeq
dht             network         ucryptolib      uwebsocket
ds18x20         nrf24101        uctypes         uzlib
esp             ntptime         uerrno          vl5311x
esp32           onewire         uhashlib        webrepl
flashbdev       paj7620         uheapq          webrepl_setup
framebuf        sgp30           uio
gc              sht21           ujson
hcsr04          sht31           umachine
Plus any modules on the filesystem
>>>
```

Let's see the available files, **View→Files.**

Note that a newly "flashed" map only contains the **`boot.py`** file.



We are going to add our **`example.1.py`** program to it. It is possible to save the program on the PC (**This computer**) or on the card (**MicroPython device**).


```
import machine                  # the main library for MCU
from machine import Pin
from time import sleep
led=Pin(13,Pin.OUT)              # Pin LED number is 12 or 13
while True:
    led.value(not led.value())  # the value to display is complemented
    sleep(1.1)                  # waiting time in seconds
```

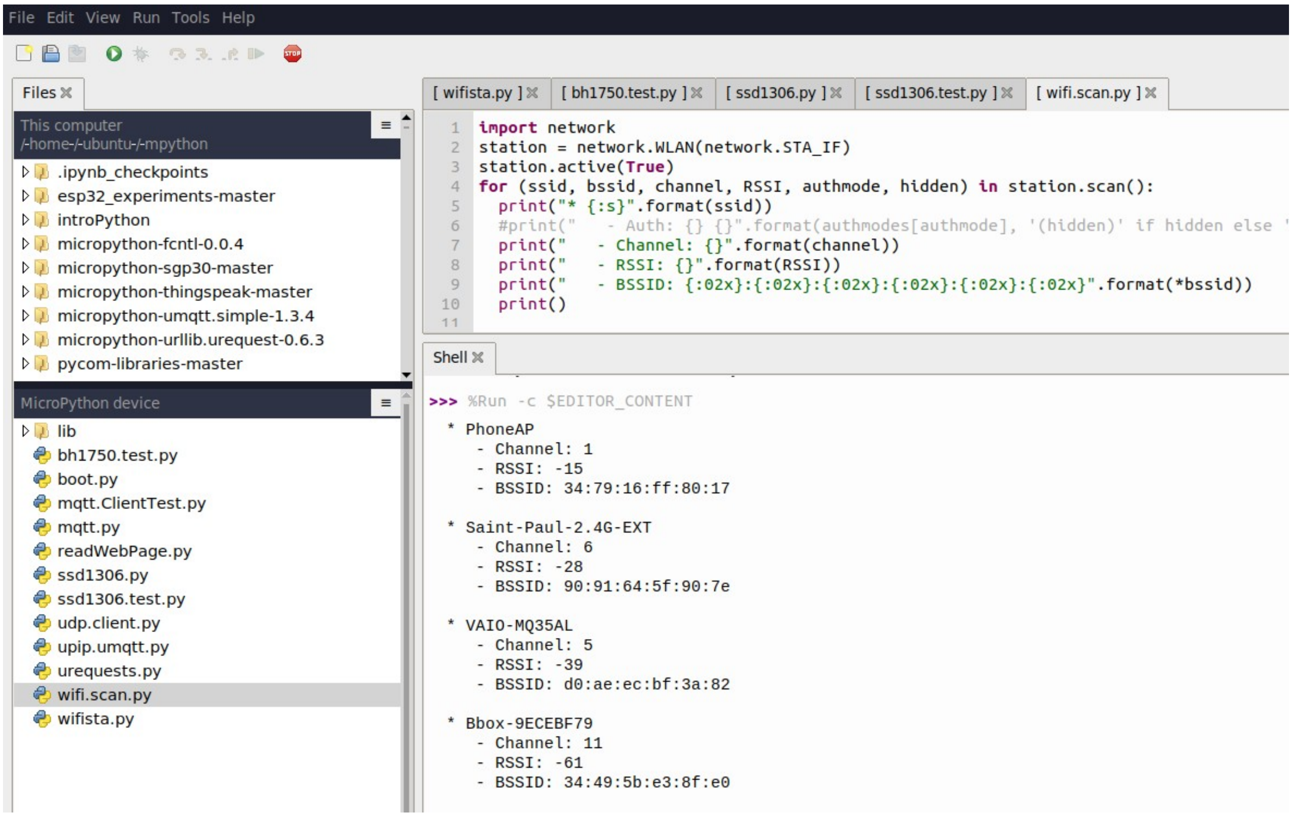
Now we can start the "interpretation" execution of our program by pressing the **green arrow**.

### 0.4.4 First example – `wifi.scan.py`

In our first example we will run **Thonny** and edit a simple **`wifi.scan.py`** program.
The following figure shows the working windows in the Thonny IDE. On the left at the top we have the contents of the **`/home/bako/monPython`** directory on our PC. At the bottom left we have the list of programs recorded on the card.
When the board boots first time there is only **`boot.py`**; other programs are loaded later.
In the main window we display the content of the last edited program, here **`wifi.scan.py`**

The code is:

```
import network
station = network.WLAN(network.STA_IF)
station.active(True)
for (ssid, bssid, channel, RSSI, authmode, hidden) in station.scan():
  print("* {:s}".format(ssid))
  #print("   - Auth: {} {}".format(authmodes[authmode], '(hidden)' if hidden else ''))
  print("   - Channel: {}".format(channel))
  print("   - RSSI: {}".format(RSSI))
  print("   - BSSID: {:02x}:{:02x}:{:02x}:{:02x}:{:02x}:{:02x}".format(*bssid))
  print()
```




**To do:**

1. Launch Thonny IDE, edit the program and save it to the card
2. Modify the program in order to execute it 5 times.

# Lab 1

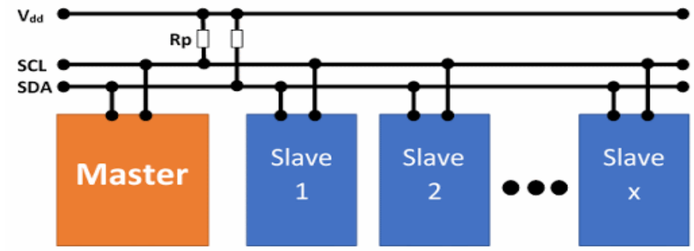
# Data display and sensor reading (i2c)

## 1.0 Introduction

In this lab we will experiment with displaying on an **OLED screen** and capturing physical data such as **temperature**, **humidity** and **brightness**.

The communication between the IoT SoC and these devices is done by sending bytes representing **addresses**, **commands** and **data** over the **I2C bus**.

**I2C bus consists of 2 lines** (signals or wires); **SCL-5** which carries the **CLock** signal and **SDA-4** which carries the information (**D**ata, **A**ddress).



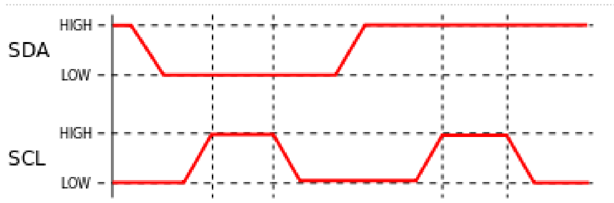




**Fig 1.1 I2C** bus configuration and logic levels on **SDA** - Data/Address and **SCL** – Clock lines

Finding **I2C** connected devices - scan code:

```
import machine
i2c = machine.I2C(scl=machine.Pin(5), sda=machine.Pin(4))
print('Scan i2c bus...')
devices = i2c.scan()
if len(devices) == 0:
  print("No i2c device !")
else:
  print('i2c devices found:',len(devices))
  for device in devices:
    print("Decimal address: ",device," | Hexa address: ",hex(device))
```

```
%Run -c $EDITOR_CONTENT
Warning: I2C(-1, ...) is deprecated, use SoftI2C(...) instead
Scan i2c bus...
i2c devices found: 2
Decimal address:  60  | Hexa address:  0x3c
Decimal address:  64  | Hexa address:  0x40
```

## 1.1 First example – data display on OLED screen (I2C)

In this exercise we will simply display a title and 2 numerical values on the OLED screen added to your **Pomme-Pi Core DevKit**.

**Before starting the execution** of the following program you must add the **SSD1306 display driver** to your files on the device. This driver is prepared in the module file called **`ssd1306.py`**.

First **edit** and **test** the following code:

```
import machine, ssd1306
from machine import Pin, SoftI2C
import esp32
print(machine.freq())
i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
oled.fill(0)
oled.text("SmartComputerLab", 0, 0)
oled.text("RISC-V mPython", 0, 16)
oled.text("Pomme-Pi Core", 0, 32)
oled.text("WiFi/BLE/LoRa", 0, 48)
oled.show()
```

Then **modify** the code to display two variables::

```
import machine, ssd1306
from machine import Pin, SoftI2C
import time

def disp(p1,p2):
    i2c = SoftI2C(scl=Pin((), sda=Pin(4), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab",0,0)   # colonne 0 et ligne 0
    oled.text("max: 16 car/line",0,16)  # colonne 0 et ligne 16
    oled.text(p1,0,32)
    oled.text(p2,0,48)
    oled.show()

d1=0
d2=0
c=0
while c<10:
    disp(str(d1),str(d2))
    c+=1
    d1+=2
    d2+=3
    time.sleep(2)
```

And save it to the directory on your PC and save it in the device.

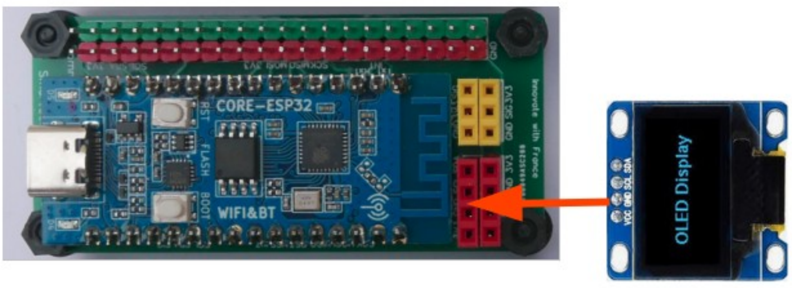
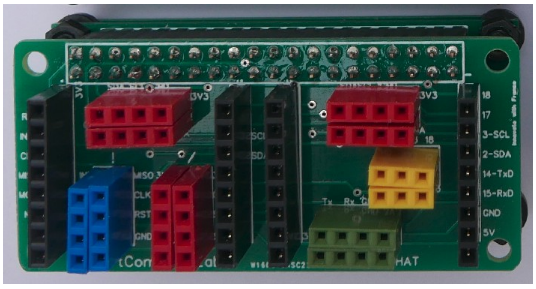
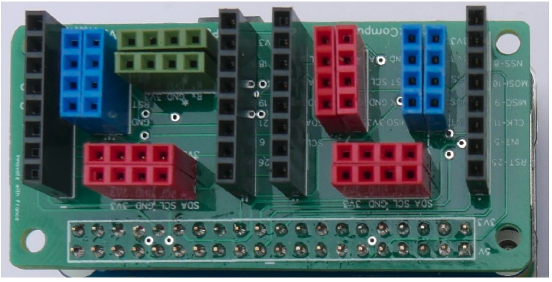



**Fig 1.2** The **Thonny IDE** for storing/flashing a **MicroPython** program. And the configuration of the **Pomme-Pi zero board** board with an OLED screen (**ssd1306**).'

**Pay attention** to the **pinout** of the **I2C** bus connectors - **SDA**,**SCL**,**GND**, and **3V3** on the board.

You can also use one of the expansion boards(HAT) and connect the SSD1306 oled to the corresponding I2C slot.

### To do :

Study the code:

The **`import`** lines ..

```
import machine, ssd1306
from machine import Pin, SoftI2C
import time
```

The function:

```
def disp(p1,p2):
    i2c = SoftI2C(scl=Pin(14), sda=Pin(12), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
```

The initialization of the I2C bus, then the instantiation of the OLED – **`SSD1306_I2C`** driver on the I2C bus. The **`SSD1306_I2C`** **class** is available in the **`ssd1306.py`** file

The **`while`** loop:

```
while c<10:
```

**Add a third row** of data with variable **`d3`**.

## 1.2 Second example – sensor reading (T/H): SHT21 (I2C)

In our second example we are going to capture the **temperature** and **humidity** values on an **SHT21** type sensor. The sensor is connected to an I2C bus (like our OLED screen).

On this bus, over the **SDA line**, the processor sends the **address of the sensor to wake it up**. On the line **SCL** (**S**-signal, **CL**-CLock) the synchronization signal is sent to synchronize the binary values transmitted on the **SDA** line (**S**-signal, **D**-data, **A**-address).

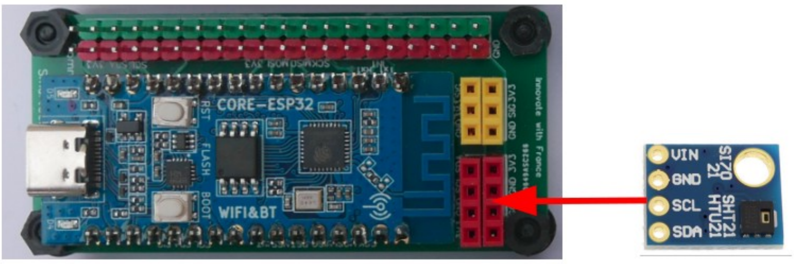



**Fig 1.3 Pomme-Pi Zero** with **SHT21** sensor on **I2C** bus

### 1.2.1 Preparing the code

For the SHT21 sensor, the reserved address is **`0x40`** in **hexadecimal** or **`64`** in **decimal**. The firmware integrates **`sht21.py`** driver.

#### Full code:

```
import machine, time
import sht21 # librairie capteur SHT21
i2c = machine.I2C(scl = machine.Pin(5), sda = machine.Pin(4), freq=100000)
c=0
while(c<20):
    if (sht21.SHT21_DETECT(i2c)):
        sht21.SHT21_RESET(i2c)
        resolution = 2
        #sht21.SHT21_SET_RESOLUTION(i2c, resolution)
        #serial_number_sht21 = sht21.SHT21_SERIAL(i2c)
        temperature = sht21.SHT21_TEMPERATURE(i2c)
        humidite = sht21.SHT21_HUMIDITE(i2c)
        print("T: {:.2f}".format(temperature))
        print("H: {:.2f}".format(humidite))
        c=c+1
        time.sleep(2)
```

#### To do:

1. Study and test the above code
2. Add an OLED display to show the sensor results

```
import machine, time, ssd1306
import sht21
i2c = machine.I2C(scl = machine.Pin(5), sda = machine.Pin(4), freq=400000)
# Declaration OLED SSD1306
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c) # 128 x 64 pixels

if (sht21.SHT21_DETECT(i2c)):
    sht21.SHT21_RESET(i2c)
    serial_number_sht21 = sht21.SHT21_SERIAL(i2c)
    temperature = sht21.SHT21_TEMPERATURE(i2c)
    humidity = sht21.SHT21_HUMIDITE(i2c)
    # Resolutions
    # 0 : Humidite=12 bits Temperature=14 bits (default)
    # 1 : Humidite=8 bits  Temperature=12 bits
    # 2 : Humidite=10 bits Temperature=13 bits
    # 3 : Humidite=11 bits Temperature=11 bits
    resolution = 0
    sht21.SHT21_SET_RESOLUTION(i2c, resolution)
    if (temperature == -1) or (humidity == -1):
        oled.fill(0)
        oled.text("I2C bus error", 0, 0)
        oled.show()
    else:
        Str_temperature = "%2.1f" % temperature + " C"
        Str_humidite = "%2.1f" % humidity + " %"
        oled.fill(0) # efface l'ecran
        oled.text(serial_number_sht21, 0, 0)
        oled.text(Str_temperature, 0, 16)
        oled.text(Str_humidity, 0, 26)
        if (sht21.SHT21_ALIMENTATION(i2c)):
            oled.text("Alimentation OK", 0, 40)
        else:
            oled.text("Alimentation NOK", O, 40)
        oled.text(sht21.SHT21_GET_RESOLUTION(i2c), 0, 50)
        oled.show()
else:
   oled.fill(0)
   oled.text("SHT21 not detected", 0, 0)
   oled.show()
```

## 1.3 Third example – reading a luminosity sensor (L) – BH1750 (I2C)

In this example we use the **BH1750** light brightness sensor

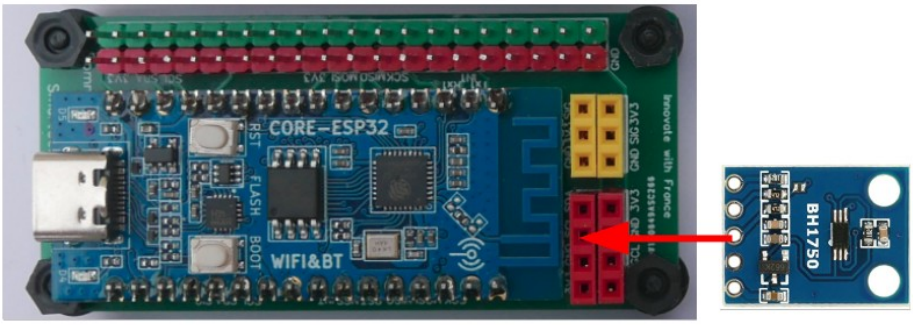



**Fig 1.5 Pomme-Pi ZERO** plus expansion board with OLED display (SSD1306) and **SHT21** and **BH1750** sensors

Here is the code with integrated **`bh1750.py`** library. The program performs 100 readings.

```
import machine
from bh1750 import BH1750
import time
sda=machine.Pin(4) # Pomme-Pi CORE
scl=machine.Pin(5) # Pomme-Pi CORE
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000) #I2C channel 0,pins,400kHz max

s = BH1750(i2c)

c=0

while c<100:
    lumi=s.luminance(BH1750.ONCE_HIRES_1)
    c+=1
    print(int(lumi))
    time.sleep(2)
```

```
%Run -c $EDITOR_CONTENT
lum
496
```

### To do:

1. Study the program to understand how it works.
2. Add the OLED screen and complete the program to show the results.

## 1.4 Fourth example – reading T/H – DHT22 sensor (simple signal)

The drivers for DHT sensors (**DHT11**, **DHT22**) are integrated into the **MicroPython** firmware, so there is no need to add the **`dht.py`** file to our example. **Note** that the output signal from the sensor is connected to **Pin 8**.

```
from machine import Pin
from time import sleep
import dht

sensor = dht.DHT22(Pin(8))   # core
while True:
  try:
    sleep(2)
    sensor.measure()
    temp = sensor.temperature()
    hum = sensor.humidity()
    temp_f = temp * (9/5) + 32.0
    print('Temperature: %3.1f C' %temp)
    print('Temperature: %3.1f F' %temp_f)
    print('Humidity: %3.1f %%' %hum)
  except OSError as e:
    print('Failed to read sensor.')
```

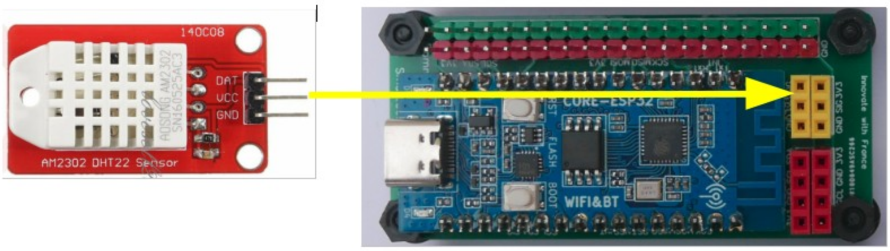



**Fig 1.6 Pomme-Pi ZERO Core board** with **DHT22** sensor

## 1.5 Fifth example – reading a PIR sensor – SR602 (simple signal)

**SR602** is a **presence sensor** activated in the presence of **Infra Red** (IR) radiation. The output signal carries a value of 1 if presence is detected, otherwise it is set to 0.

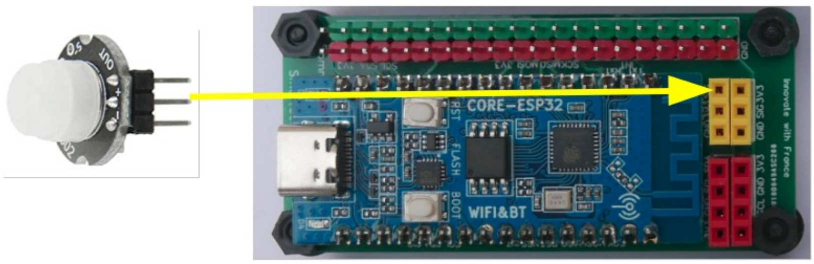



**Fig 1.7 Pomme-Pi ZERO Core board** with **PIR** motion/presence sensor: **SR602**

**Note** the use of a **three-pin slot (GND, 3V3, SIG).**

```
from machine import Pin
import time

ldr = Pin(8, Pin.IN) # create input pin on GPIO2
while True:
    if ldr.value():
        print('OBJECT DETECTED')
    else:
        print('ALL CLEAR')
    time.sleep(1)
```

```
>>> %Run -c $EDITOR_CONTENT
ALL CLEAR
ALL CLEAR
ALL CLEAR
OBJECT DETECTED
OBJECT DETECTED
OBJECT DETECTED
OBJECT DETECTED
ALL CLEAR
ALL CLEAR
ALL CLEAR
OBJECT DETECTED
..
```

### To do:

1. Study and test the program. Note the delay (about 3 sec) between the consecutive detection.

## 1.6 Sixth example – using RGB ring (`neopixel`)

The following example shows how to use (drive) an RGB LED ring. Note that the RING contains just one input signal that carries all necessary data to drive the indicated number of LEDs; each led providing 3 colors (RGB) with up to **255 light intensity levels**. Note that the **`neopixel`** driver is integrated into **MicroPython** firmware.

```
import time
import machine, neopixel
np = neopixel.NeoPixel(machine.Pin(8), 12)

def reset_ring():
    for i in range(12):
        np[i]=(0, 0, 0)
    np.write()

def set_all_red():
    for i in range(12):
        np[i]=(255, 0, 0)
    np.write()

def set_all_green():
    for i in range(12):
        np[i]=(0, 255, 0)
    np.write()

def set_all_blue():
    for i in range(12):
        np[i]=(0, 0, 255)
    np.write()

c=0
while c<60:
    reset_ring()
    time.sleep(1)
    set_all_red()
    time.sleep(1)
    set_all_green()
    time.sleep(1)
    set_all_blue()
    time.sleep(1)
    c+=1
```

### To do:

1. Study and test the program.
2. Modify the code to change the intensity of colors.
3. Write a simple **counter** (modulo) 12 and activate in turn the corresponding leds.

## 1.7 Seventh example – GPS module: NEO-6M (UART)

The following example shows the use of a GPS module connected to the extension board with **UART** bus. Note that **UART** bus is connected to the `UART_TxD` and `UART_RxD` pins on `GPIO_01` and `GPIO_00` .

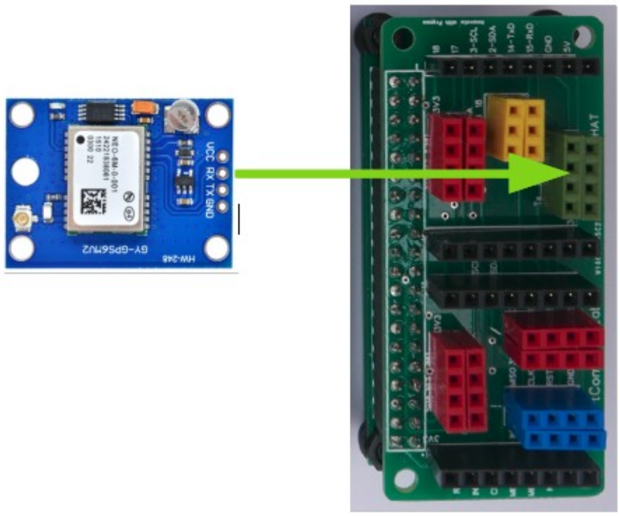


**Fig 1.8 Pomme-Pi ZERO Core board** with **GPS** module: **NEO-6M**

The code uses `MicropyGPS` class from `micropyGPS.py` file.

```
import time
import machine
from machine import Pin, SoftI2C
from micropyGPS import MicropyGPS
import ssd1306
import _thread
import time

WIDTH  = 128
HEIGHT = 64

def main():
    i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)
    dsp = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    uart = machine.UART(1, rx=0, tx=1, baudrate=9600, bits=8, parity=None, stop=1, timeout=5000,
rxbuf=1024)

    gps = MicropyGPS()

    while True:
      buf = uart.readline()
      for char in buf:
        gps.update(chr(char))  # Note the conversion to to chr, UART outputs ints normally
      #print('UTC Timestamp:', gps.timestamp)
      #print('Date:', gps.date_string('long'))
      #print('Latitude:', gps.latitude)
      #print('Longitude:', gps.longitude_string())
      #print('Horizontal Dilution of Precision:', gps.hdop)
      #print('Altitude:', gps.altitude)
      #print('Satellites:', gps.satellites_in_use)
      #print()
      dsp.fill(0)
      y = 0
      dy = 10
      dsp.text("{}".format(gps.date_string('s_mdy')), 0, y)
      dsp.text("Sat:{}".format(gps.satellites_in_use), 80, y)
      y += dy
      dsp.text("{:02d}:{:02d}:{:02.0f}".format(gps.timestamp[0],gps.timestamp[1],gps.timestamp[2]),
0, y)
```

```
      y += dy
      dsp.text("Lat:{}{:3d}'{:02.4f}".format(gps.latitude[2],gps.latitude[0],gps.latitude[1]),0,y)
      y += dy
      dsp.text("Lo:{}{:3d}'{:02.4f}".format(gps.longitude[2],gps.longitude[0],gps.longitude[1]),0,y)
      y += dy
      dsp.text("Alt:{:0.0f}ft".format(gps.altitude * 1000 / (12*25.4)),  0, y)
      y += dy
      dsp.text("HDP:{:0.2f}".format(gps.hdop),  0, y)
      dsp.show()


def startGPSthread():
    _thread.start_new_thread(main, ())

if __name__ == "__main__":
  print('...running main, GPS testing')
  main()
```

Run (after 5 min !):

```
UTC Timestamp: [14, 9, 59.0]
Date: October 9th, 2022
Latitude: [47, 13.00707, 'N']
Longitude: 1° 41.61422' W
Horizontal Dilution of Precision: 2.66
Altitude: 63.2
Satellites: 4
..
```

# Lab 2

# WiFi communication and WEB servers

## 2.0 Introduction

In this lab we will study and experiment with the WiFi features integrated into the ESP32-C3 SoC. First we are going to scan (scan) the networks available with **WiFi.scan.** Then we are going to build simple applications to read WEB pages and to send arguments to WEB servers.
Finally we will build simple WEB servers operating on the local WiFi network or even create our own access points with simple WEB servers.

## 2.1 Network scan

Edit and run the following program – **`wifiscan.py`**

```
import network
station = network.WLAN(network.STA_IF)
station.active(True)

for (ssid, bssid, channel, RSSI, authmode, hidden) in station.scan():
  print("* {:s}".format(ssid))
  print("   - Channel: {}".format(channel))
  print("   - RSSI: {}".format(RSSI))
  print("   - BSSID: {:02x}:{:02x}:{:02x}:{:02x}:{:02x}:{:02x}".format(*bssid))
  print()
```

```
>>> %Run -c $EDITOR_CONTENT
* DIRECT-G8M2070 Series
   - Channel: 11
   - RSSI: -62
   - BSSID: 86:25:19:53:78:8f

* VAIO-MQ35AL
   - Channel: 5
   - RSSI: -72
   - BSSID: d0:ae:ec:bf:3a:82

* PIX-LINK-2.4G
   - Channel: 11
   - RSSI: -76
   - BSSID: 90:91:64:50:7e:04

..
```

### To do:

1. Study and test the program. Try to understand the formatting of the data returned by the **`station.scan()`** method

### Note:

The WiFi scan program "**cleans**" the WiFi modem by putting it in the initial state with no **WiFi credentials** (**ssid**, **password**) stored in EEPROM memory.

You can use it in case of connection problems in the examples of the code to follow.

## 2.2 Connection to the WiFi network, station mode – **STA**

Our **Pomme-Pi** board can connect to the WiFi network in **station mode** (**STA**). In this case the modem can automatically retrieve (via the **DHCP** protocol) an IP address and the addresses of the router and the **DNS** server.
The modem can also impose a **static configuration** with a static IP address chosen by the user.

The following program demonstrates these features:

```
def connect():
    import network

    ip        = '192.168.1.110'
    subnet    = '255.255.255.0'
    gateway   = '192.168.1.1'
    dns       = '8.8.8.8'
    ssid      = "PhoneAP"        # replace by your SSID
    password  = "smartcomputerlab"  # and its password


    station = network.WLAN(network.STA_IF)

    if station.isconnected() == True:
        print("Already connected")
         print(station.ifconfig())
        return

    station.active(True)
    # station.ifconfig((ip,subnet,gateway,dns))  # uncomment to set static configuration
    station.connect(ssid,password)
    while station.isconnected() == False:
        pass
    print("Connection successful")
    print(station.ifconfig())

def disconnect():
    import network
    station = network.WLAN(network.STA_IF)
    station.disconnect()
    station.active(False)

# disconnect()
# connect()                    # to test operation
```

```
>>> %Run -c $EDITOR_CONTENT
disconnected - start connection
>>> %Run -c $EDITOR_CONTENT
Connection successful
('192.168.43.136', '255.255.255.0', '192.168.43.1', '192.168.43.1')
```

### To do:

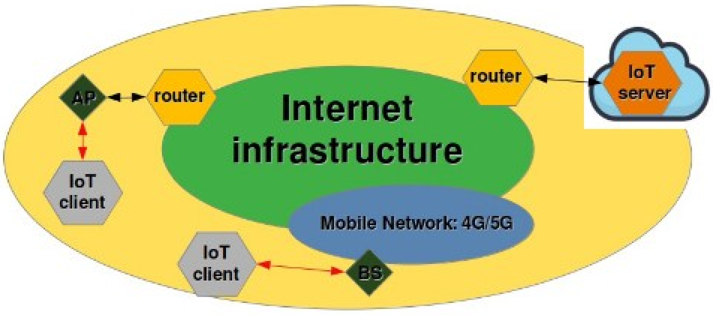
1. Study and test the program with your access point.
2. Save the main code with def connect() and def disconnect() in a **`wifista.py`** python module.

### Important Note

We will use this module (**`wifista.py`**) in many examples requiring WiFi connection in **STA** mode.

## 2.3 Getting time from NTP server

The **Network Time Protocol** (**NTP**) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. In operation since before 1985, NTP is one of the oldest Internet protocols in current use.




```
import ntptime
import wifista
import time
#wifista.scan()
wifista.disconnect()
wifista.connect()
set=1
while set:
        (year,montth,day,hour,min,sec,val1,val2)=time.localtime()
        print("hour: "+ str(hour))
        print("min: "+ str(min))
        print("sec: "+ str(sec))
        time.sleep(1)
```

```
>>> %Run -c $EDITOR_CONTENT
Connection successful
('192.168.43.136', '255.255.255.0', '192.168.43.1', '192.168.43.1')
hour: 9
min: 38
sec: 47
hour: 9
min: 38
sec: 48
hour: 9
```

### To do:

Use the **Buzzer** to signal each second/minute.

```
        buz = Pin(8, Pin.OUT) # create input pin on GPIO0
        ..
        buz.on() # to buzz
        buz.off() # not to buzz
```

#### Using `neopixel` LED-ring (reminder):

The LED-ring with RGB LEDs allows us to build a kind of wall-clock with different colors associated for different time units; for example – hours in RED, minutes in GREEN and seconds in BLUE.

The RGB colors are defined by 3 bytes, one byte per color. If the byte is set to zero there is no light associated to the given color. The maximum value (brightness) is 255.

Use the hour, minute, second values obtained from NTP server and project them on the LED-ring. To facilitate the task we provide you with almost complete code for this application.

Note that the LED activation should be aligned to the vertical axis.

```
import ntptime
import wifista
import time
```

```
import machine, neopixel
np = neopixel.NeoPixel(machine.Pin(8), 12)

def reset_clock():
        for i in range(12):
        np[i]=(0, 0, 0)

def set_clock(h,m,s,lum):
        reset_clock()
        np[s] = (0, 0, lum) # set to blue, quarter brightness
        np[m] = (0, lum, 0) # set to green, half brightness
        np[h] = (lum, 0, 0) # set to red, full brightness
        np.write()

wifista.disconnect()
wifista.connect()
set=0
print("Local time before synchronization: %s" %str(time.localtime()))
ntptime.settime()
set=1
while set:
        #print("Local time after synchronization: %s" %str(time.localtime()))
        (year,montth,day,hour,min,sec,val1,val2)=time.localtime()
        print("hour: "+ str(hour))
        print("min: "+ str(min))
        print("sec: "+ str(sec))
        ledmin= .. # to complete
        ledsec= .. # to complete
        ledhour= .. # to complete
        print(int(ledhour),int(ledmin),int(ledsec))
        set_clock(int(ledhour),int(ledmin),int(ledsec),64) # 64 is proposed brightness
        time.sleep(5)
```

## 2.4 Reading a WEB page

The following example shows how to connect to a WiFi AP and how to send an HTTP request to receive a WEB page.
To facilitate development, we use the `urequests.py` library which contains the methods for connecting to WEB servers and sending **HTTP requests** (`GET, POST`).

```
import machine
import sys
import network
import utime, time
import urequests
import wifista

# Pin definitions
led = machine.Pin(12,machine.Pin.OUT)  // LED 12, 13

# Network settings
wifista.disconnect()
wifista.connect()

# Web page (non-SSL) to get
url = "http://www.smartcomputerlab.org"
# Continually print out HTML from web page as long as we have a connection
c=0
while c<4:
    wifista.connect()
    # Perform HTTP GET request on a non-SSL web
    response = urequests.get(url)
    # Display the contents of the page
    print(response.text)
    c+=1
    time.sleep(6)

print("End of program.")
```

### To do:

Use the LED to signal the reading of a page.

Example of a code with the **LED** on pin 12 (or 13).

```
from machine import Pin
from time import sleep

led=Pin(12,Pin.OUT)

while True:
    led.value(not led.value())
    sleep(1.1)
```

## 2.5 Simple WEB server – reading a variable

It is possible to create an **HTTP server** (or **WEB server**). The HTML code is written directly in the main program or contained in a separate file. Communication between client and server uses **socket** mechanism:

- The server is listening on the port. It is waiting for a client connection.
- As long as no client shows up, the program remains blocked (accept)
- The client sends a request.
- The server processes the request and then sends the response.

```
from machine import Pin
import usocket as socket
import wifista

def web_page():
    pot = 55
    print("CAN =", pot)
    html = """
    <!DOCTYPE html>
    <html>
        <head>
            <meta name="viewport" content="width=device-width, initial-scale=1">
            <title>ESP32 WEB server</title>
            <style>
                p { font-size: 36px; }
            </style>
        </head>
        <body>
            <h2>Hello from Pomme-Pi</h2>
            <h3>A variable = </h3>
            <p><span>""" + str(pot) + """</span></p>
        </body>
    </html>
    """
    return html

wifista.connect()
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind(('', 80))
serverSocket.listen(5)
while True:
    try:
        if gc.mem_free() < 102000:
            gc.collect()
        print("Waiting for client")
        clientConnection, adresse = serverSocket.accept()  # accept TCP connection request
        clientConnection.settimeout(4.0)
        print("Connected with client", adresse)
        print("Waiting for client request")
        request = clientConnection.recv(1024)              # receiving client request - HTTP
        request = str(request)
        print("Client request= ", request)
        clientConnection.settimeout(None)
        print("Sending response to client : HTML code to display")
        clientConnection.send('HTTP/1.1 200 OK\n')
        clientConnection.send('Content-Type: text/html\n')
        clientConnection.send("Connection: close\n\n")
        reponse = web_page()
        clientConnection.sendall(reponse)
        clientConnection.close()
        print("Connection with client closed")

    except:
        clientConnection.close()
        print("Connection closed, program error")
```

### To do:

1. Analyze and test the program with your smartphone (why we use: `try` and `except`)
2. Edit the text on the HTML page.

## 2.6 Simple WEB server – sending an order

In the previous example we read a value generated by our **Pomme-Pi**. In this section we will send, from our smartphone, a command to display on local OLED screen. Below is the code of the WEB server which allows to receive HTTP requests and display the corresponding messages on the OLED screen.

### 2.6.1 Mini WEB server en mode station – RGB LED management

```
from machine import Pin, SoftI2C
import time
import ssd1306
import usocket as socket
import wifista

def web_page():
    html = """
    <!DOCTYPE html>
    <html>
        <head
            <meta name="viewport" content="width=device-width, initial-scale=1">
            <title>ESP32 Serveur Web</title>
            <style>
                p { font-size: 36px; }
            </style>
        </head>
        <body>
            <h1>Commande LED</h1>
            <p><a href="/?led=green">LED GREEN</a></p>
            <P><a href="/?led=red">LED RED</a></p>
            <p><a href="/?led=blue">LED BLUE</a></p>
        </body>
    </html>
    """
    return html

i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
oled.fill(0)
oled.text("SmartComputerLab", 0, 0)
oled.show()
time.sleep(1)
wifista.connect()
time.sleep(1)
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind(('', 80))
serverSocket.listen(5)
while True:
    try:
        if gc.mem_free() < 102000:
            gc.collect()
        print("Attente connexion d'un client")
        clientConnection, adresse = serverSocket.accept()
        clientConnection.settimeout(4.0)
        print("Connected to client", adresse)
        print("Waiting for client")
        request = clientConnection.recv(1024)     # request from client
        request = str(request)
        print("Request from client = ", request)
        clientConnection.settimeout(None)
        #analyse de la requête, recherche de led=on ou led=off
        if "GET /?led=green" in request:
            print("LED GREEN")
            oled.fill(0)
            oled.text("LED GREEN", 0, 0)
            oled.show()
        if "GET /?led=red" in request:
            print("LED RED")
            oled.fill(0)
            oled.text("LED RED", 0, 0)
```

```
            oled.show()
        if "GET /?led=blue" in request:
            print("LED BLUE")
            oled.fill(0)
            oled.text("LED BLUE", 0, 0)
            oled.show()

        print("Sending response to server : HTML code to display")
        clientConnection.send('HTTP/1.1 200 OK\n')
        clientConnection.send('Content-Type: text/html\n')
        clientConnection.send("Connection: close\n\n")
        reponse = web_page()
        clientConnection.sendall(reponse)
        clientConnection.close()
        print("Connexion avec le client fermee")

    except:
        clientConnection.close()
        print("Connection closed, program error")
```

**To do:**

1. Test the program
2. Display the IP address and SSID name on OLED screen
3. Use RGB led to signal message (color)

## 2.6.2 Mini WEB server with Access Point – RGB LED management

The following program is almost identical to the one shown in the previous section, but it creates its own access point with **`ssid="Pomme-Pi AP"`** and the default IP address: **`192.168.4.1`**; the default password is **`"smarcomputertlab"`**.

Here goes the code:

```
from machine import Pin, SoftI2C
import network,ssd1306
import usocket as socket

i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
oled.fill(0)
oled.text("SmartComputerLab", 0, 0)
oled.show()


def web_page():

    html = """
    <!DOCTYPE html>
    <html>
        <head
            <meta name="viewport" content="width=device-width, initial-scale=1">
            <title>ESP32 WEB server</title>
            <style>
                p { font-size: 36px; }
            </style>
        </head>
        <body>
            <h1>Commande LED</h1>
            <p><a href="/?led=green">LED GREEN</a></p>
            <P><a href="/?led=red">LED RED</a></p>
            <p><a href="/?led=blue">LED BLUE</a></p>
        </body>
    </html>
    """
```

```
    return html

ssid="Pomme-Pi AP"
password="smarcomputertlab"
ap = network.WLAN(network.AP_IF)    # set WiFi as Access Point
ap.active(True)
ap.config(essid=ssid, password=password)
print(ap.ifconfig())

serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind(('', 80))
serverSocket.listen(5)

while True:
    try:
        if gc.mem_free() < 102000:
            gc.collect()

        print("Waiting for client")
        clientConnection, adresse = serverSocket.accept()
        clientConnection.settimeout(4.0)
        print("Connected to client", adresse)
        print("Waiting for client request")
        request = clientConnection.recv(1024)     #requête du client
        request = str(request)
        print("Client request = ", request)
        clientConnection.settimeout(None)
        #request analyzis: led=on ou led=off
        if "GET /?led=green" in request:
            print("LED GREEN")
            oled.fill(0)
            oled.text("LED GREEN", 0, 0)
            oled.show()
        if "GET /?led=red" in request:
            print("LED RED")
            oled.fill(0)
            oled.text("LED RED", 0, 0)
            oled.show()
        if "GET /?led=blue" in request:
            print("LED BLUE")
            oled.fill(0)
            oled.text("LED BLUE", 0, 0)
            oled.show()

        print("Sending response to server : HTML code to display")
        clientConnection.send('HTTP/1.1 200 OK\n')
        clientConnection.send('Content-Type: text/html\n')
        clientConnection.send("Connection: close\n\n")
        reponse = web_page()
        clientConnection.sendall(reponse)
        clientConnection.close()
        print("Connection closed")

    except:
        clientConnection.close()
        print("Conneclosed, program error")
```

## To do:

1. Test the program
2. Display the IP address and SSID name on OLED screen
3. Use RGB led to signal message (color)

# Lab 3

## MQTT Broker and ThingSpeak Server

In this lab we will study and experiment with IoT servers such as MQTT and ThingSpeak.

## 3.1 MQTT Protocol and MQTT Client

**MQTT,** that stands for '**M**essage **Q**ueuing **T**elemetry **T**ransport', is a **publish/subscribe messaging protocol** based on the **TCP/IP** protocol. A client, called publisher, first establishes a '**publish**' type connection with the MQTT server, called broker.

The publisher transmits the messages to the broker on a **specific channel**, called **topic**. Subsequently, these messages can be read by subscribers, called subscribers, who have previously established a 'subscribe' type connection with the broker.

In this section we will study the **MQTT protocol** and we will **write a program** that allows you to send (**publish**) MQTT messages on an MQTT server-broker, then retrieve the latest messages posted by **subscribing** to the given topic.

The transmission and consumption of messages is done asynchronously.
The operation we have just detailed is illustrated in the diagram below.

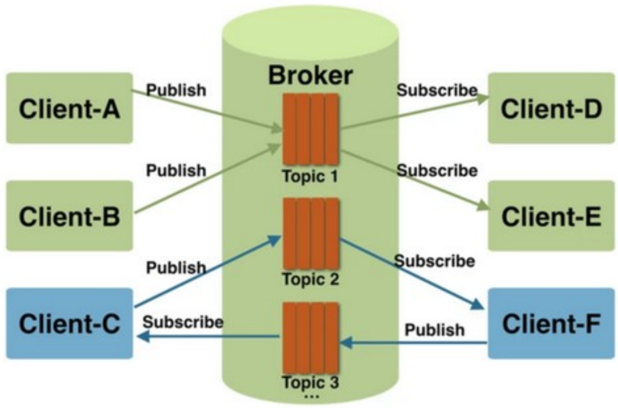



**Fig. 3.1 Client-A**, **Client-B** and **Client-F** are publishers while **Client-C**, **Client-D** and **Client-E** are subscribers.

To prepare our program we need the library – **`umqtt`** .

### 3.1.1 MQTT client – the code

In this example, we will connect our Pomme-Pi to the **free public MQTT server** operated and maintained by **`EMQX MQTT` Cloud**.

Here is an example of the program that uses the **`umqtt`** library and its **`MQTTClient`** class.

Note that **`umqtt.simple`** and **`umqtt.robust`** are already integrated into our firmware.

#### 3.1.1.1 The code of MQTT Client

```
from umqtt.robust import MQTTClient
import machine
import wifista
import utime as time
import gc

wifista.connect()
broker = "broker.emqx.io"
client = MQTTClient("Pomme-Pi", broker)

def sub_cb(topic, msg):
    print((topic, msg))
    if topic == b'pomme-pi/test' :
        print('Pomme-Pi received '+ str(msg))

def subscribe_publish():
    count = 1
```

```
    client.set_callback(sub_cb)
    client.subscribe(b"pomme-pi/test")
    while True:
        client.check_msg()
        mess="hello: " + str(count)
        client.publish(b"pomme-pi/test", mess)
        count = count + 1
        time.sleep(20)


client.reconnect()
subscribe_publish()
```

```
>>> %Run -c $EDITOR_CONTENT
Already connected
('192.168.43.136', '255.255.255.0', '192.168.43.1', '192.168.43.1')
(b'pomme-pi/test', b'hello: 1')
Pomme-Pi received b'hello: 1'
(b'pomme-pi/test', b'hello: 2')
Pomme-Pi received b'hello: 2'
```

### To do:

1. Test the program on your smartphone with the `MyMQTT` application
2. Add the display of messages received on the **OLED** screen
3. Add a sensor and `publish` the captured values on a `topic`

## 3.1.2 Broker MQTT on a PC

It is very easy to install your own **MQTT broker** on a PC; it is called `mosquitto`.
The download page that explains the installation of `mosquitto` broker (and client) is available here:

`https://mosquitto.org/download/`

### To do:

1. Download and install `mosquitto`
2. Test MQTT client programs with `mosquitto` broker (example)

```
ubuntu@bako:~/esptool-master$ mosquitto_sub -h broker.emqx.io -t pomme-pi/test
hello: 13
hello: 14
hello: 15
```

## 3.2 ThingSpeak server

**ThingSpeak** is an **open source** API and application for the "**Internet of Things**", allowing data to be stored and collected from connected objects with HTTP protocol via the Internet or a local network.
With **ThingSpeak**, the user can create sensor data logging apps, location tracking apps, and a social network for IoT devices with status updates.

**ThingSpeak** Features:

- Open API
- Real-time data collection
- Geo-location data
- Data processing
- Data visualizations
- Circuit status messages
- Plugins

**ThingSpeak** can be integrated with RISC-V, ESP32, Raspberry Pi, .. platforms, mobile/web applications, social networks and data analysis with **MATLAB** (`ThingSpeak.com`)

### 3.2.1 Preparation for sending data with `thingspeak.py` library

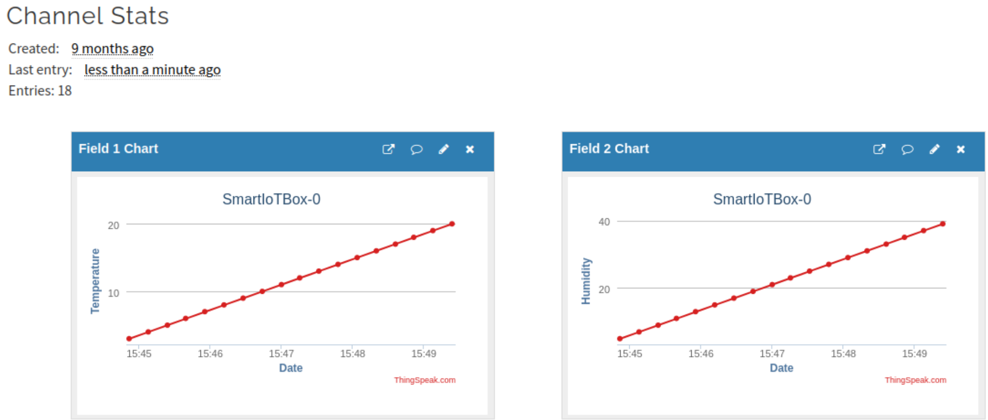
The **easiest way** to send the data to ThingSpeak server is touse `thingspeak.py` library available here:

`https://raw.githubusercontent.com/radeklat/micropython-thingspeak/master/src/lib/thingspeak.py`

Download it and save it on your PC and on the **Pomme-Pi** board.
An example of use of `thingspeak.py` library is given below:

```
import machine
import time
import wifista
import thingspeak
from thingspeak import ThingSpeakAPI, Channel, ProtoHTTP
channel_living_room = "1538804"
field_temperature = "Temperature"
field_humidity = "Humidity"
thing_speak = ThingSpeakAPI([
    Channel(channel_living_room , 'YOX31M0EDKO0JATK', [field_temperature, field_humidity])],
    protocol_class=ProtoHTTP, log=True)

wifista.connect()
active_channel = channel_living_room
temperature = 2.0
humidity=3.0
c=0
while c<20:
    thing_speak.send(active_channel, {
        field_temperature: temperature,
        field_humidity: humidity
    })
    temperature=temperature+1.0
    humidity=humidity+2.0
    c=c+1
    time.sleep(thing_speak.free_api_delay)

>>> %Run -c $EDITOR_CONTENT
Already connected
('192.168.43.136', '255.255.255.0', '192.168.43.1', '192.168.43.1')
ThingSpeak at 3.90.157.224:80
1538804 {'Humidity': 33.7, 'Temperature': 21.4} #1, took 0.55s, next in 15.45s
1538804 {'Humidity': 33.7, 'Temperature': 21.4} #2, took 0.55s, next in 15.45s
1538804 {'Humidity': 33.7, 'Temperature': 21.4} #3, took 0.50s, next in 15.50s
1538804 {'Humidity': 33.7, 'Temperature': 21.4} #4, took 0.48s, next in 15.52s
```

In the last statement you can note the delay value of `sleep()` required for a **free** account:




```
time.sleep(thing_speak.free_api_delay)
```

## To do:

1. Test the programs above with **your ThingSpeak** account
2. Add one or more sensors to send the actual data

### 3.2.2 Preparation for sending and receiving data as simple HTTP requests

The **simplest way** to send and receive data on the **ThingSpeak** server is to use directly the **socket** library. A TCP connection with socket makes it possible to establish a link with the **ThingSpeak** server (`s.connect(addr)`), then transmit HTTP requests to send/receive data.

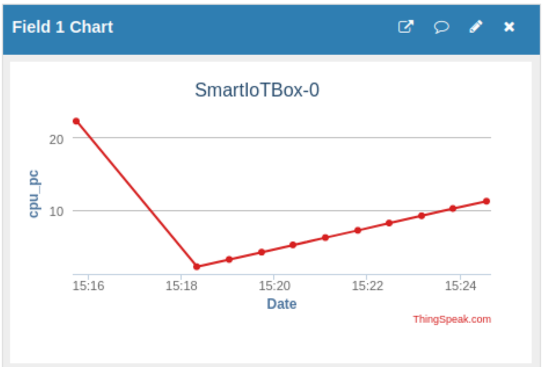
Here is a simple, but complete example, with a function `http_get(url)` allowing to establish a **TCP/HTTP connection**, then to send the data (t,h) and finally to read a data (field) on the requested channel in `json` format.

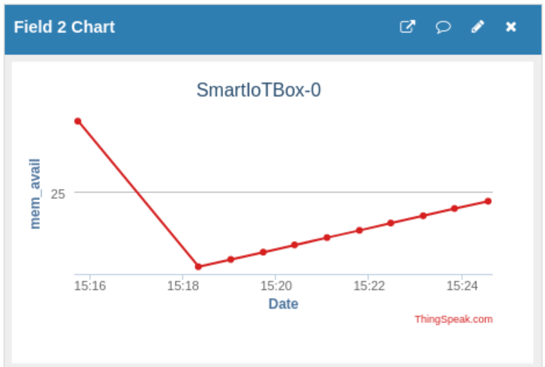
```
import socket
import wifista
import time

def http_get(url):
    import socket
    _, _, host, path = url.split('/', 3)
    print(path)
    print(host)
    addr = socket.getaddrinfo(host, 80)[0][-1]
    s = socket.socket()
    s.connect(addr)
    s.send(bytes('GET /%s HTTP/1.0\r\nHost: %s\r\n\r\n' % (path, host), 'utf8'))
    while True:
            data = s.recv(100)
            if data:
              print(str(data, 'utf8'), end='')
            else:
              break
    s.close()

t=2.2
h=4.4
c=0
while c<10:
    wifista.connect()
    urlkey='https://api.thingspeak.com/update?api_key=YOX31M0EDKO0JATK'
    fields='&field1='+str(t)+'&field2='+str(h)
    http_get(urlkey+fields)
    time.sleep(15)
    http_get('https://api.thingspeak.com/channels/1538804/fields/2/last.json?
api_key=20E9AQVFW7Z6XXOM')
    t=t+1.0
    h=h+2.0
    c=c+1
    time.sleep(25)
```

## Channel Stats

Created: 9 months ago

Entries: 11




Execution result (one complete cycle: **send** and **receive** **`last`** item in JSON format :

```
Already connected
('192.168.43.136', '255.255.255.0', '192.168.43.1', '192.168.43.1')
update?api_key=YOX31M0EDKO0JATK&field1=2.2&field2=4.4
api.thingspeak.com
HTTP/1.1 200 OK
Date: Fri, 29 Jul 2022 13:18:20 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 1
Connection: close
Status: 200 OK
Cache-Control: max-age=0, private, must-revalidate
Access-Control-Allow-Origin: *
Access-Control-Max-Age: 1800
X-Request-Id: de6d7767-fb3e-4002-8887-5ae50271616d
Access-Control-Allow-Headers: origin, content-type, X-Requested-With
Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS, DELETE, PATCH
ETag: W/"d4735e3a265e16eee03f59718b9b5d03"
X-Frame-Options: SAMEORIGIN

2channels/1538804/fields/2/last.json?api_key=20E9AQVFW7Z6XXOM
api.thingspeak.com
HTTP/1.1 200 OK
Date: Fri, 29 Jul 2022 13:18:36 GMT
Content-Type: application/json; charset=utf-8
Connection: close
Status: 200 OK
Cache-Control: max-age=7, private
Access-Control-Allow-Origin: *
Access-Control-Max-Age: 1800
X-Request-Id: b42bff93-34e7-4416-94fb-ec856e8c84a6
Access-Control-Allow-Headers: origin, content-type, X-Requested-With
Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS, DELETE, PATCH
ETag: W/"33fe122f595ed87e04c7a493503e1096"
X-Frame-Options: SAMEORIGIN

{"created_at":"2022-07-29T13:18:20Z","entry_id":2,"field2":"4.4"}

---------------------------------------------------------------------
```

### To do:

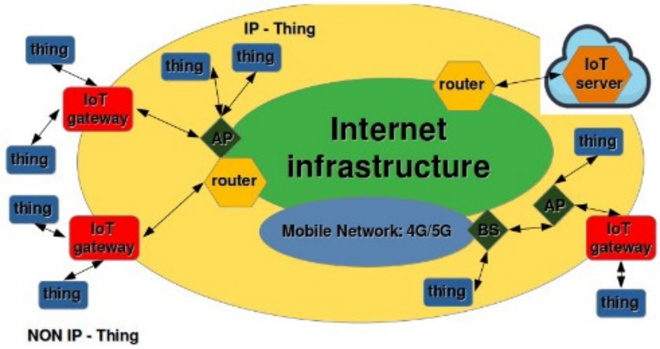
1. Test the above programs with your ThingSpeak account
2. Add one or more sensors to send the actual data
3. Parse the result in **`json`** format with a **decode function**

# Lab 4

## LoRa technology for Long Range communication

## 4.0 Introduction

In this lab we will focus on the **Long Range** transmission technology essential for communication between remote objects that are not connected directly to the Internet Infrastructure.
**Long Range** or **LoRa** allows data to be transmitted over a distance of one kilometer or more with speeds ranging from a few hundred bits per second to a few tens of Kilo-bits (100bit – 75Kbit).




## 4.1 LoRa  Modulation

LoRa modulation has **three basic parameters (there are many others)**:

- **`freq`** – frequency or carrier frequency from 868 to 870 MHz,
- **`sf`** – spreading factor or spreading of the spectrum or the number of modulations per bit sent (64-4096 expressed in powers of 2 – 7 to 12)
- **`sb`** – signal bandwidth or signal bandwidth (31250 Hz to 500KHz)

By default we use: **freq=434MHz** or **868MHz**, **sf=7**, and **sb=125KHz**

To provide LoRa communication our DevKiT can be completed with an expansion board – **LoRa modem.**

## 4.2 `sx127x.py` driver library

The **`sx127x.py`** library makes it possible to integrate the functionalities of the **sx1276/8 modem** into our applications.
The modem-circuit is connected to our base board by  **SPI bu**s. An SPI bus operates on **3 basic lines** (signals): **SCK** – clock, **MISO** – Master_In_Slave_Out, **MOSI** – Master_Out_Slave_In, and on **three control lines**: **NSS** – Slave output selection or activation, **RST** – signal d initialization, and **DIO0/INT** – interrupt signal sent by the activated Slave.

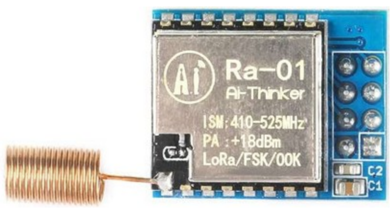




**Fig 4.1** The LoRa modem-module (**Ra-01**) with its **SPI** connector
Here are some excerpts from the **`sx127x.py`** library

```
class SX127x:

    default_parameters = {
```

```
        "frequency": 434500000,  # to be overloaded
        "frequency_offset": 0,
        "tx_power_level": 14,
        "signal_bandwidth": 125e3,
        "spreading_factor": 9,
        "coding_rate": 5,
        "preamble_length": 8,
        "implicitHeader": False,
        "sync_word": 0x12,
        "enable_CRC": True,
        "invert_IQ": False,
    }
```

The default (radio) **settings** can be changed through the functions available in the same library:

```
        self.setFrequency(self.parameters["frequency"])
        self.setSignalBandwidth(self.parameters["signal_bandwidth"])
        # set LNA boost
        self.writeRegister(REG_LNA, self.readRegister(REG_LNA) | 0x03)
        # set auto AGC
        self.writeRegister(REG_MODEM_CONFIG_3, 0x04)
        self.setTxPower(self.parameters["tx_power_level"])
        self.implicitHeaderMode(self.parameters["implicitHeader"])
        self.setSpreadingFactor(self.parameters["spreading_factor"])
        self.setCodingRate(self.parameters["coding_rate"])
        self.setPreambleLength(self.parameters["preamble_length"])
        self.setSyncWord(self.parameters["sync_word"])
        self.enableCRC(self.parameters["enable_CRC"])
        self.invertIQ(self.parameters["invert_IQ"])
```

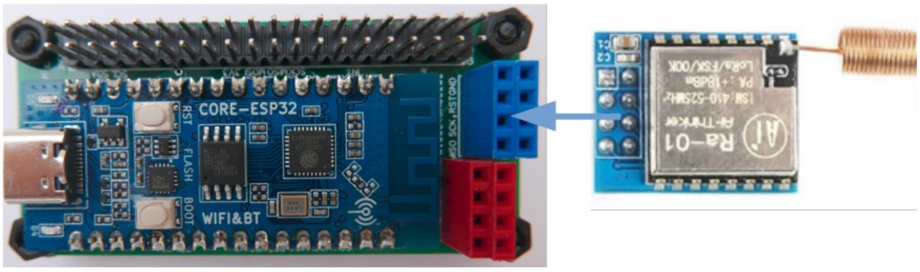



**Fig 4.2** Connecting the LoRa (Ra-01) (SPI) module to the **Pomme-Pi Zero Core board (V1).**

## 4.3 Main program

In the program that we are going to study we find all the parameters and the initialization of the operating functions of the LoRa module.

In the **`lora_default`** list we offer the parameters compatible with our LoRa modem (ISM: 434MHz).

For now we will only focus on **3 parameters**:

- **frequency**,
- **signal bandwidth** and
- the **spreading factor**

The modem is connected on the SPI bus; the **`lora_pins`** **list** identifies the **signal numbers** used to connect our modem to the **PYCOM-X board**.

Finally, the parameters and control signals associated with the SPI bus – **`lora_spi`** are determined.

With all the parameters initialized, the communication between the card and the LoRa modem (**sx127x**) is activated. Once the connection is activated we can call different LoRa communication functions, such as:

```
        # type = 'sender'
        # type = 'receiver'
        # type = 'receiver_callback'
```

Here is the full code, predefined for the use of **`LoRaSender`** function (and module):

```
from machine import Pin,SPI
from sx127x import SX127x
from time import sleep

import LoRaSender
# import LoRaReceiver
# import LoRaReceiverCallback
# radio - modulation parameters

lora_default = {                    # our settings
    'frequency': 434500000,   #869525000,
    'frequency_offset':0,
    'tx_power_level': 14,
    'signal_bandwidth': 125e3,       # 125 KHz
    'spreading_factor': 9,           # 2 to power 9
    'coding_rate': 5,                # 4 data bits over a symbol with 5 bits
    'preamble_length': 8,
    'implicitHeader': False,
    'sync_word': 0x12,
    'enable_CRC': False,
    'invert_IQ': False,
    'debug': False,
}

# modem - connection wires-pins on SPI bus - case Pomme-Pi Zero Core
lora_pins = {
    'dio_0':11,
    'ss':7,      # 16 on SPI-LoRa ext. card
    'reset':6,   #RST
    'sck':2,
    'miso':10,
    'mosi':3,
}

lora_spi = SPI(
    baudrate=10000000, polarity=0, phase=0,
    bits=8, firstbit=SPI.MSB,
    sck=Pin(lora_pins['sck'], Pin.OUT, Pin.PULL_DOWN),
    mosi=Pin(lora_pins['mosi'], Pin.OUT, Pin.PULL_UP),
    miso=Pin(lora_pins['miso'], Pin.IN, Pin.PULL_UP),
)

lora = SX127x(lora_spi, pins=lora_pins, parameters=lora_default)

# type = 'receiver'
# type = 'ping_master'
# type = 'ping_slave'
type = 'sender'     # let us select sender method

if __name__ == '__main__':
    if type == 'sender':
        LoRaSender.send(lora)
#     if type == 'receiver':
#         LoRaReceiver.receive(lora)
#     if type == 'receiver_callback':
#         LoRaReceiverCallback.receiveCallback(lora)


Type "help()" for more information.
>>> %Run -c $EDITOR_CONTENT
Warning: SPI(-1, ...) is deprecated, use SoftSPI(...) instead
```

```
SX version: 18
LoRa Sender
TX: Long long Hello (0)
TX: Long long Hello (1)
TX: Long long Hello (2)
T..
```

### Remark:

The **correct execution** of this program is indicated by this line:

```
SX version: 18
```

It indicates that the modem is **correctly connected and initialized**. If the indicated version is `255` the modem is not activated.

In the program above we have chosen the elements to configure the code as **`sender`** - **`LoRaSender`**.
The -**`type`** **switch** at the end of the program will select the **`LoRaSender.py`** module

## 4.4 LoRa functional modules

### 4.4.1 Transmitter – `sender()` `(LoRaSender.py)`

Our transmitter module (sender) uses the OLED screen to present the value of the LoRa message counter. Data is sent as character strings.

```
from time import sleep
import machine, ssd1306
from machine import Pin, SoftI2C
import esp32

def disp(c):
    i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text("LoRa sender", 0, 16)
    oled.text("Packet Nr:", 0, 32)
    oled.text(str(c), 0, 48)
    oled.show()

def send(lora):
    print("LoRa Sender")
    counter = 0
    while True:
        payload = 'Long long Hello ({0})'.format(counter)
        print('TX: {}'.format(payload))
        lora.println(payload)
        counter += 1
        disp(counter)
        sleep(5)
```

### To do:

1. Test the **main** program with the **`LoRaSender.py`** module above.
2. Add the reading of a sensor (**`sht21.py`**) and send the captured values.
3. Save the main program as **`main.py`** , launch its execution, then detach the card from your PC so that it runs autonomously on its battery.

### 4.4.2 Receiver – `receive (LoRaReceiver.py)`

Our receiver module (**`receive()`**) uses the OLED screen to present the **RSSI** (**R**eceived **S**ignal **S**trength **I**ndicator) value corresponding to the received LoRa messages.

```
from time import sleep
import machine, ssd1306
from machine import Pin, SoftI2C
import esp32

def disp(p):
    i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text("LoRa receiver", 0, 16)
    oled.text("Packet Nr:", 0, 32)
    oled.text(format(p), 0, 48)
    oled.show()

def receive(lora):
    print("LoRa Receiver")

    while True:
        if lora.receivedPacket():
            try:
                payload = lora.readPayload().decode()
                rssi = lora.packetRssi()
                print("RX: {} | RSSI: {}".format(payload, rssi))
                disp(payload)
            except Exception as e:
                print(e)
```

### To do:

1. Test the main program with the **`LoRaReceiver.py`** module above.
2. Add payload value presentation on OLED screen.
3. Save the main program as **`main.py`** , run it, then detach the board from your PC to run on battery power.

### 4.4.3 Receiver – `onReceive (LoRaReceiverCallback.py)`

The reception of a LoRa packet can be performed **asynchronously** by means of the **interrupt signal** generated by the **sx127x modem** (**`INT/DIO0`**) at the time of reception of the physical frame and its recording in the reception buffer.

Here is the code:

```
from time import sleep
import machine, ssd1306
from machine import Pin, SoftI2C
import esp32

def disp(p):
    i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text("LoRa receiver", 0, 16)
    oled.text("Packet Nr:", 0, 32)
    oled.text(format(p), 0, 48)
    oled.show()

def receiveCallback(lora):
    print("LoRa Receiver Callback")
```

```
    lora.onReceive(onReceive)
    lora.receive()


def onReceive(lora, payload):
    try:
        payload = payload.decode()
        rssi = lora.packetRssi()
        print("RX: {} | RSSI: {}".format(payload, rssi))
        disp(rssi)
    except Exception as e:
        print(e)
```

### To do:

1. Test the **main** program with the above **`LoRaReceiverCallback.py`** module.
2. Add the presentation of the payload value (data received from the **SHT21** sensor) on the OLED screen.
3. Save the main program as **`main.py`** , run it, then detach the board from your PC to run on battery power.

# Lab 5

# Development of simple IoT gateways

In this lab we will develop an architecture integrating several essential devices for the creation of a **complete IoT system**. The central device will be the gateway between the LoRa links and the WiFI communication.

## 5.1 LoRa-WiFi Gateway (MQTT)

Our first example illustrates the construction of a **LoRa-WiF gateway to an MQTT broker**.
The **gateway** (G) receives the **LoRa packets** with a payload containing the data from the sensors associated with the LoRa terminal.
The principal modules to import are:

```
from umqtt.robust import MQTTClient
```

This module is used to define the **broker** to use (**IP address**) and to establish a TCP connection on port **1883**. (unsecured version)
WiFi connection is realized by our **`wifista`** module; this module can be modified in order to be able to choose between a static or dynamic address.
We need two serial buses: **SPI** and **I2C** (soft version). The **OLED** screen is attached to the **`SoftI2C`** bus.
The **LoRa** modem is connected by the **SPI** bus whose parameters are defined in the code. The default radio settings are also set in code (**`lora_default`**).

Below is the **main** program which can be modified to make it work with another functional module.
To start we will choose the **`LoRaReceiverGatewayMqtt.py`** module

```
from machine import Pin,SPI
from sx127x import SX127x
from time import sleep
# import LoRaSender
# import LoRaReceiver
# import LoRaReceiverCallback
# import LoRaReceiverGatewayTsMqtt    # to import gateway LoRa-WiFi to TS with MQTT
# import LoRaReceiverGatewayMqtt      # to import gateway LoRa-WiFi to MQTT

# radio - modulation parameters
lora_default = {
    'frequency': 434500000,   #869525000,
    'frequency_offset':0,
    'tx_power_level': 14,
    'signal_bandwidth': 125e3,
    'spreading_factor': 9,
    'coding_rate': 5,
    'preamble_length': 8,
    'implicitHeader': False,
    'sync_word': 0x12,
    'enable_CRC': False,
    'invert_IQ': False,
    'debug': False,
}

# modem - connection wires-pins on SPI bus - case Pomme-Pi Zero Core
lora_pins = {
    'dio_0':11,
    'ss':7,      # 16 on SPI-LoRa ext. card
    'reset':6,   #RST
    'sck':2,
    'miso':10,
    'mosi':3,
}

lora_spi = SPI(
    baudrate=10000000, polarity=0, phase=0,
```

```
    bits=8, firstbit=SPI.MSB,
    sck=Pin(lora_pins['sck'], Pin.OUT, Pin.PULL_DOWN),
    mosi=Pin(lora_pins['mosi'], Pin.OUT, Pin.PULL_UP),
    miso=Pin(lora_pins['miso'], Pin.IN, Pin.PULL_UP),
)

lora = SX127x(lora_spi, pins=lora_pins, parameters=lora_default)

# type = 'sender'
# type = 'receiver'
# type = 'receiver_callback'
type = 'gatewaytsmqtt'
# type = 'gatewaymqtt'
# type = 'sender'     # let us select sender method

if __name__ == '__main__':
#     if type == 'sender':
#         LoRaSender.send(lora)
#     if type == 'receiver':
#         LoRaReceiver.receive(lora)
#     if type == 'ping_master':
#         LoRaPing.ping(lora, master=True)
#     if type == 'ping_slave':
#         LoRaPing.ping(lora, master=False)
#     if type == 'receiver_callback':
#         LoRaReceiverCallback.receiveCallback(lora)
#     if type == 'gatewaytsmqtt':
#         LoRaReceiverGatewayTsMqtt.receive(lora)
#     if type == 'gatewaymqtt':
#         LoRaReceiverGatewayMqtt.receive(lora)
#
```

The module called in the **main** program - `LoRaReceiverGatewayMqtt.py` is as follows:

```
from umqtt.robust import MQTTClient
import wifista
from time import sleep
import machine, ssd1306
from machine import Pin, SoftI2C
import esp32

def disp(p):
    i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text("LoRa receiver", 0, 16)
    oled.text("Packet Nr:", 0, 32)
    oled.text("{}".format(p), 0, 48)
    oled.show()

def receive(lora):
    print("LoRa Receiver")
    broker = "broker.emqx.io"
    client = MQTTClient("PYCOM-X", broker)
    count = 1
    rssi =0

    while True:
        if lora.receivedPacket():
            try:
                payload = lora.readPayload().decode()
                rssi = lora.packetRssi()
                print("RX: {} | RSSI: {}".format(payload, rssi))
                mess="RSSI: " + str(rssi)
                wifista.connect()
                client.connect()
                client.publish(b"pycom-x/test", mess)
                disp(rssi)
```

```
                count=count+1
                sleep(15)
            except Exception as e:
                print(e)
```

```
..
RX: Long long Hello (324) | RSSI: -61
Already connected
('192.168.1.36', '255.255.255.0', '192.168.1.1', '192.168.1.1')
RX: Long long Hello (328) | RSSI: -58
Already connected
('192.168.1.36', '255.255.255.0', '192.168.1.1', '192.168.1.1')
RX: Long long Hello (332) | RSSI: -58
Already connected
('192.168.1.36', '255.255.255.0', '192.168.1.1', '192.168.1.1')
..
```

Note that only the **RSSI** value is transmitted to the **MQTT broker**.

### To do:

1. Test the above program.
2. Retrieve the payload value (data received from the SHT21 sensor) and send it in the MQTT message.
3. Write the same application (gateway) with the reception of LoRa packets by the **callback** function (interrupt)

## 5.2 LoRa-WiFi gateway (ThingSpeak)

The **LoRa-WiFi gateway** (ThingSpeak) will resend the data received on a LoRa link over a WiFi connection to a ThingSpeak server.

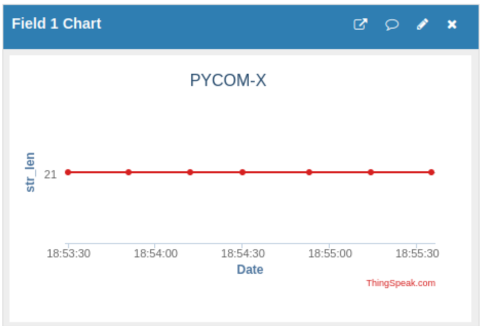
The following program allows you to receive LoRa packets and relay them over a WiFi connection to the ThingSpeak server.

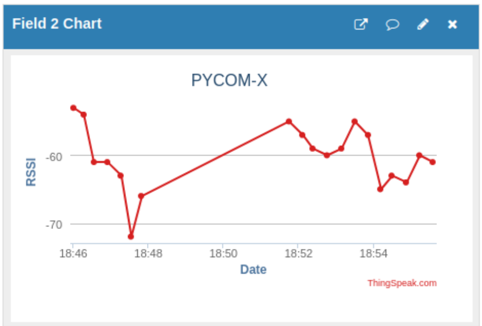
Note that we are using **`thingspeak.py`** library to send data to the server.

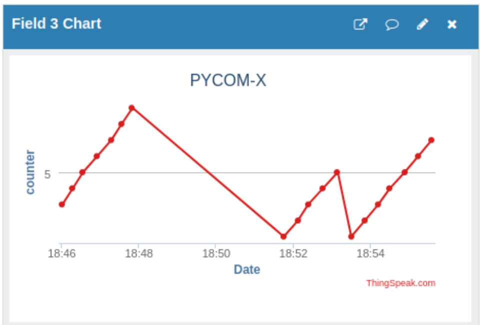
Here is the full code:

```
import wifista
from time import sleep
import machine, ssd1306
from machine import Pin, SoftI2C
import esp32
import thingspeak
from thingspeak import ThingSpeakAPI, Channel, ProtoHTTP
channel_living_room = "1538804"
field_temperature = "Temperature"
field_humidity = "Humidity"
thing_speak = ThingSpeakAPI([
    Channel(channel_living_room , 'YOX31M0EDKO0JATK', [field_temperature, field_humidity])],
    protocol_class=ProtoHTTP, log=True)

def disp(p):
    i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text("LoRa receiver", 0, 16)
    oled.text("Packet Nr:", 0, 32)
    oled.text("{}".format(p), 0, 48)
    oled.show()


def receive(lora):
    channel_living_room = "1538804"
    field_temperature = "Temperature"
    field_humidity = "Humidity"
    thing_speak = ThingSpeakAPI([
    Channel(channel_living_room , 'YOX31M0EDKO0JATK', [field_temperature, field_humidity])],
    protocol_class=ProtoHTTP, log=True)

    print("LoRa Receiver")

    wifista.connect()
    active_channel = channel_living_room
    temperature = 2.0
    humidity=3.0
    rssi =0

    while True:
        if lora.receivedPacket():
            try:
                payload = lora.readPayload().decode()
                rssi = lora.packetRssi()
                print("RX: {} | RSSI: {}".format(payload, rssi))
               # here - decompose payload into temperature and humidity fields !!
                wifista.connect()
                thing_speak.send(active_channel, {
                              field_temperature: temperature,
                              field_humidity: humidity
                              }) ts_payload)
                disp(payload)
                time.sleep(thing_speak.free_api_delay)
            except Exception as e:
                print(e)
```

The ThingSpeak chart corresponding to the execution sequence of our gateway.




## To do:

1. Complete and test the above program:
   Retrieve the payload value (data received from the SHT21 sensor) as temperature and humidity values.
2. Write the same application while receiving the LoRa packets by the **callback** function (interrupt)