

# Laboratoires IoT avec PYCOM-V

SmartComputerLab

## Contenu

<b>0. Introduction.....</b>	<b>3</b>
0.1 ESP32-C3 Soc – une unité avancée pour les architectures IoT.....	4
0.2 Carte pyWiFi-ESP32-C3.....	4
0.3 IoT DevKit PYCOM-V une plate-forme de développement IoT.....	5
0.4 L'installation de Thonny.....	6
0.4.1 Conditions préalables.....	6
0.4.2 Installation de Thonny IDE dans le système d'exploitation Windows.....	6
0.4.3 Installation de Thonny IDE sous Linux.....	6
0.4.4 Premier exemple.....	6
A faire:.....	7
Remarque:.....	7
<b>Laboratoire 1 – Lecture des capteurs et affichage (i2c).....</b>	<b>8</b>
1.0 Introduction.....	8
1.1 Premier exemple – l'affichage des données.....	8
A faire:.....	9
1.2 Deuxième exemple – lecture d'un capteur (T/H) - SHT21.....	10
A faire:.....	11
1.3 Troisième exemple – lecture d'un capteur (L) - BH1750.....	12
A faire:.....	12
1.4 Quatrième exemple – lecture d'un capteur PIR - SR602.....	13
A faire:.....	13
<b>Laboratoire 2 – Communication WiFi et broker MQTT.....</b>	<b>14</b>
2.1 Scrutation <i>scan</i> du réseau.....	14
A faire:.....	14
Remarque :.....	14
2.2 Connexion au réseau WiFi , mode station - STA.....	15
A faire:.....	15
2.3 Lecture d'une page WEB.....	16
A faire:.....	16
2.3 Simple serveur WEB – lecture d'une variable.....	17
A faire:.....	17
2.4 Simple serveur WEB – l'envoi d'une commande.....	18
2.4.1 Une LED RGB.....	18
A faire:.....	18
2.4.2 Mini serveur WEB – gestion de la LED RGB.....	19
A faire:.....	20
2.4.3 Mini serveur WEB avec Point d'Accès – gestion de la LED RGB.....	20
A faire:.....	21
<b>Laboratoire 3 - Broker MQTT et serveur ThingSpeak.....</b>	<b>22</b>
3.1 Protocole MQTT et Client MQTT.....	22
3.1.1 Chargement et installation de bibliothèques par upip.....	22
3.1.2 Client MQTT – le code.....	23
A faire:.....	23
3.1.3 Broker MQTT sur un PC.....	23
A faire:.....	23
<b>3.2 Serveur ThingSpeak.....</b>	<b>24</b>
3.2.1 Préparation pour l'envoi des données comme messages MQTT.....	24
3.2.2 Préparation pour l'envoi des données comme simples requêtes HTTP.....	25
A faire:.....	25
3.2.3 Préparation pour l'envoi des données avec thingspeak.py.....	26
A faire:.....	26

<b>Laboratoire 4 - Technologie LoRa et Communication Long Range</b> .....	<b>27</b>
4.0 Introduction.....	27
4.1 Modulation LoRa.....	27
4.2 Bibliothèque sx127x.py.....	27
4.3 Programme principal.....	28
4.4 Modules fonctionnels LoRa.....	30
4.4.1 Emetteur - sender() (LoRaSender.py).....	30
A faire:.....	30
4.4.2 Récepteur - receive (LoRaReceiver.py).....	31
A faire:.....	31
4.4.3 Récepteur - onReceive (LoRaReceiverCallback.py).....	32
A faire:.....	32
Laboratoire 5 - Développement de simples passerelles IoT.....	33
5.1 Passerelle LoRa-WiFi (MQTT).....	33
A faire:.....	34
5.2 Passerelle LoRa-WiFi (ThingSpeak).....	35
A faire:.....	36

# Laboratoires IoT avec PYCOM-V

SmartComputerLab

## 0. Introduction

Dans les laboratoires IoT nous allons mettre en œuvre plusieurs architectures IoT intégrant les terminaux (**T**), les passerelles (*gateways* - **G**), et les serveurs-brokers (**S,B**) IoT type **MQTT**, **ThingSpeak**. Le développement sera réalisé sur les **IoT DevKit PYCOM-V** de **SmartComputerLab**.

Le kit de développement contient une carte de base "basecard" pour y accueillir une unité centrale et un ensemble de cartes d'extension pour les capteurs, les actionneurs et les modems supplémentaires. L'unité centrale est une carte équipée d'un **SoC ESP32-C3 (RISC-V)** et des modems **WiFi/BLE**.

Dans l'introduction nous présentons l'environnement de développement **Thonny**.

**PThonnyIDE** intègre l'ensemble d'outils qui permet d'éditer, et de charger (flasher) les codes micro-Python sur **PYCOM-V**.

**Le premier laboratoire** permet de mettre en œuvre l'environnement de travail et de tester l'utilisation des capteurs connectés sur le bus I2C (température/humidité/luminosité/pression) et d'un afficheur. L'environnement **Thonny** permet de charger les bibliothèques nécessaires pour l'interfaçage et l'exploitation des capteurs/afficheurs.

**Le deuxième laboratoire** est consacré à la prise en main du modem **WiFi** intégré dans le SoC RISC-V (carte **pyWiFi-ESP32-C3**). La communication **WiFi** en mode station (**STA**) permet d'envoyer les données – messages vers un broker **MQTT**. Un broker **MQTT** reçoit et renvoie les messages postés sur les **topic** données. Il ne possède pas d'une base de données pour y enregistrer les messages reçus.

L'utilisation d'une autre carte ESP basée sur le SoC ESP8266 permet le déploiement d'un micro-broker MQTT local fonctionnant en mode **WiFi station (STA)** et point d'accès (**AP**).

**Le troisième laboratoire** est dédié à l'utilisation du serveur **IoT ThingSpeak**. Les serveurs type **ThingSpeak** contiennent une base de données et offrent une interface graphique pour la visualisation des données enregistrées. **ThingSpeak.com** est accessible gratuitement, mais sa fréquence de réception de messages et le nombre des messages sont limités.

**Le quatrième laboratoire** permet d'expérimenter avec les liens à longue distance (1Km). Il s'agit de la technologie (et modulation) **LoRa**. Le modem LoRa est intégré sur la carte d'extension de notre **PYCOM-V**. Nous allons envoyer les données structurées d'un capteur géré par un terminal sur un lien LoRa vers une autre carte PYCOM-V. La carte de destination va afficher les données reçus, ainsi que la force du signal associée au paquet reçu.

**Le cinquième laboratoire** permettra d'intégrer l'ensemble de liens **WiFi et LoRa** pour créer des applications complètes avec les terminaux et les passerelles **LoRa-WiFi** vers les brokers-serveurs **MQTT** et **ThingSpeak**.

Nous y développerons deux applications, une pour la passerelle type **LoRa-WiFi** (broker **MQTT**) et une pour la passerelle de type **LoRa-WiFi** (serveur **ThingSpeak**).

## 0.1 ESP32-C3 Soc – une unité avancée pour les architectures IoT

ESP32-C3 est une unité de microcontrôleur avancée conçue pour le développement d'architectures IoT. Un SoC intègre un processeur **RISC-V** 32-bit fonctionnant à 180 MHz et plusieurs unités de traitement et de communication supplémentaires, notamment un processeur ULP (*Ultra Low Power*), des modems WiFi / BLE et un ensemble de contrôleurs E/S pour bus série (UART, I2C, SPI), ...). Ces blocs fonctionnels sont décrits ci-dessous dans la figure suivante.

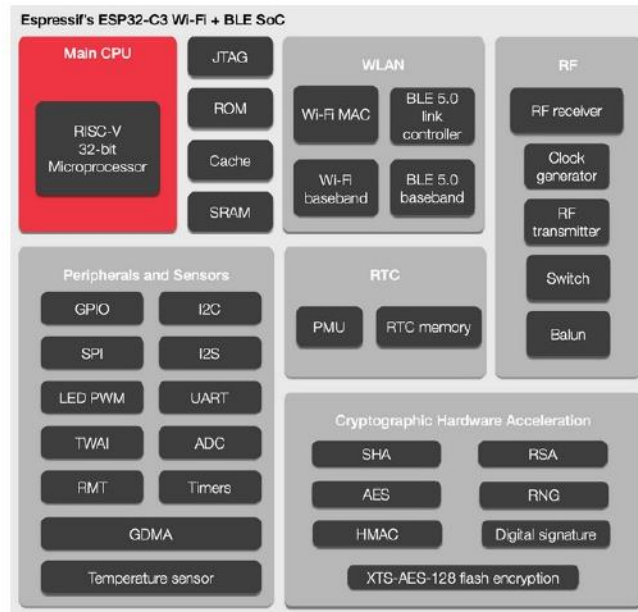


Figure 0.1 ESP32-C3 SoC – architecture interne

## 0.2 Carte pyWiFi-ESP32-C3

Les SoC ESP32-C3 sont intégrés dans un certain nombre de cartes de développement qui incluent des circuits supplémentaires et des modems de communication. Notre choix est la carte pyWiFi-ESP32-C3 qui intègre une interface avec les batteries LiPo (3V7)

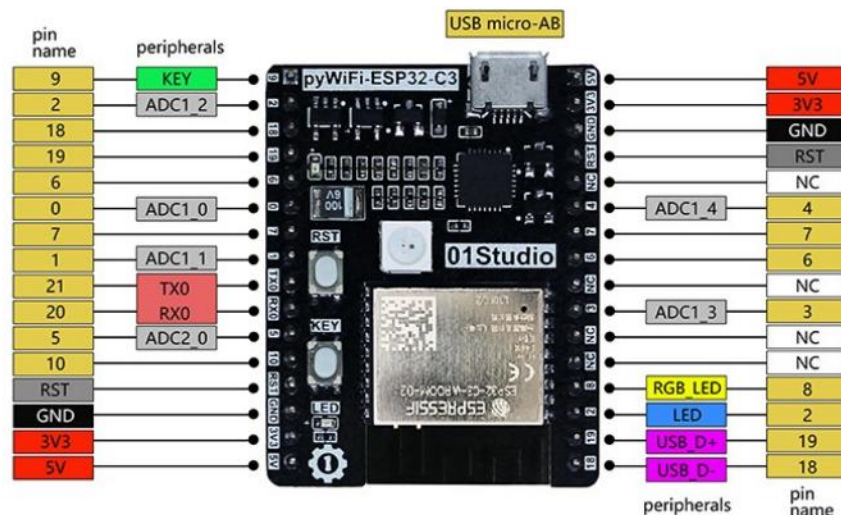


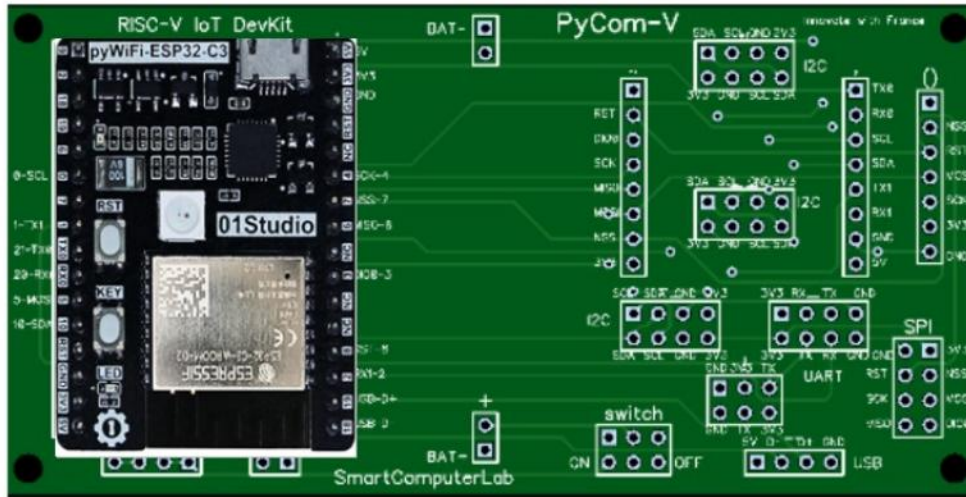
Figure 0.2 Carte MCU pyWiFi-ESP32-C3 et son pinout

Comme nous pouvons le voir sur la figure ci-dessus, la carte expose 2x16. Ces broches portent les bus **I2C** (10,0), **SPI** (18,19,23), **UART** (21,20) plus les signaux de contrôle ((NSS,RST,INT,...). LED est connectée sur pin 2, et le KEY sur le pin 9.

### 0.3 IoT DevKit PYCOM-V une plate-forme de développement IoT

Une intégration efficace de la carte **pyWiFi-ESP32-C3** sélectionnée dans les architectures IoT nécessite l'utilisation d'une plate-forme de développement telle que IoT DevKit proposée par **SmartComputerLab**.

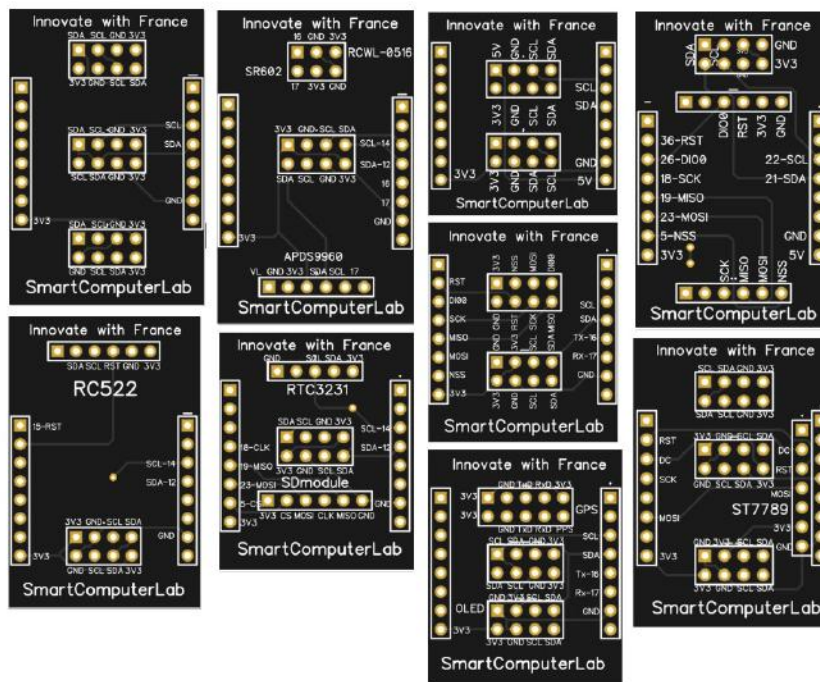
L'**IoTDevKit** est composé d'une carte de base et d'un grand nombre de cartes d'extension conçues pour l'utilisation efficace des bus de connexion et de tous les types de capteurs et d'actionneurs.



**Figure 0.3** Carte de base de **PYCOM-V** et ses interfaces intégrées

La carte de base peut accueillir directement plusieurs types de capteurs ou modems de communication. Pour y connecter un ensemble plus complet de capteurs/modems/afficheurs on utilise les cartes d'extension. Le chevalier (**jmp**) permet de connecter un multimètre et d'effectuer les mesures du courant. En mode faible consommation le courant descend à quelques dizaines de micro-ampères.

Ci dessous quelques exemples de cartes d'extension.



**Figure 0.4** Cartes d'extension pour les différents composants IoT: capteurs, afficheurs, modems, ..

## 0.4 L'installation de Thonny

**Thonny** est un **IDE** open source qui est utilisé pour écrire et télécharger des programmes **MicroPython** sur différentes cartes de développement telles que ESP32 et ESP8266. C'est un IDE extrêmement interactif et facile à apprendre, car il est connu comme l'IDE convivial pour les débutants pour les nouveaux programmeurs.

Avec l'aide de **Thonny**, il devient très facile de coder en **Micropython** car il possède un débogueur intégré qui aide à trouver toute erreur dans le programme en déboguant le script ligne par ligne.

Vous pouvez réaliser la popularité de **Thonny IDE** à partir de ce nouveau qu'il est pré-installé dans Raspian OS qui est un système d'exploitation pour un Raspberry Pi. Il est disponible pour l'installation sur Windows, Linux et Mac OS.

### 0.4.1 Conditions préalables

Avant d'installer Thonny IDE sur vos PC Windows, Linux et Mac, assurez-vous que la dernière version de Python3 est téléchargée et installée.

### 0.4.2 Installation de Thonny IDE dans le système d'exploitation Windows

Nous allons vous montrer comment télécharger et installer Thonny dans le système d'exploitation Windows. En cliquant [ici](#), vous serez redirigé vers la page officielle de Thonny IDE. Vous pouvez télécharger la dernière version de l'IDE. Une fenêtre s'affiche. Cliquez sur télécharger la version 3.3.6 pour Windows. Installer thonny ide.

### 0.4.3 Installation de Thonny IDE sous Linux

Afin d'installer Thonny IDE sur le système d'exploitation Linux, vous devez d'abord vérifier la distribution Linux que vous utilisez et installer les dépendances Python en conséquence.

Pour le système d'exploitation Linux, vous devez d'abord installer ces dépendances Python pour passer à l'étape suivante :

Cela peut être réalisé dans Ubuntu en tapant la commande suivante dans le terminal :

```
sudo apt installer python3 python3-pip python3-tk
```

Appuyez sur Entrée et tapez la commande suivante pour installer le Thonny ID

On peut aussi effectuer l'installation via pip3 en donnant la commande :

```
sudo pip3 installer thonny
```

### 0.4.4 Premier exemple

Dans notre premier exemple nous allons lancer **Thonny** et éditer un simple programme blink.py.

La figure suivante montre les fenêtres de travail dans l'**IDE Thonny**.

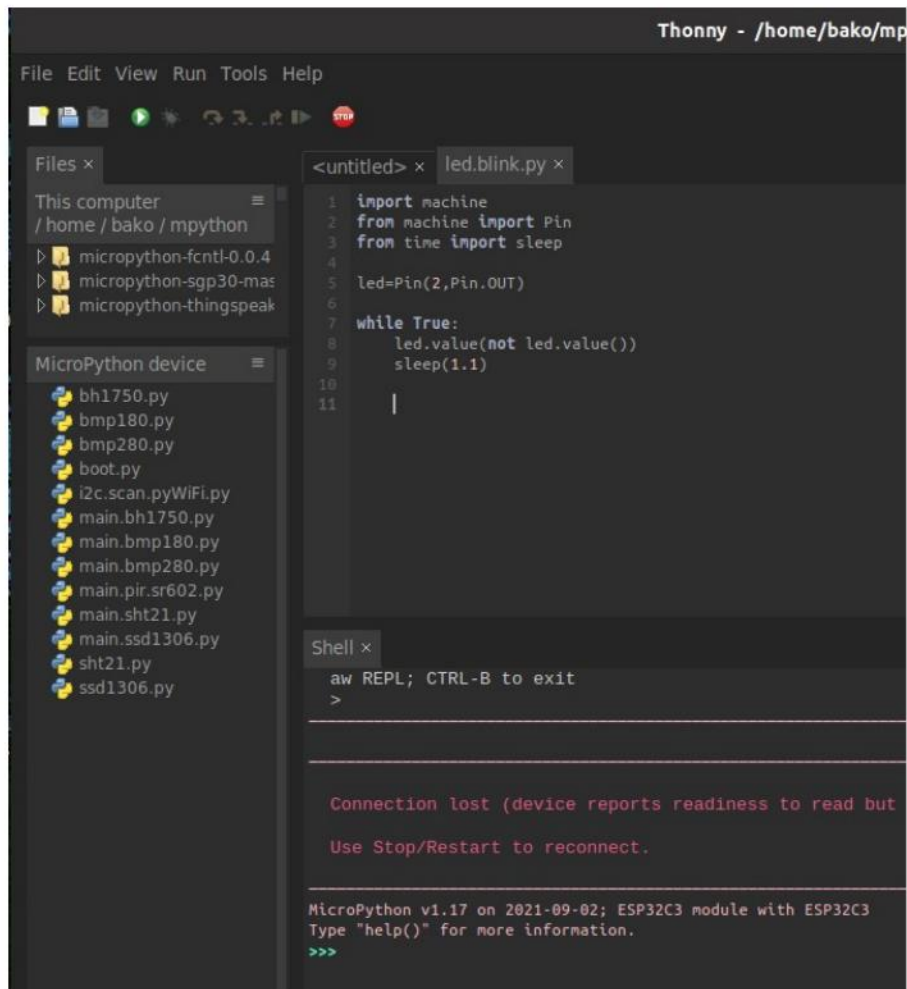
A gauche en haut nous avons le contenu du répertoire **mpython** sur notre PC. A gauche en bas nous avons la liste de programmes enregistrés sur la carte.

Au démarrage de la carte il n'existe que **boot . py** ; d'autres programmes sont chargés ultérieurement.

Dans la fenêtre principale nous affichons le contenu du dernier programme édité, ici **led.blink.py**.

Le code est :

```
import machine                # la bibliothèque principale
from machine import Pin
from time import sleep
led=Pin(2,Pin.OUT)           # le numéro du Pin LED est 2
while True:
    led.value(not led.value()) # la valeur à afficher est complémenté
    sleep(1.1)                # le temps d'attente en secondes
```



### A faire:

1. Lancer Thonny, éditer le programme et l'enregistrer sur la carte
2. Activer (charger, flasher) le programme
3. Modifier le programme, la valeur de `sleep()` et ajouter un `print()`

### Remarque:

La préparation de la carte pyWiFi-ESP32-C3 peut exiger le chargement (flash) du firmware. Le firmware est disponible sur la page :

<https://micropython.org/download/?mcu=esp32c3>

Il doit être « flashé » avec `esptool.py` :

```
python -m pip install esptool
```

```
esptool.py --chip esp32-c3 --port /dev/ttyUSB0 erase_flash
```

```
esptool.py --chip esp32c3 -p /dev/ttyUSB0 -b 460800 --before=default_reset --
after=hard_reset write_flash --flash_mode dio --flash_freq 80m --flash_size 4MB
0x0 esp32c3-20210902-v1.17.bin
```

# Laboratoire 1 – Lecture des capteurs et affichage (i2c)

## 1.0 Introduction

Dans ce laboratoire nous allons expérimenter avec l'affichage sur un écran OLED et la capture des données physiques telles que la température, humidité et la luminosité.

La communication entre le SoC IoT et ces dispositifs est effectué par l'envoi des octets représentant des adresses, des commandes et de données sur le bus I2C.

Ce bus est composé de 2 lignes (signaux) ; **SCL** qui porte le signal d'horloge et **SDA** qui porte l'information.

## 1.1 Premier exemple – l'affichage des données

Dans cet exercice nous allons simplement afficher un titre et 2 valeurs numériques sur l'écran **OLED** ajouté à notre **PYCOM-V**.

Tout d'abord il faut télécharger le driver de l'écran `ssd1306.py` disponible ici :

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

Après son chargement il faut le stocker dans votre répertoire sur PC puis il faut l'ouvrir dans l'éditeur Thonny et le **flasher** sur la carte.

Maintenant éditez la code suivant :

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

print(machine.freq())
i2c = SoftI2C(scl=Pin(0), sda=Pin(10), freq=100000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
oled.fill(0)
oled.text("SmartComputerLab", 0, 0)
oled.text("RISC-V mPython", 0, 16)
oled.text("PYCOM-V.io", 0, 32)
oled.text("WiFi/BLE/LoRa", 0, 48)
oled.show()
```

Et l'enregistrez dans le répertoire sur votre PC et flashez sur la carte.

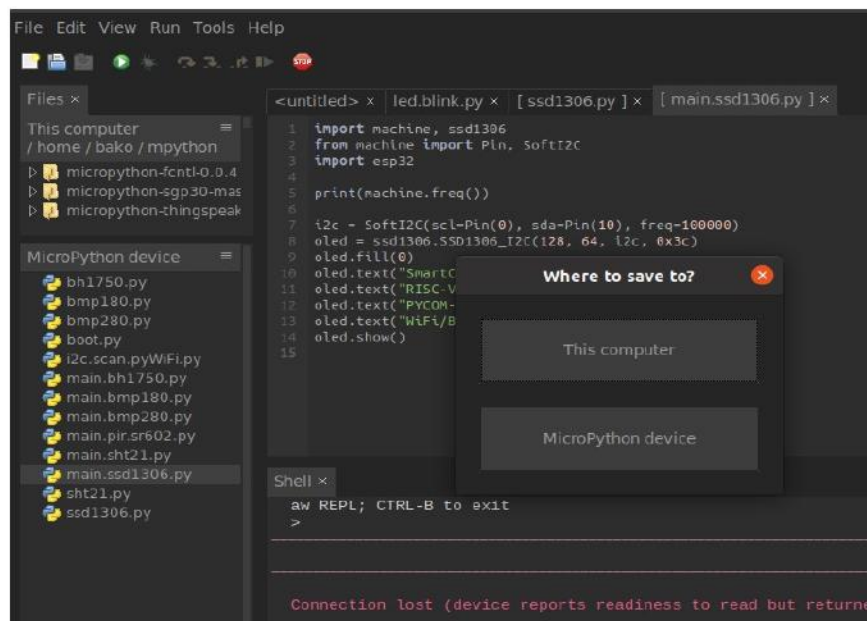


Fig 1.1 L'environnement Thonny pour le stockage/flashage d'un programme MicroPython.



Sur la figure suivante vous avez la configuration de la carte PYCOM-V avec un écran OLED (ssd1306)

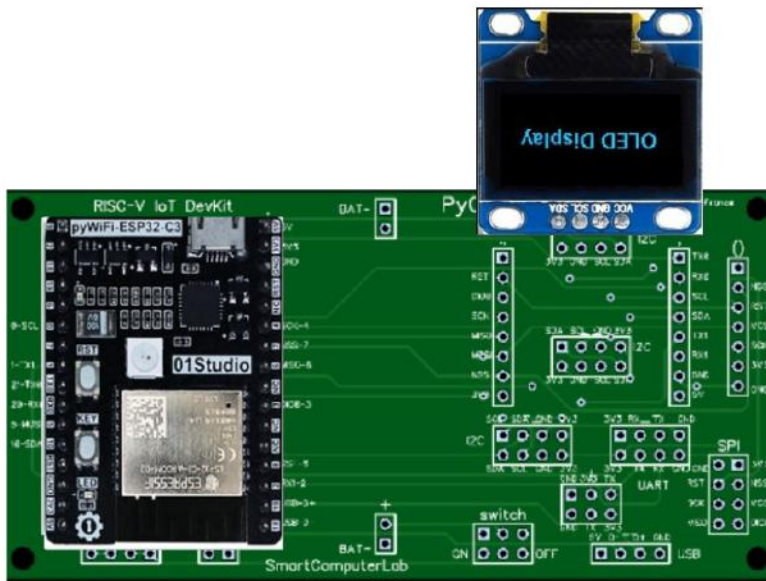


Fig 1.2 PYCOM-V avec l'écran OLED (SSD1306).

### A faire:

1. Compléter le programme en ajoutant trois variables (**d1**, **d2**, **d3**) qui sont affichées en **boucle de 16 pas**. A chaque pas incrémentez les variables par **1 (d1)**, **2 (d2)**, **3 (d3)**.
2. Editer, et charger ce programme.

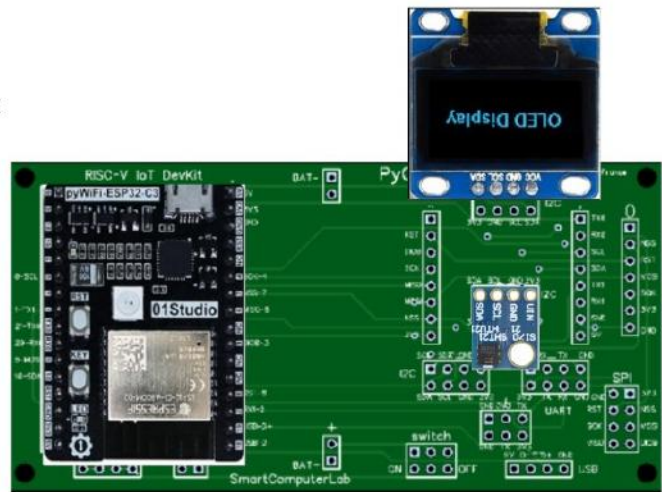
```
d1=0
d2=0
for i in range(10):
    oled.fill(0)
    oled.text(str(d1),0,0)
    oled.text(str(d2),0,16)
    oled.show()
    d1+=1
    d2+=2
    time.sleep(2)
```

## 1.2 Deuxième exemple – lecture d'un capteur (T/H) - SHT21

Dans notre deuxième exemple nous allons capter les valeurs de la température et de la luminosité sur un capteur type **SHT21**. Le capteur est connecté sur un bus I2C (comme notre écran OLED).

Sur ce bus, ligne **SDA**, le processeur envoie l'adresse du capteur pour le réveiller. Sur la ligne **SCL (S-signal, CL-Clock)** on envoie le signal de synchronisation pour synchroniser les valeurs binaires transmises sur la ligne **SDA (S-signal, D-données, A-adresse)**.

Fig 1.3 PYCOM-V avec l'écran OLED (SSD1306) et un capteur SHT21



Pour le capteur SHT21 l'adresse réservée est **0x40** en hexadécimal ou **64** en décimal.

```
data= [] # data est déclarée comme un tableau - la taille n'est pas déterminée
```

Le processeur envoie la commande de lecture de l'humidité:

```
i2c.writeto(address, b'\xF5') # Read user register for humidity
```

Après avoir reçu la commande le processeur lit la donnée (deux octets) :

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

Notez que les données de l'humidité et de la température sont reçues sur **2 octets**.

Ces 2 octets (16 bits) il faut convertir et ajuster en une valeur entière (**int**) pour l'affichage et l'exploitation à suivre.

Voici le code complet:

```
import machine
STATUS_BITS_MASK = 0xFFFC
from machine import Pin, I2C
import utime

sda=machine.Pin(10) # PYCOM-V
scl=machine.Pin(0) # PYCOM-V
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000) #I2C channel 0,pins,400kHz max
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
    #print ("Humidity = %.1f" % adjusted)
    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
    #print ("Temperature = %.1f" % adjusted)
    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
>>>

```

## A faire:

1. Etudiez le programme pour comprendre le fonctionnement du bus I2C.
2. Ajoutez l'écran OLED et complétez le programme pour pouvoir afficher les résultats.
3. Ajoutez une boucle pour lire et afficher les résultats plusieurs fois.
4. Editez le programme suivant , charger le sur la carte PYCOM-V et vérifiez son fonctionnement.

```

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))

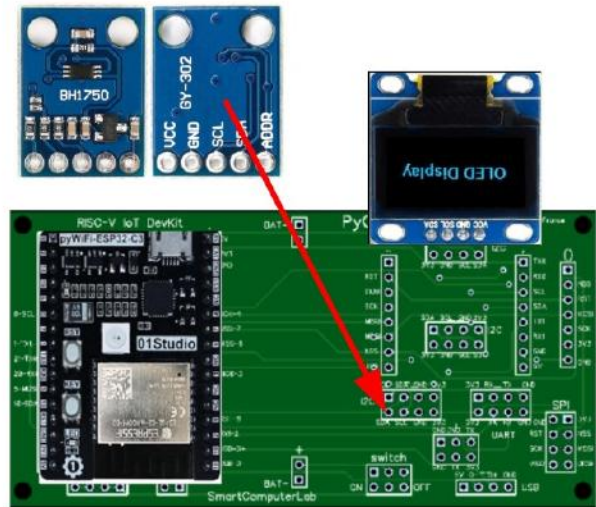
%Run -c $EDITOR_CONTENT
Warning: I2C(-1, ...) is deprecated, use SoftI2C(...) instead
Scan i2c bus...
i2c devices found: 2
Decimal address: 60 | Hexa address: 0x3c
Decimal address: 64 | Hexa address: 0x40

```

## 1.3 Troisième exemple – lecture d'un capteur (L) - BH1750

Dans cet exemple nous utilisons le capteur de la luminosité **BH1750**

Fig 1.5 **PYCOM-V** avec l'écran OLED (SSD1306) et un capteur **BH1750**



Voici le code. Le programme effectue une simple lecture par la fonction `sample()`. Cette fonction commence par l'initialisation du capteur à l'adresse - `0x23` avec 3 octets envoyés séparément par `i2c.writeto()` : `clear, power up, mode`

Ensuite on lit 2 octets - données par `i2c.readfrom(i2c_addr, 2)`, et on termine par l'envoi de la commande `power down`.

```
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):
    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

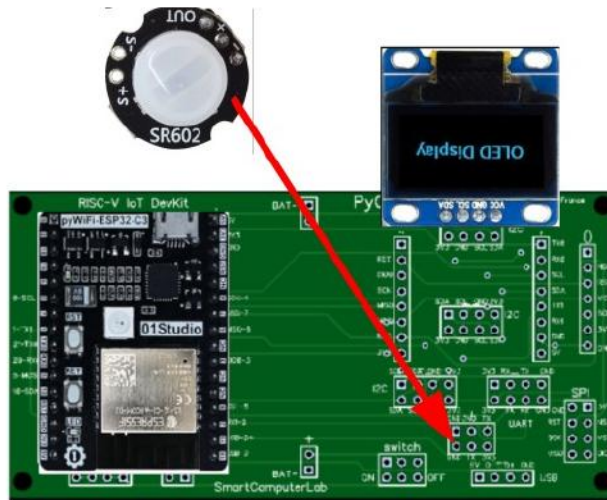
sda=machine.Pin(10)
scl=machine.Pin(0)
i2c=machine.I2C(0,sda=sda,scl=scl,freq=400000) #I2C channel 0,pins,400kHz max
lum=sample(i2c,mode=OP_SINGLE_HRES1, i2c_addr=0x23)
print('lum')
print(lum)
```

### A faire:

1. Etudiez le programme pour comprendre le fonctionnement du bus I2C.
2. Ajoutez l'écran OLED et complétez le programme pour pouvoir afficher les résultats.
3. Ajoutez une boucle pour lire et afficher les résultats plusieurs fois.

## 1.4 Quatrième exemple – lecture d'un capteur PIR - SR602

**SR602** est un capteur de présence s'active en présence de rayonnement Infra Rouge (IR). Le signal de sortie porte une valeur 1 si la présence est détectée sinon il est mis à 0.



```
from machine import Pin
import time

ldr = Pin(1, 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
```

### A faire:

1. Etudiez et testez le programme.

## Laboratoire 2 – Communication WiFi et broker MQTT

Dans ce laboratoire nous allons étudier et expérimenter avec les fonctionnalités WiFi intégrées dans le SoC ESP32 RISC-V.

Nous allons scruter les réseaux disponible avec `WiFi.scan` puis nous allons nous connecter à un **serveur/broker IoT** de type MQTT.

La communication avec un broker MQTT passe par l'utilisation de deux opérations : abonnement - **subscribe** et publication - **publish** des messages MQTT sur le **topic** – sujets proposés.

Dans cet exercice nous pouvons utiliser un smartphone avec une application **MyMQTT** pour s'abonner au sujets disponible et de poster nos message sur ces sujets.

Dans la deuxième partie du laboratoire nous allons montrer comment développer un mini-serveur WEB sur notre réseau local ou monter notre propre point d'accès.

### 2.1 Scrutation *scan* du réseau

```
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

* Livebox-08B0
  - Channel: 1
  - RSSI: -80
  - BSSID: 78:81:02:31:08:b0
```

#### A faire:

1. Etudiez et testez le programme.

Asseyez de comprendre le formatage des données retournées par la méthode `station.scan()`

#### Remarque :

Le programme de **WiFi scan** « **nettoie** » le modem WiFi en le mettant en état de base.

Vous pouvez l'utilisez en cas de problèmes de connexion dans les exemples du code à suivre.

## 2.2 Connexion au réseau WiFi , mode station - STA

Notre carte PYCOM-V 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 :

```
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"
    password    = "G79ji6..EptVTPWmZP"

    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))
    station.connect(ssid, password)

    while station.isconnected() == False:
        pass

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

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

disconnect()
print('disconnected - start connection')
connect()

>>> %Run -c $EDITOR_CONTENT
disconnected - start connection
Connection successful
('192.168.1.110', '255.255.255.0', '192.168.1.1', '8.8.8.8')
>>>
```

### A faire:

1. Etudiez et testez le programme avec votre point d'accès.
2. Enregistrer le code principal avec `def connect()` et `def disconnect()` dans un module python `wifista.py`. On va utiliser ce module dans les exemples à suivre.

## 2.3 Lecture d'une page WEB

L'exemple suivant montre comment se connecter à un pont d'accès WiFi et comment envoyer une requête HTTP pour recevoir une page WEB.

Pour faciliter le développement nous utilisons la bibliothèque `urequests.py` qui contient des méthodes de connexion au serveurs WEB et d'envoi des requêtes HTTP (`GET`,`POST`).

Voici le lien vers cette bibliothèque à télécharger sur votre PC et flasher sur la carte **PYCOM-V**:  
<https://github.com/micropython/micropython-lib/blob/master/python-ecosys/urequests/urequests.py>

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

# Pin definitions
bouton = machine.Pin(9, machine.Pin.IN, machine.Pin.PULL_UP)
led = machine.Pin(2, machine.Pin.OUT)

# Network settings
wifi_ssid = "Livebox-08B0"
wifi_password = "G79ji6dtE..VTPWmZP"

# Web page (non-SSL) to get
url = "http://www.smartcomputerlab.org"

# Create a station object to store our connection
station = network.WLAN(network.STA_IF)
station.active(True)
print("Connecting...")
station.connect(wifi_ssid, wifi_password)
print("Connected!")
print("My IP Address:", station.ifconfig()[0])
# Continually print out HTML from web page as long as we have a connection
while station.isconnected():
    # Display connection details
    print("Connected!")
    print("My IP Address:", station.ifconfig()[0])
    # Perform HTTP GET request on a non-SSL web
    response = urequests.get(url)
    # Display the contents of the page
    print(response.text)
    time.sleep(6)

# If we lose connection, repeat this main.py and retry for a connection
print("Connection lost. Trying again.")
```

### A faire:

1. Testez le programme avec votre point d'accès.
2. Utilisez le bouton et la LED pour arrêter le défilement du texte et/ou signaler un problème.
2. Modifiez le programme en utilisant notre module `wifista.py`

```
#         if bouton.value() == 0:
#             print("Dropping to REPL")
#             led.value(1)
#             sys.exit()
```



## 2.3 Simple serveur WEB – lecture d’une variable

Il est possible de créer un serveur HTTP (ou serveur WEB). Le code HTML est directement écrit dans le programme principale ou contenu dans un fichier séparé.

Communication entre client et serveur :

- Le serveur écoute le port. Il attend la connexion d'un client.
- Tant qu'aucun client ne se présente, on reste bloqué (**accept**)
- Le client envoie une requête.
- Le serveur traite la requête puis envoie la réponse.

```
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 Serveur Web</title>
    <style>
      p { font-size: 36px; }
    </style>
  </head>
  <body>
    <h2>Une variable = </h2>
    <p><span>""" + str(pot) + """</span></p>
  </body>
</html>
"""
    return html

wifista.connect()
socketServeur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
socketServeur.bind(('', 80))
socketServeur.listen(5)

while True:
    try:
        if gc.mem_free() < 102000:
            gc.collect()
        print("Attente d'une connexion client")
        connexionClient, adresse = socketServeur.accept()
        connexionClient.settimeout(4.0)
        print("Connecté avec le client", adresse)
        print("Attente requete du client")
        requete = connexionClient.recv(1024)    #requête du client
        requete = str(requete)
        print("Requete du client = ", requete)
        connexionClient.settimeout(None)
        print("Envoi reponse du serveur : code HTML a afficher")
        connexionClient.send('HTTP/1.1 200 OK\n')
        connexionClient.send('Content-Type: text/html\n')
        connexionClient.send("Connection: close\n\n")
        reponse = web_page()
        connexionClient.sendall(reponse)
        connexionClient.close()
        print("Connexion avec le client fermee")

    except:
        connexionClient.close()
        print("Connexion avec le client fermee, le programme a declenché une erreur")
```

### A faire:

1. Testez le programme avec votre smartphone
2. Utilisez le bouton et la LED pour arrêter le défilement du texte et/ou signaler un problème.

## 2.4 Simple serveur WEB – l’envoi d’une commande

Dans l’exemple précédent nous avons lit une valeur générée par notre PYCOM-V. Dans cette section nous allons envoyer une commande d’affichage (activation) à partir de notre smartphone.

### 2.4.1 Une LED RGB

Pour commencer regardons l’utilisation d’une LED-RGB intégrée sur notre carte et connectée sur la Pin 8.

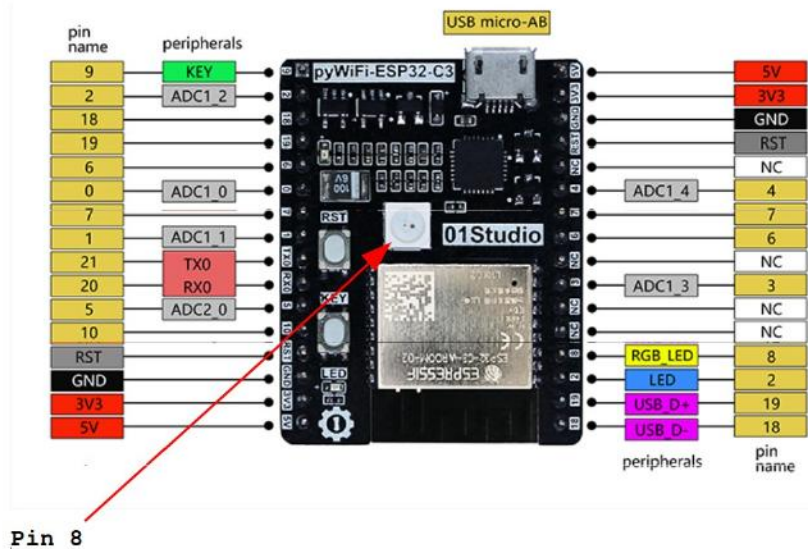


Fig 2.1. RGB LED sur Pin 8

Voici un code permettant d’allumer la LED avec la couleur RGB.

```
from machine import Pin
import neopixel

pixels = neopixel.NeoPixel(Pin(8, Pin.OUT), 1)
pixels[0] = (0xff, 0x00, 0x22)
pixels.write()
```

#### A faire:

1. Testez le programme avec différentes valeurs RGB
2. Ecrivez une boucle d’affichage pour générer une suite – arc en ciel.

## 2.4.2 Mini serveur WEB – gestion de la LED RGB

Voici le code du serveur WEB qui permet de recevoir les requêtes HTTP et d'allumer la LED avec la couleur selon la commande préprogrammée.

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

pixels = neopixel.NeoPixel(Pin(8, Pin.OUT), 1)
pixels[0] = (0xff, 0x00, 0x22)
pixels.write()

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>
            <p><a href="/?led=black">LED BLACK</a></p>
        </body>
    </html>
    """
    return html

wifista.connect()
pixels = neopixel.NeoPixel(Pin(8, Pin.OUT), 1)
pixels[0] = (0xff, 0xff, 0xff) # blanc
pixels.write()
socketServeur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
socketServeur.bind(('', 80))
socketServeur.listen(5)

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

        print("Attente connexion d'un client")
        connexionClient, adresse = socketServeur.accept()
        connexionClient.settimeout(4.0)
        print("Connecté avec le client", adresse)

        print("Attente requete du client")
        requete = connexionClient.recv(1024) #requête du client
        requete = str(requete)
        print("Requete du client = ", requete)
        connexionClient.settimeout(None)

        #analyse de la requête, recherche de led=on ou led=off
        if "GET /?led=green" in requete:
            print("LED GREEN")
            pixels[0] = (0x00, 0xff, 0x00) # RGB GREEN max
            pixels.write()
        if "GET /?led=red" in requete:
            print("LED RED")
            pixels[0] = (0xff, 0x0, 0x00) # RGB RED max
            pixels.write()
        if "GET /?led=blue" in requete:
```

```

        print("LED BLUE")
        pixels[0] = (0x00, 0x0, 0xff) # RGB BLUE max
        pixels.write()
    if "GET /?led=black" in requete:
        print("LED BLACK")
        pixels[0] = (0x00, 0x0, 0x00) # RGB BLACK max
        pixels.write()

    print("Envoi reponse du serveur : code HTML a afficher")
    connexionClient.send('HTTP/1.1 200 OK\n')
    connexionClient.send('Content-Type: text/html\n')
    connexionClient.send("Connection: close\n\n")
    reponse = web_page()
    connexionClient.sendall(reponse)
    connexionClient.close()
    print("Connexion avec le client fermee")

except:
    connexionClient.close()
    print("Connexion avec le client fermee, le programme a declenché une erreur")

```

## A faire:

1. Testez le programme avec différentes valeurs RGB
2. Ajoutez l'affichage des commandes reçues sur l'écran OLED

## 2.4.3 Mini serveur WEB avec Point d'Accès – gestion de la LED RGB

Le programme suivant est presque identique à celui présenté sans le point 2.4.2, mais il crée son propre point d'accès avec `ssid = MyAP` et l'adresse IP par défaut : `192.168.4.1` ; le mot de passe est « `smartlab` ».

Voici le code :

```

from machine import Pin
import network
import usocket as socket
import neopixel

pixels = neopixel.NeoPixel(Pin(8, Pin.OUT), 1)
pixels[0] = (0xff, 0x00, 0x22)
pixels.write()

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>
            <p><a href="/?led=black">LED BLACK</a></p>
        </body>
    </html>
    """
    return html

ssid="MyAP"
password="smartlab"
ap = network.WLAN(network.AP_IF)

```

```

ap.active(True)
ap.config(essid=ssid, password=password)
print(ap.ifconfig())

pixels = neopixel.NeoPixel(Pin(8, Pin.OUT), 1)
pixels[0] = (0xff, 0xff, 0xff) # blanc
pixels.write()
socketServeur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
socketServeur.bind(('', 80))
socketServeur.listen(5)

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

        print("Attente connexion d'un client")
        connexionClient, adresse = socketServeur.accept()
        connexionClient.settimeout(4.0)
        print("Connecté avec le client", adresse)
        print("Attente requete du client")
        requete = connexionClient.recv(1024) #requête du client
        requete = str(requete)
        print("Requete du client = ", requete)
        connexionClient.settimeout(None)

        #analyse de la requête, recherche de led=on ou led=off
        if "GET /?led=green" in requete:
            print("LED GREEN")
            pixels[0] = (0x00, 0xff, 0x00) # RGB GREEN max
            pixels.write()
        if "GET /?led=red" in requete:
            print("LED RED")
            pixels[0] = (0xff, 0x0, 0x00) # RGB RED max
            pixels.write()
        if "GET /?led=blue" in requete:
            print("LED BLUE")
            pixels[0] = (0x00, 0x0, 0xff) # RGB BLUE max
            pixels.write()
        if "GET /?led=black" in requete:
            print("LED BLACK")
            pixels[0] = (0x00, 0x0, 0x00) # RGB BLACK max
            pixels.write()

        print("Envoi reponse du serveur : code HTML a afficher")
        connexionClient.send('HTTP/1.1 200 OK\n')
        connexionClient.send('Content-Type: text/html\n')
        connexionClient.send("Connection: close\n\n")
        reponse = web_page()
        connexionClient.sendall(reponse)
        connexionClient.close()
        print("Connexion avec le client fermee")

    except:
        connexionClient.close()
        print("Connexion avec le client fermee, le programme a declenché une erreur")

```

## A faire:

1. Testez le programme
2. Ajoutez l'affichage des commandes reçues sur l'écran OLED
3. Ajoutez une carte relai permettant de piloter un équipement électrique – MAX 220V/10A

# Laboratoire 3 - Broker MQTT et serveur ThingSpeak

Dans ce laboratoire nous allons étudier et expérimenter avec les serveurs IoT type MQTT et ThingSpeak.

## 3.1 Protocole MQTT et Client MQTT

Dans cette section nous allons étudier le protocole MQTT et nous allons écrire un programme qui permet d'envoyer (publier) les messages MQTT sur un serveur-broker MQTT, puis récupérer les derniers messages postés en s'abonnant (subscribe) au topic donné.

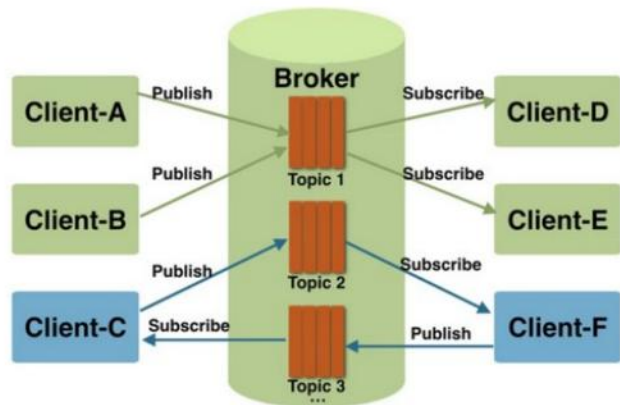
MQTT, pour 'Message Queuing Telemetry Transport', est un protocole de messagerie de publication et d'abonnement (publish/subscribe) basé sur le protocole TCP/IP. Un client, appelé *publisher*, établi dans un premier temps une connexion de type 'publish' avec le serveur MQTT, appelé *broker*.

Puis, le publisher transmet les messages au broker sur un canal spécifique, appelé *topic*. Par la suite, ces messages peuvent être lus par des abonnés, appelés *subscribers*, qui au préalable ont établi une connexion de type 'subscribe' avec le broker.

Ainsi, la transmission et la consommation des messages se font de manière asynchrone.

Le fonctionnement que nous venons de détailler est illustré dans le schéma ci-dessous.

Client-A, Client-B et Client-F sont des publishers alors que Client-C, Client-D et Client-E sont des subscribers.



Pour préparer notre programme nous avons besoin de la bibliothèque - `umqtt` .

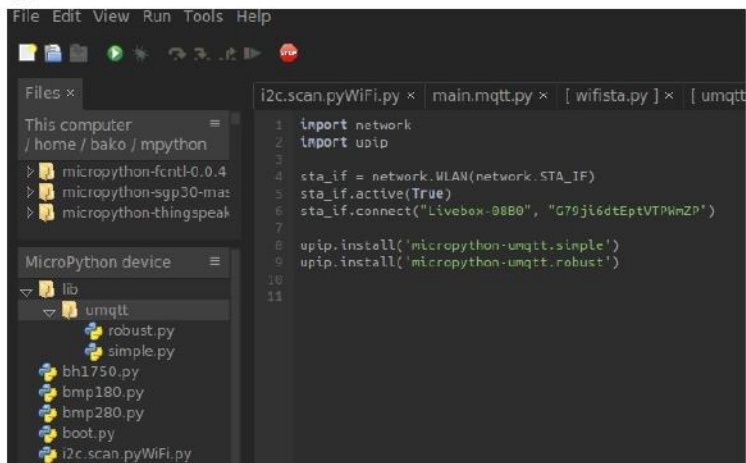
### 3.1.1 Chargement et installation de bibliothèques par upip

Dans un programme MicroPython il est possible d'installer les nouvelles bibliothèques par le gestionnaire `upip`.

Après une connexion au réseau il suffit d'appeler `upip.install(nom_du_package)` pour télécharger et installer le package demandé dans le répertoire spécifique - `lib`.

```
import upip
upip.install('micropython-your_package')
```

Voici la préparation et l'exécution d'un programme qui va chercher et installer les bibliothèques `umqtt.robust.py` et `umqtt.simple.py` .



### 3.1.2 Client MQTT – le code

Dans cet exemple, nous allons connecter notre **PYCOM-V** au serveur public gratuit **MQTT** exploité et maintenu par **EMQ X MQTT Cloud**. **EMQ X Cloud** est une plate-forme de service cloud sécurisée MQTT IoT lancée par **EMQ**.

Voici un exemple du programme qui exploite la bibliothèque `umqtt` et sa classe `MQTTClient`.

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

wifista.connect()
broker = "broker.emqx.io"
client = MQTTClient("PYCOM-V", broker)

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

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

client.reconnect()
subscribe_publish()
```

#### A faire:

1. Testez le programme sur votre smartphone avec l'application **MyMQTT**
2. Ajoutez l'affichage des messages reçues sur l'écran OLED
3. Ajoutez un capteur et postez les valeurs captées sur un **topic** (sujet)

### 3.1.3 Broker MQTT sur un PC

Il est très facile d'installer son propre broker MQTT sur un PC ; il s'appelle **mosquitto**. La page de **download** qui explique l'installation de broker (et client) **mosquitto** est disponible ici :

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

#### A faire:

1. Télécharger et installer **mosquitto**
2. Tester les programme de client MQTT avec le broker **mosquitto**

## 3.2 Serveur ThingSpeak

**ThingSpeak** est une API et une application open source pour l'«Internet des objets», permettant de **stocker** et de **collecter** les données des objets connectés en passant par le protocole HTTP via Internet ou un réseau local.

Avec **ThingSpeak**, l'utilisateur peut créer des **applications d'enregistrement** de données capteurs, des applications de suivi d'emplacements et un réseau social pour objets connectés, avec mises à jour de l'état.

Fonctions de **ThingSpeak**:

- API ouverte
- Collecte de données en temps réel
- Données de géolocalisation
- Traitement des données
- Visualisations de données
- Messages d'état des circuits
- Plugins

**ThingSpeak** peut être intégré aux plates-formes **RISC-V, ESP32, Raspberry Pi**, ..., aux applications mobiles/Web, aux réseaux sociaux et aux analyses de données avec **MATLAB (ThingSpeak.com)**

### 3.2.1 Préparation pour l'envoi des données comme messages MQTT

Pour pouvoir utiliser **ThingSpeak.com**, il faut créer un compte (gratuit) et configurer un canal - **channel** avec ses champs - **fields**. Puis il faut récupérer l'identificateur du canal et les clé d'écriture et lecture.

Dans notre exemple nous avons créé un **channel** numéro **1626377** avec une clé d'écriture **3IN09682SQX3PT4Z**.

Dans le programme suivant nous utilisons les messages type MQTT pour envoyer les données dans notre canal avec 2 champs (**température** et **humidité**).

Le topic est une chaîne :

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

et le message lui-même est :

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

Voici le code complet :

```
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)
```



### 3.2.2 Préparation pour l'envoi des données comme simples requêtes HTTP

La manière la plus simple d'envoi et réception des données sur le serveur **ThingSpeak** est l'utilisation directe de la bibliothèque **socket**.

Une connexion TCP avec socket permet d'établir lien avec le serveur ThingSpeak ( `s.connect(addr)` ), puis transmettre les **requêtes HTTP** pour envoyer/recevoir les données.

Voici un exemple simple, mais complet, avec une fonction `http_get(url)` permettant d'établir une connexion TCP/HTTP, puis d'envoyer les données (t,h) et enfin de lire une donnée (**field**) sur le canal demandé en format **json**.

```
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')
```

Résultat d'exécution

```
...
channels/1626377/fields/2/last.json?api_key=9JVTP8ZHVTB9G4TT
api.thingspeak.com
HTTP/1.1 200 OK
Date: Fri, 07 Jan 2022 15:21:16 GMT
Content-Type: application/json; charset=utf-8
Connection: close
Status: 200 OK
..
{"created_at": "2022-01-07T15:21:00Z", "entry_id": 5, "field2": "44.4"}
```

#### A faire:

1. Testez les programme ci-dessus avec votre compte ThinSpeak
2. Ajoutez un ou plusieurs capteurs pour envoyer les données réelles
3. Analysez le résultat en **json** et appliquez une fonction de décodage.

### 3.2.3 Préparation pour l'envoi des données avec thingspeak.py

Dans cet exemple nous allons utiliser une bibliothèque `thingspeak.py` disponible ici :

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

Téléchargez la et l'enregistrez sur votre PC et sur la carte PYCOM-V.

Ensuite éditez le code suivant :

```
import machine
import time
import wifista
import thingspeak

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

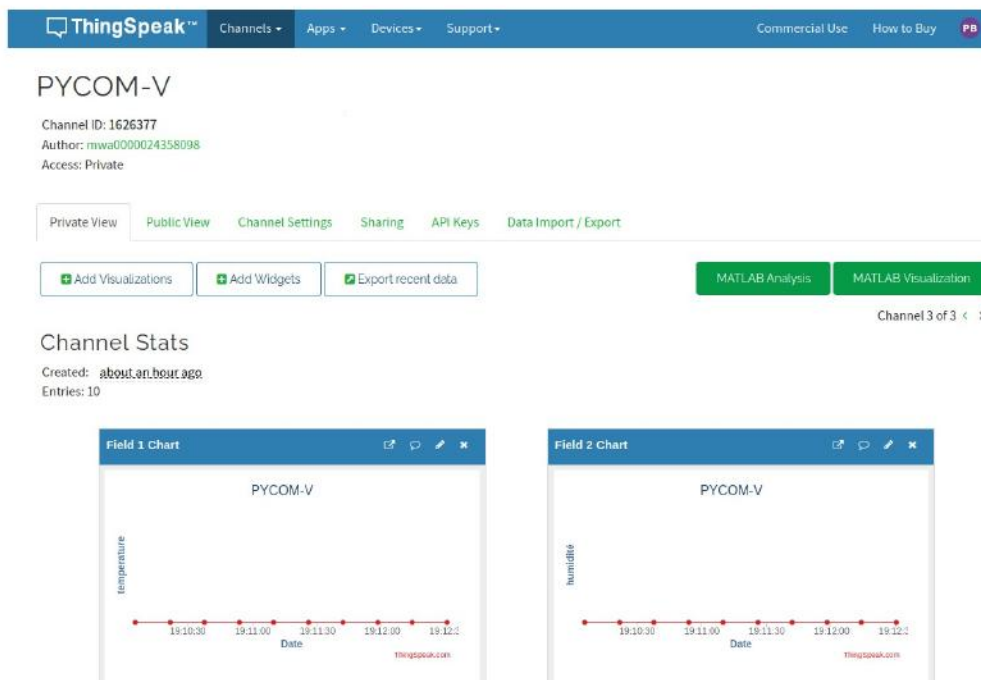
thing_speak = ThingSpeakAPI([
    Channel(channel_living_room , '3IN09682SQX3PT4Z', [field_temperature, field_humidity]),
    protocol_class=ProtoHTTP, log=True)

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)
```

Dans la dernière instruction vous pouvez noter la valeur de `sleep()` disponible pour un compte gratuit (**free**) :

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



#### A faire:

1. Testez les programme ci-dessus avec votre compte ThinSpeak
2. Ajoutez un ou plusieurs capteurs pour envoyer les données réelles

# Laboratoire 4 - Technologie LoRa et Communication Long Range

## 4.0 Introduction

Dans ce laboratoire nous allons nous intéresser à la technologie de transmission **Long Range** essentielle pour la communication entre les objets.

Longe Range ou **LoRa** permet de transmettre les données à la distance d'un kilomètre ou plus avec les débits allant de quelques centaines de bits par seconde aux quelques dizaines de kilobits (100bit - 75Kbit).

## 4.1 Modulation LoRa

Modulation LoRa a trois paramètres :

- **freq** - *frequency* ou fréquence de la porteuse de 868 à 870 MHz,
- **sf** - *spreading factor* ou étalement du spectre ou le nombre de modulations par un bit envoyé (64-4096 exprimé en puissances de 2 - 7 à 12)
- **sb** - *signal bandwidth* ou bande passante du signal (31250 Hz à 500KHz)

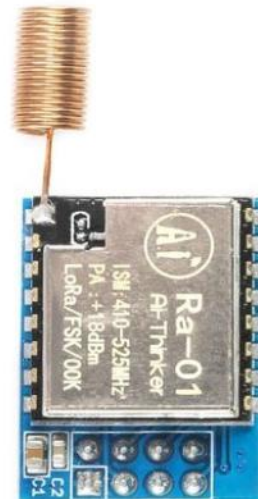
Par défaut on utilise : **freq**=434MHz ou 868MHz , **sf**=7, et **sb**=125KHz

Notre DevKiT **PYCOM-V** peut être complété par une carte d'extension - **modem LoRa**.

## 4.2 Bibliothèque **sx127x.py**

La bibliothèque **sx127x.py** permet d'intégrer les fonctionnalités du modem **sx1276/8** dans nos applications.

Ce modem-circuit est connecté à notre carte de base par le bus **SPI**. Un bus SPI fonctionne sur 3 lignes (signaux) de base : **SCK** - clock/horloge, **MISO** - Master\_In\_Slave\_Out, **MOSI** - Master\_Out\_Slave\_In, et sur sur trois lignes de contrôle : **NSS** - sélection ou activation de la sortie de Slave, **RST** - signal d'initialisation, et **DIO0/INT** - signal d'interruption envoyé par Slave activé.



Voici quelques extraits de la bibliothèque **sx127x.py**

```
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,
    }
```

Les paramètres (radio) par défaut peuvent être modifiés moyennant les fonctions disponibles dans la même bibliothèque :

```
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"])
```

### 4.3 Programme principal

Dans le programme que nous allons étudier nous trouvons l'ensemble des paramètres et des fonctions d'initialisation et d'exploitation du module LoRa.

Dans la liste `lora_default` nous proposons les paramètres compatibles avec notre modem LoRa (ISM :434MHz).

Pour l'instant on va s'intéresser seulement au 3 paramètres:

- la fréquence,
- la bande passante du signal et
- le facteur d'étalement (*spreading factor*)

Le modem est connecté sur le bus SPI ; la liste `lora_pins` identifie les numéros des signaux utilisés pour la connexion de notre modem à la carte PYCOM-V.

Finalement on détermine les paramètres et les signaux de contrôle associés au bus SPI – `lora_spi` . Avec tous les paramètres initialisés on active la communication entre la carte et le modem LoRa ( `SX127x`). Une fois la connexion activée nous pouvons appeler différentes fonctions de communication LoRa :

```
# type = 'sender'
# type = 'receiver'
# type = 'ping_master'
# type = 'ping_slave'
```

Voici le code complet :

```
from machine import Pin,SPI
from sx127x import SX127x
from time import sleep
import LoRaSender
import LoRaReceiver
import LoRaPing
import LoRaReceiverCallback
# 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': True,
    'invert_IQ': False,
    'debug': False,
}
```

```

# pins for PYCOM-V on SPI bus
lora_pins = {
    'dio_0':3,
    'ss':7,      # 16 on SPI-LoRa ext. card
    'reset':8,   # RST
    'sck':4,
    'miso':6,
    'mosi':5,
}

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 = 'ping_master'
# type = 'ping_slave'
# 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 == 'ping_master':
        LoRaPing.ping(lora, master=True)
    if type == 'ping_slave':
        LoRaPing.ping(lora, master=False)
    if type == 'receiver_callback':
        LoRaReceiverCallback.receiveCallback(lora)

```

## 4.4 Modules fonctionnels LoRa

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

Notre module émetteur (`sender`) utilise l'écran OLED pour présenter la valeur du compteur des messages LoRa.

Les données sont envoyées comme chaînes de caractères.

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

def disp(c):
    i2c = SoftI2C(scl=Pin(0), sda=Pin(10), 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)

def send_payload(lora, payload):
    print('TX: {}'.format(payload))
    lora.println(payload)
```

### A faire:

1. Testez le programme principal avec le module `LoRaSender.py` ci-dessus.
2. Ajoutez la lecture d'un capteur (`sht21.py`) et l'envoi de la valeur captée.
3. Enregistrez le programme principal (4.3) comme `main.py`, lancez son exécution, puis détachez la carte de votre PC pour qu'il fonctionne de façon autonome sur sa batterie.

## 4.4.2 Récepteur – receive (LoRaReceiver.py)

Notre module récepteur (`receive()`) utilise l'écran OLED pour présenter la valeur de RSSI correspondant aux messages LoRa reçus.

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

def disp(p):
    i2c = SoftI2C(scl=Pin(0), sda=Pin(10), 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("RX:{}".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(rssi)
            except Exception as e:
                print(e)
```

### A faire:

1. Testez le programme principal avec le module `LoRaReceiver.py` ci-dessus.
2. Ajoutez la présentation de la valeur de payload sur l'écran OLED.
3. Enregistrez le programme principal (4.3) comme `main.py`, lancez son exécution, puis détachez la carte de votre PC pour qu'il fonctionne de façon autonome sur sa batterie.

### 4.4.3 Récepteur – onReceive (LoRaReceiverCallback.py)

La réception d'un paquet LoRa peut être effectuée de façon asynchrone moyennant le signal d'interruption généré par le modem **sx127x** (INT/DIO0) au moment de la réception de la trame physique et son enregistrement dans le tampon de réception.

Voici le code :

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

def disp(p):
    i2c = SoftI2C(scl=Pin(0), sda=Pin(10), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text("LoRa onReceive", 0, 16)
    oled.text("Packet Nr:", 0, 32)
    oled.text("RX:{}".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)
```

#### A faire:

1. Testez le programme principal avec le module **LoRaReceiverCallback.py** ci-dessus.
2. Ajoutez la présentation de la valeur de payload (donnée reçue du capteur **SHT21**) sur l'écran OLED.
3. Enregistrez le programme principal (4.3) comme **main.py**, lancez son exécution, puis détachez la carte de votre PC pour qu'il fonctionne de façon autonome sur sa batterie.



# Laboratoire 5 - Développement de simples passerelles IoT

Dans ce laboratoire nous allons développer une architecture intégrant plusieurs dispositifs essentiels pour la création d'un système IoT complet. Le dispositif central sera la passerelle (*gateway*) entre les liens LoRa et la communication par WiFi.

## 5.1 Passerelle LoRa-WiFi (MQTT)

Notre premier exemple illustre la construction d'une passerelle LoRa-WiFi vers un broker MQTT. La passerelle (G - gateway) reçoit les paquets LoRa avec un payload prévu pour contenir les données des capteurs associés au terminal LoRa.

Les modules principaux à importer sont :

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

Ce module permet de définir le broker à utiliser (adresse IP) et d'établir une connexion **TCP** sur le port **1883**. (version non sécurisée)

La connexion WiFi est réalisée par notre module **wifista** ; ce module peut être modifié afin de pouvoir choisir entre une adresse statique ou dynamique.

Nous avons besoin de deux bus série : SPI et I2C (version soft). L'écran **OLED** est attaché sur le bus **SoftI2C**.

Le modem LoRa est connecté par le bus **SPI** dont paramètres sont définies dans le code. Les paramètres radio par défaut sont également définis dans le code (**lora\_default**).

```
from umqtt.robust import MQTTClient
import machine
from machine import Pin, SPI, SoftI2C
import ssd1306
import wifista
import utime as time
import gc
from sx127x import SX127x

def disp(p):
    i2c = SoftI2C(scl=Pin(0), sda=Pin(10), 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("RSSI: {}".format(p), 0, 48)
    oled.show()

# 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': True,
    'invert_IQ': False,
    'debug': False,
}

# modem - connection wires-pins on SPI bus
# pins for PYCOM-V
```

```

lora_pins = {
    'dio_0':3,
    'ss':7,      # 16 on SPI-LoRa ext. card
    'reset':8,   #RST
    'sck':4,
    'miso':6,
    'mosi':5,
}

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)

wifista.connect()
broker = "broker.emqx.io"
client = MQTTClient("PYCOM-V", broker)

def only_publish():
    count = 1
    rssi =0
    while True:
        if lora.receivedPacket():
            payload = lora.readPayload().decode()
            rssi = lora.packetRssi()
            print("RX: {} | RSSI: {}".format(payload, rssi))
            mess="RSSI: " + str(rssi)
            wifista.connect()
            client.publish(b"pycom-v/test", mess)
            count = count + 1
            disp(rssi)
            time.sleep(10)

client.reconnect()
only_publish()

```

## A faire:

1. Testez le programme ci-dessus.
2. Récupérez la valeur de **payload** (donnée reçue du capteur **SHT21**) et l'envoyez dans le message de MQTT.
3. Ecrivez la même application (apsserelle) moyennant la réception des paquets LoRa par la fonction callback (interruption)

## 5.2 Passerelle LoRa-WiFi (ThingSpeak)

La passerelle **LoRa-WiFi (ThingSpeak)** permettra de retransmettre les données reçues sur un lien **LoRa** et de les envoyer par sur une connexion WiFi vers un serveur **ThingSpeak**.

Le program suivant permet de recevoir les paquets LoRa et de le retransmettre sur un lin WiFi vers le serveur ThingSpeak. Nous utilisons ici également la notion de **topic (MQTT)** et de messages.

Le **topic** est une chaîne de caractères incluant le numéro du canal la commande publish et la clé d'écriture.

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

Le message correspond au **payload** avec les champs (**fields**) associés à chaque valeur :

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

Voici le code complet :

```
from umqtt.robust import MQTTClient
import machine
from machine import Pin,SPI,SoftI2C
import ssd1306
import wifista
import utime as time
import gc
from sx127x import SX127x

def disp(p):
    i2c = SoftI2C(scl=Pin(0), sda=Pin(10), 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("RSSI: {}".format(p), 0, 48)
    oled.show()

# 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': True,
    'invert_IQ': False,
    'debug': False,
}

# pins for PYCOM-V on SPI bus

lora_pins = {
    'dio_0':3,
    'ss':7, # 16 on SPI-LoRa ext. card
    'reset':8, #RST
    'sck':4,
    'miso':6,
    'mosi':5,
}

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)

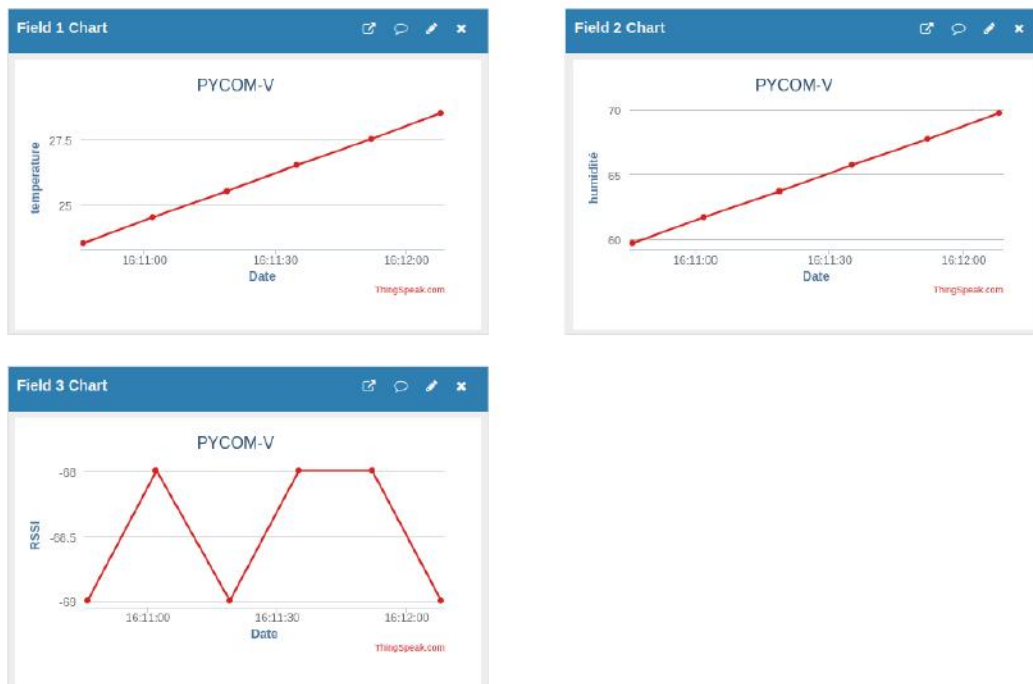
wifista.connect()
server = "mqtt.thingspeak.com"
client = MQTTClient("umqtt_client", server)
CHANNEL_ID = "1626377"
WRITE_API_KEY = "3IN09682SQX3PT4Z"

def only_publish():
    topic = "channels/" + CHANNEL_ID + "/publish/" + WRITE_API_KEY
    temp =21.5
    hum =55.7
    count = 1
    rssi =0
    while True:
        if lora.receivedPacket():
            buff = lora.readPayload().decode()
            rssi = lora.packetRssi()
            print("RX: {} | RSSI: {}".format(buff, rssi))
            mess="RSSI: " + str(rssi)
            wifista.connect()
            payload = "field1="+str(temp)+"&field2="+str(hum)+"&field3="+str(rssi)
            client.connect()
            client.publish(topic, payload)
            client.disconnect()
            temp=temp+1.0
            hum=hum+2.0
            time.sleep(15)

only_publish()

```

Le résultat d'affichage pour une séquence d'exécution de notre passerelle.



## A faire:

1. Testez le programme ci-dessus.
2. Récupérez la valeur de payload (donnée reçue du capteur **SHT21**) et l'envoyez dans le message de **MQTT**.
3. Ecrivez la même application moyennant la réception des paquets LoRa par la fonction **callback** (interruption)