Practical IoT Labs for Business Schools

SmartComputerLab

There are two kinds of people: those who understand technology and those who don't. People who understand technology can design and control the very structure of the world around them. People who don't understand it are controlled by those who do"

Mattan Griffel (Columbia Business School)

0. Introduction

Business schools are well aware of the necessity to teach the essential technological knowledge in the domain of **Computer Science**. Renown business schools have already introduced the teaching of programming for **Decision**, **Risk** and **Business Operations**.
Artificial Intelligence - **AI** and **Internet of Things** – **IoT** are becoming more and more important and are also progressively introduced on their syllabus.

The pedagogical content and the related pedagogical platform presented in this document are already in use in several "Grands Ecoles" from **Bachelor** level to **Master Degrees**.
The experience shows that the **practical approach is necessary**. The **theory about the technology** even for the most motivated and attentive enters *in one ear and out the other*.

Our pedagogical task is facilitated by the fact that programming skills in a language such as **Python** are already taught and practiced in secondary schools.

In this section we are going to introduce the overall architecture of **IoT infrastructure** and **devices**. The following section will present the IoT platform designed for "smart" (easy) teaching of IoT architectures covering all essential features of IoT technologies. The IoT platform with all supplementary components is provided by **SmartComputerLab** .
The pedagogical content including codes is available on `github.com/smartcomputerlab` server.

0.1 IoT Architecture

IoT Architecture may be seen as an addition to or an extension of the **Internet Infrastructure**.   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
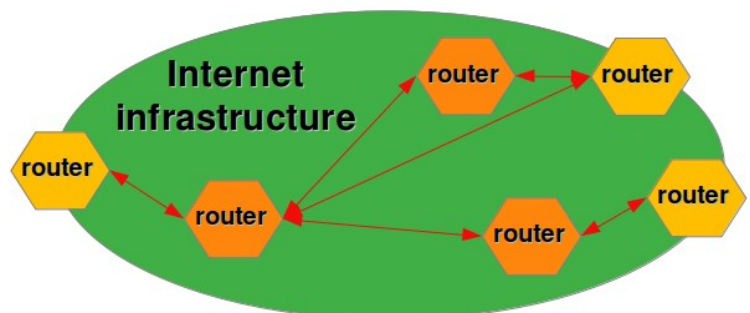side are processing and data centers ("clouds").

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.
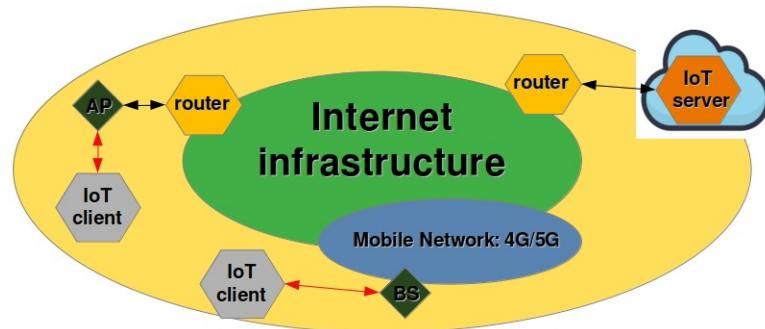

Fig 0.2 Entry points to Internet Infrastructure (**AP**, **BS**)

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**.
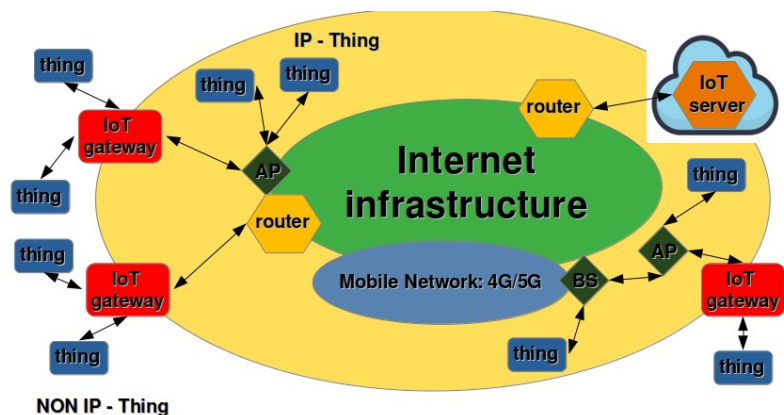

Fig 0.3 Communication links with IoT devices: **IP Things** and **NON-IP Things**

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.
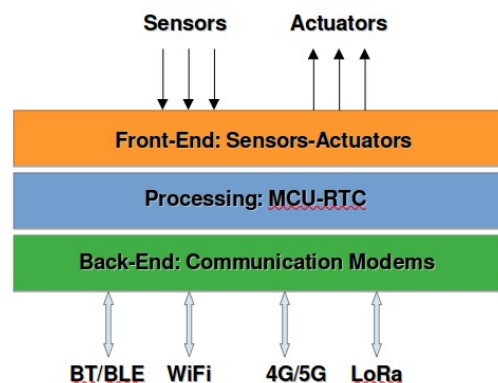

Fig 0.4 The **SoC** core of an IoT device

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.
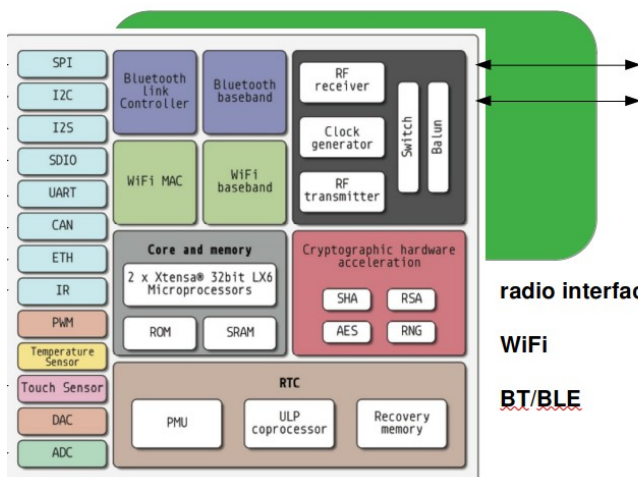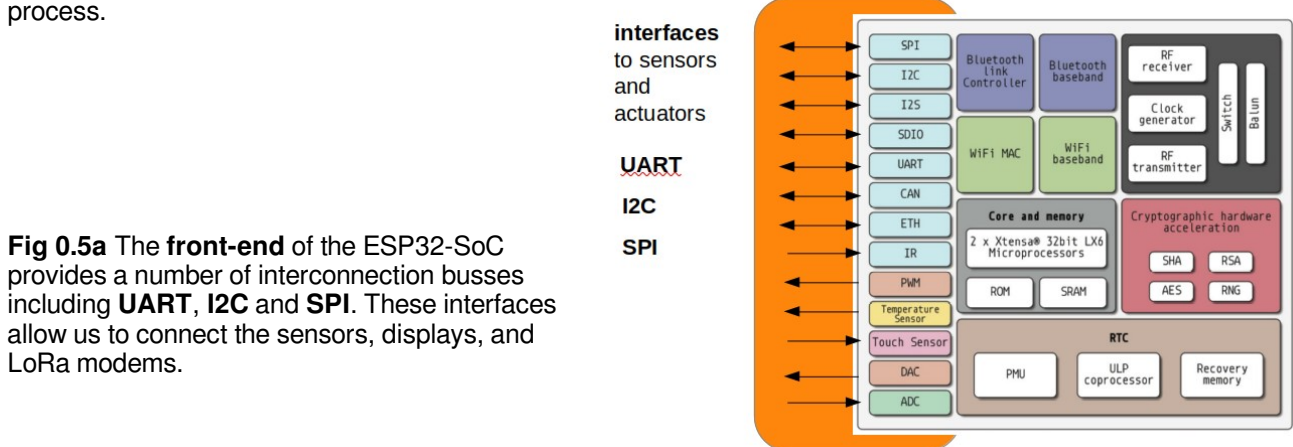
Fig 0.5a The **front-end** of the ESP32-SoC provides a number of interconnection busses including **UART**, **I2C** and **SPI**. These interfaces allow us to connect the sensors, displays, and LoRa modems.

Fig 0.5b The **back-end** provides the radio interfaces : **WiFi** and **BT/BLE** .

Fig 0.5c Finally the **MCU part** provides the processing power with 2 Xtensa processors and cryptographic accelerators.

ROM : read only

SRAM : read_write

processors
Xtensa LX6
ULP

cryptographic
accelerators
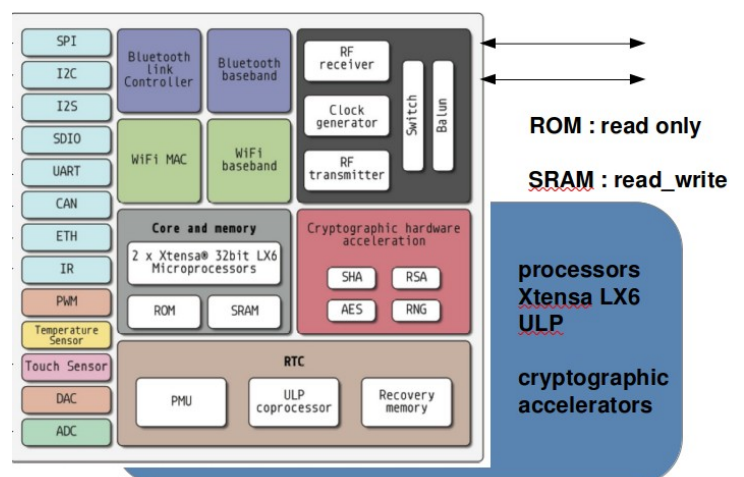
0.3 ESP32 LOLIN32 board

ESP32 SoCs are integrated into a number of development boards that include additional circuitry and communication modems. Our choice is the **LOLIN32** board which integrates an interface with **LiPo** batteries (3.7V)
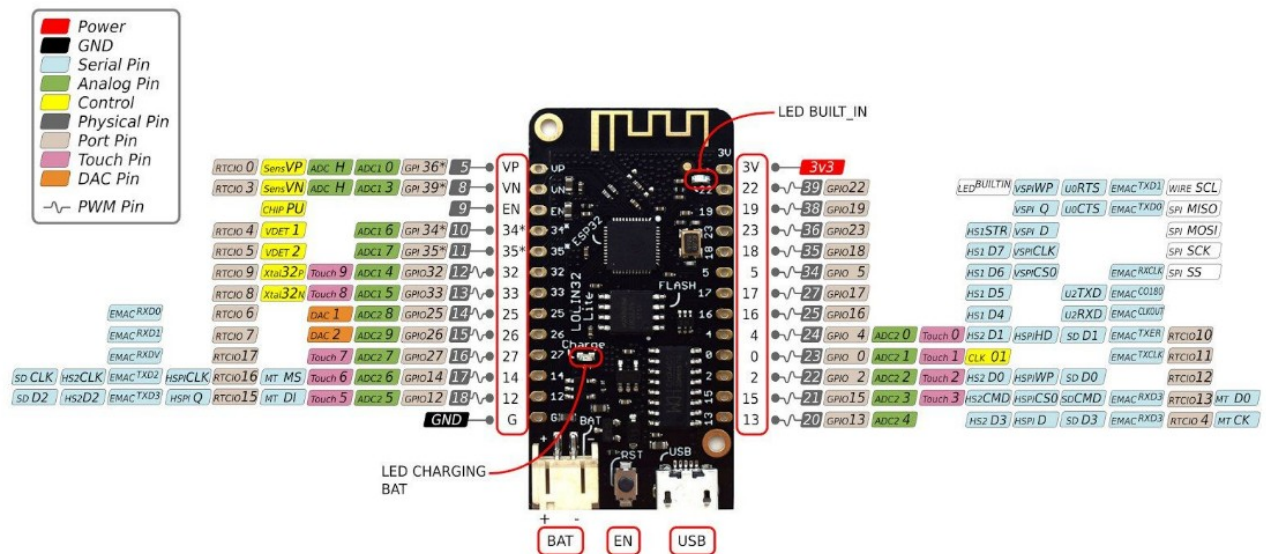
Fig 0.6 ESP32 LOLIN32 Lite MCU board and its **pinout**

As we can see in the figure above, the board exposes 2x13 pins. These pins carry the **I2C** (**SDA**-12,**SCL**-14), **UART** (**RX**-16,**TX**-17), **SPI** (**SCK**-18,**MISO**-19,**MOSI**-23) busses, plus control signals (**NSS**- 5,**RST**-15,**INT**-26,..). The **LED** is connected to pin 22.

0.4 IoT laboratories

In the presented **IoT laboratories** we will implement several IoT architectures integrating or using **IoT terminals** (**T**), **IoT gateways** (**G**), and **IoT broker-servers** (**B,S**) type **MQTT**, and **ThingSpeak**.

The development will be carried out on the **IoT DevKit** (**PYCOM-X)** from **SmartComputerLab**. **PYCOM-X** contains a **base card** to accommodate a central board with its **IoT SoC** (**S**ystem **o**n **C**hip) and a set of interfaces for sensors/actuators plus the expansion cards for additional sensors, actuators and modems. The central unit is a board equipped with an **ESP32 SoC** integrating **WiFi/BT/BLE** modems. In the introduction we have presented **Thonny IDE** (Integrated Development Environment) . Thonny IDE provides us with the set of tools to **edit**, and to **load** (flash) the micro-Python codes on PYCOM-X.

During the first lab we are going to set up the work environment and test the use of an **OLED display** and some **sensors** connected to the **I2C bus** (temperature/humidity/luminosity/movement).

The firmware provided for the labs contains all necessary libraries for interfacing and operating the sensors/displays.

The second lab is devoted to **getting started with the WiFi modem** integrated into the SoC ESP32. WiFi communication in **station mode** (**STA**) allows us to read the WEB pages and send the arguments to the WEB servers. We can also build a simple WEB server used to interact with our **smartphone**. The WEB server may operate on the local WiFi network or it may be associated the the Access Point (mode **AP**) created on the board.

The third lab is dedicated to the use of the **IoT broker – MQTT** and **ThingSpeak server**. **MQTT** is a simple **Client-Server publish/subscribe messaging transport protocol**. **ThingSpeak** type servers contain a **database** and offer a **graphical interface** for viewing recorded data. `ThingSpeak.com` is accessible free of charge, but its frequency of reception of messages and the number of messages are limited.

The fourth laboratory deals with **long-range radio links** based on **LoRa modems** with the communication range up to 3 Km (**NON-IP Things**). **LoRa** modems are integrated in the expansion boards provided for PYCOM-X. With LoRa we can send the **structured data** from sensors on terminal nodes to another PYCOM-X board with the same type of modem. The **destination node** may display the received data, as well as the signal strength corresponding to the received packet.

During the fifth lab we are going to integrate the WiFi and LoRa links in order to create complete applications with LoRa-WiFi **terminals** and the **gateways** to **MQTT** and **ThingSpeak** broker-servers.

We will develop two applications : one for the **LoRa-WiFi** type gateway (**MQTT** broker) and one for the **LoRa-WiFi** type gateway (**ThingSpeak** server).

0.5 IoT development platform

Efficient integration of the selected **ESP32 LOLIN32** board into IoT architectures requires the use of a development platform such as **IoT DevKit** provided by **SmartComputerLab**.

The **IoT DevKit** 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.
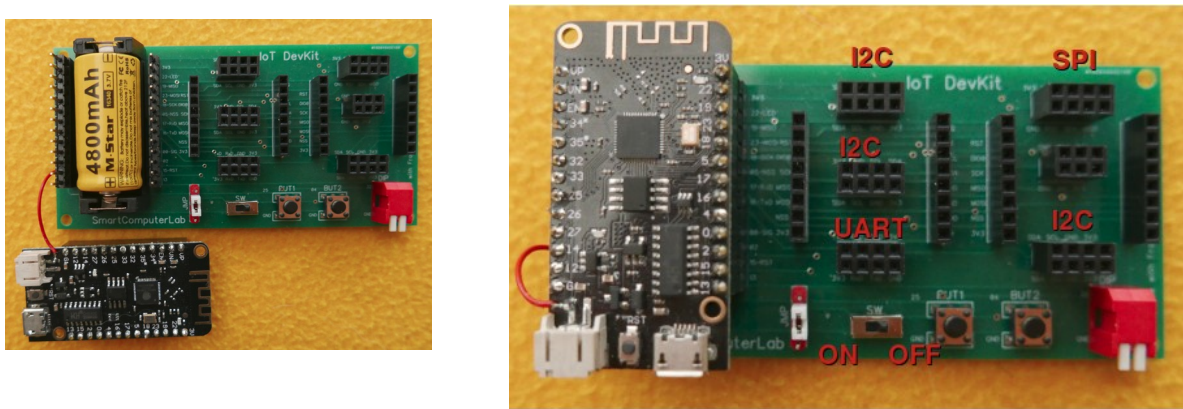

Figure 0.7 IoT DevKit base board with the battery and its integrated interfaces

The base card can directly accommodate several types of sensors or communication modems. To connect a more complete set of sensors/modems/displays, **expansion cards** are used.
The **jumper** (JMP) is used to connect a multi-meter and perform current measurements. In **low consumption** mode (**deep_sleep**) the current drops to a few tens of micro-amperes.
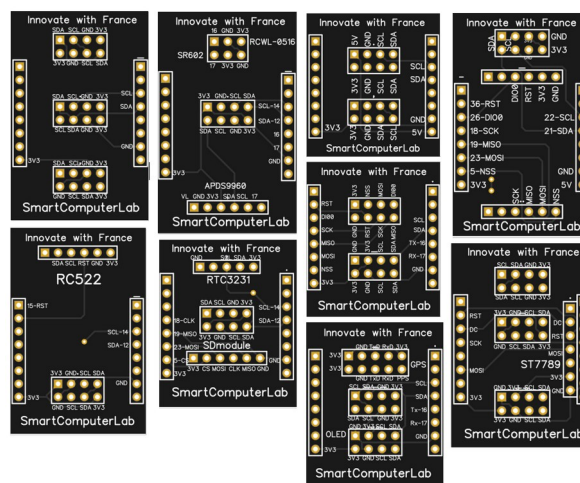
Below are some examples of expansion cards.

Figure 0.8 Expansion cards for various IoT components: sensors, displays, modems, ..

Note the expansion boards are **not required for the presented labs**, the base board provides all necessary interfaces to connect our sensors, screens and modems.

0.6 Software – Thonny IDE

0.6.1 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.
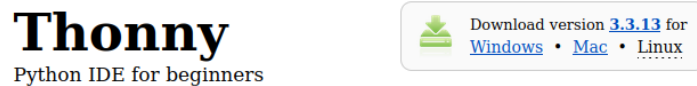
The installation of Thonny IDE includes the installation of Python 3.7 (built in).

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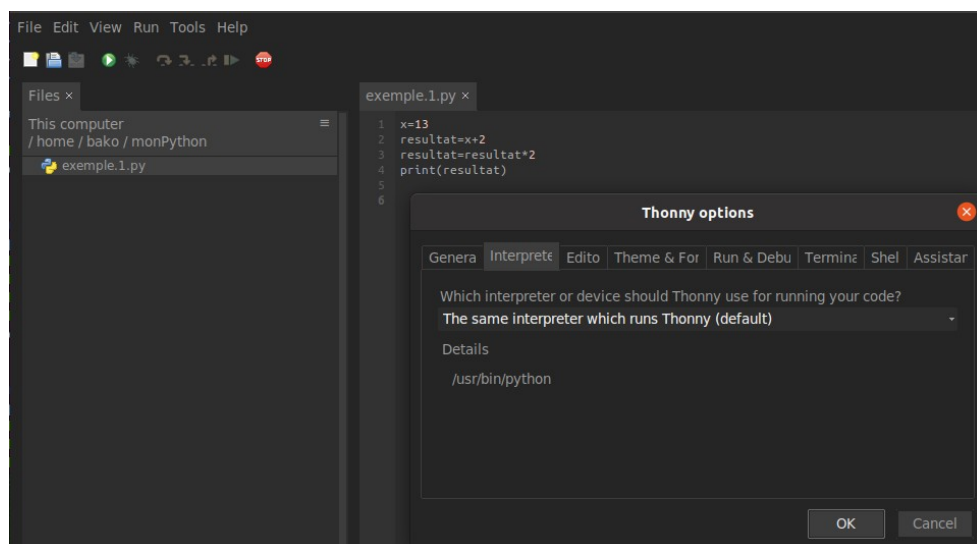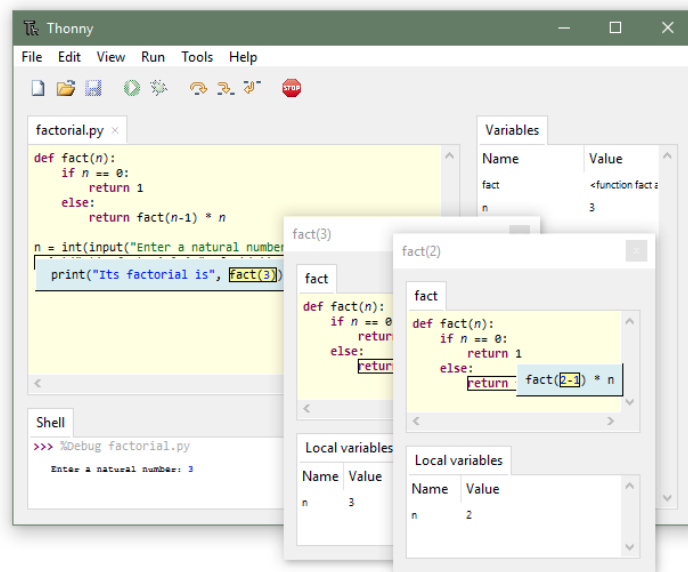

The above figure shows the **default selection of interpreter** running on your PC. With this interpreter you run your code directly on the PC. This feature is useful to start the programming in Python.
Above is an example of programming in Python. The 4 lines are saved in the file **`example.1.py`**.

0.6.2 Preparing the ESP32 LOLIN32 board

Thonny IDE allows you to install the **MicroPython interpreter** corresponding to our card (ESP32).
Go to **Tools→Options** then **Interpreter**.
To start, choose Interpreter **MicroPython (ESP32)** then go to **Install or update firmware**.
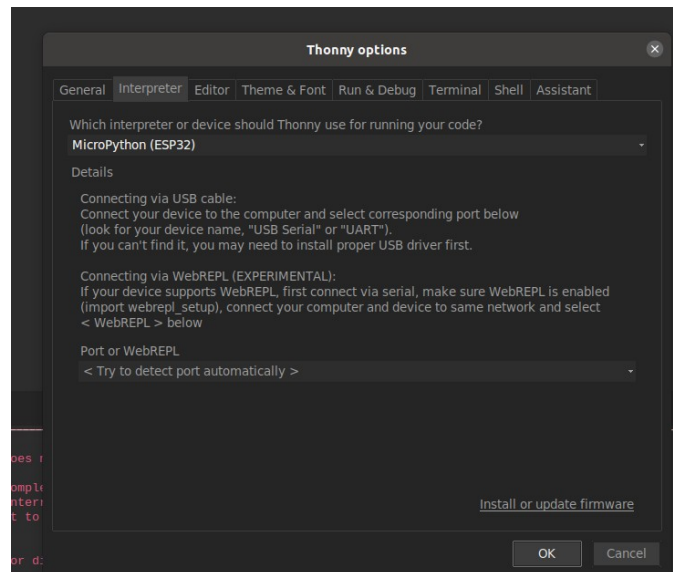
In the interpreter installation phase, you
must **connect** your card to the PC and choose the **USB interface**.
Then you have to **download the binary code** of the interpreter from the **GitHub** page:

https://github.com/smartcomputerlab/Practical-IoT-Audencia

For our card we have prepared **specific firmware**: that contains all necessary modules for the following IoT
laboratories.

firmware.220315.bin

We download the file then we indicate its
location in the frame as below.

The installation must be preceded by
Erase flash before installing.

Attention: Flash mode must be **Dual I/O**
(**dio**).

Then we click on **Install.**
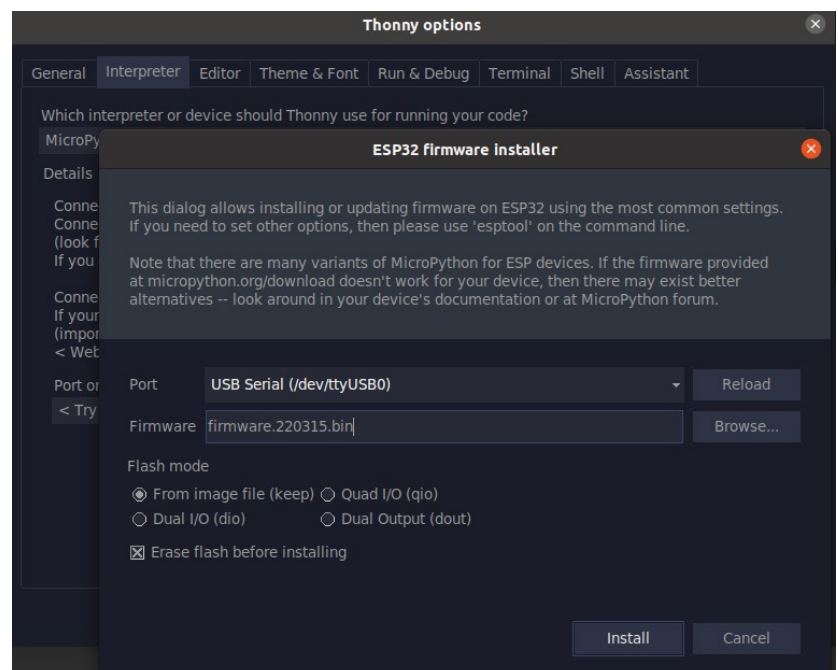
After loading the MicroPython interpreter on the ESP32 board we can connect our board with the USB cable to
our PC and launch Thonny IDE again.
First let us verify the presence of different modules in our firmware via :

help('modules') command.

```
>>> help('modules')
VL53L0X          gc              uasyncio/__init__ uos
__main__         inisetup        uasyncio/core     upip
_boot            machine         uasyncio/event    upip_utarfile
_onewire         maes            uasyncio/funcs    urandom
_thread          math            uasyncio/lock     ure
_uasyncio        micropyGPS      uasyncio/stream   urequests
_webrepl         micropython     ubinascii         uselect
apa106           mpu6050         ubluetooth        usocket
bh1750           neopixel        ucollections      ussl
bmp280           network         ucryptolib        ustruct
btree            ntptime         uctypes           usys
builtins         onewire         uerrno            utime
cmath            paj7620         uhashlib          utimeq
dht              sht21           uheapq            uwebsocket
ds18x20          ssd1306         uio               uzlib
esp              sx127x          ujson             webrepl
esp32            thingspeak      ulab              webrepl_setup
flashbdev        uSGP30          umqtt/robust      websocket_helper
framebuf         uarray          umqtt/simple      wifimgr
Plus any modules on the filesystem
>>>
```
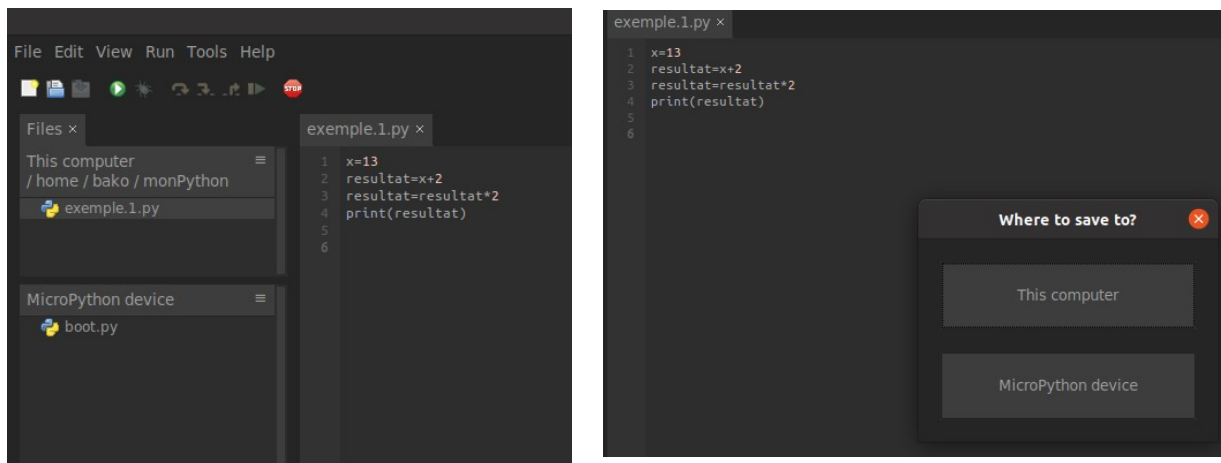
This time we go to **Tools->Options** to look for **Interpreter** and we will choose **MicroPython (ESP32)**.

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

Our 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**).
Let's do both.
Now we can start the "interpretation" execution of our program by pressing the **green arrow**.

0.6.4 First example – `x.led.blink.py`

In our first example we will run **Thonny** and edit a simple `blink.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 `x.led.blink.py`.

Note: note the name of the program which starts with `x`...) this prefix makes it possible to specify that the code is intended for the **PYCOM-X** board with **ESP32 LOLIN32**.

The code is:

```
import machine                  # the main library for MCU
from machine import Pin
from time import sleep
led=Pin(22,Pin.OUT)             # Pin LED number is 22
while True:
    led.value(not led.value())  # the value to display is complemented
    sleep(1.1)                  # waiting time in seconds
```
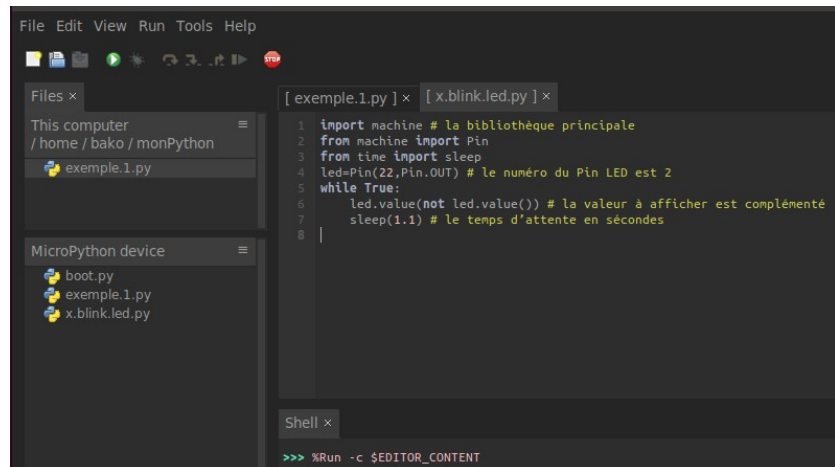

To do:

1. Launch Thonny IDE, edit the program and save it to the card
2. Modify the program, the value of `sleep()` and add a `print()`

0.6.5 Second example – `x.led.button.py`
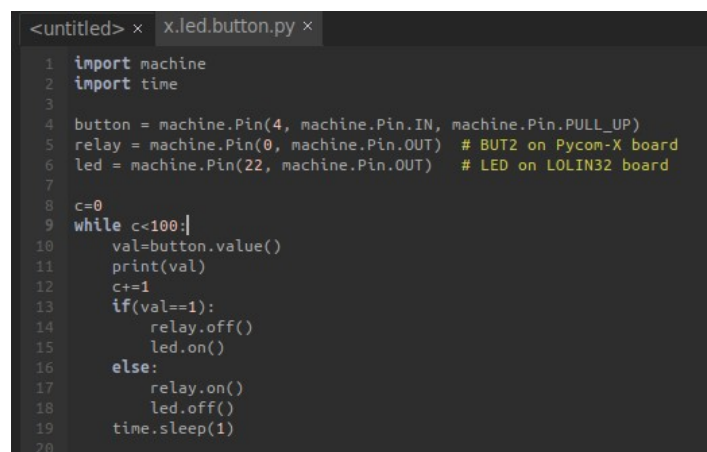
In the above example we test the use of integrated button – **BUT2**

To do:

1. Launch Thonny IDE, edit the program and save it to the card
2. Modify the program, the value of **counter** `c`
3. transform the loop `while` into loop `for` …

Lab 1

Sensor reading and data display (`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**.
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-14** which carries the **CLock** signal and **SDA-12** which carries the information (**D**ata, **A**ddress).

1.1 First example – data display on OLED screen

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

`ssd1306.py` module is already integrated into our firmware.

Edit the following code:

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

def disp(p1,p2):
    i2c = SoftI2C(scl=Pin(14), sda=Pin(12), 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)
```

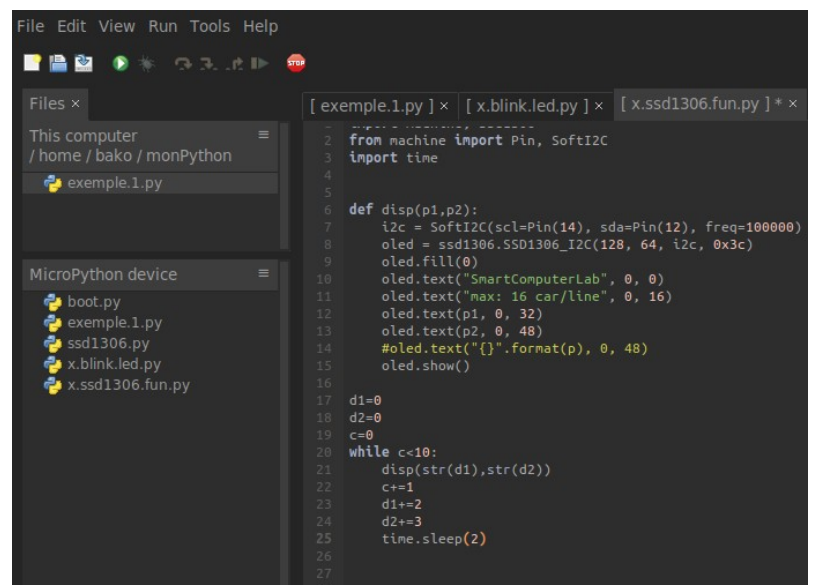And save it to the directory on your PC and re-flash to the card.
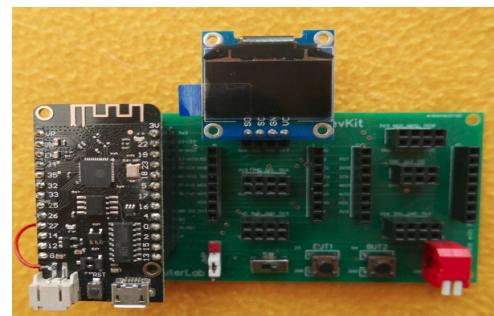
Fig 1.1 The **Thonny IDE** for storing/flashing a **MicroPython** program. And the configuration of the **IoT DevKit** board with an OLED screen (**ssd1306**).'

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

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

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).
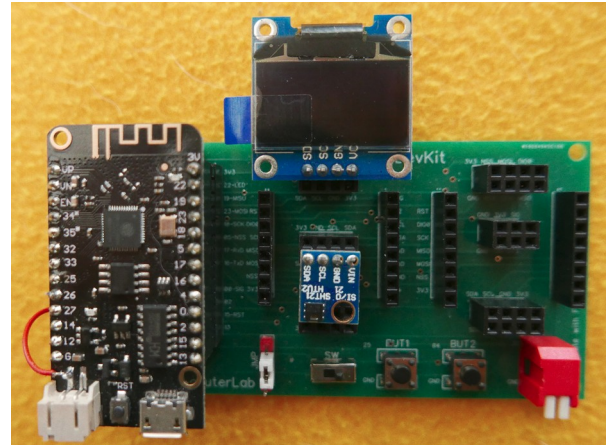

Fig 1.3 IoT DevKit with OLED display (**SSD1306**) and **SHT21** sensor on the same I2C bus

1.2.1 Preparing the code

For the SHT21 sensor, the reserved address is **0x40** in **hexadecimal** or **64** in **decimal**.

Full code:

```
import machine
import ssd1306
import sht21

from machine import Pin,I2C
import utime
sda=machine.Pin(12) # PYCOM-X
scl=machine.Pin(14) # PYCOM-X
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000) #I2C channel 0,pins,400kHz max
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)

def disp(d1,d2,d3):
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text(d1, 0, 16)
    oled.text(d2, 0, 32)
    oled.text(d3, 0, 48)
    oled.show()

c=0
disp("ESP32 mPython","Audencia-2022","WiFi/BLE/LoRa")
while c<100:
    temp = sht21.SHT21_TEMPERATURE(i2c)
    humi = sht21.SHT21_HUMIDITE(i2c)
    print ("Temperature = %.1f" % temp)
    print ("Humidity = %.1f" % humi)
    c+=1
    disp("SHT21 sensor","T: "+str(temp),"H: "+str(humi))
    utime.sleep_ms(1000)
```

To do:

1. Study and test the above code
2. Edit the following program, load it onto the board and check how it works.

```
import machine
i2c = machine.I2C(scl=machine.Pin(14), sda=machine.Pin(12))
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
```

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

In this example we use the **BH1750** light brightness sensor
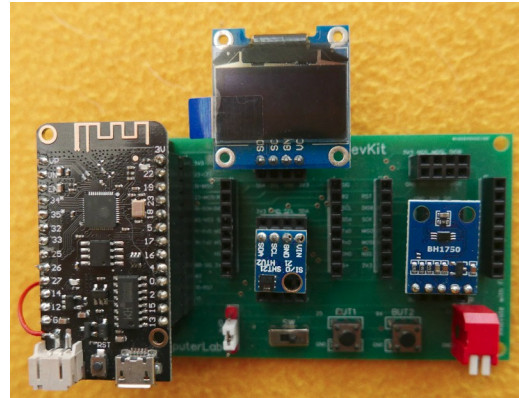
Fig 1.5 PYCOM-X with OLED display (SSD1306) and **SHT21** and **BH1750** sensors

Here is the code. The program performs a 100 readings.

```
import machine
from bh1750 import BH1750
import time

sda=machine.Pin(12) # PYCOM-X
scl=machine.Pin(14) # PYCOM-X
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. Transform the while loop into a function type **read_lum()**.
3. Add the OLED screen and complete the program to show the results.

1.4 Fourth example – reading a PIR sensor – SR602

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.
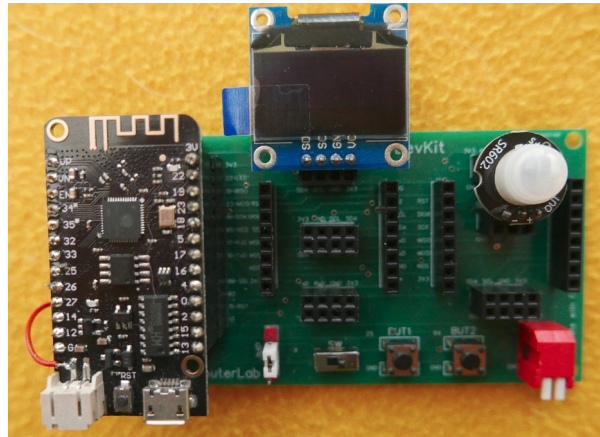
Fig 1.6 IoT DevKit with OLED display (SSD1306) and PIR motion/presence sensor: **SR602**

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

```
from machine import Pin
import time

ldr = Pin(0, 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.
2. Replace the **SR602 sensor** by **micro-radar sensor** : **RCWL-0516**
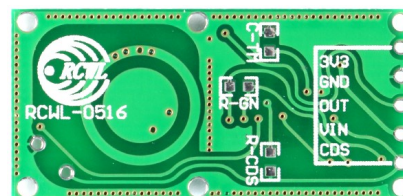
Lab 2

WiFi communication and WEB servers

In this lab we will study and experiment with the WiFi features integrated into the ESP32 SoC. First we are going to scan (scan) the networks available with **WiFi.scan.** Then we are going to build simple applications. The first one allows us to get/read the wall-time from NTP server; the following examples allows us to read WEB pages and to send arguments to WEB servers.

Finally we are going to build simple WEB servers operating on the local WiFi network or even create our own access points with simple WEB servers.

Note that for run these examples we can use of an Access Point at your **smartphone**.

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 **PYCOM-X** 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 provide a **static configuration** with a **static IP address** chosen by the user.
The following program demonstrates these features:

```
from machine import Pin

def scan():
    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()

def connect():
    import network
    ssid     = "SmartIoTLab"
    password  =  ""

    station = network.WLAN(network.STA_IF)

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

    station.active(True)
    station.connect(ssid,password)

    while station.isconnected() == False:
        pass

    print("Connection successful")
    print(station.ifconfig())
    return station

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

# scan()
# disconnect()
# connect()


>>> %Run –c $EDITOR_CONTENT
disconnected – start connection
Connection successful
('192.168.1.37', '255.255.255.0', '192.168.1.1', '8.8.8.8')
>>>
```

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)
```
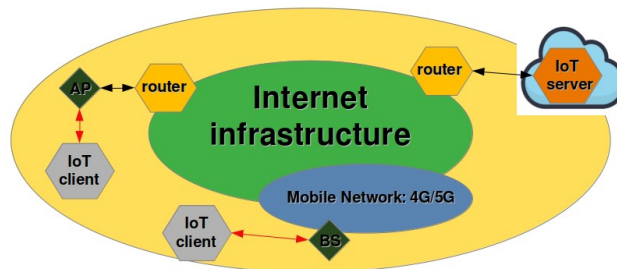

To do:

Use the Buzzer to signal each second/minute.

```
buz = Pin(0, Pin.OUT) # create input pin on GPIO2
..
buz.on()        # to buzz
buz.off()       # not to buzz
```

2.3.1 Using neopixel LED-ring

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.

The following is an example of code to show the usage of neopixel LEDs.

```
import time
import machine, neopixel
np = neopixel.NeoPixel(machine.Pin(0), 12)

def reset_ring():
    for i in range(12):
        np[i]=(0, 0, 0)  # RGB bytes – colors
    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:

Test the above code.
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.

```
import ntptime
import wifista
import time
import machine, neopixel
np = neopixel.NeoPixel(machine.Pin(0), 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(22,machine.Pin.OUT)

# 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 22.

```
from machine import Pin
from time import sleep

led=Pin(22,Pin.OUT)

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

2.3 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:
  • 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.

```python
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 PYCOM-X</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.

2.4 Simple WEB server – sending an order

In the previous example we read a value generated by the board . 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.

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

i2c = SoftI2C(scl=Pin(14), sda=Pin(12), 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 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

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("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. Analyze the program
1. Test the program with different messages to display

2.4.3 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=MyAP** and the default IP address: **192.168.4.1**; the default password is **"smarcomputertlab"**.

Here goes the code:

```
import machine,neopixel,network,ssd1306
from machine import Pin, SoftI2C
import usocket as socket
i2c = SoftI2C(scl=Pin(14), sda=Pin(12), freq=100000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
oled.fill(0)
oled.text("SmartComputerLab", 0, 0)
oled.show()
np = neopixel.NeoPixel(machine.Pin(0), 12)

def reset_ring():
    for i in range(12):
        np[i]=(0, 0, 0)   # RGB bytes - colors
    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()

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="MyAP"
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()
            set_all_green()
        if "GET /?led=red" in request:
            print("LED RED")
            oled.fill(0)
            oled.text("LED RED", 0, 0)
            oled.show()
            set_all_red()
        if "GET /?led=blue" in request:
            print("LED BLUE")
            oled.fill(0)
            oled.text("LED BLUE", 0, 0)
            oled.show()
            set_all_blue()
        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

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 'Message Queuing Telemetry Transport', 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.
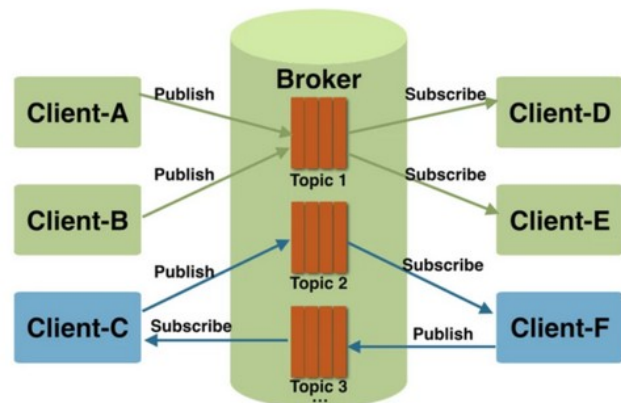The operation we have just detailed is illustrated in the diagram below.

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 PYCOM-X 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.

```
from umqtt.robust import MQTTClient
import machine
import wifista
import utime as time
import gc
wifista.connect()
broker = "broker.emqx.io"
#client = MQTTClient("pycom/esp32",
"broker.hivemq.com")
client = MQTTClient("PYCOM-X", broker)

def sub_cb(topic, msg):
    print((topic, msg))
    if topic == b'pycom-x/test' :
        print('ESP received '+ str(msg))

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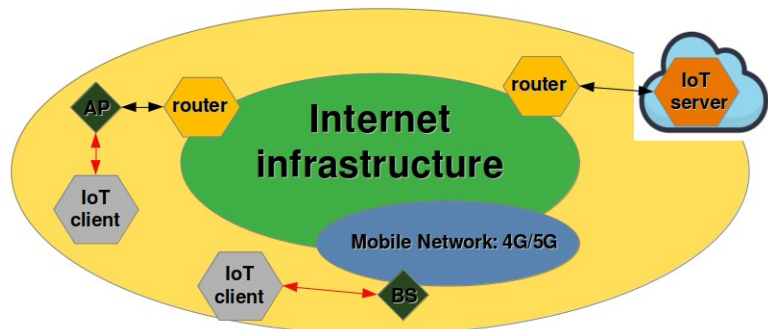
```
        count = count + 1
        time.sleep(20)
```

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

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
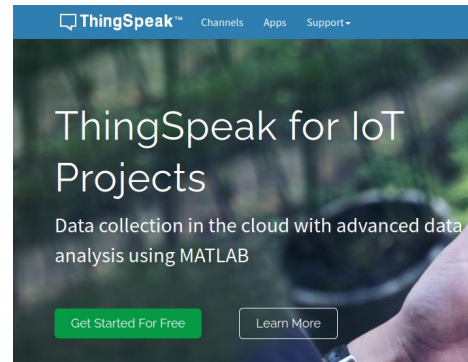
3.2.1 Preparation for sending data as MQTT messages

To be able to use `ThingSpeak.com`, you must create an **account** (free) and configure a **channel** - channel with its **fields** - fields. Then you have to retrieve the **channel identifier** and the **write and read keys**.
In our example we have created a channel number `1626377` with a write key `3IN09682SQX3PT4Z`.
In the following program we use **MQTT** type messages to send data in our channel with 2 fields (**temperature** and **humidity**).

Topic is a **string**:

```
topic = "channels/" + CHANNEL_ID + "/publish/" + WRITE_API_KEY
```

and the **message** itself is:

```
payload = "field1="+str(temp)+"&field2="+str(hum)
```

Attention:
We have prepared 4 public `Thingspeak.com` channels:
(**https://thingspeak.com/channels/public**)   for your work, they are:

```
SmartIoTBox-0: CHANNEL_ID – 1538804, WRITE_API_KEY: YOX31M0EDKO0JATK, READ_API_KEY: 20E9AQVFW7Z6XXOM
SmartIoTBox-1: CHANNEL_ID – 1626377, WRITE_API_KEY: 3IN09682SQX3PT4Z, READ_API_KEY: 9JVTP8ZHVTB9G4TT
SmartIoTBox-2: CHANNEL_ID – 1697980, WRITE_API_KEY: 4K897XNNHTW7I4NO, READ_API_KEY: V9GT1RN2FIP0YRAY
SmartIoTBox-3: CHANNEL_ID – 1697981, WRITE_API_KEY: A1G48I8FNSKGLRUB, READ_API_KEY: QKHDAYFFXPG1IGBK
```

Their **tags** for (**https://thingspeak.com/channels/public**) are:
smartiotlabs, zero ; smartiotlabs, one ; smartiotlabs, two ; smartiotlabs, three

Complete code:

```
from umqtt.simple import MQTTClient
import wifista, time
server = "mqtt.thingspeak.com"
client = MQTTClient("umqtt_client", server)
CHANNEL_ID = "1626377"
WRITE_API_KEY = "3IN09682SQX3PT4Z"
topic = "channels/" + CHANNEL_ID + "/publish/" + WRITE_API_KEY
temp =21.5
hum =55.7

for i in range(60):
    wifista.connect()
    payload = "field1="+str(temp)+"&field2="+str(hum)
    client.connect()
    client.publish(topic, payload)
    client.disconnect()
    temp=temp+1.0
    hum=hum+2.0
    time.sleep(15)
```
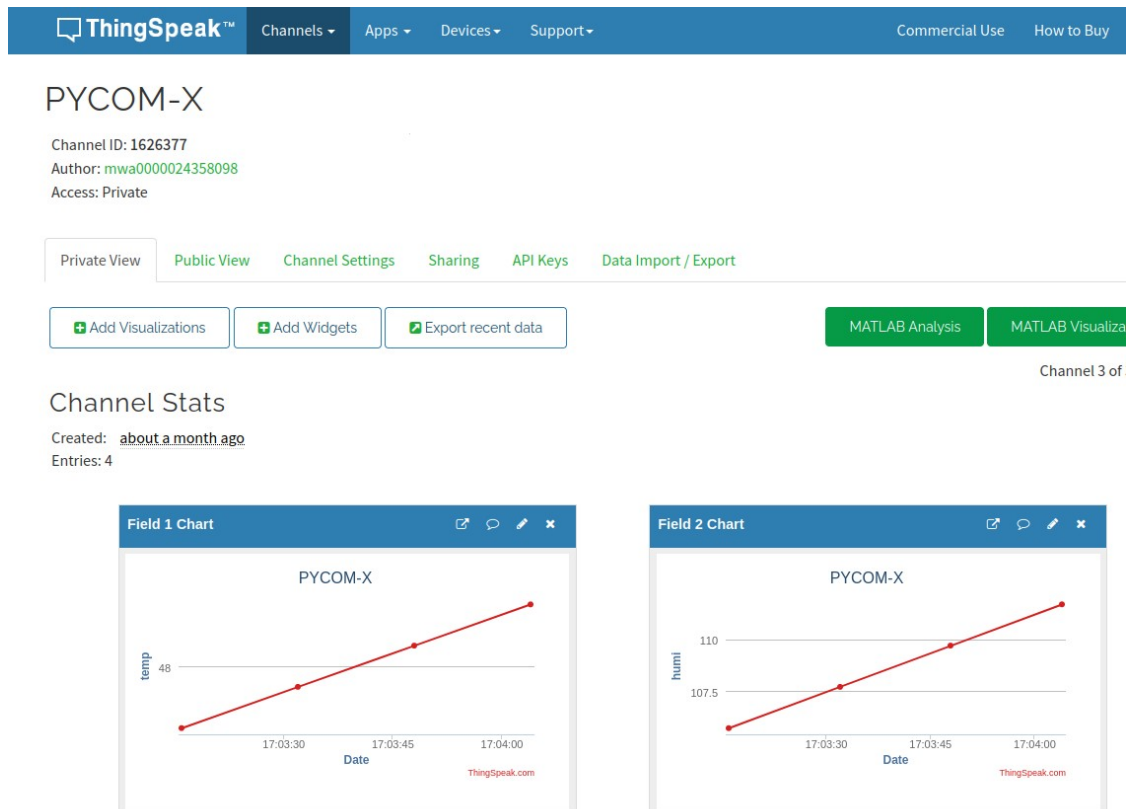
Fig. 3.2 ThingSpeak diagrams for our application program.

To do:

1. Log in to **your `ThingSpeak.com`** account and test the program
2. Add a sensor and post the values captured in a topic with the channel and the corresponding fields

3.2.2 Preparation for sending data as simple HTTP requests

The easiest 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()

wifista.connect()
t=22.2
h=44.4
urlkey='https://api.thingspeak.com/update?api_key=3IN09682SQX3PT4Z'
fields='&field1='+str(t)+'&field2='+str(h)
#http_get('https://api.thingspeak.com/update?api_key=3IN09682SQX3PT4Z&field1=0')
http_get(urlkey+fields)
time.sleep(15)
http_get('https://api.thingspeak.com/channels/1626377/fields/2/last.json?api_key=9JVTP8ZHVTB9G4TT')
```

Execution result:

```
%Run -c $EDITOR_CONTENT
Already connected
('192.168.1.36', '255.255.255.0', '192.168.1.1', '192.168.1.1')
Already connected
('192.168.1.36', '255.255.255.0', '192.168.1.1', '192.168.1.1')
update?api_key=3IN09682SQX3PT4Z&field1=22.2&field2=44.4
api.thingspeak.com
HTTP/1.1 200 OK
Date: Sun, 06 Feb 2022 16:09:35 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 2
Connection: close
Status: 200 OK
X-Frame-Options: SAMEORIGIN
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS, DELETE, PATCH
Access-Control-Allow-Headers: origin, content-type, X-Requested-With
Access-Control-Max-Age: 1800
ETag: W/"c2356069e9d1e79ca924378153cfbbfb"
Cache-Control: max-age=0, private, must-revalidate
X-Request-Id: 581bea14-20c5-46a9-8d9c-8844d58aef13
X-Runtime: 0.104615
X-Powered-By: Phusion Passenger 4.0.57
Server: nginx/1.9.3 + Phusion Passenger 4.0.57

24channels/1626377/fields/2/last.json?api_key=9JVTP8ZHVTB9G4TT
api.thingspeak.com
HTTP/1.1 200 OK
Date: Sun, 06 Feb 2022 16:09:51 GMT
Content-Type: application/json; charset=utf-8
Connection: close
Status: 200 OK
X-Frame-Options: SAMEORIGIN
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS, DELETE, PATCH
Access-Control-Allow-Headers: origin, content-type, X-Requested-With
Access-Control-Max-Age: 1800
Cache-Control: max-age=7, private
ETag: W/"3c729f09e7a7d82847c98c647b419168"
X-Request-Id: fd0e3003-2be3-49d6-9cd5-05d06bbfaec5
X-Runtime: 0.005083
X-Powered-By: Phusion Passenger 4.0.57
Server: nginx/1.9.3 + Phusion Passenger 4.0.57

{"created_at":"2022-02-06T16:09:35Z","entry_id":24,"field2":"44.4"}
```
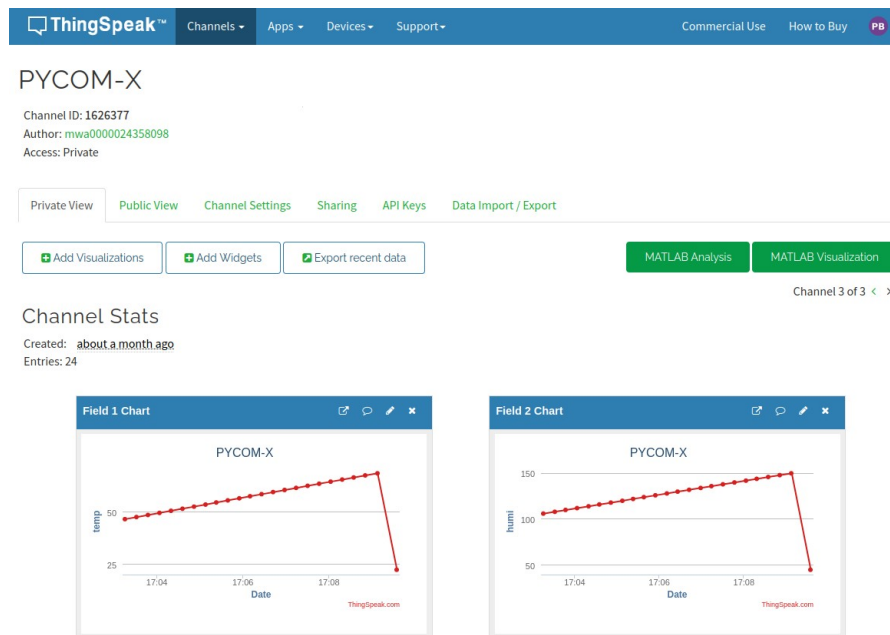
To do:

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**

3.2.3 Preparation for sending data with `thingspeak.py` library

In this example we will use a `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 **PYCOM-X** card.

Then edit the following code:

```
import machine
import time
import wifista
import thingspeak

from thingspeak import ThingSpeakAPI, Channel, ProtoHTTP
channel_living_room = "1626377"
field_temperature = "Temperature"
field_humidity = "Humidity"

thing_speak = ThingSpeakAPI([
    Channel(channel_living_room , '3IN09682SQX3PT4Z', [field_temperature, field_humidity])],
    protocol_class=ProtoHTTP, log=True)

wifista.connect()
active_channel = channel_living_room
temperature = 21.4
humidity=33.7
while True:
    thing_speak.send(active_channel, {
        field_temperature: temperature,
        field_humidity: humidity
    })

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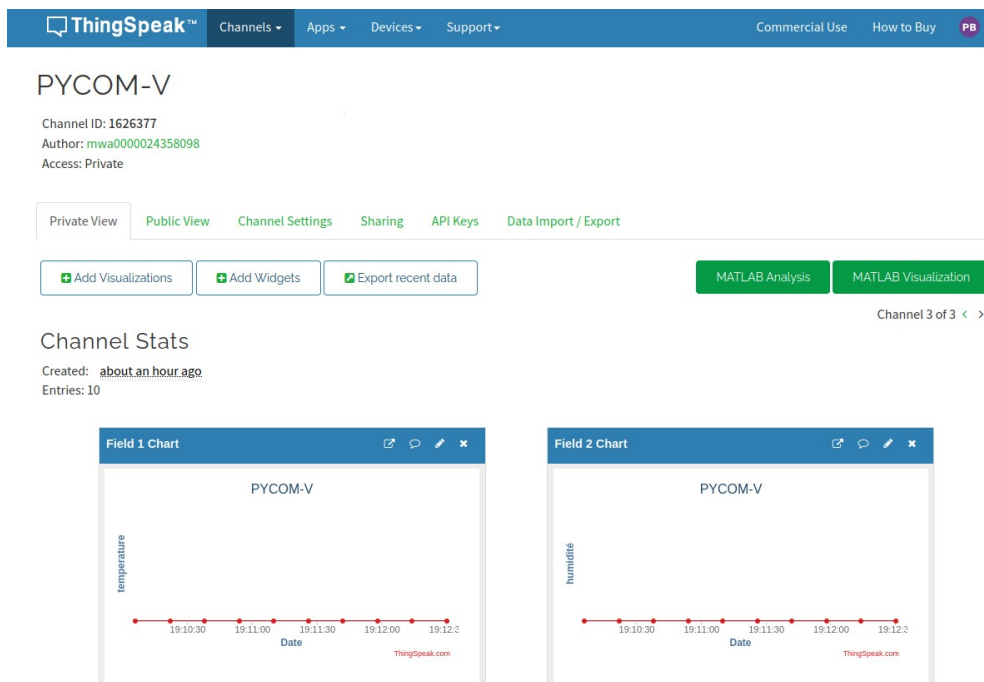
In the last statement you can note the delay value of `sleep()` required for a **free** account:

```
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

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 objects. 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**
LoRa communication on our **IoT DevKiT** (PYCOM-X) is provided by an additional – **LoRa modem** connected via **SPI** bus.

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

```python
class SX127x:

    default_parameters = {
        "frequency": 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": True,
        "invert_IQ": False,
    }
```

The default (radio) **settings** can be changed through the functions available in the same library:
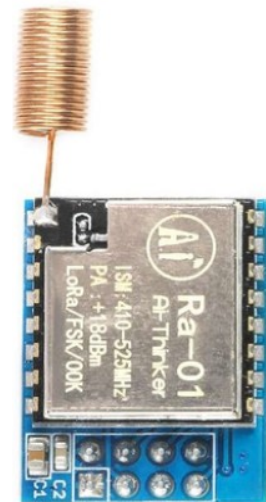
```
            self.setFrequency(self.parameters["frequency"])
            self.setSignalBandwidth(self.parameters["signal_bandwidth"])
            # set LNA boost
            self.writeRegister(REG_LNA, self.readRegister(REG_LNA) | 0x03)
            # set auto AGC
            self.writeRegister(REG_MODEM_CONFIG_3, 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 **PYCOM-X DevKit**
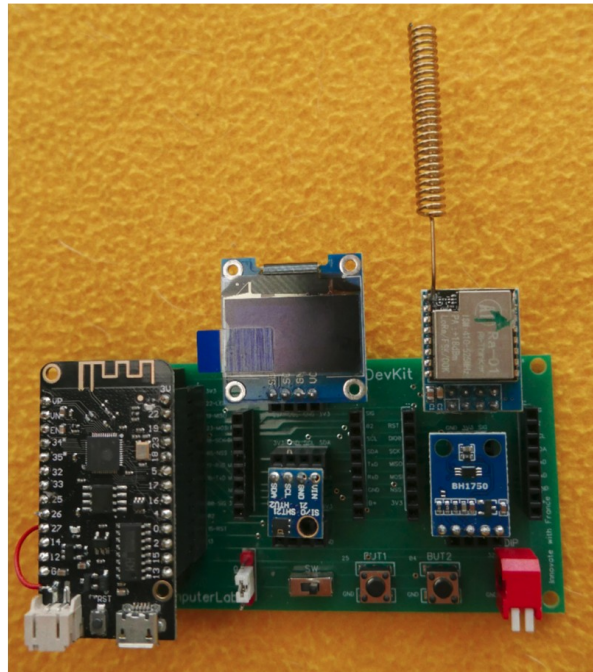
4.3 Main program

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

lora_pins = {  # pycom-x
    'dio_0':26,
    'ss':5,        #
    'reset':15,  #
    'sck':18,
    'miso':19,
    'mosi':23,
}

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 = '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)
```

Dans le **programme** ci-dessus nous avons choisie les fonctions permettant de configurer le code comme
Sender – LoRaSender.

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

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.

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

def disp(c):
    i2c = SoftI2C(scl=Pin(14), sda=Pin(12), 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.

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

def disp(p):
    i2c = SoftI2C(scl=Pin(14), sda=Pin(12), 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(14), sda=Pin(12), 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

lora_pins = {  # pycom-x
    'dio_0':26,
    'ss':5,        #
    'reset':15,   #
    'sck':18,
    'miso':19,
    'mosi':23,
}

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(14), sda=Pin(12), 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 use here the notion of **topic** (**MQTT**) and **messages**. The **topic** is a character string including the **channel number**, the **publish** command and the **write key**.

```
topic = "channels/" + CHANNEL_ID + "/publish/" + WRITE_API_KEY
```

The message is the **payload** with the **fields** associated with each **value**:

```
payload = "field1="+str(temp)+"&field2="+str(hum)+"&field3="+str(rssi)
```

Here is the full code:

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

CHANNEL_ID = "1626377"
WRITE_API_KEY = "3IN09682SQX3PT4Z"

def disp(p):
    i2c = SoftI2C(scl=Pin(14), sda=Pin(12), 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")
   # wifista.disconnect()
    wifista.connect()
    server = "mqtt.thingspeak.com"
    client = MQTTClient("umqtt_client", server)
    topic = "channels/" + CHANNEL_ID + "/publish/" + WRITE_API_KEY

    temp =21.5
    hum =55.7
    count = 1
    rssi =0

    while True:
        if lora.receivedPacket():
            try:
                payload = lora.readPayload().decode()
                rssi = lora.packetRssi()
                print("RX: {} | RSSI: {}".format(payload, rssi))
                wifista.connect()
                ts_payload = "field1="+str(len(str(payload)))+"&field2="+str(rssi)
+"&field3="+str(count)
                client.connect()
                client.publish(topic, ts_payload)
                client.disconnect()
                disp(payload)
                count=count+1
                sleep(15)
            except Exception as e:
                print(e)
```

The **ThingSpeak** chart corresponding to the execution sequence of our gateway.
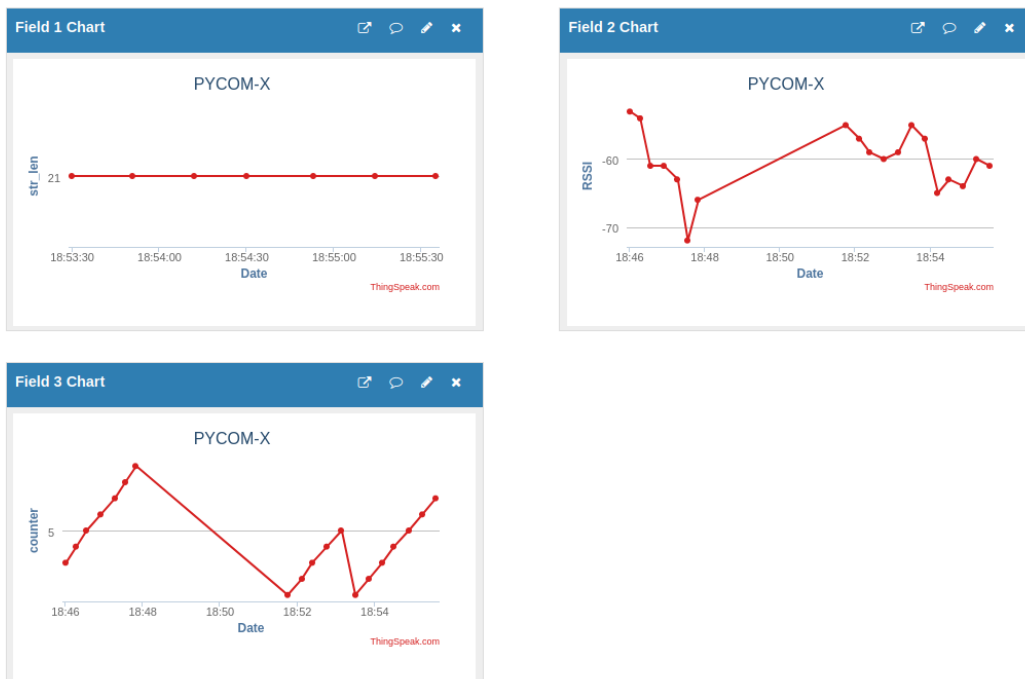
To do:

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

Lab 6

More sensors and modems

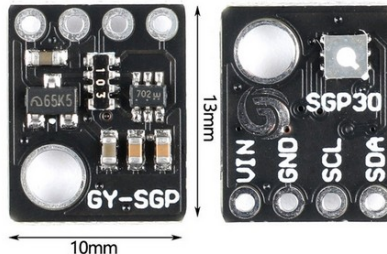In this lab we are going to study and use more advanced sensor.

6.1 SGP30 : Air Quality Sensor - VOC and eCO2

```
import uSGP30
from machine import I2C, Pin
import machine, time

i2c = I2C(0, sda=Pin(12), scl=Pin(14))

sgp30 = uSGP30.SGP30(i2c)
c=0

while c<100:
    co2eq_ppm, tvoc_ppb = sgp30.measure_iaq()
    print(co2eq_ppm, tvoc_ppb)
    time.sleep(2)
    c+=1
```

6.2 MPU-6050 : 3 Axis Gyro With Accelerometer Sensor

```
from machine import I2C
from machine import Pin
from machine import sleep

import mpu6050
i2c = I2C(scl=Pin(14), sda=Pin(12))      #initializing the I2C method for ESP32

mpu= mpu6050.accel(i2c)

while True:
 mpu.get_values()
 print(mpu.get_values())
 sleep(500)
```
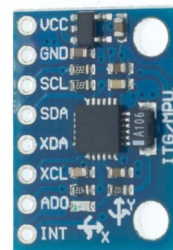
To do:

1. Add OLED display to show the capture data

6.3 PAJ76620: Hand gesture recognition sensor

```python
import machine, time, ssd1306, paj7620

i2c = machine.I2C(scl=machine.Pin(14), sda=machine.Pin(12))
oled = ssd1306.SSD1306_I2C(128, 64, i2c)  # 128 x 64 pixels
g = paj7620.PAJ7620(i2c = i2c)
while True:
    geste = g.gesture()
    oled.fill(0)
    if geste == 1:
        oled.text("Forward", 0, 0)
    elif geste == 2:
        oled.text("Backward", 0, 0)
    elif geste == 3:
        oled.text("Right", 0, 0)
    elif geste == 4:
        oled.text("Left", 0, 0)
    elif geste == 5:
        oled.text("Up", 0, 0)
    elif geste == 6:
        oled.text("Down", 0, 0)
    elif geste == 7:
        oled.text("Clockwise", 0, 0)
    elif geste == 8:
        oled.text("anti-clockwise", 0, 0)
    elif geste == 9:
        oled.text("wave", 0, 0)
    else:
        oled.text("nothing", 0, 0)
    oled.show()
    time.sleep(.5)
```

6.4 VL53L0X – Time of Flight (distance) laser module

```python
import time
from machine import Pin
from machine import I2C
import VL53L0X


i2c= I2C(0, sda=Pin(12), scl=Pin(14))

# Create a VL53L0X object
tof = VL53L0X.VL53L0X(i2c)

while True:
# Start ranging
    tof.start()
    tof.read()
    print(tof.read())
    tof.stop()
```

6.5 BMP280 – barometric pressure sensor

```
from machine import Pin
from machine import I2C
from bmp280 import BMP280

i2c_bus = I2C(0, sda=Pin(12), scl=Pin(14))
bmp = BMP280(i2c_bus)

# 1. To get environment temperature (^C):
print(bmp.getTemp())
# 2. To get Pressure (hPa):
print(bmp.getPress())
```

To do:

1. Add OLED display to show the capture data

6.6 GY-NEO6MV2 GPS module

```
import time
import machine
from micropyGPS import MicropyGPS
import ssd1306
import _thread
import time

WIDTH  = 128
HEIGHT = 64

def main():
    i2c = machine.I2C(scl=machine.Pin(14), sda=machine.Pin(12))
    dsp = ssd1306.SSD1306_I2C(WIDTH, HEIGHT, i2c, 0x3c, False)
    uart = machine.UART(1,rx=16,tx=17,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("Lon:{}{: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()
```

To do:

1. Test the above code. Beware of long initialization time required to receive satellite data.

6.7 AT24C256 : EEPROM module (32 KB)


```
import machine
from machine import Pin,I2C
import os
from eeprom_i2c import EEPROM, T24C256

sda=machine.Pin(12) # PYCOM-X
scl=machine.Pin(14) # PYCOM-X
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000) #I2C channel 0,pins,400kHz max
eep = EEPROM(i2c, T24C256)

eep[2000] = 42
eep[2001] = 2*42
print(eep[2000])  # Return an integer
print(eep[2001])  # Return an integer

%Run -c $EDITOR_CONTENT
1 chips detected. Total EEPROM size 32768bytes.
42
84


import machine
from machine import Pin,I2C
import os
from eeprom_i2c import EEPROM, T24C256

sda=machine.Pin(12) # PYCOM-X
scl=machine.Pin(14) # PYCOM-X
```

```
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000) #I2C channel 0,pins,400kHz max
eep = EEPROM(i2c, T24C256)
eep[2000:2002] = bytearray((42, 43))
print(eep[2000:2002])  # Returns a bytearray

>>> %Run -c $EDITOR_CONTENT
1 chips detected. Total EEPROM size 32768bytes.
bytearray(b'*+')
```

6.8 NeoPixels : WS2812 – 12 LEDS

6.8.1 Simple test

```
import machine, neopixel
np = neopixel.NeoPixel(machine.Pin(0), 12)

np[0] = (255, 0, 0) # set to red, full brightness
np[1] = (0, 128, 0) # set to green, half brightness
np[2] = (0, 0, 64)  # set to blue, quarter brightness

np.write()
```

6.8.2 12 LED Clock with NTP protocol

The following example exploit the Network Time Protocol to get the current time via a WiFi connection.
The received data is transformed into the pixels on 12 LED ring. The hours are displayed directly, the minutes
and the seconds every fifth unit (0,5,10, ..) to cover 60 min and 60 sec.

```
import ntptime
import wifista
import time
import machine, neopixel
np = neopixel.NeoPixel(machine.Pin(0), 12)

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

def set_clock(h,m,s,l):
    reset_clock()
    np[s] = (0, 0, 1)  # set to blue, quarter brightness
    np[m] = (0, 1, 0) # set to green, half brightness
    np[h] = (1, 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=(min/5+7)%12  # +7 ring shift
    ledsec=(sec/5+7)%12  # +7 ring shift
    ledhour=(hour+9)%12  # +7 ring shift +2 GMT time
    print(int(ledhour),int(ledmin),int(ledsec))
    set_clock(int(ledhour),int(ledmin),int(ledsec),100)
    time.sleep(5)
```

6.9 CardKB – micro keyboard

6.9.1 Test keyboard program

```
import machine
from machine import I2C
from cardkb import *

sda=machine.Pin(12) # PYCOM-X
scl=machine.Pin(14) # PYCOM-X
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000)
#I2C channel 0,pins,400kHz max

MOD_TEXT = { MOD_NONE : 'none', MOD_SYM : 'SYMBOL',
MOD_FN : 'FUNCTION'}

keyb = CardKB( i2c )

# Ctrl key as text
CTRL_NAME =




{0xB5:'UP',0xB4:'LEFT',0xB6:'DOWN',0xB7:'RIGHT',0x1B:'ESC',0x09:'TAB',0x08:'BS',0x7F:'DEL',0x0D:'CR'
}

print( 'Keycode | Ascii | Modifier' )
print( '--------------------------')
while True:
        keycode,ascii,modifier = keyb.read_key()
        if keycode == None:
                continue # restart the loop
        if keyb.is_ctrl( keycode ): # Ctrl char cannot be displayed safely!!!
                if keycode in CTRL_NAME:
                        ascii = CTRL_NAME[keycode]
                else: # we do not know the name for that KeyCode
                        ascii = 'ctrl' # so we replace it with "ctrl" string

        print( "  %5s | %5s | %s" %(hex(keycode), ascii, MOD_TEXT[modifier]) )
```
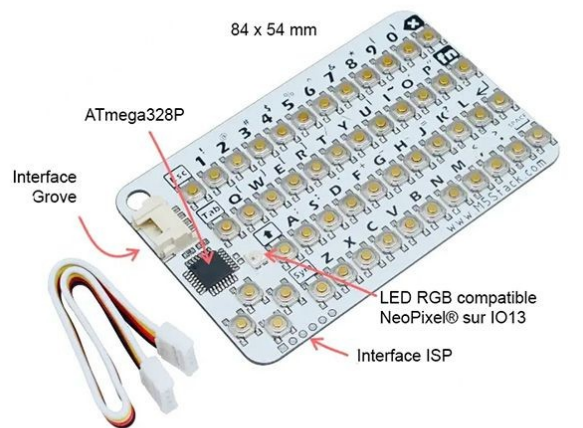
6.9.2 Read character/string from keyboard

```
import machine
from machine import I2C
from cardkb import *

sda=machine.Pin(12) # PYCOM-X
scl=machine.Pin(14) # PYCOM-X
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000) #I2C channel 0,pins,400kHz max

s = ''

keyb = CardKB( i2c )
while True:
        ch = keyb.read_char( wait=True ) # Wait for a key to be pressed (by default)
        if ord(ch) == RETURN:
                print( 'Return pressed! Blank string')
                s = ''
        elif ord(ch) == BACKSPACE:
                s = s[:-1] # remove last char
        else:
                s = s + ch # Add the char to the string
        print( s )
```

6.10 IR temperature sensor: MLX90614

6.10.1 Simple reading from temperature IR sensor.

Attention the I2C bus speed-clock must be set to 100 kHz max

```
import machine
from machine import I2C
import mlx90614
import time

sda=machine.Pin(12) # PYCOM-X
scl=machine.Pin(14) # PYCOM-X
i2c=machine.I2C(0,sda=sda, scl=scl, freq=100000) #I2C channel 0,pins,100kHz max

sensor = mlx90614.MLX90614(i2c)
time.sleep(1)
print(sensor.read_ambient_temp())
time.sleep(1)
print(sensor.read_object_temp())
time.sleep(1)
if sensor.dual_zone:
    print(sensor.object2_temp)
```

6.10.2 Continuous reading from temperature IR sensor.

```
import machine
from machine import I2C
import mlx90614
import time

sda=machine.Pin(12) # PYCOM-X
scl=machine.Pin(14) # PYCOM-X
i2c=machine.I2C(0,sda=sda, scl=scl, freq=100000) #I2C channel 0,pins,100kHz max

sensor = mlx90614.MLX90614(i2c)
while True:
        print(sensor.read_ambient_temp(), sensor.read_object_temp())
        time.sleep_ms(1000)
```

6.11 Radio modem: NRF24I01

The nRF24L01+ is a single chip 2.4GHz transceiver with an embedded baseband protocol engine suitable for ultra low power wireless applications. The nRF24L01+ is designed for operation in the world wide ISM frequency band at 2.400 – 2.4835GHz.
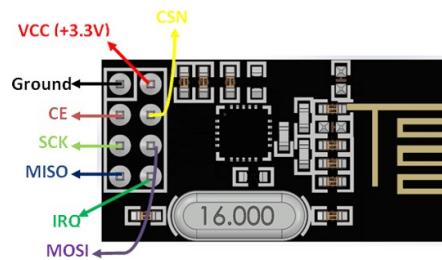
The test code with master and slave functions for **nrf24l01** module. Portable between MicroPython targets. The **Slave node** pause between receiving data and checking for further packets. It pauses an additional **_SLAVE_SEND_DELAY** ms after receiving data and before transmitting to allow the (remote) **Master node** time to get into receive mode.

```
"""Test for nrf24l01 module.  Portable between MicroPython targets."""

import usys
import ustruct as struct
import utime
from machine import Pin, SPI
from nrf24l01 import NRF24L01
from micropython import const

_RX_POLL_DELAY = const(15)
_SLAVE_SEND_DELAY = const(10)
led=Pin(22,Pin.OUT)

# lora_pins = {
#     'dio_0':26,
#     'ss':5,
#     'reset':15,   #RST
#     'sck':18,
#     'miso':19,
#     'mosi':23,
# }

cfg = {"spi": -1, "miso": 19, "mosi": 23, "sck": 18, "csn": 5, "ce": 15}

# Addresses are in little-endian format. They correspond to big-endian
# 0xf0f0f0f0e1, 0xf0f0f0f0d2
pipes = (b"\xe1\xf0\xf0\xf0\xf0", b"\xd2\xf0\xf0\xf0\xf0")

def master():
    csn = Pin(cfg["csn"], mode=Pin.OUT, value=1)
    ce = Pin(cfg["ce"], mode=Pin.OUT, value=0)
    if cfg["spi"] == -1:
        spi = SPI(-1, sck=Pin(cfg["sck"]), mosi=Pin(cfg["mosi"]), miso=Pin(cfg["miso"]))
        nrf = NRF24L01(spi, csn, ce, payload_size=8)
    else:
        nrf = NRF24L01(SPI(cfg["spi"]), csn, ce, payload_size=8)

    nrf.open_tx_pipe(pipes[0])
    nrf.open_rx_pipe(1, pipes[1])
    nrf.start_listening()
    num_needed = 16
    num_successes = 0
    num_failures = 0
    led_state = 0
```

```python
    print("NRF24L01 master mode, sending %d packets..." % num_needed)

    while num_successes < num_needed and num_failures < num_needed:
        # stop listening and send packet
        nrf.stop_listening()
        millis = utime.ticks_ms()
        led_state = max(1, (led_state << 1) & 0x0F)
        print("sending:", millis, led_state)
        try:
            nrf.send(struct.pack("ii", millis, led_state))
        except OSError:
            pass

        # start listening again
        nrf.start_listening()

        # wait for response, with 250ms timeout
        start_time = utime.ticks_ms()
        timeout = False
        while not nrf.any() and not timeout:
            if utime.ticks_diff(utime.ticks_ms(), start_time) > 250:
                timeout = True

        if timeout:
            print("failed, response timed out")
            num_failures += 1
        else:
            # recv packet
            (got_millis,) = struct.unpack("i", nrf.recv())

            # print response and round-trip delay
            print(
                "got response:",
                got_millis,
                "(delay",
                utime.ticks_diff(utime.ticks_ms(), got_millis),
                "ms)",
            )
            num_successes += 1

        # delay then loop
        utime.sleep_ms(250)

    print("master finished sending; successes=%d, failures=%d" % (num_successes, num_failures))


def slave():
    csn = Pin(cfg["csn"], mode=Pin.OUT, value=1)
    ce = Pin(cfg["ce"], mode=Pin.OUT, value=0)
    if cfg["spi"] == -1:
        spi = SPI(-1, sck=Pin(cfg["sck"]), mosi=Pin(cfg["mosi"]), miso=Pin(cfg["miso"]))
        nrf = NRF24L01(spi, csn, ce, payload_size=8)
    else:
        nrf = NRF24L01(SPI(cfg["spi"]), csn, ce, payload_size=8)

    nrf.open_tx_pipe(pipes[1])
    nrf.open_rx_pipe(1, pipes[0])
    nrf.start_listening()

    print("NRF24L01 slave mode, waiting for packets... (ctrl-C to stop)")

    while True:
        if nrf.any():
            while nrf.any():
                buf = nrf.recv()
                millis, led_state = struct.unpack("ii", buf)
                print("received:", millis, led_state)
                for led in leds:
                    if led_state & 1:
                        led.on()
                    else:
                        led.off()
```

```
                led_state >>= 1
            utime.sleep_ms(_RX_POLL_DELAY)

        # Give master time to get into receive mode.
        utime.sleep_ms(_SLAVE_SEND_DELAY)
        nrf.stop_listening()
        try:
            nrf.send(struct.pack("i", millis))
        except OSError:
            pass
        print("sent response")
        nrf.start_listening()


print("NRF24L01 test module loaded")
print("NRF24L01 pinout for test:")
print("    CE on", cfg["ce"])
print("    CSN on", cfg["csn"])
print("    SCK on", cfg["sck"])
print("    MISO on", cfg["miso"])
print("    MOSI on", cfg["mosi"])
print("run nrf24l01test.slave() on slave, then nrf24l01test.master() on master")

master()
```

The execution result (incomplete):

```
>>> %Run -c $EDITOR_CONTENT
NRF24L01 test module loaded
NRF24L01 pinout for test:
    CE on 15
    CSN on 5
    SCK on 18
    MISO on 19
    MOSI on 23
run nrf24l01test.slave() on slave, then nrf24l01test.master() on master
Warning: SPI(-1, ...) is deprecated, use SoftSPI(...) instead
NRF24L01 master mode, sending 16 packets...
sending: 203650 1
got response: 203650 (delay 48 ms)
sending: 203949 2
got response: 203949 (delay 45 ms)
sending: 204245 4
got response: 204245 (delay 43 ms)
sending: 204539 8
got response: 204539 (delay 43 ms)
sending: 204833 1
got response: 204833 (delay 45 ms)
sending: 205129 2
got response: 205129 (delay 43 ms)
sending: 205423 4
```

6.12 LED dot matrix: MAX7219

The **MAX7219** are compact, serial input/output common-cathode display drivers that interface IoT board to 64 individual LEDs that may be chained in up to 8 units. The MAX7221 is compatible with SPI bus.

The following example uses 4 units – dot matrix to display a down counter for 3600 seconds.
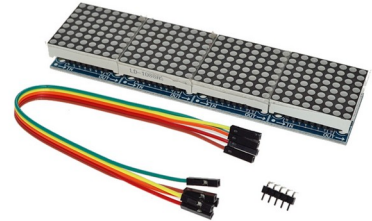

```
import max7219
from machine import Pin, SPI
import time

cfg = {"spi": -1, "miso": 19, "mosi": 23, "sck": 18, "csn": 5, "ce": 15}

spi = SPI(1, baudrate=10000000, polarity=1, phase=0, sck=Pin(18), mosi=Pin(23))
ss = Pin(5, Pin.OUT)
display = max7219.Matrix8x8(spi, ss, 4)
c=3600
while c>0:
    count=str(c)
    display.fill(0)
    #display.text('1234',0,0,1)
    display.text(count,0,0,1)
    display.show()
    time.sleep(1)
    c-=1
```

Lab 7

Getting data from services

In this lab we are going to exploit a number of IoT services (data) available on Internet.

7.1 Time service - NTP

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 (UDP, port number: 123) .
NTP is intended to synchronize all participating computers to within a few milliseconds of Coordinated Universal Time (UTC, previously called GMT-Greenwich Mean Time).
NTP uses the intersection algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency. NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy in local area networks under ideal conditions.

Time service with NTP server:

```
import ntptime
import wifista
import time
import machine, neopixel
np = neopixel.NeoPixel(machine.Pin(0), 12)

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

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

wifista.scan()
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=(min/5+7)%12  # +7 ring shift
    ledsec=(sec/5+7)%12  # +7 ring shift
    ledhour=(hour+9)%12  # +7 ring shift +2 GMT time
    print(int(ledhour),int(ledmin),int(ledsec))
    set_clock(int(ledhour),int(ledmin),int(ledsec),100)
    time.sleep(5)
```


To do:

1. Try to understand the lines used to "move" the LED position to correct place:

```
ledmin=(min/5+7)%12  # +7 ring shift
ledsec=(sec/5+7)%12  # +7 ring shift
ledhour=(hour+9)%12  # +7 ring shift +2 GMT time
```

2. Display the time parameters on OLED screen.

7.2 Reading data from Thingspeak server

Thingspeak service (**read channels):**

```
import socket, wifista, time
import json
import gc

def http_get(url):
    scheme, _, host, path = url.split('/', 3)
    wifista.connect()
    s = socket.socket()
    s.connect((host, 80))
    request=bytes('GET /%s HTTP/1.1\r\nHost: %s\r\n\r\n' % (path, host), 'utf8')    # request header
string into bytes
    print("Requesting /%s from host %s\n" % (path, host))
    s.send(request)
    val=0
    c=0
    while True:
        data = s.recv(500)
        if data:
            val=data
            if(c<2):
                c+=1
                if (c==2):
                    s.close()
                    return val
        else:
            break
    s.close()
    return val
c=0
gc.enable()
wifista.scan()
wifista.disconnect()
wifista.connect()

while c<1000:
    try:
        print(gc.mem_free())
        gc.collect()
        #ret=http_get("https://api.thingspeak.com/channels/1538804/fields/1/last.json")
        ret=http_get("https://api.thingspeak.com/channels/1538804/fields/2/last.json")
        #ret=http_get("https://api.thingspeak.com/channels/1538804/fields/2.json?results=3")
        mstr=str(ret, 'utf8')
        print(mstr[-69:])               # to show the received string
        data=json.loads(mstr[-69:])  # string into list; depends on the requested json: fields, ..
        #print(data["field1"])         # printing the field from  list
        print(data["field2"])
        print(data["entry_id"])
        print(data["created_at"])
        c+=1
        time.sleep(20)
    except Exception as e:
        print("ERROR: "+str(e))
        if (str(e) == "-202" or str(e) == "23"):  # page or socket errors
            wifista.scan()
            wifista.disconnect()
            wifista.connect()
        time.sleep(1)
```
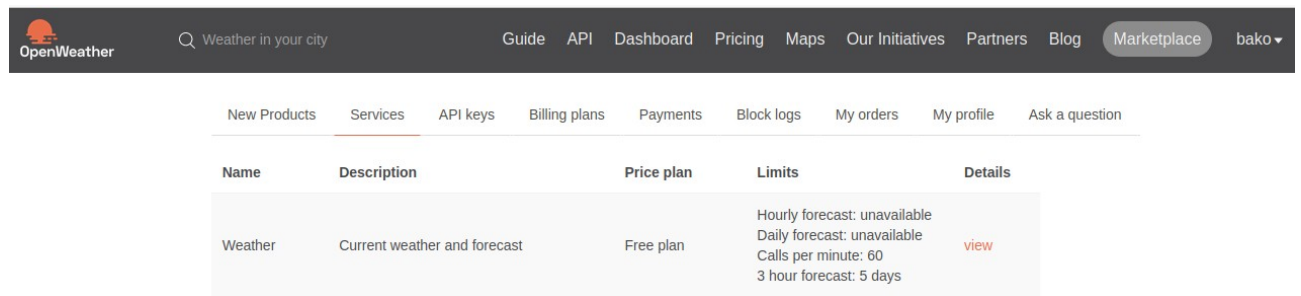
To do:

1. Modify the code to read the data from **field1.**
2. Study how to use of **try** and **except** to catch the errors and "**repair**" them.
3. Notice the use of garbage collector (**gc.collect()**)to "clean" the RAM memory.
4. Notice the use of **json.loads** to transform JSON **string** into an **array**.
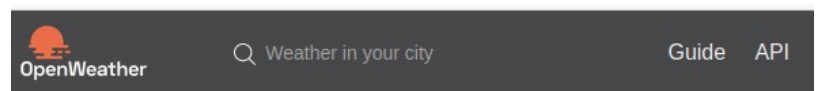
7.3 Reading weather data from `openweathermap` server

Weather service from `openweathermap.org`.
First you need to subscribe to the "**Current Weather Data**" service with a free account.


```
# Load necessary libraries
import machine
import wifista
import ssd1306
import network
import urequests
import ujson as json
import time


# **************************************
# Create objects:
led = machine.Pin(22,machine.Pin.OUT)

scl = machine.Pin(14, machine.Pin.OUT, machine.Pin.PULL_UP)
sda = machine.Pin(12, machine.Pin.OUT, machine.Pin.PULL_UP)
i2c = machine.I2C(scl=scl, sda=sda, freq=400000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c, addr=0x3C)

def print_text(msg, x, y, clr):
    if clr:
        oled.fill(0)
    oled.text(msg, x, y)
    oled.show()


# Configure the ESP32 wifi as STAtion
wifista.scan()
wifista.disconnect()
wifista.connect()

# Global variables and constants:
#api.openweathermap.org/data/2.5/weather?q={city name}&appid={API key}
#https://api.openweathermap.org/data/2.5/weather?q=paris&appid=ac07acbb25501a599aad09b6f6d290fb

BASE_URL = "https://api.openweathermap.org/data/2.5/weather?"
```

```python
API_KEY = "d0f71269f05816fa65ebafa10f1513d5"  # put here your api key

machine.Pin(32,machine.Pin.IN,machine.Pin.PULL_UP)  # DIP switch to select the location

UPDATE_INTERVAL_ms = 5000 # in ms time unit
last_update = time.ticks_ms()

while True:
    if machine.Pin(32).value()==0:  # test the DIP switch to change the CITY
        CITY_NAME = "HongKong"
    else:
        CITY_NAME = "Nantes"
    URL = BASE_URL + "q=" + CITY_NAME + "&appid=" + API_KEY
    if time.ticks_ms() - last_update >= UPDATE_INTERVAL_ms:
        # send a weather data request
        response = urequests.get(URL)
        # check status code of the request
        if response.status_code == 200:
            # get the json format of data
            data = response.json()
            # get the main dict block
            # it contains the weather data
            main = data['main']
            # get the temperature and subtract
            # 273.15 to get temp in Celcius
            # originally temp is in Kelvin
            temperature = main['temp'] - 273.15
            # get the humidity in %
            humidity = main['humidity']
            # get the pressure in hPA
            pressure = main['pressure']
            # weather report
            report = data['weather']

            ## OLED display
            print_text('**OpenWeather**', 0, 0, 1)
            print_text('City:{}' .format(CITY_NAME), 3, 16, 0)
            print_text('Temp:{} oC' .format(temperature), 3, 26, 0)
            print_text('Humi:{} %' .format(humidity), 3, 36, 0)
            print_text('Pres:{} hPa' .format(pressure), 3, 46, 0)
            print_text('"{}."' .format(report[0]['description']), 3, 56, 0)

            ## debug messages
            print('')
            print('**OpenWeather**')
            print('City:{}' .format(CITY_NAME))
            print('Temperature:{} oC' .format(temperature))
            print('Humidity:{} %' .format(humidity))
            print('Pressure:{} hPa' .format(pressure))
            print('Weather Report:{}.' .format(report[0]['description']))
        else:
            # show error message
            print_text('Error in HTTP request.', 3, 20, 1)
            print('Error in HTTP request.')

        led.value(not led.value())
        last_update = time.ticks_ms()
```

To do:

Modify the code to select different places: **Paris**, **London**, ..
You may use one more DIP switch (Pin 33) to select more places.

Table of Contents