

# IoT Labs with PYCOM-X

## SmartComputerLab

### Table of Contents

<b>0. Introduction.....</b>	<b>3</b>
0.1 ESP32 Soc – an advanced unit for IoT architectures.....	3
0.2 ESP32 LOLIN32 board.....	4
0.3 IoT DevKit PYCOM-V an IoT development platform.....	5
0.4 Software – Thonny IDE.....	6
0.4.1 Installing Thonny IDE - thonny.org.....	6
0.4.2 Preparing the ESP32 LOLIN32 board.....	7
0.4.4 First example – x.led.blink.py.....	8
<b>Lab 1.....</b>	<b>10</b>
<b>Sensor reading and data display (i2c).....</b>	<b>10</b>
1.0 Introduction.....	10
1.1 First example – data display on OLED screen.....	10
1.2 Second example – sensor reading (T/H): SHT21.....	12
1.2.1 Preparing the code.....	12
1.3 Third example – reading a luminosity sensor (L) - BH1750.....	14
1.4 Fourth example – reading a PIR sensor – SR602.....	15
<b>Lab 2.....</b>	<b>16</b>
<b>WiFi communication and WEB servers.....</b>	<b>16</b>
2.1 Network scan.....	16
2.2 Connection to the WiFi network, station mode – STA.....	17
2.3 Reading a WEB page.....	18
2.3 Simple WEB server – reading a variable.....	19
2.4 Simple WEB server – sending an order.....	20
2.4.3 Mini WEB server with Access Point – RGB LED management.....	21
<b>Lab 3.....</b>	<b>23</b>
<b>MQTT Broker and ThingSpeak Server.....</b>	<b>23</b>
3.1 MQTT Protocol and MQTT Client.....	23
3.1.1 MQTT client – the code.....	23
3.1.2 Broker MQTT on a PC.....	24
3.2 ThingSpeak server.....	24
3.2.1 Preparation for sending data as MQTT messages.....	24
3.2.2 Preparation for sending data as simple HTTP requests.....	26
3.2.3 Preparation for sending data with thingspeak.py library.....	27
<b>Lab 4.....</b>	<b>29</b>
<b>LoRa technology for Long Range communication.....</b>	<b>29</b>
4.0 Introduction.....	29
4.1 LoRa Modulation.....	29
4.2 sx127x.py driver library.....	29
4.3 Main program.....	30
4.4 LoRa functional modules.....	32
4.4.1 Transmitter – sender() (LoRaSender.py).....	32
4.4.2 Receiver – receive (LoRaReceiver.py).....	32
4.4.3 Receiver – onReceive (LoRaReceiverCallback.py).....	33
<b>Lab 5.....</b>	<b>34</b>
<b>Development of simple IoT gateways.....</b>	<b>34</b>
5.1 LoRa-WiFi Gateway (MQTT).....	34
5.2 LoRa-WiFi gateway (ThingSpeak).....	37

# IoT Labs with PYCOM-X

SmartComputerLab

## 0. Introduction

In these **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 **PYCOM-X DevKit** from **SmartComputerLab**.

**PYCOM-X** contains a base card to accommodate a central board with its **IoT SoC (System on Chip)** and a set of 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 present the 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.

**The first lab** is used to set up the work environment and test the use of an **OLED display** and the set of **sensors** connected to the **I2C bus** (temperature/humidity/luminosity/movement).

**Thonny IDE** makes it possible to load the libraries necessary 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 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** (1 km). **LoRa** modems are integrated in the expansion boards provided for PYCOM-X.

With LoRa we can send structured data from sensors managed on terminal nodes over 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.

**The fifth lab** will integrate the set of WiFi and LoRa links 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).

### 0.1 ESP32 Soc – an advanced unit for IoT architectures

ESP32 is an **advanced micro-controller** unit designed for the **development of IoT architectures**. The ESP32 SoC integrates two 32-bit RISC processors of the Xtensa LX6 type operating at **240MHz** and several additional processing and communication units, in particular an **ULP (Ultra Low Power)** processor, **WiFi/BT/BLE** modems and a set of I/O controllers for serial busses (**UART, I2C, SPI,..**).

These functional blocks are depicted in the following figure.

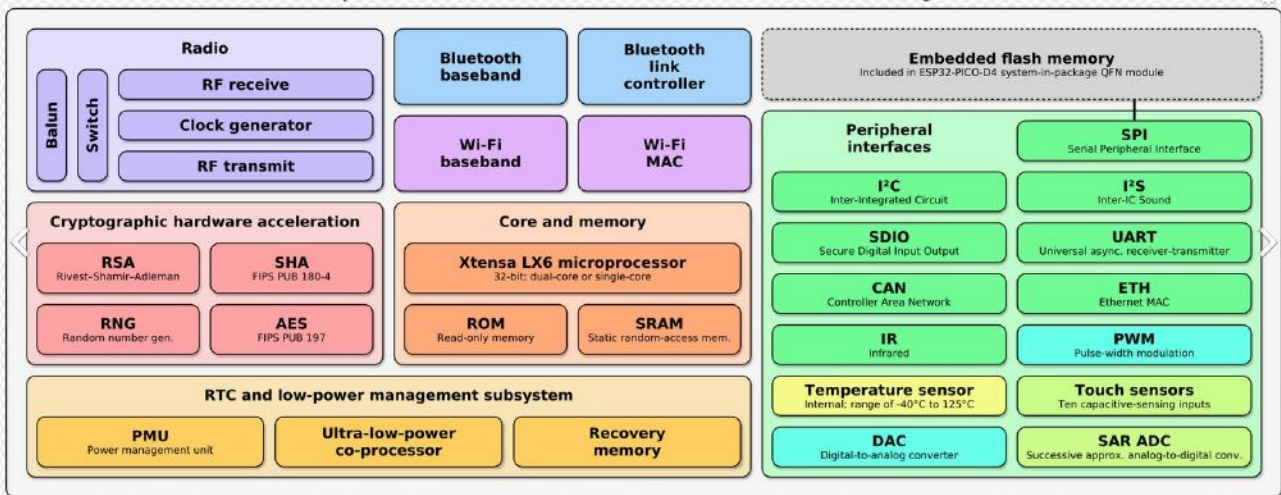


Figure 0.1 ESP32 SoC – internal architecture

## 0.2 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 (3V7)

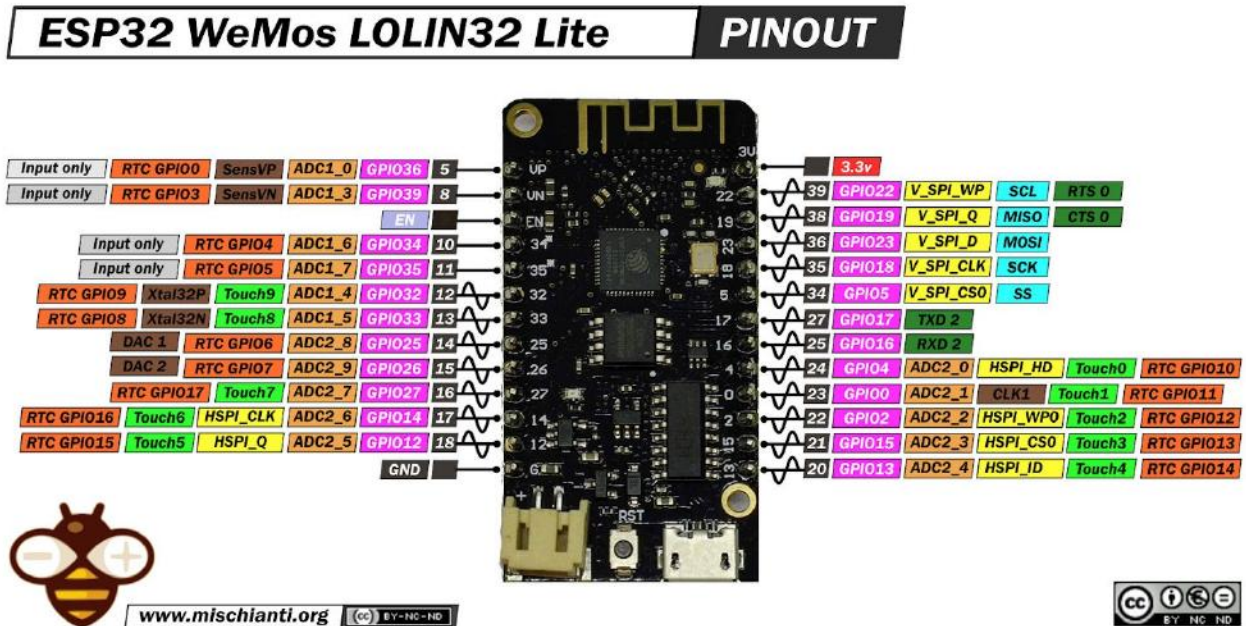


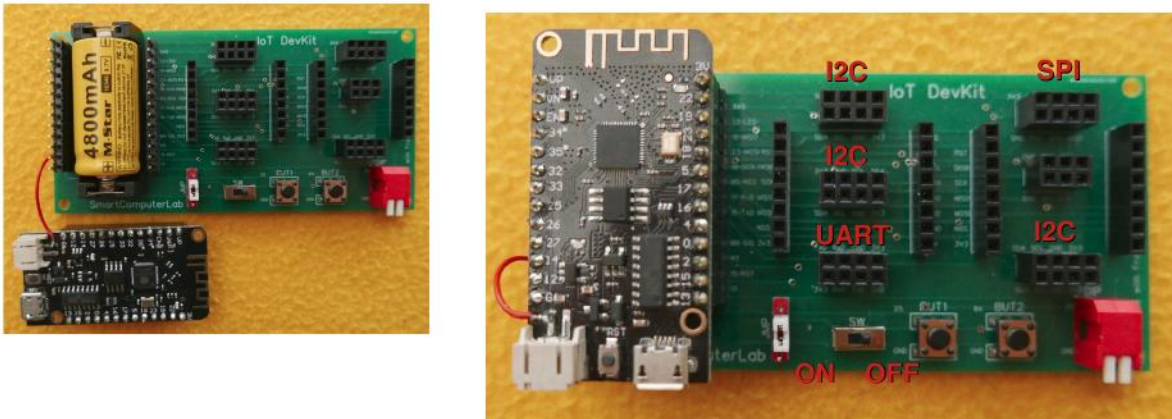
Figure 0.2 ESP32 LOLIN32 Lite MCU board and its pinout

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.

### 0.3 IoT DevKit PYCOM-V and 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 IoTDevKit 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.

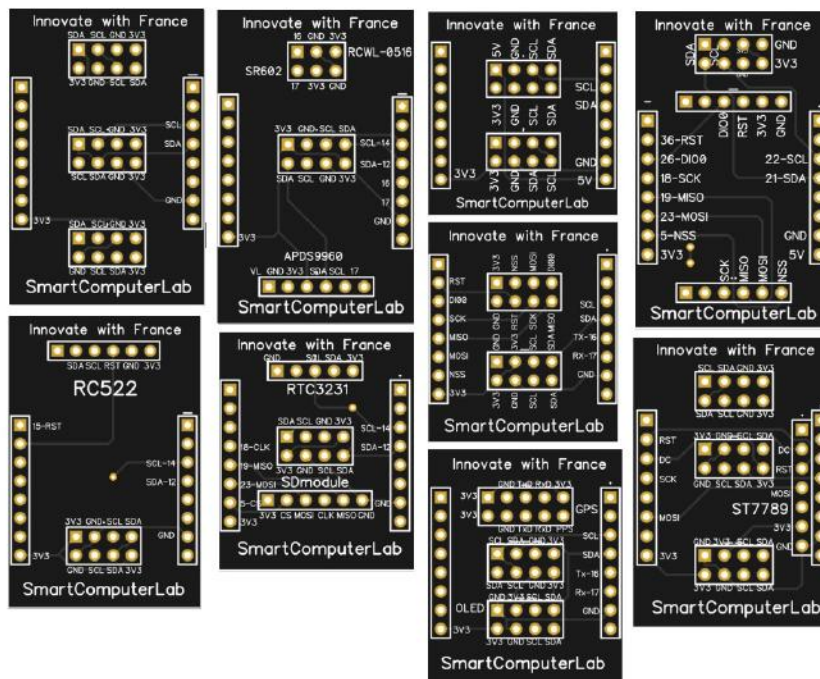


**Figure 0.3** PYCOM-X base board with the battery and its integrated interfaces

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

## 0.4 Software – Thonny IDE

### 0.4.1 Installing Thonny IDE - [thonny.org](https://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.

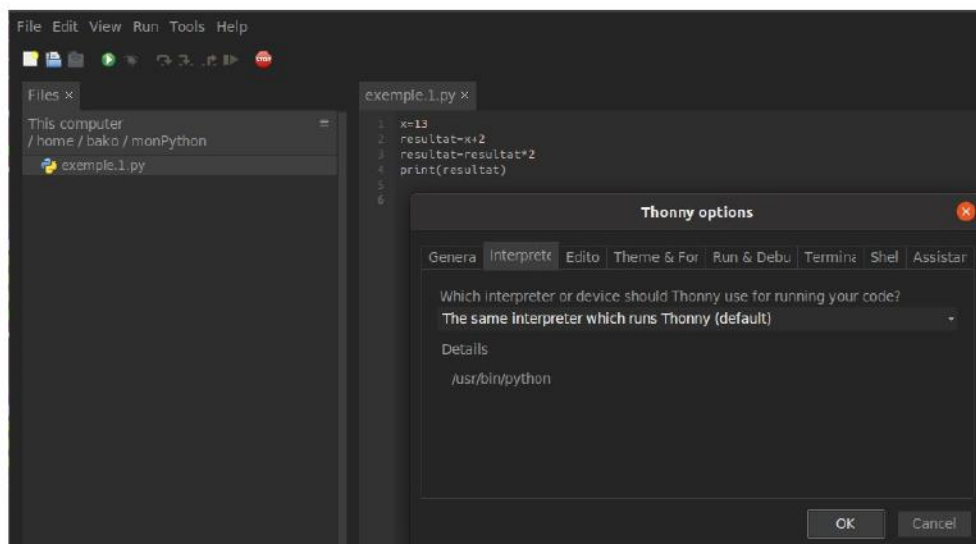
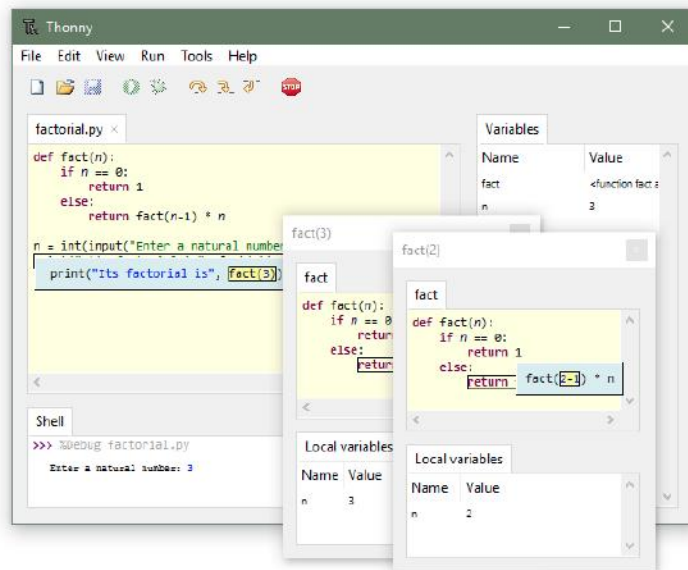
Here is the installation page of the Thonny IDE. You follow the instructions.

**Thonny**  
Python IDE for beginners

Download version **3.3.13** for  
[Windows](#) • [Mac](#) • [Linux](#)

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.



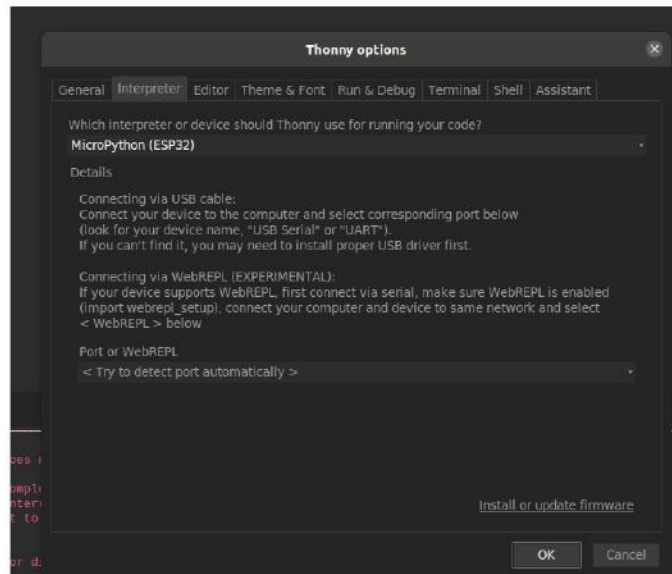
The above figure shows the **default selection of interpreter** running on your PC. With this interpreter you run your code directly on the PC. This feature is useful to start the programming in Python.

Above is an example of programming in Python. The 4 lines are saved in the file **example.1.py**.

## 0.4.2 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 on the page:

<https://micropython.org/download/>.

For our card we choose MCU and esp32:

<https://micropython.org/download/?mcu=esp32>.

On this page we will select: ESP32 with **OTA** support:

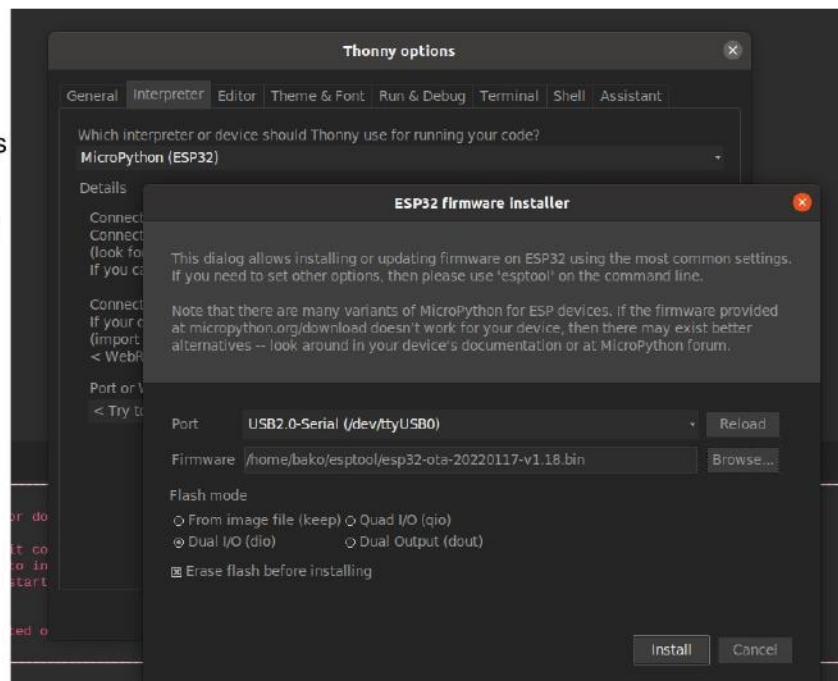
<https://micropython.org/download/esp32-ota/>

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

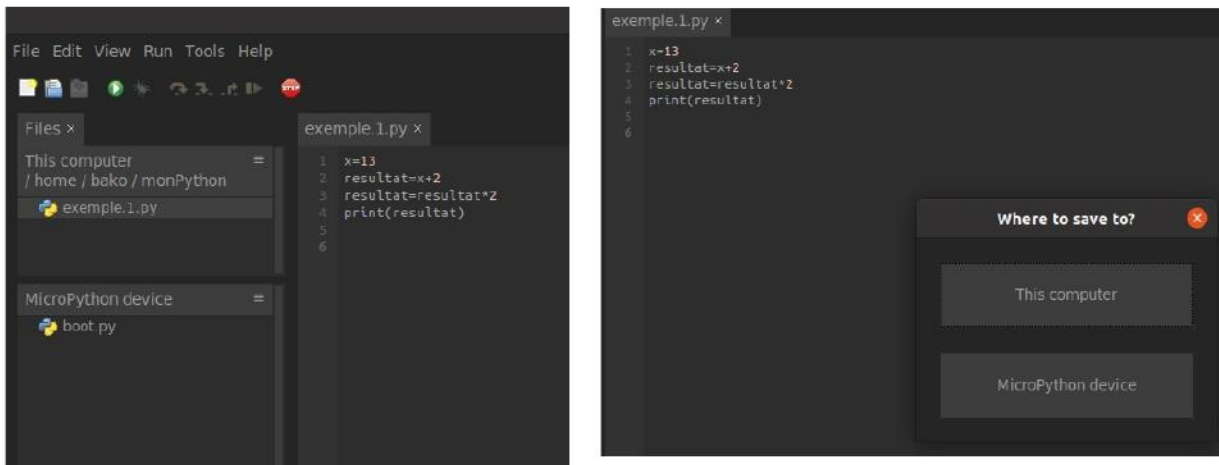
The installation must be preceded by **Erase flash** before installing.

**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. This time we go to **Tools->Options** to look for **Interpreter** and we will choose **MicroPython (ESP32)**.

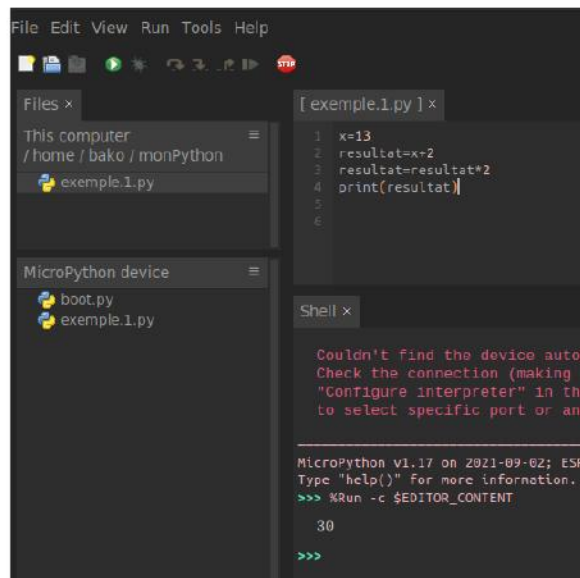


Let's see the available files, **View->Files**.

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**.



#### 0.4.4 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.

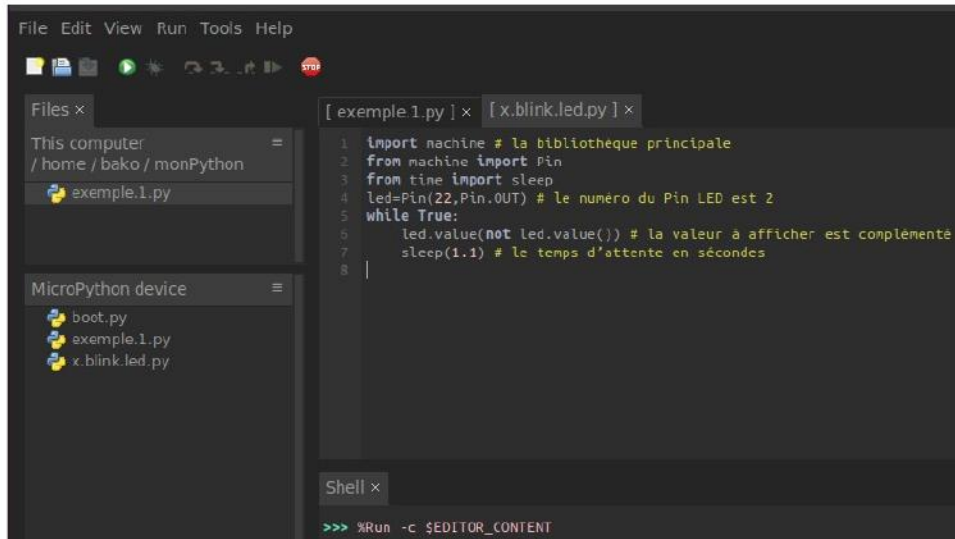
When the board boots first time there is only **boot.py**; other programs are loaded later.

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()`



# 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 (Data, Address).

### 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 **PYCOM-X DevKit**.

First of all you have to download the `ssd1306.py` screen driver available on our **GitHub**

<https://github.com/smartcomputerlab/Practical-IoT>

or

<https://github.com/micropython/micropython/tree/master/drivers/display>

After loading it, you have to store it in your PC directory, then open it in the Thonny editor and flash it on the card.

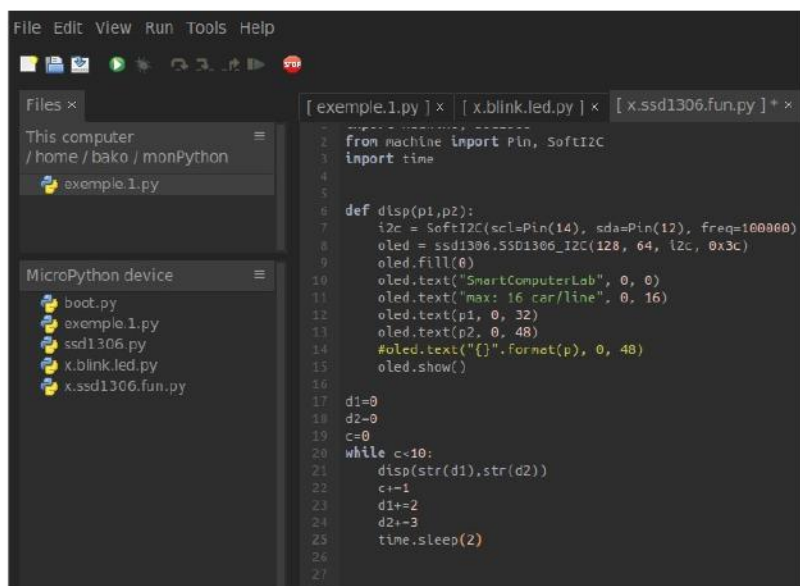
Now edit the following code:

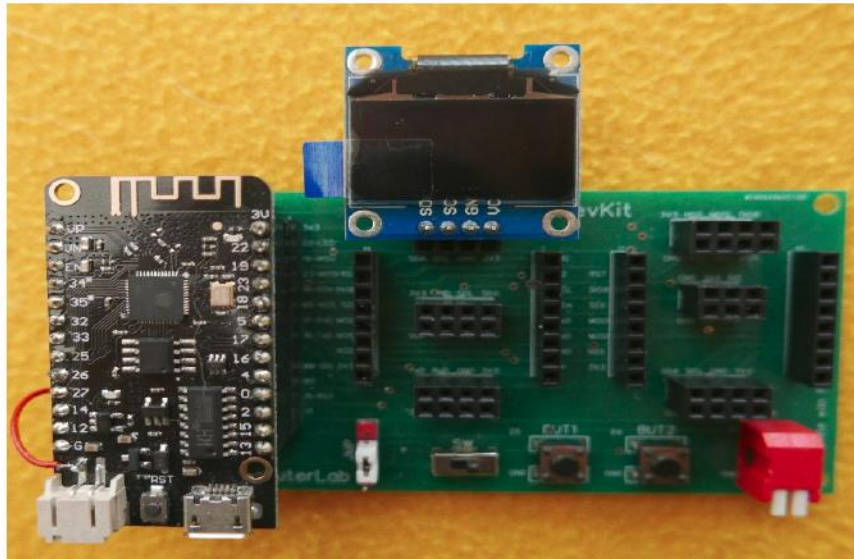
```
import machine, ssd1306
from machine import Pin, SoftI2C
import time

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 **PYCOM-X** 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)**.

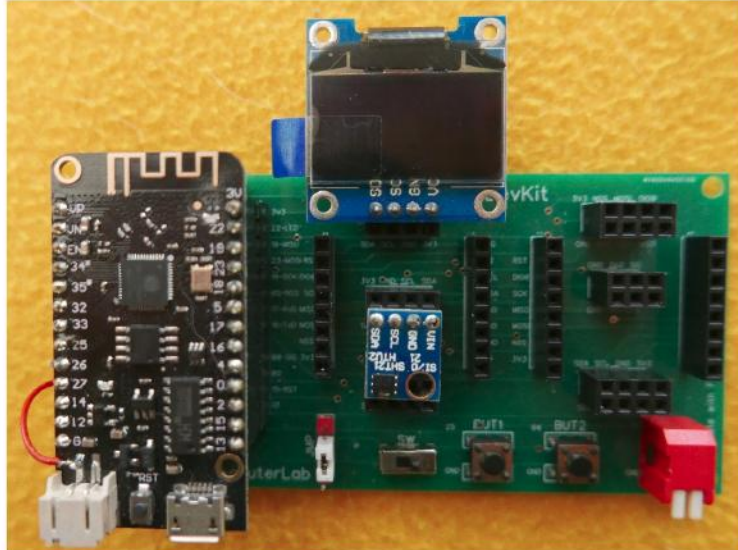


Fig 1.3 PYCOM-X with OLED display (SSD1306) and SHT21 sensor on the same I2C bus

### 1.2.1 Preparing the code

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

```
data= [] # data is declared as an array - size is not determined
```

The processor sends the **humidity** reading **command – one byte**:

```
i2c.writeto(address, b'\xf5') # read user register for humidity
```

After sending the command the processor reads **two bytes of data**:

```
data = i2c.readfrom(address, 2) # get the 2-byte result
```

Note that humidity and temperature data are received in 2 bytes.

These 2 bytes (16 bits) **must be converted and adjusted** into an integer value (int) for the display and the operations to follow.

#### Full code:

```
from machine import Pin, SoftI2C
import utime

i2c = SoftI2C(scl=Pin(14), sda=Pin(12), freq=40000) #I2C channel 0,pins,400kHz max
STATUS_BITS_MASK = 0xFFFC
data = []
address = 64

def get_humi():
    # Read humidity
    i2c.writeto(address, b'\xf5') # Trigger humidity measurement
    utime.sleep_ms(29) # Wait for it to finish (29ms max)
    data = i2c.readfrom(address, 2) # Get the 2 byte result
    adjusted = (data[0] << 8) + data[1] # convert to 16 bit value
    adjusted &= STATUS_BITS_MASK # zero the status bits
    adjusted *= 125 # scale
```

```

    adjusted /= 1 << 16          # divide by 2^16
    adjusted -= 6                # subtract 6
    return adjusted

def get_temp():
    # Read temperature
    i2c.writeto(address, b'\xF3') # Trigger temperature measurement
    utime.sleep_ms(85)           # Wait for it to finish (85ms max)
    data = i2c.readfrom(address, 2) # Get the 2 byte result
    ## Compute temperature
    adjusted = (data[0] << 8) + data[1] # convert to 16 bit value
    adjusted &= STATUS_BITS_MASK      # zero the status bits
    adjusted *= 175.72                # scale
    adjusted /= 1 << 16              # divide by 2^16
    adjusted -= 46.85                 # subtract offset
    return adjusted

temp=get_temp()
print ("Temperature = %.1f" % temp)
humi=get_humi()
print ("Humidity = %.1f" % humi)

```

Voici l'affichage du résultat:

```

>>> %Run -c $EDITOR_CONTENT
Humidity    = 55.8
Temperature = 22.0
>>>

```

### To do:

1. Study the program to understand how the I2C bus works with the functions

```

    i2c.writeto(address,b'\xF3')    # send command to sensor
    and
    data = i2c.readfrom(address,2)  # read data from sensor

```

2. Add the OLED display and complete the program to show the results.
3. Add a loop to read and display the results multiple times.
4. Edit the following program, load it onto the PYCOM-X board and check how it works.

```

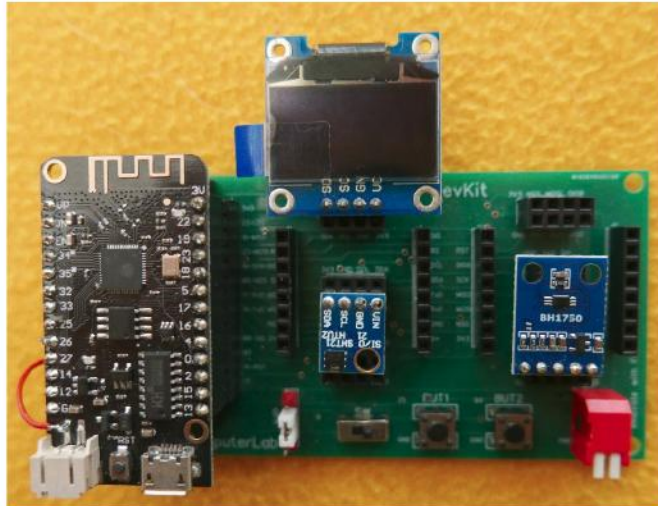
import machine
i2c = machine.I2C(scl=machine.Pin(0), sda=machine.Pin(10))
print('Scan i2c bus...')
devices = i2c.scan()
if len(devices) == 0:
    print("No i2c device !")
else:
    print('i2c devices found:',len(devices))
    for device in devices:
        print("Decimal address: ",device," | Hexa address: ",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 simple reading done by `sample()` function. This function starts by initializing the sensor at address – `0x23` with **3 bytes** sent separately by `i2c.writeto(): clear, power up, mode`

Then we read 2 bytes – given by `i2c.readfrom(i2c_addr, 2)`, and we finish by sending the **power down command**.

```
import time
import machine

OP_SINGLE_HRES1 = 0x20
OP_SINGLE_HRES2 = 0x21
OP_SINGLE_LRES = 0x23
DELAY_HMODE = 180 # 180ms in H-mode
DELAY_LMODE = 24 # 24ms in L-mode

def sample(i2c, mode=OP_SINGLE_HRES1, i2c_addr=0x23): # sample() function with default values
    i2c.writeto(i2c_addr, b"\x00") # make sure device is in a clean state
    i2c.writeto(i2c_addr, b"\x01") # power up
    i2c.writeto(i2c_addr, bytes([mode])) # set measurement mode
    time.sleep_ms(DELAY_LMODE if mode == OP_SINGLE_LRES else DELAY_HMODE)
    raw = i2c.readfrom(i2c_addr, 2)
    i2c.writeto(i2c_addr, b"\x00") # power down again
    # we must divide the end result by 1.2 to get the lux
    return ((raw[0] << 24) | (raw[1] << 16)) // 78642

sda=machine.Pin(12)
scl=machine.Pin(14)
i2c=machine.I2C(0,sda=sda,scl=scl,freq=400000) #I2C channel 0,pins,400kHz max
lum=sample(i2c,mode=OP_SINGLE_HRES1, i2c_addr=0x23)
print('lum')
print(lum)

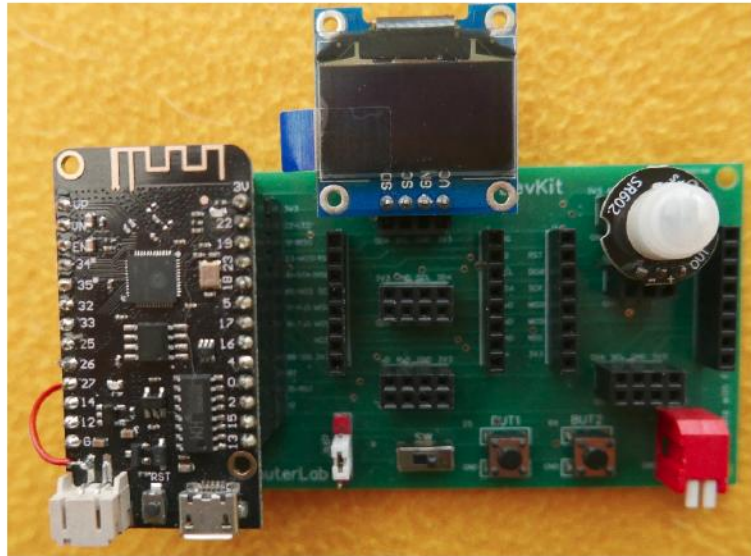
%Run -c $EDITOR_CONTENT
lum
496
```

### To do:

1. Study the program to understand how the I2C bus works.  
Note that the `sample(. . .)` function embeds the parameters with the **default values**.
2. Add the OLED screen and complete the program to show the results.
3. Add a **while loop** to read and display the results multiple times.

## 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** PYCOM-X 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
OBJECT DETECTED
OBJECT DETECTED
ALL CLEAR
```

### To do:

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

## 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 to read WEB pages and to send arguments to WEB servers.

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

#### 2.1 Network scan

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

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

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

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

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

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

..
```

#### To do:

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

#### Note:

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

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

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

Notre carte **PYCOM-X** peut se connecter au réseau WiFi en **mode station (STA)**. Dans ce cas le modem peut récupérer automatiquement (par le biais du protocole **DHCP**) une adresse IP et les adresses du router et du serveur DNS.

Le modem peut également imposer une configuration avec son adresse statique choisie par l'utilisateur.

Le programme suivant montre ces fonctionnalités :

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 impose a **static configuration** with a static IP address chosen by the user.

The following program demonstrates these features:

```
def connect():
    import network

    ip      = '192.168.1.110'
    subnet  = '255.255.255.0'
    gateway = '192.168.1.1'
    dns     = '8.8.8.8'
    ssid    = "Livebox-08B0"      # replace by your SSID
    password = "G79ji6dtEptVTPWmZP" # and its password

    station = network.WLAN(network.STA_IF)

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

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

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

# connect() # test operation
```

```
>>> %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 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.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.

```
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 our PYCOM-X. 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:

```

from machine import Pin, SoftI2C
import network, ssd1306
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()

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

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

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

```

### To do:

1. Test the program
2. Display the IP address and SSID name on OLED screen

## Lab 3

# MQTT Broker and ThingSpeak Server

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

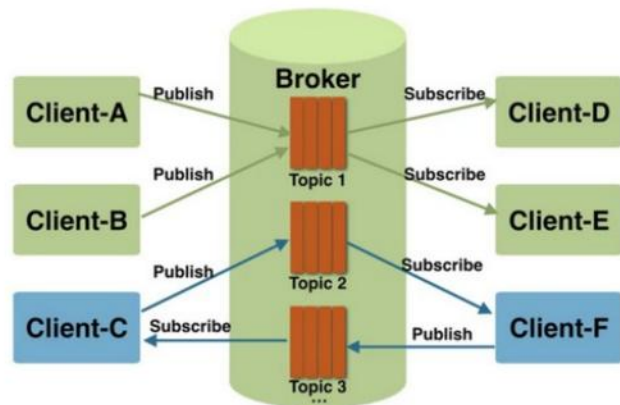
### 3.1 MQTT Protocol and MQTT Client

**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-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

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

### 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)
```

## Complete code:

```
from umqtt.simple import MQTTClient
import wifista
import 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)
```

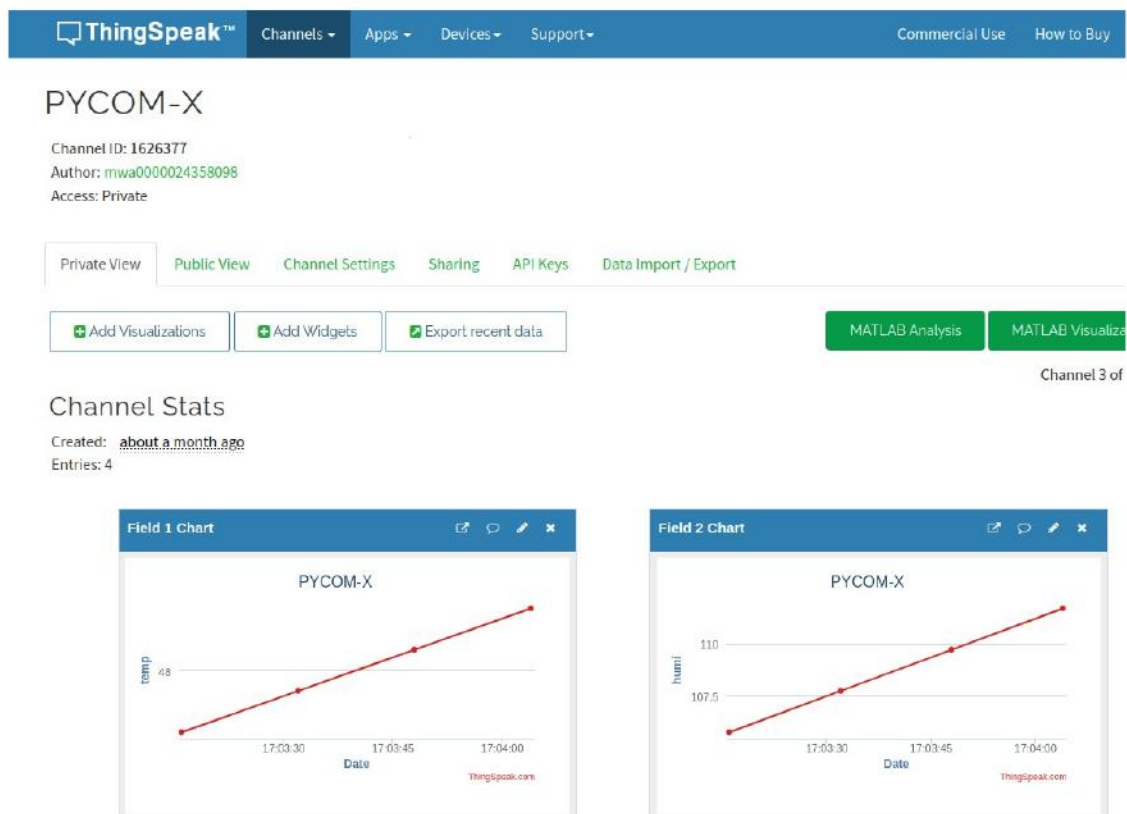


Fig. 3.2 ThingSpeak diagrams with our app

## 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

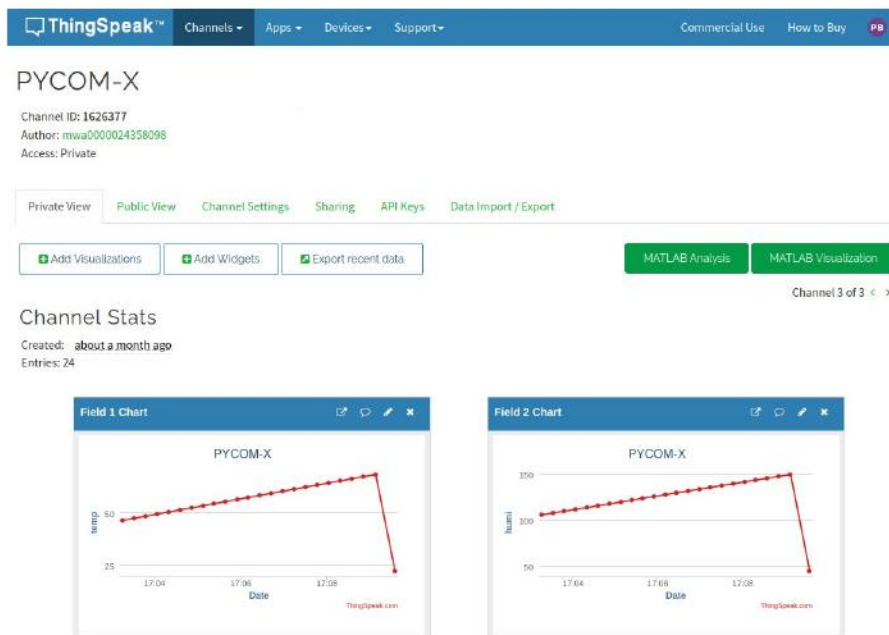
24channels/1626377/fields/2/last.json?api_key=9JVTP8ZHVTB9G4TT
api.thingspeak.com
```

```

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)

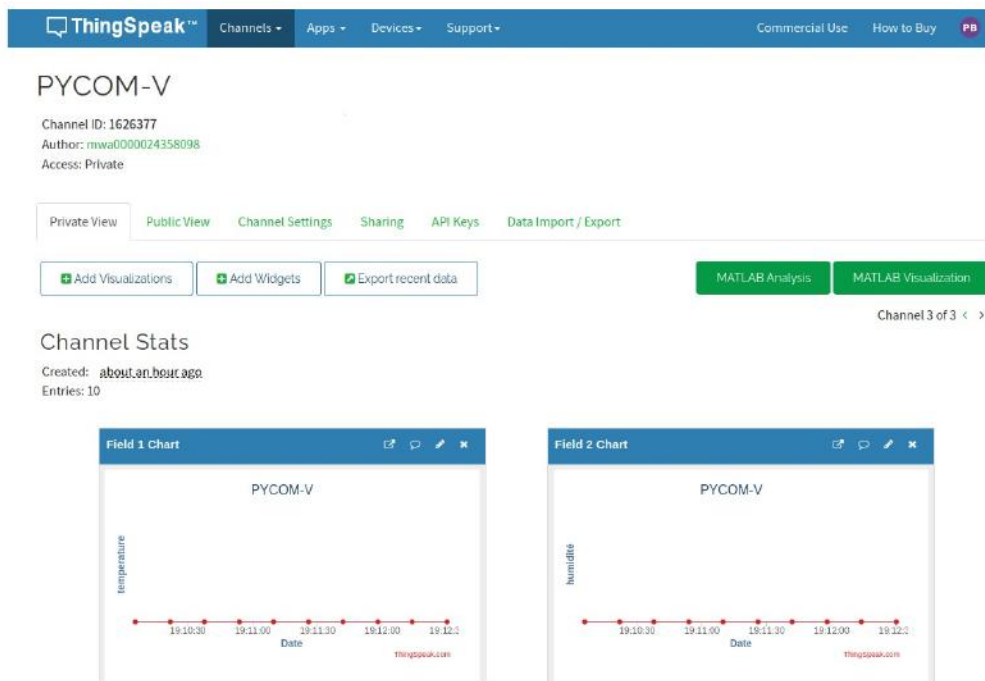
wifista.connect()
active_channel = channel_living_room
temperature = 21.4
humidity=33.7
while True:
    thing_speak.send(active_channel, {
        field_temperature: temperature,
        field_humidity: humidity
    })

    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**

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

## 4.2 sx127x.py driver library

The **sx127x.py** library makes it possible to integrate the functionalities of the **sx1276/8 modem** into our applications.

The modem-circuit is connected to our base board by **SPI bus**. An SPI bus operates on **3 basic lines** (signals): **SCK** – clock, **MISO** – Master\_In\_Slave\_Out, **MOSI** – Master\_Out\_Slave\_In, and on **three control lines**: **NSS** – Slave output selection or activation, **RST** – signal d initialization, and **DIO0/INT** – interrupt signal sent by the activated Slave.



Fig 4.1 The LoRa modem-module (Ra-01) with its SPI connector

Here are some excerpts from the **sx127x.py** library

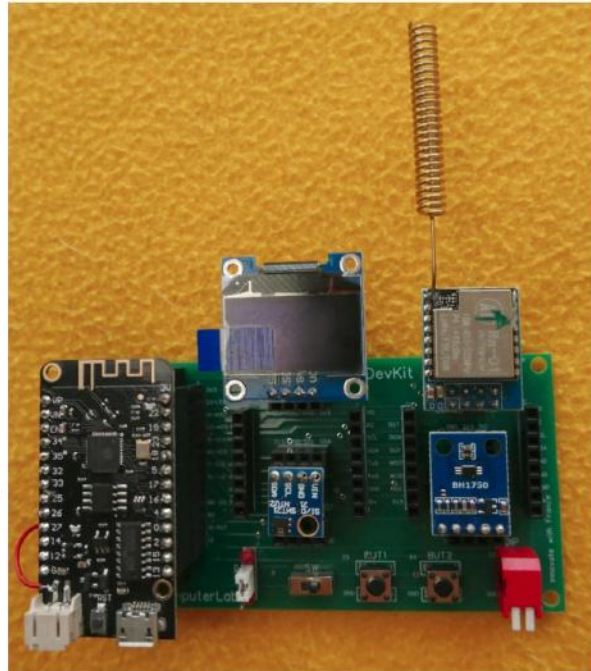
```
class SX127x:
    default_parameters = {
        "frequency": 869525000,
        "frequency_offset": 0,
        "tx_power_level": 14,
        "signal_bandwidth": 125e3,
        "spreading_factor": 9,
        "coding_rate": 5,
        "preamble_length": 8,
        "implicitHeader": False,
        "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 = {
    '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**.

Le **switch - type** à la fin du programme va sélectionner le module de **LoRaSender .py**

In the program above we have chosen the elemnts to configure the code as **sender - LoRaSender**.

The **-type** switch at the end of the program will select the **LoRaSender .py** module

## 4.4 LoRa functional modules

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

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

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

def disp(c):
    i2c = SoftI2C(scl=Pin(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 (Received Signal Strength Indicator)** value corresponding to the received LoRa messages.

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

def disp(p):
    i2c = SoftI2C(scl=Pin(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 = 'gatewaymqtt'
# 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 == 'gatewaymqtt':
#         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)

# Table of Contents

SmartComputerLab.....	1
SmartComputerLab.....	3
0. Introduction.....	3
0.1 ESP32 Soc – an advanced unit for IoT architectures.....	3
0.2 ESP32 LOLIN32 board.....	4
0.3 IoT DevKit PYCOM-V an IoT development platform.....	5
0.4 Software – Thonny IDE.....	6
0.4.1 Installing Thonny IDE - thonny.org.....	6
0.4.2 Preparing the ESP32 LOLIN32 board.....	7
0.4.4 First example – x.led.blink.py.....	8
Lab 1.....	10
Sensor reading and data display (i2c).....	10
1.0 Introduction.....	10
1.1 First example – data display on OLED screen.....	10
1.2 Second example – sensor reading (T/H): SHT21.....	12
1.2.1 Preparing the code.....	12
1.3 Third example – reading a luminosity sensor (L) - BH1750.....	14
1.4 Fourth example – reading a PIR sensor – SR602.....	15
Lab 2.....	16
WiFi communication and WEB servers.....	16
2.1 Network scan.....	16
2.2 Connection to the WiFi network, station mode – STA.....	17
2.3 Reading a WEB page.....	18
2.3 Simple WEB server – reading a variable.....	19
2.4 Simple WEB server – sending an order.....	20
2.4.3 Mini WEB server with Access Point – RGB LED management.....	21
Lab 3.....	23
MQTT Broker and ThingSpeak Server.....	23
3.1 MQTT Protocol and MQTT Client.....	23
3.1.1 MQTT client – the code.....	23
3.1.2 Broker MQTT on a PC.....	24
3.2 ThingSpeak server.....	24
3.2.1 Preparation for sending data as MQTT messages.....	24
3.2.2 Preparation for sending data as simple HTTP requests.....	26
3.2.3 Preparation for sending data with thingspeak.py library.....	27
Lab 4.....	29
LoRa technology for Long Range communication.....	29
4.0 Introduction.....	29
4.1 LoRa Modulation.....	29
4.2 sx127x.py driver library.....	29
4.3 Main program.....	30
4.4 LoRa functional modules.....	32
4.4.1 Transmitter – sender() (LoRaSender.py).....	32
4.4.2 Receiver – receive (LoRaReceiver.py).....	32
4.4.3 Receiver – onReceive (LoRaReceiverCallback.py).....	33
Lab 5.....	34
Development of simple IoT gateways.....	34
5.1 LoRa-WiFi Gateway (MQTT).....	34
5.2 LoRa-WiFi gateway (ThingSpeak).....	37