IoT Labs with ESP32 (S1) and µPython

SmartComputerLab

Table of Contents

IoT Labs with ESP32 (S1) and µPython

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 one of the IoT **ESP32.D32 based IoT DevKits** from **SmartComputerLab**.

ESP32.D32 IoT DevKits contain a base card to accommodate a central board with its **IoT SoC** (**S**ystem **o**n **C**hip) and a set of expansion cards for additional sensors, actuators and modems.

The central unit is a board equipped with an **ESP32S1 SoC** integrating **WiFi**/**BT**/**BLE** modems.

In this series of IoT labs we are going to work with micro-Python using 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** and the initial **firmware** (**ESP32S1**).

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 and displays. However we have prepared an **augmented firmware** with all (almost) drivers and libraries integrated into it.

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 C**lient 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 different **LoRa modems** (ISM:434/868/2400 MHz). **LoRa** modems are integrated in the expansion boards provided for our IoT DevKits. With LoRa we can send structured data from sensors managed on terminal nodes over to another board with the same type of modem with the same frequency band . 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 (S1) 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.

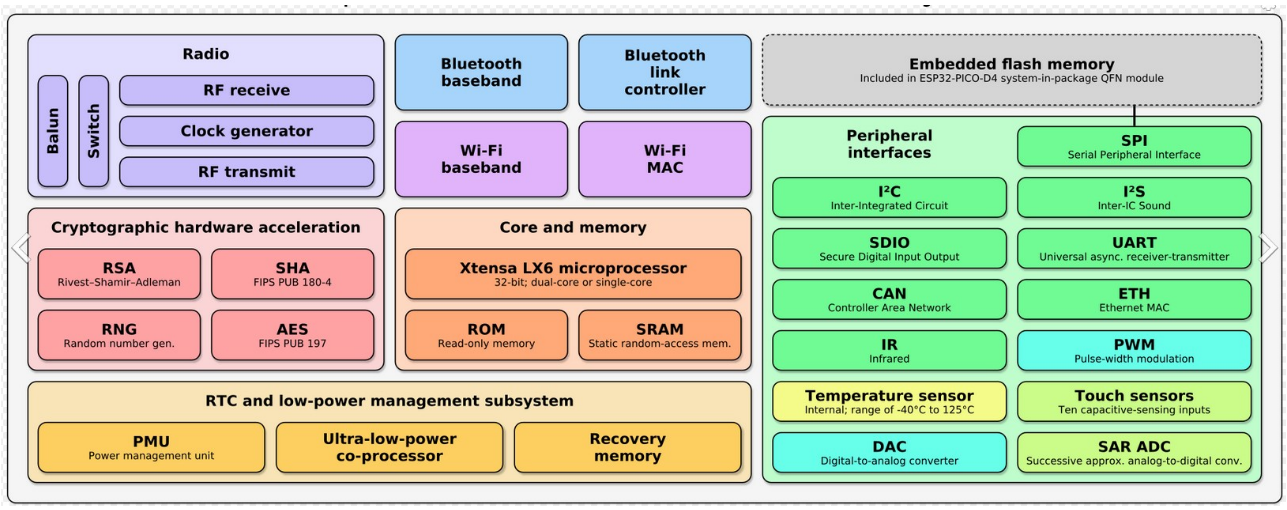

Radio
Balun
Switch
RF receive
Clock generator
RF transmit
Bluetooth baseband
Bluetooth link controller
Wi-Fi baseband
Wi-Fi MAC
Embedded flash memory
Included in ESP32-PICO-D4 system-in-package QFN module
Peripheral interfaces
SPI
Serial Peripheral Interface
I²C
Inter-Integrated Circuit
I²S
Inter-IC Sound
SDIO
Secure Digital Input Output
UART
Universal async. receiver-transmitter
CAN
Controller Area Network
ETH
Ethernet MAC
IR
Infrared
PWM
Pulse-width modulation
Temperature sensor
Internal; range of -40°C to 125°C
Touch sensors
Ten capacitive-sensing inputs
DAC
Digital-to-analog converter
SAR ADC
Successive approx. analog-to-digital conv.
Cryptographic hardware acceleration
RSA
Rivest-Shamir-Adleman
SHA
FIPS PUB 180-4
RNG
Random number gen.
AES
FIPS PUB 197
Core and memory
Xtensa LX6 microprocessor
32-bit; dual-core or single-core
ROM
Read-only memory
SRAM
Static random-access mem.
RTC and low-power management subsystem
PMU
Power management unit
Ultra-low-power co-processor
Recovery memory

Figure 0.1 ESP32S1 SoC – internal architecture

0.2 ESP32 LOLIN32 (D32) 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)

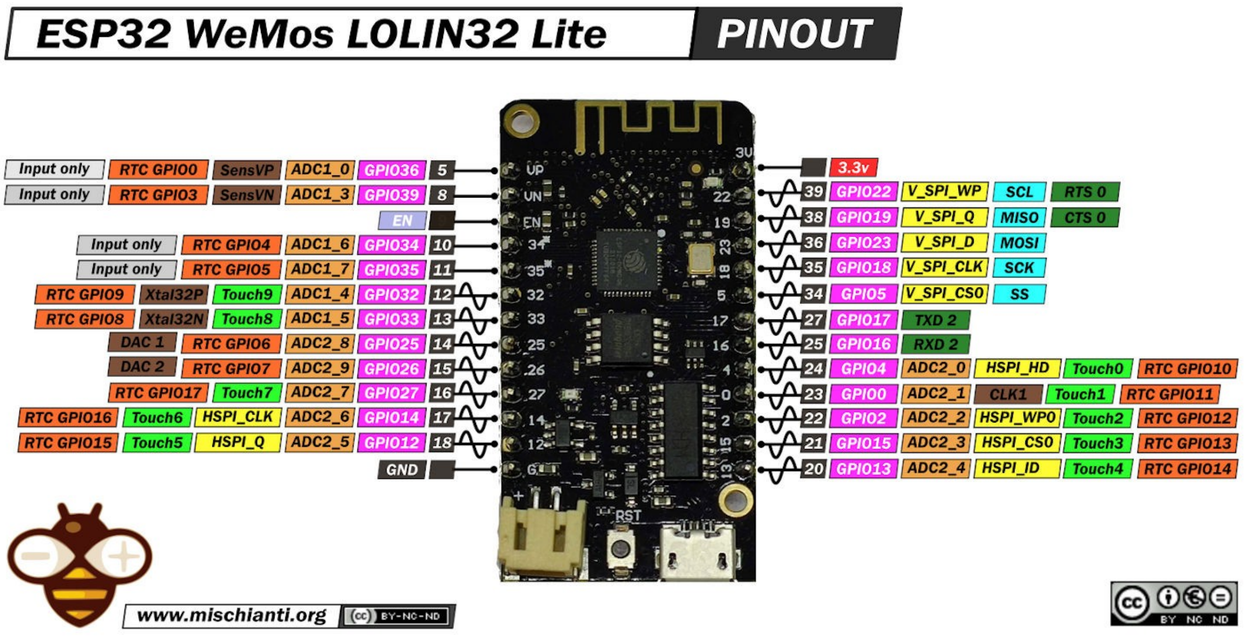
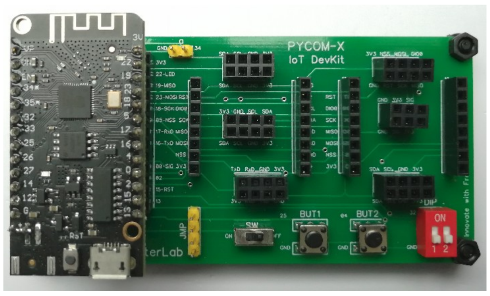

ESP32 WeMos LOLIN32 Lite
PINOUT
Input only | RTC GPIO0 | SensVP | ADC1_0 | GPIO36 | 5 | VP
Input only | RTC GPIO3 | SensVN | ADC1_3 | GPIO39 | 8 | VN
EN | EN
Input only | RTC GPIO4 | ADC1_6 | GPIO34 | 10 | 34
Input only | RTC GPIO5 | ADC1_7 | GPIO35 | 11 | 35
RTC GPIO9 | Xtal32P | Touch9 | ADC1_4 | GPIO32 | 12 | 32
RTC GPIO8 | Xtal32N | Touch8 | ADC1_5 | GPIO33 | 13 | 33
DAC 1 | RTC GPIO6 | ADC2_8 | GPIO25 | 14 | 25
DAC 2 | RTC GPIO7 | ADC2_9 | GPIO26 | 15 | 26
RTC GPIO17 | Touch7 | ADC2_7 | GPIO27 | 16 | 27
RTC GPIO16 | Touch6 | HSPI_CLK | ADC2_6 | GPIO14 | 17 | 14
RTC GPIO15 | Touch5 | HSPI_Q | ADC2_5 | GPIO12 | 18 | 12
GND | G
3V | 3.3v
22 | 39 | GPIO22 | V_SPI_WP | SCL | RTS 0
19 | 38 | GPIO19 | V_SPI_Q | MISO | CTS 0
23 | 36 | GPIO23 | V_SPI_D | MOSI
18 | 35 | GPIO18 | V_SPI_CLK | SCK
5 | 34 | GPIO5 | V_SPI_CS0 | SS
17 | 27 | GPIO17 | TXD 2
16 | 25 | GPIO16 | RXD 2
4 | 24 | GPIO4 | ADC2_0 | HSPI_HD | Touch0 | RTC GPIO10
0 | 23 | GPIO0 | ADC2_1 | CLK1 | Touch1 | RTC GPIO11
2 | 22 | GPIO2 | ADC2_2 | HSPI_WP0 | Touch2 | RTC GPIO12
15 | 21 | GPIO15 | ADC2_3 | HSPI_CS0 | Touch3 | RTC GPIO13
13 | 20 | GPIO13 | ADC2_4 | HSPI_ID | Touch4 | RTC GPIO14
RST
www.mischianti.org
BY-NC-ND

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 DevKits with ESP32S1 (D32) main board

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



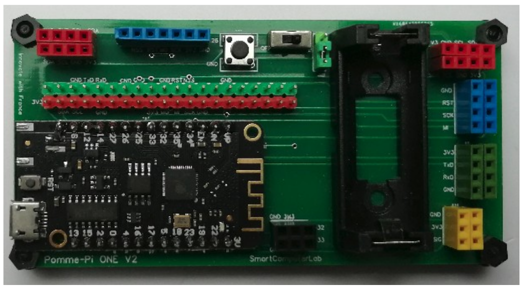
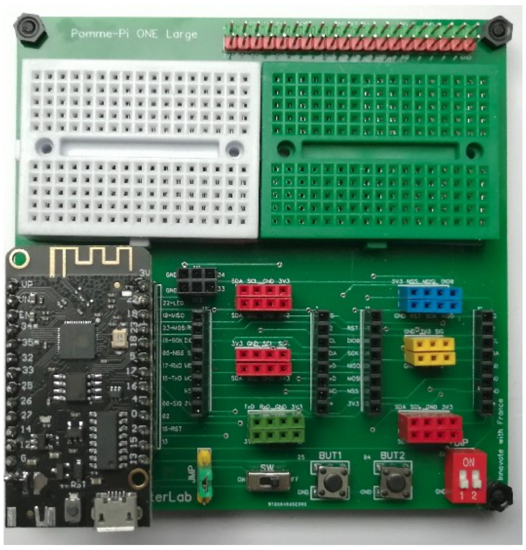


Fig 0.3 Two basic “narrow” IoT DevKits (ESP32S1) : PYCOM-X and Pomme-Pi ONE with integrated battery slot with jumper for power measurement and serial interfaces



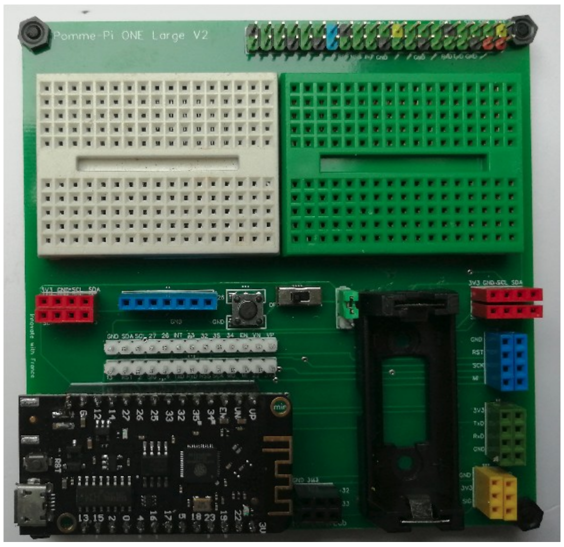


Fig 0.4 Two basic “large” Pomme-Pi ONE DevKits (ESP32S1) : with integrated battery slot with jumper for power measurement and serial interfaces including **GPIO**

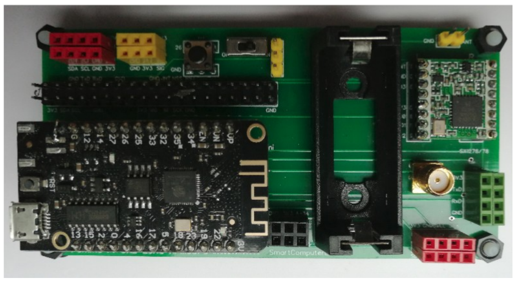
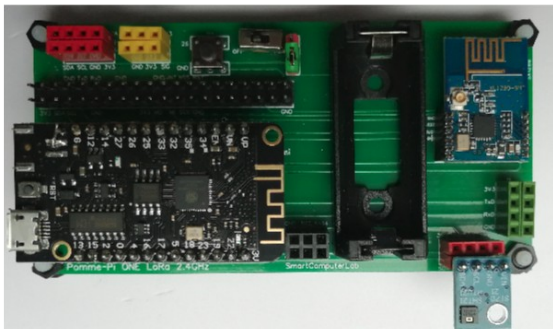



Figure 0.5 Two Pomme-Pi ONE LoRa DevKits (ESP32S1) with integrated interfaces for LoRa modems (SX1278/6 – 343/686 MHz or SX1280- 2.4 GHz)

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.

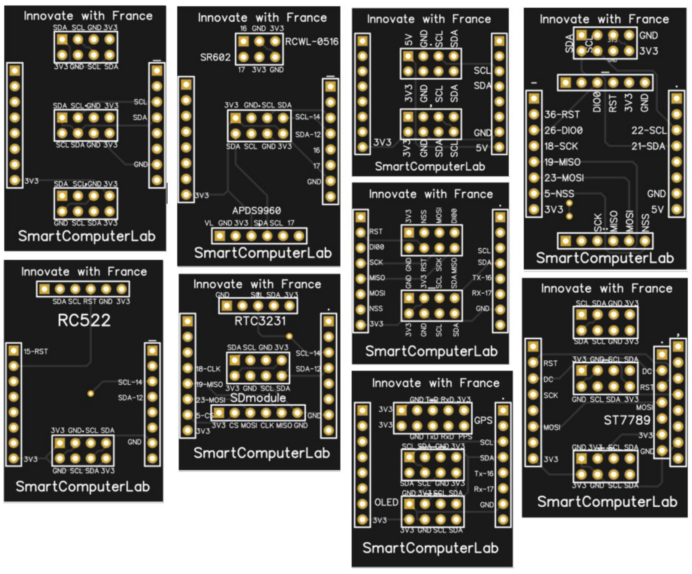


Figure 0.6 Additional expansion cards for various IoT components: sensors, displays, modems, ..

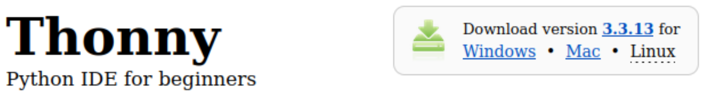
0.4 Software – Thonny IDE

0.4.1 Installing Thonny IDE - `thonny.org`

Thonny is an **open source IDE** which is used to write and upload **MicroPython** programs to different development boards such as ESP32S1/S2/S3 and C3 (RISC-V). It is an extremely interactive and easy-to-learn IDE, as it is known as the beginner-friendly IDE for new programmers.

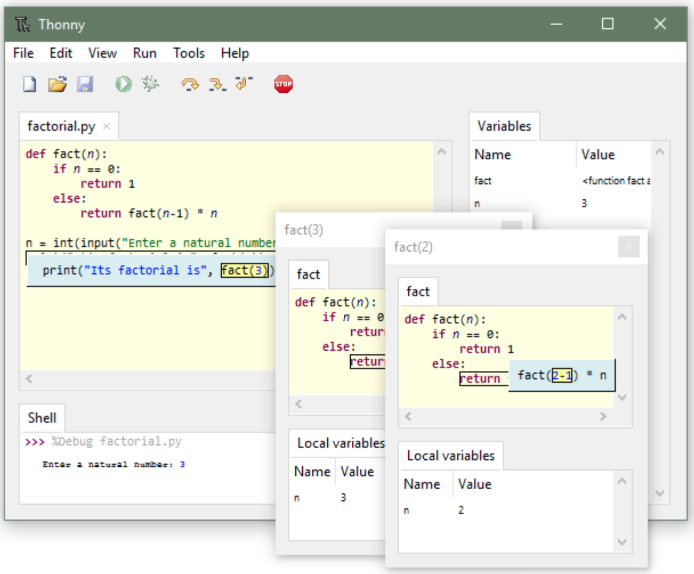
With the help of Thonny, it becomes very easy to code in MicroPython as it has an inbuilt debugger which helps to find any error in the program by debugging the script line by line.

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.



```
def fact(n):
    if n == 0:
        return 1
    else:
        return fact(n-1) * n

n = int(input("Enter a natural number
print("Its factorial is", fact(3))
```

```
>>> %Debug factorial.py
  Enter a natural number: 3
```

Name	Value
fact	<function fact a
n	3

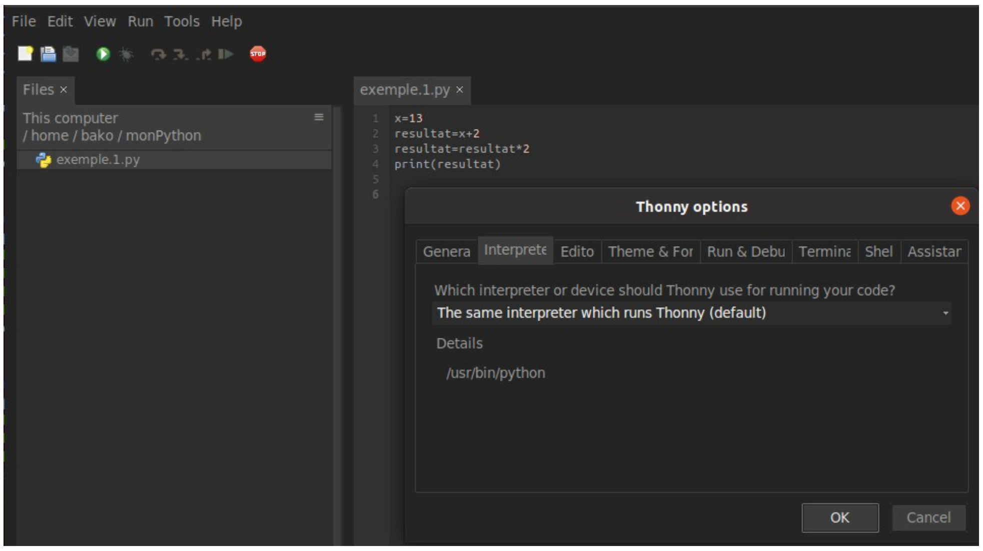
fact(2-1) * n

Local variables

Name	Value
n	3

Local variables

Name	Value
n	2



```
x=13
resultat=x+2
resultat=resultat*2
print(resultat)
```

Which interpreter or device should Thonny use for running your code?
The same interpreter which runs Thonny (default)
Details
/usr/bin/python

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

Attention:

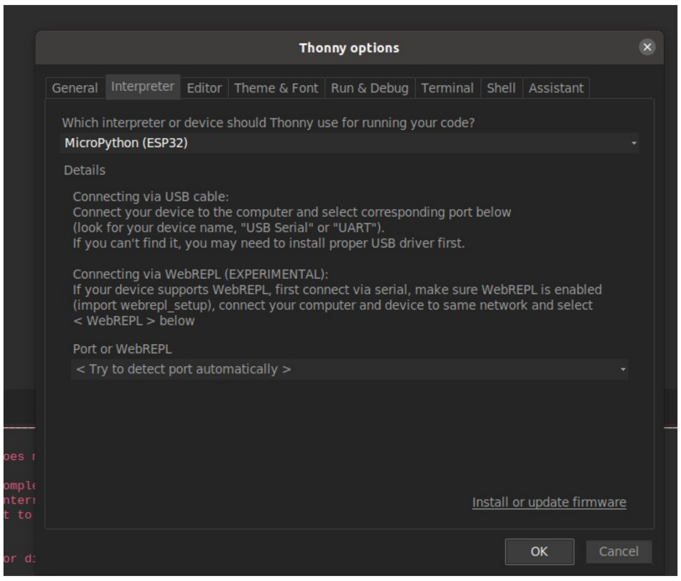
All necessary preparations for IoT Labs (and not only) including the firmware to flash on your board are available at:

`github.com/smartcomputerlab/Smart-IoT-Labs`

First download the required interpreter from Smart-IoT-Labs/firmware/**esp32**/ version:

`firmware.esp32.all.221023.bin`

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**.



Which interpreter or device should Thonny use for running your code?
MicroPython (ESP32)
Details
Connecting via USB cable:
Connect your device to the computer and select corresponding port below
(look for your device name, "USB Serial" or "UART").
If you can't find it, you may need to install proper USB driver first.

Connecting via WebREPL (EXPERIMENTAL):
If your device supports WebREPL, first connect via serial, make sure WebREPL is enabled
(import webrepl_setup), connect your computer and device to same network and select
< WebREPL > below

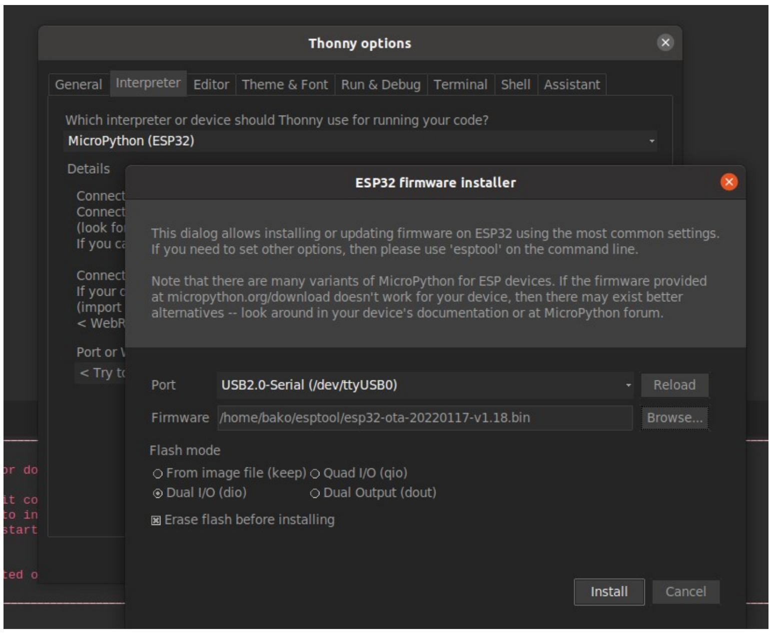
Port or WebREPL
< Try to detect port automatically >

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:

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.**



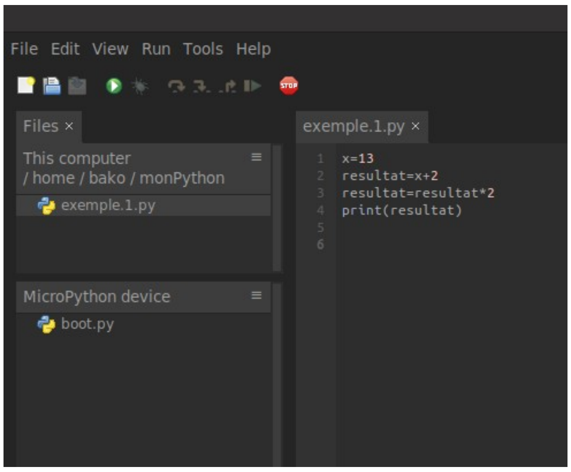
Which interpreter or device should Thonny use for running your code?
MicroPython (ESP32)

This dialog allows installing or updating firmware on ESP32 using the most common settings.
If you need to set other options, then please use 'esptool' on the command line.

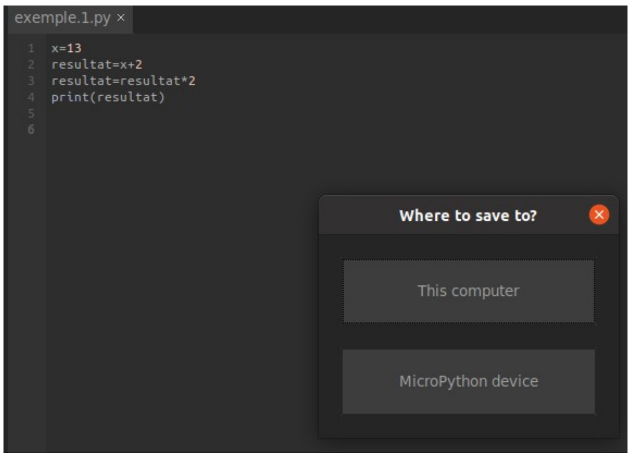
Note that there are many variants of MicroPython for ESP devices. If the firmware provided at micropython.org/download doesn't work for your device, then there may exist better alternatives -- look around in your device's documentation or at MicroPython forum.

Port: USB2.0-Serial (/dev/ttyUSB0)
Firmware: /home/bako/esptool/esp32-ota-20220117-v1.18.bin
Flash mode: ○ From image file (keep) ○ Quad I/O (qio) [✓ Dual I/O (dio)] ○ Dual Output (dout)
☒ Erase flash before installing

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)**.



```
x=13
resultat=x+2
resultat=resultat*2
print(resultat)
```

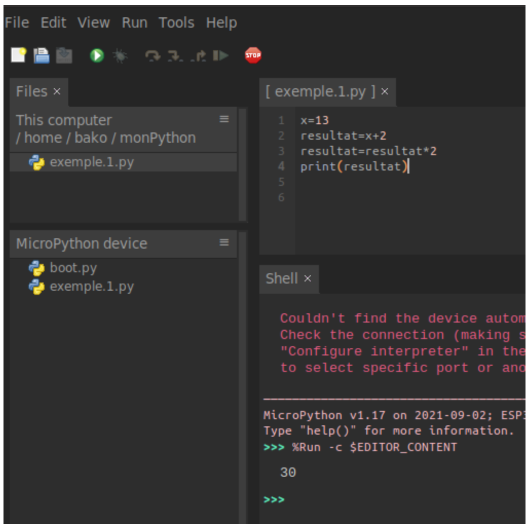


```
x=13
resultat=x+2
resultat=resultat*2
print(resultat)
```

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**.



```
x=13
resultat=x+2
resultat=resultat*2
print(resultat)
```

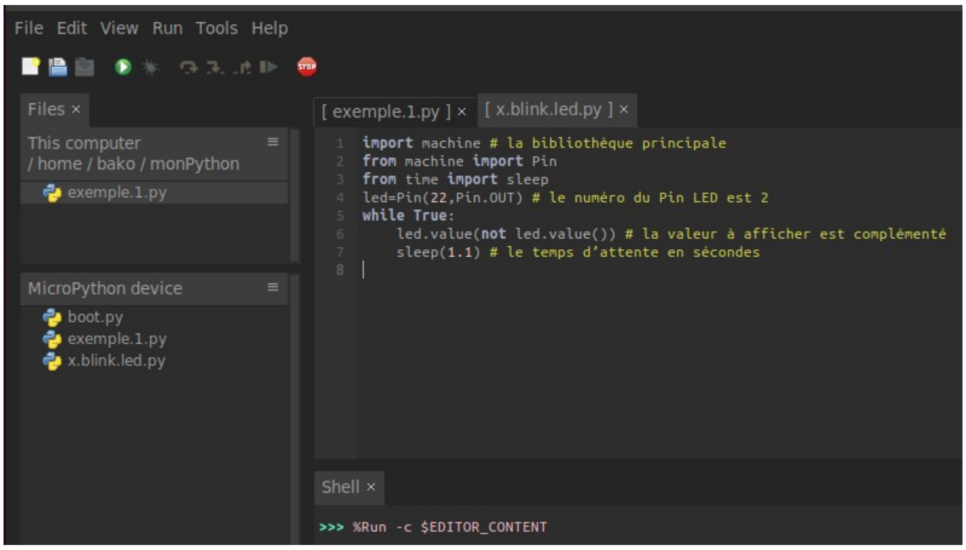
```
Couldn't find the device autom
Check the connection (making s
"Configure interpreter" in the
to select specific port or ano

MicroPython v1.17 on 2021-09-02; ESP3
Type "help()" for more information.
>>> %Run -c $EDITOR_CONTENT
  30
>>>
```

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 board with **ESP32 LOLIN32 (D32)**.
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
```



```
import machine # la bibliothèque principale
from machine import Pin
from time import sleep
led=Pin(22,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 sécondes
```

```
>>> %Run -c $EDITOR_CONTENT
```

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()`**

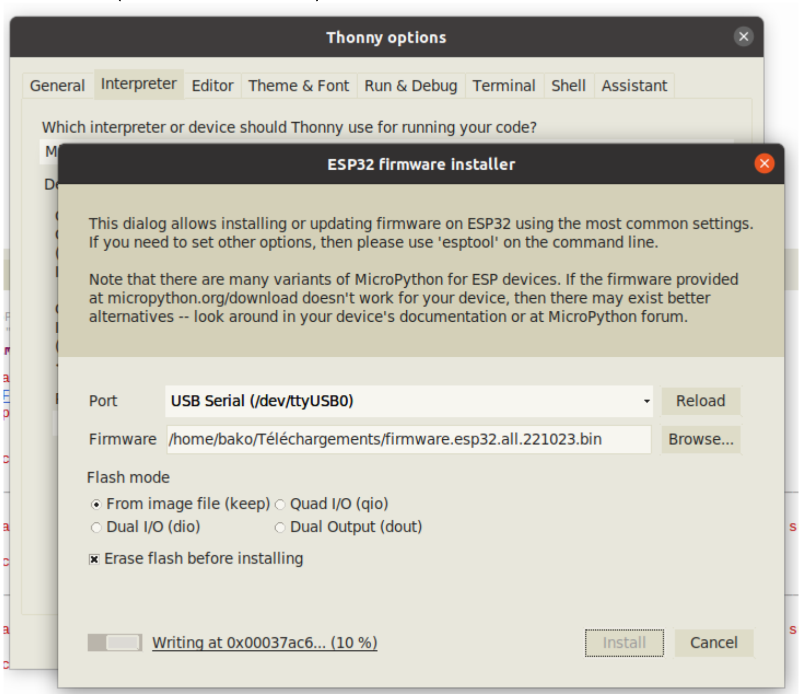
Lab 1

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 (**D**ata, **A**ddress).

Before starting the experiments with the sensor, displays, and additional libraries let us look what we have installed on our board with the provided firmware (version with `all`)



Thonny options

ESP32 firmware installer

This dialog allows installing or updating firmware on ESP32 using the most common settings. If you need to set other options, then please use 'esptool' on the command line.

Note that there are many variants of MicroPython for ESP devices. If the firmware provided at micropython.org/download doesn't work for your device, then there may exist better alternatives -- look around in your device's documentation or at MicroPython forum.

Port: USB Serial (/dev/ttyUSB0)

Firmware: /home/bako/Téléchargements/firmware.esp32.all.221023.bin

Flash mode: [✓ From image file (keep)] Quad I/O (qio) Dual I/O (dio) Dual Output (dout)

[✓ Erase flash before installing]

Writing at 0x00037ac6... (10 %)

Fig 1.1 Connecting to the board and loading the firmware with Thonny options.

Then we may launch a micropython command:

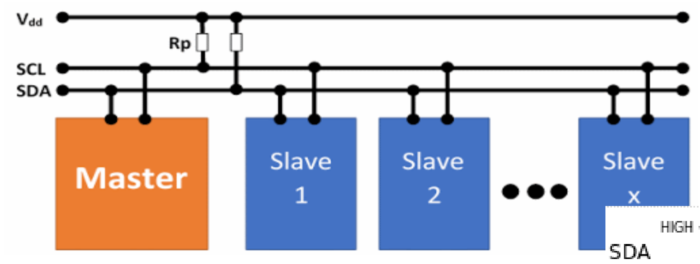
```
>>> help('modules')
CCS811            hcsr04            sht31             umachine
VL53L0X           htu21d            ssd1306           umqtt/robust
__main__          inisetup          st7789            umqtt/simple
_boot             math              sx127x            uos
_onewire          max30100          tsl2561           uplatform
_thread           max30102          uSGP30            upysh
_uasyncio         max30120          uarray            urandom
_webrepl          max44009          uasyncio/__init__ ure
apa106            max7219           uasyncio/core     urequests
bh1750            mcp9808           uasyncio/event    uselect
bme280            mfrc522           uasyncio/funcs    usocket
bmp180            micropyGPS        uasyncio/lock     ussl
bmp280            micropython       uasyncio/stream   ustruct
btree             mip               ubinascii         usys
builtins          mpu6050           ubluetooth        utime
cmath             neopixel          ucollections      utimeq
dht               network           ucryptolib        uwebsocket
ds18x20           nrf24101          uctypes           uzlib
esp               ntptime           uerrno            vl5311x
esp32             onewire           uhashlib          webrepl
flashbdev         paj7620           uheapq            webrepl_setup
framebuf          sgp30             uio
gc                sht21             ujson
Plus any modules on the filesystem
```

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 **DevKit**. With the previous command we see the **`ssd1306`** driver is already available in the firmware, no need to flash it on the board.

Remainder:

The communication between the IoT SoC and these devices is done by sending bytes representing **addresses**, **commands** and **data** over the **I2C bus**.
I2C bus consists of 2 lines (signals or wires); **SCL-5** which carries the **CLock** signal and **SDA-4** which carries the information (**D**ata, **A**ddress).



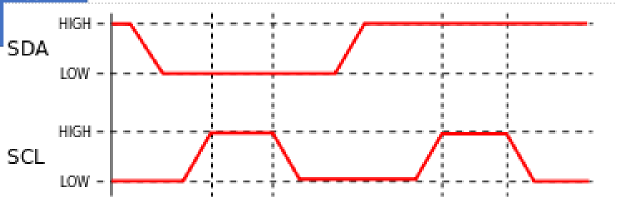


Fig 1.1 I2C bus configuration and logic levels on **SDA** - Data/Address and **SCL** – Clock lines

Before trying to display something on the OLED screen let us edit load and test if the **`ssd1306`** oled is correctly connected via the provided I2C interface: **`SDA=12, SCL=14`**.
Pay attention to the pinout of the I2C bus connectors - **SDA**,**SCL**,**GND**, and **3V3** on the board.

```
import machine
i2c = machine.I2C(scl=machine.Pin(14), sda=machine.Pin(12))
print('Scan i2c bus...')
devices = i2c.scan()
if len(devices) == 0:
  print("No i2c device !")
else:
  print('i2c devices found:',len(devices))
  for device in devices:
    print("Decimal address: ",device," | Hexa address: ",hex(device))
```

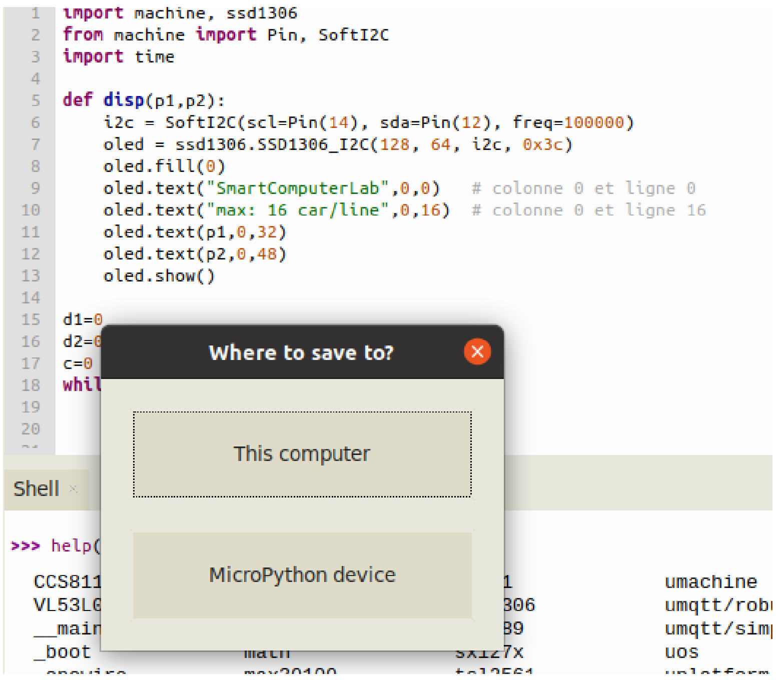
Execution:

```
Warning: I2C(-1, ...) is deprecated, use SoftI2C(...) instead
Scan i2c bus...
i2c devices found: 1
Decimal address:  60  | Hexa address:  0x3c
```

Now let us 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
    time.sleep(2)
```



```
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
whil
```

```
>>> help(

  CCS811                                  1                    umachine
  VL53L0                                  306                  umqtt/rob
  __main                                  39                   umqtt/sim
  _boot                                   sx127x               uos
```

Fig 1.1 Saving code to the selected directory **on your PC** and re-**flashing it to the card (MicroPython device)**.

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` **and display it on the OLED screen.**

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).

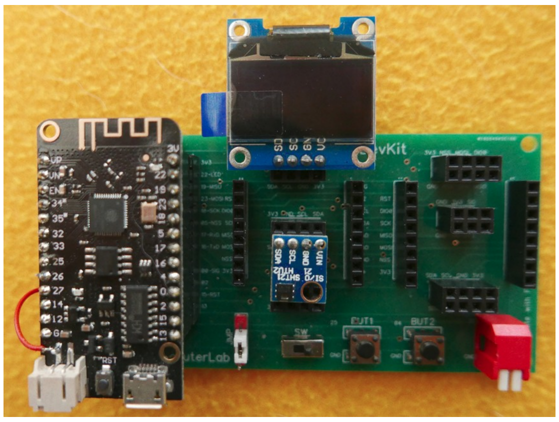


Fig 1.3 ESP32S1 DevKit with OLED display (**SSD1306**) and **SHT21** sensor **on the same I2C bus**

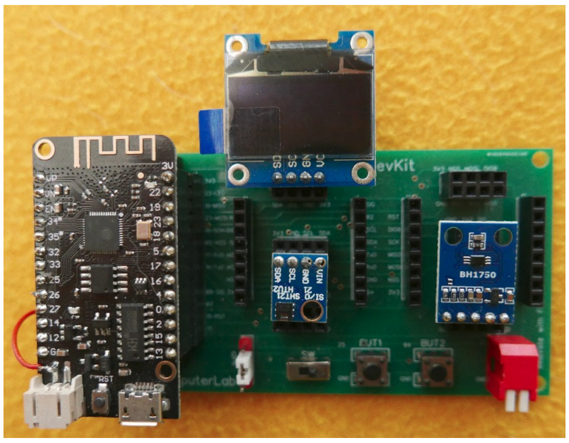
1.2.1 Preparing the code

```
import machine, time
import sht21 # librairie capteur SHT21
i2c = machine.I2C(scl = machine.Pin(14), sda = machine.Pin(12), freq=100000)
c=0
while(c<20):
    if (sht21.SHT21_DETECT(i2c)):
        sht21.SHT21_RESET(i2c)
        resolution = 2
        #sht21.SHT21_SET_RESOLUTION(i2c, resolution)
        #serial_number_sht21 = sht21.SHT21_SERIAL(i2c)
        temperature = sht21.SHT21_TEMPERATURE(i2c)
        humidity = sht21.SHT21_HUMIDITE(i2c)
        print("T: {:.2f}".format(temperature))
        print("H: {:.2f}".format(humidity))
        c=c+1
        time.sleep(2)
```

To do :

1. Add the OLED display and complete the program to show the results.
2. Add a loop to read and display the results multiple times.

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



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

Fig 1.5 The board with OLED display (SSD1306) and **SHT21** and **BH1750** sensors

Here is the code using integrated BH1750 driver.

```
import machine
from bh1750 import BH1750
import time
sda=machine.Pin(12) # ESP32S1 D32
scl=machine.Pin(14)
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000) #I2C channel 0,pins,400kHz max
s = BH1750(i2c)
c=0
while c<100:
    lumi=s.luminance(BH1750.ONCE_HIRES_1)
    c+=1
    print(int(lumi))
    time.sleep(2)
```

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

To do:

1. Add the OLED screen and complete the program to show the results.
2. 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.

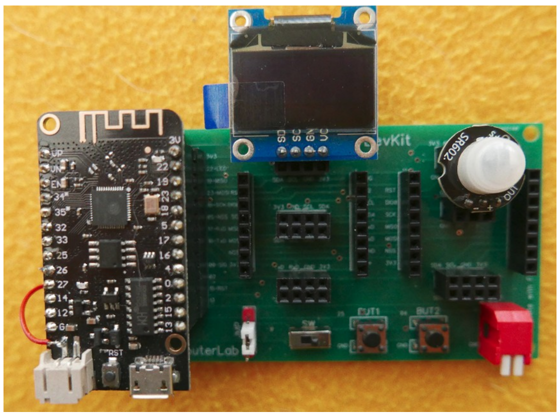


Fig 1.6 IoT DevKit with OLED display (**SSD1306**) and **PIR** motion/presence sensor: **SR602**

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

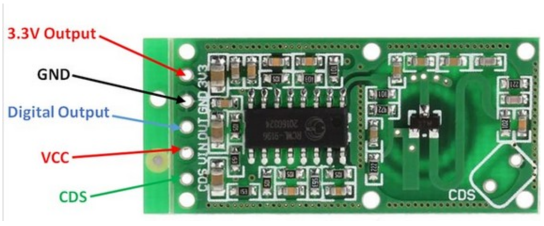
```
from machine import Pin
import time
ldr = Pin(0, Pin.IN) # create input pin on GPIO2
while True:
    if ldr.value():
        print('OBJECT DETECTED')
    else:
        print('ALL CLEAR')
    time.sleep(1)
```

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

To do:

1. Study and test the program. Note the delay (about 3 sec) between the consecutive detections.
2. Use the micro Radar sensor with the same code; discuss the **behavioral differences** between the **SR602** (PIR) and **RCWL-0516** radar sensor

Fig 1.7 RCWL-0516 radar sensor



1.5 Fifth example – using RGB ring (`neopixel`)

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

Fig 1.8 RGB led ring

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

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

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

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

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

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

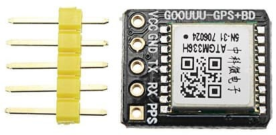
To do:

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

1.6 Sixth example – GPS module: NEO-6M (UART)

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

Fig 1.8 GPS module: **ATGM336H GPS+BD**



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

```
import time
import machine
from machine import Pin, SoftI2C
from micropyGPS import MicropyGPS
import ssd1306
import _thread
import time
WIDTH  = 128
HEIGHT = 64

def main():
    i2c = SoftI2C(scl=Pin(5), sda=Pin(4), freq=100000)
    dsp = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    uart = machine.UART(1, rx=0, tx=1, baudrate=9600, bits=8, parity=None, stop=1, timeout=5000,
rxbuf=1024)
    gps = MicropyGPS()
    while True:
      buf = uart.readline()
      for char in buf:
        gps.update(chr(char))  # Note the conversion to to chr, UART outputs ints normally
      #print('UTC Timestamp:', gps.timestamp)
      #print('Date:', gps.date_string('long'))
      #print('Latitude:', gps.latitude)
      #print('Longitude:', gps.longitude_string())
      #print('Horizontal Dilution of Precision:', gps.hdop)
      #print('Altitude:', gps.altitude)
      #print('Satellites:', gps.satellites_in_use)
      #print()
      dsp.fill(0)
      y = 0
      dy = 10
      dsp.text("{}".format(gps.date_string('s_mdy')), 0, y)
      dsp.text("Sat:{}".format(gps.satellites_in_use), 80, y)
      y += dy
      dsp.text("{:02d}:{:02d}:{:02.0f}".format(gps.timestamp[0],gps.timestamp[1],gps.timestamp[2]),
0, y)
      y += dy
      dsp.text("Lat:{}{:3d}'{:02.4f}".format(gps.latitude[2],gps.latitude[0],gps.latitude[1]),0,y)
      y += dy
      dsp.text("Lo:{}{:3d}'{:02.4f}".format(gps.longitude[2],gps.longitude[0],gps.longitude[1]),0,y)
      y += dy
      dsp.text("Alt:{:0.0f}ft".format(gps.altitude * 1000 / (12*25.4)),  0, y)
      y += dy
      dsp.text("HDP:{:0.2f}".format(gps.hdop),  0, y)
      dsp.show()

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

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

Run and be patient (after 5 min , at least !):

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

Lab 2

WiFi communication and WEB servers

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

Our 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:

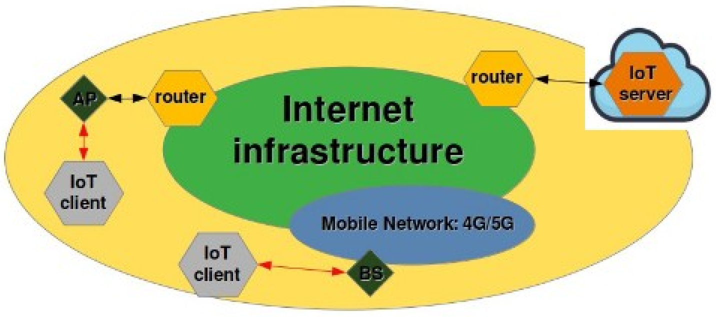
Use the LED to signal the reading of a page.

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)
```

2.4 Getting time from NTP server

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



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

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

To do:

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

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

Remainder:

Using `neopixel` LED-ring:

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

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

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

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

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

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

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

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

2.5 Simple WEB server – reading a variable

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

- 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 ESP32</h2>
            <h3>A variable = </h3>
            <p><span>""" + str(pot) + """</span></p>
        </body>
    </html>
    """
    return html

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

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

To do:

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

2.6 Simple WEB server – sending an order

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

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

i2c = SoftI2C(scl=Pin(14), sda=Pin(12), freq=100000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
oled.fill(0)
oled.text("SmartComputerLab", 0, 0)
oled.show()

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

wifista.connect()
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind(('', 80))
serverSocket.listen(5)
while True:
    try:
        if gc.mem_free() < 102000:
            gc.collect()
        print("Attente connexion d'un client")
        clientConnection, adresse = serverSocket.accept()
        clientConnection.settimeout(4.0)
        print("Connected to client", adresse)
        print("Waiting for client")
        request = clientConnection.recv(1024)     # request from client
        request = str(request)
        print("Request from client = ", request)
        clientConnection.settimeout(None)
        #analyse de la requête, recherche de led=on ou led=off
        if "GET /?led=green" in request:
            print("LED GREEN")
            oled.fill(0)
            oled.text("LED GREEN", 0, 0)
            oled.show()
        if "GET /?led=red" in request:
            print("LED RED")
            oled.fill(0)
            oled.text("LED RED", 0, 0)
            oled.show()
        if "GET /?led=blue" in request:
            print("LED BLUE")
            oled.fill(0)
            oled.text("LED BLUE", 0, 0)
            oled.show()
        print("Sending response to server : HTML code to display")
        clientConnection.send('HTTP/1.1 200 OK\n')
        clientConnection.send('Content-Type: text/html\n')
        clientConnection.send("Connection: close\n\n")
        reponse = web_page()
        clientConnection.sendall(reponse)
        clientConnection.close()
        print("Connexion avec le client fermee")

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

To do:
1. Analyze the program
1. Test the program with different messages to display

2.4.3 Mini WEB server with Access Point – RGB LED management

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

Here goes the code:

```
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
3. **Use RGB led** ring to signal message (color)

Lab 3

MQTT Broker and ThingSpeak Server

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

3.1 MQTT Protocol and MQTT Client

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

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

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

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

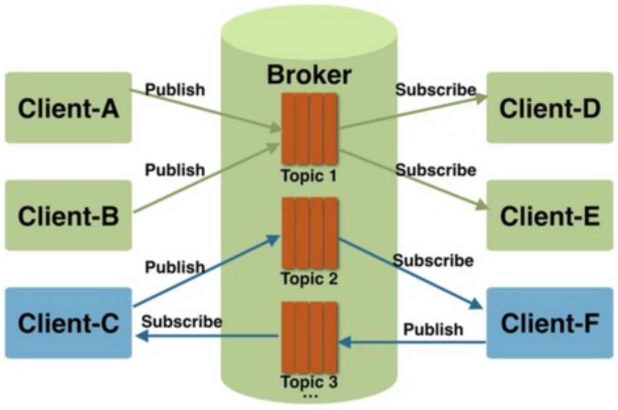


Fig. 3.1 Client-A, Client-B and Client-F are publishers while Client-C, Client-D and Client-E are subscribers.

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

3.1.1 MQTT client – the code

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

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

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

```
from umqtt.robust import MQTTClient
import machine
import wifista
import utime as time
import gc
wifista.connect()
broker = "broker.emqx.io"
client = MQTTClient("ESP32S1", 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"esp32s1/test")
    while True:
        client.check_msg()
        mess="hello: " + str(count)
        client.publish(b"esp32s1/test", mess)
        count = count + 1
        time.sleep(20)

client.reconnect()
subscribe_publish()

>>> %Run -c $EDITOR_CONTENT
Already connected
('192.168.43.136', '255.255.255.0', '192.168.43.1', '192.168.43.1')
(b'esp32s1/test', b'hello: 1')
ESP received b'hello: 1'
(b'esp32s1/test', b'hello: 2')
ESP received b'hello: 2'
```

To do:

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

3.1.2 Broker MQTT on a PC

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

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

To do:

1. Download and install **`mosquitto`**
2. Test MQTT client programs with **`mosquitto`** broker

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

3.2 ThingSpeak server

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

ThingSpeak Features:

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

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

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

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

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

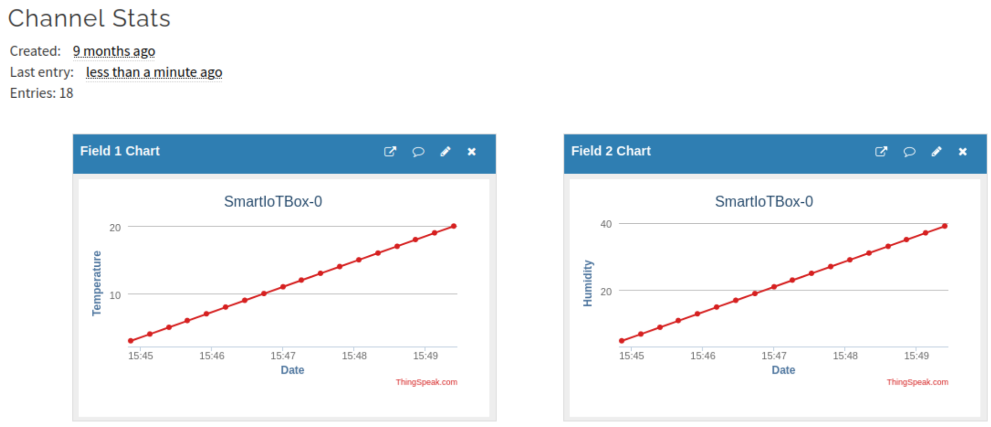
Download it and save it on your PC and on the **IoT DevKit** board. An example of use of **`thingspeak.py`** library is given below:

```
import machine
import time
import wifista
import thingspeak
from thingspeak import ThingSpeakAPI, Channel, ProtoHTTP
channel_living_room = "1538804"
field_temperature = "Temperature"
field_humidity = "Humidity"
thing_speak = ThingSpeakAPI([
    Channel(channel_living_room , 'YOX31M0EDKO0JATK', [field_temperature, field_humidity])],
    protocol_class=ProtoHTTP, log=True)
wifista.connect()
active_channel = channel_living_room
temperature = 2.0
humidity=3.0
c=0
while c<20:
    thing_speak.send(active_channel, {
        field_temperature: temperature,
        field_humidity: humidity
    })
    temperature=temperature+1.0
    humidity=humidity+2.0
    c=c+1
    time.sleep(thing_speak.free_api_delay)
```

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

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

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


Channel Stats

Created: 9 months ago

Last entry: less than a minute ago

Entries: 18

To do:

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

3.2.2 Preparation for sending 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 `1538804` with a write key `YOX31M0EDKO0JATK`.

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 = "1538804"
WRITE_API_KEY = "YOX31M0EDKO0JATK"
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)
```

To do:

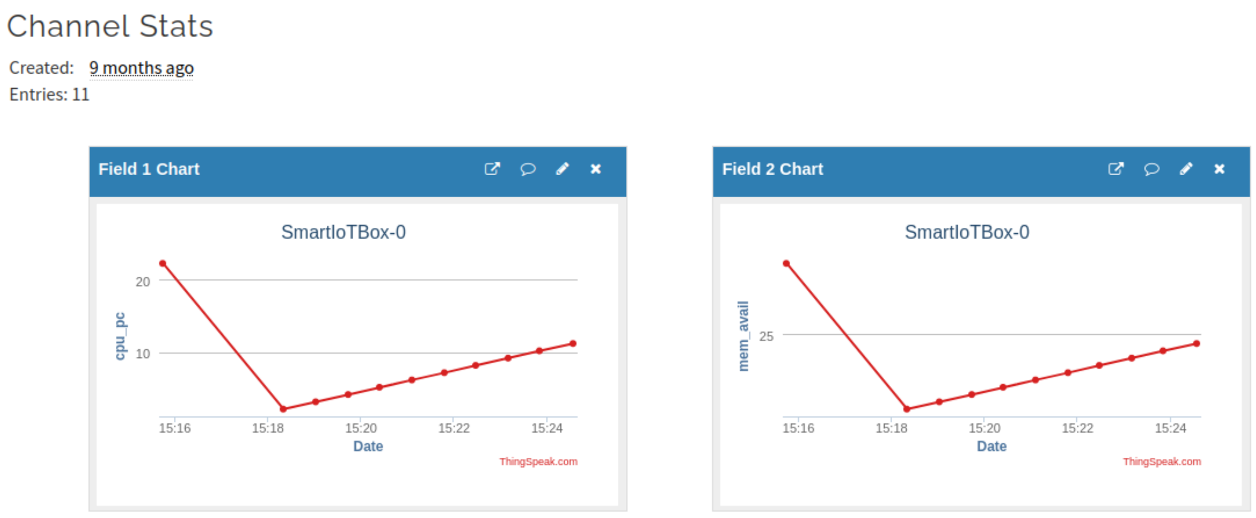
1. Log in to **your** `ThingSpeak.com` account and test the program
2. Add a sensor and post the values captured in a topic with the channel and the corresponding fields

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()

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


Channel Stats

Created: 9 months ago

Entries: 11

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

```
Already connected
```

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

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

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

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

To do:

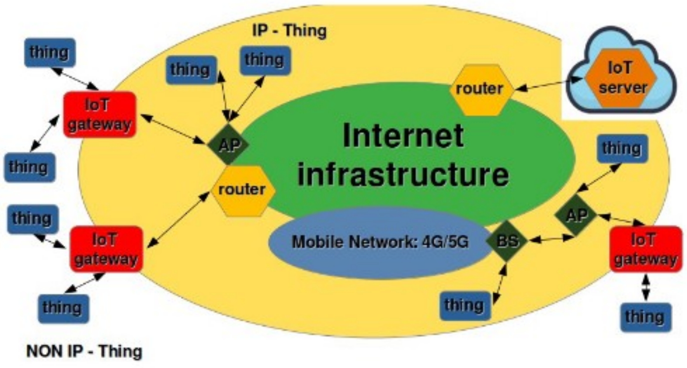
1. Test the above programs with your ThingSpeak account
2. Add one or more sensors to send the actual data
3. Parse the result in **json** format with a **decode function**

Lab 4

LoRa technology for Long Range communication

4.0 Introduction

In this lab we will focus on the **Long Range** transmission technology essential for communication between remote objects that are not connected directly to the Internet Infrastructure.
LoRa (from "long range") is a physical proprietary radio communication technique. It is based on spread spectrum modulation techniques derived from chirp spread spectrum (CSS) technology. It was developed by **Cycleo** (patent 9647718-B2), a company of Grenoble, France , later acquired by **Semtech** (USA) for 5 million dollars.
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"])
```

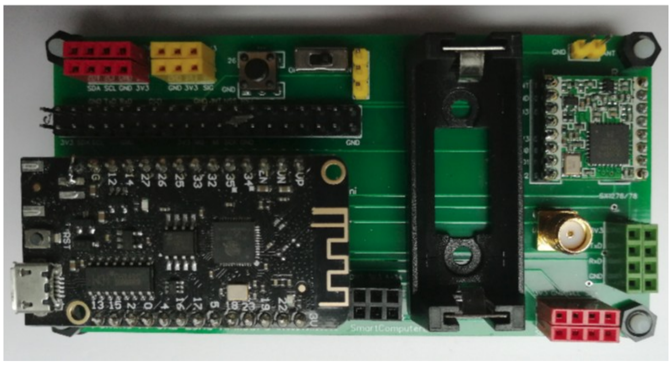
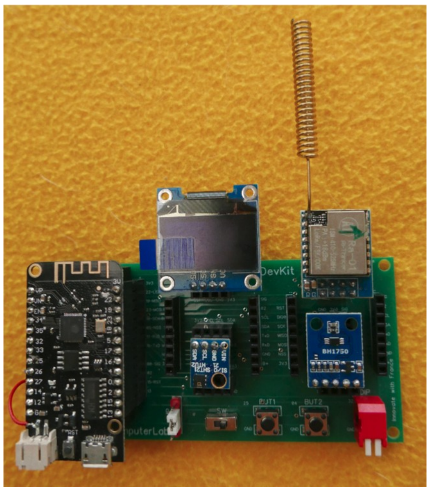



Fig 4.2 (a) Integrated LoRa module (SPI)**, (b)** Connecting the external LoRa (Ra-01) (SPI) module to the **DevKit**

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** (**R**eceived **S**ignal **S**trength **I**ndicator) value corresponding to the received LoRa messages.

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

def disp(p):
    i2c = SoftI2C(scl=Pin(14), sda=Pin(12), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text("LoRa receiver", 0, 16)
    oled.text("Packet Nr:", 0, 32)
    oled.text(format(p), 0, 48)
    oled.show()

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

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

To do:

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

4.4.3 Receiver – `onReceive (LoRaReceiverCallback.py)`

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

Here is the code:

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

def disp(p):
    i2c = SoftI2C(scl=Pin(14), sda=Pin(12), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text("LoRa receiver", 0, 16)
    oled.text("Packet Nr:", 0, 32)
    oled.text(format(p), 0, 48)
    oled.show()

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

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

To do:

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

Lab 5

Development of simple IoT gateways

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

5.1 LoRa-WiFi Gateway (MQTT)

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

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

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

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

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

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

# modem - connection wires-pins on SPI bus

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

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

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

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

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

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

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

def disp(p):
    i2c = SoftI2C(scl=Pin(14), sda=Pin(12), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text("LoRa receiver", 0, 16)
    oled.text("Packet Nr:", 0, 32)
    oled.text("{}".format(p), 0, 48)
    oled.show()

def receive(lora):
    print("LoRa Receiver")
    broker = "broker.emqx.io"
    client = MQTTClient("ESP32S1", 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"ESP32S1/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) and MQTT

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
```

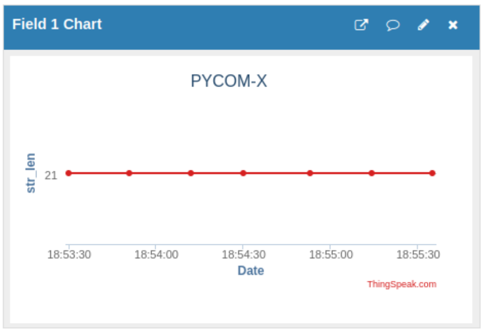
The message is the **payload** with the **fields** associated with each **value**:

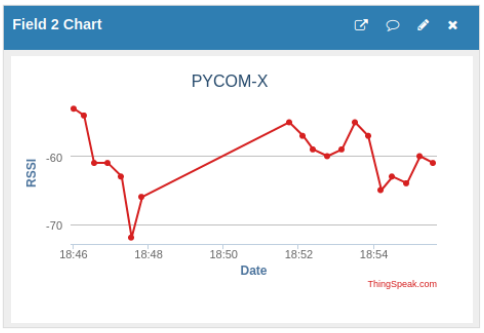
```
payload = "field1="+str(temp)+"&field2="+str(hum)+"&field3="+str(rssi)
```

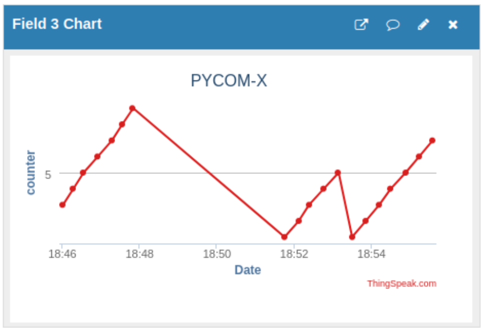
Here is the full code:

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

CHANNEL_ID = "1538804"
WRITE_API_KEY = "YOX31M0EDKO0JATK"

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)

5.3 LoRa-WiFi gateway (ThingSpeak) and `thingspeak.py` library

The **LoRa-WiFi gateway** (ThingSpeak) will resend the data received on a LoRa link over a WiFi connection to a ThingSpeak server.
The following program allows you to receive LoRa packets and relay them over a WiFi connection to the ThingSpeak server.
Note that we are using `thingspeak.py` library to send data to the server.

Here is the full code:

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

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

def receive(lora):
    channel_living_room = "1538804"
    field_temperature = "Temperature"
    field_humidity = "Humidity"
    thing_speak = ThingSpeakAPI([
    Channel(channel_living_room , 'YOX31M0EDKO0JATK', [field_temperature, field_humidity])],
    protocol_class=ProtoHTTP, log=True)
    print("LoRa Receiver")
    wifista.connect()
    active_channel = channel_living_room
    temperature = 2.0
    humidity=3.0
    rssi =0
    while True:
        if lora.receivedPacket():
            try:
                payload = lora.readPayload().decode()
                rssi = lora.packetRssi()
                print("RX: {} | RSSI: {}".format(payload, rssi))
               # here - decompose payload into temperature and humidity fields !!
                wifista.connect()
               thing_speak.send(active_channel, {
                            field_temperature: temperature,
                            field_humidity: humidity
                            }) ts_payload)
                disp(payload)
                time.sleep(thing_speak.free_api_delay)
            except Exception as e:
                print(e)
```

To do:

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

Table of Contents