

# IoT Labs with ESP32 (S1) and µPython

## SmartComputerLab

### Table of Contents

<b>0. Introduction.....</b>	<b>2</b>
0.1 ESP32 (S1) Soc – an advanced unit for IoT architectures.....	3
0.2 ESP32 LOLIN32 (D32) board.....	3
0.3 IoT DevKits with ESP32S1 (D32) main board.....	4
0.4 Software – Thonny IDE.....	5
0.4.1 Installing Thonny IDE - thonny.org.....	5
0.4.2 Preparing the ESP32 LOLIN32 board.....	6
0.4.4 First example – x.led.blink.py.....	8
<b>Lab 1: Sensor reading and data display (i2c).....</b>	<b>9</b>
1.0 Introduction.....	9
1.1 First example – data display on OLED screen.....	10
1.2 Second example – sensor reading (T/H): SHT21.....	12
1.2.1 Preparing the code.....	12
1.3 Third example – reading a luminosity sensor (L) - BH1750.....	12
1.4 Fourth example – reading a PIR sensor – SR602.....	13
1.5 Fifth example – using RGB ring (neopixel).....	14
1.6 Sixth example – GPS module: NEO-6M (UART).....	15
<b>Lab 2: WiFi communication and WEB servers.....</b>	<b>16</b>
2.1 Network scan.....	16
2.2 Connection to the WiFi network, station mode – STA.....	17
2.3 Reading a WEB page.....	18
2.4 Getting time from NTP server.....	19
To do:.....	19
2.5 Simple WEB server – reading a variable.....	21
2.6 Simple WEB server – sending an order.....	22
2.4.3 Mini WEB server with Access Point – RGB LED management.....	23
<b>Lab 3: MQTT Broker and ThingSpeak Server.....</b>	<b>25</b>
3.1 MQTT Protocol and MQTT Client.....	25
3.1.1 MQTT client – the code.....	25
3.1.2 Broker MQTT on a PC.....	26
3.2 ThingSpeak server.....	27
3.2.1 Preparation for sending data with thingspeak.py library.....	27
3.2.2 Preparation for sending data as MQTT messages.....	28
3.2.2 Preparation for sending data as simple HTTP requests.....	29
<b>Lab 4: LoRa technology for Long Range communication.....</b>	<b>31</b>
4.0 Introduction.....	31
4.1 LoRa Modulation.....	31
4.2 sx127x.py driver library.....	31
4.3 Main program.....	33
4.4 LoRa functional modules.....	34
4.4.1 Transmitter – sender() (LoRaSender.py).....	34
4.4.2 Receiver – receive (LoRaReceiver.py).....	35
4.4.3 Receiver – onReceive (LoRaReceiverCallback.py).....	35
<b>Lab 5: Development of simple IoT gateways.....</b>	<b>37</b>
5.1 LoRa-WiFi Gateway (MQTT).....	37
5.2 LoRa-WiFi gateway (ThingSpeak) and MQTT.....	40
5.3 LoRa-WiFi gateway (ThingSpeak) and thingspeak.py library.....	42

<b>Appendix: More on sensors.....</b>	<b>43</b>
A.1 MPU6050 – accelerometer, position and temperature sensor.....	43
A.1.1 File mpu6050.py.....	43
A.1.2 Test mpu6050.py.....	43
A.2 DHT driver and test.....	44
A.3 BMP180 driver and test.....	44
A.3.1 Module for the BMP180 pressure sensor : bmp180.py.....	44
A.4 ADC converter – ADS1115.....	47
A.4.1 File ADS1115.py.....	47
A.5 Micro Laser – VL53L0X.....	50
A.5.1 File VL53L0X.py.....	

# 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 (System on Chip)** 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 **Client Server publish/subscribe messaging transport protocol**. ThingSpeak type servers contain a **database** and offer a **graphical interface** for viewing recorded data.

**ThingSpeak.com** is accessible free of charge, but its frequency of reception of messages and the number of messages are limited.

**The fourth laboratory** deals with **long-range radio links** based on 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.

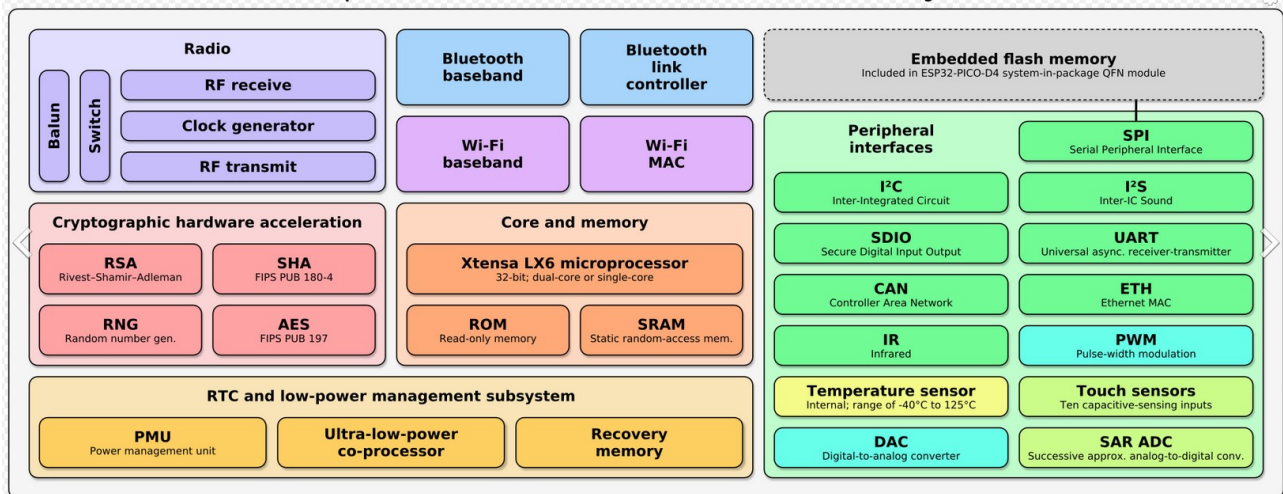


Figure 0.1 ESP32S1 SoC – internal architecture

## 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)

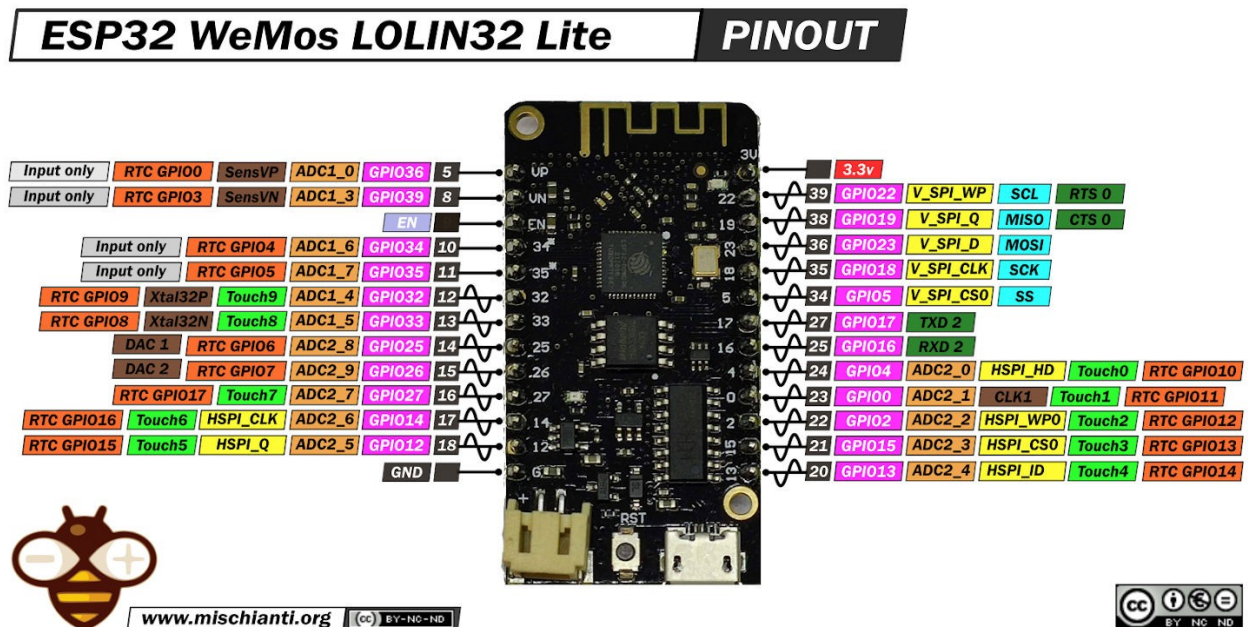
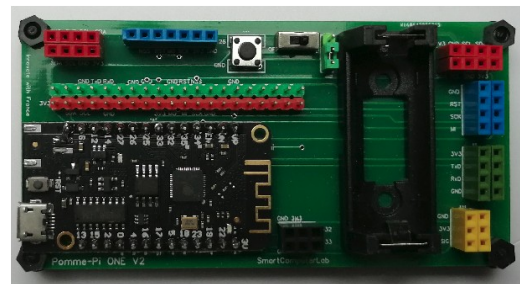
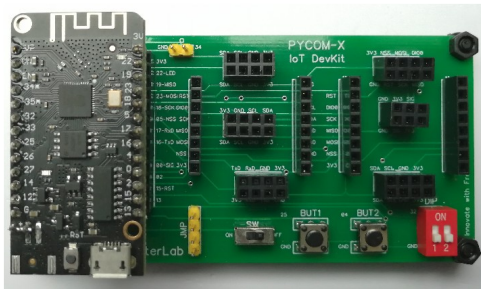


Figure 0.2 ESP32 LOLIN32 Lite MCU board and its pinout

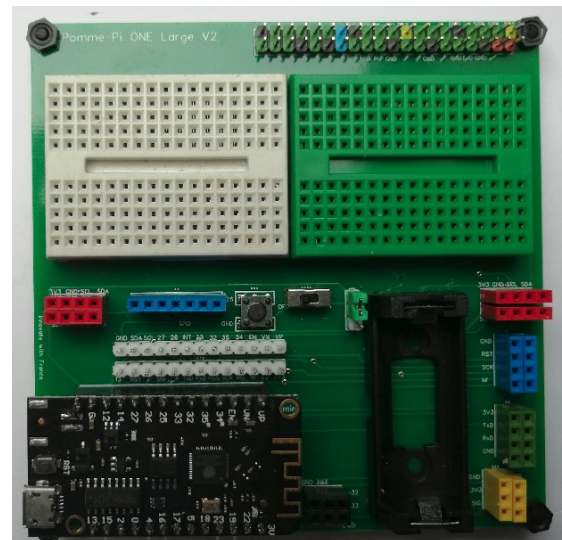
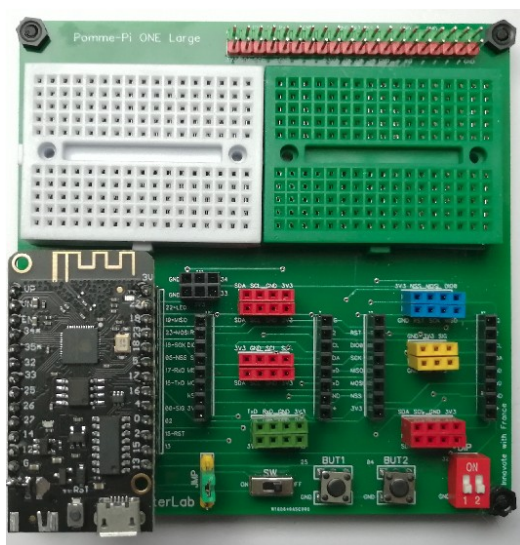
As we can see in the figure above, the board exposes 2x13 pins. These pins carry the **I2C** (**SDA-12,SCL-14**), **UART** (**RX-16,TX-17**), **SPI** (**SCK-18,MISO-19,MOSI-23**) busses, plus control signals (**NSS- 5,RST-15,INT-26,...**). The **LED** is connected to **pin 22**. **Simple signal (SIG)** is connected to **pin 0**.

### 0.3 IoT DevKits with ESP32S1 (D32) main board

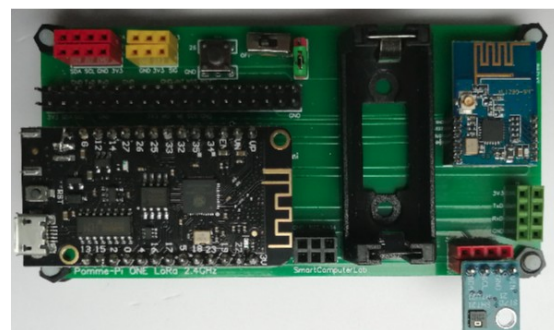
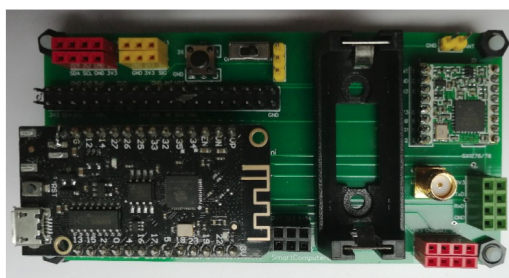
Efficient integration of the selected ESP32 LOLIN32 board into IoT architectures requires the use of a development platform such as **IoT DevKit** provided by **SmartComputerLab**. The IoT DevKits are composed of a base board and a large number of extension boards designed for the efficient use of connection 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**

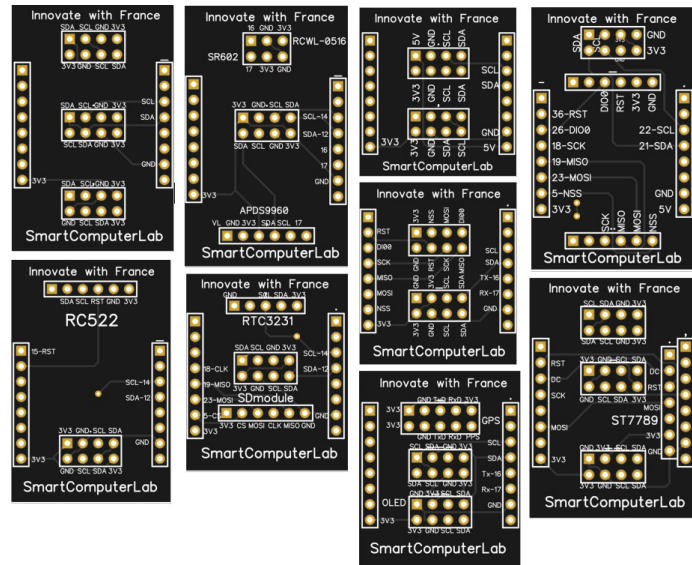


**Figure 0.5 Two Pomme-Pi ONE LoRa DevKits (ESP32S1) with integrated interfaces for LoRa modems (SX1278/6 – 343/686 MHz or SX1280- 2.4 GHz)**

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](https://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.

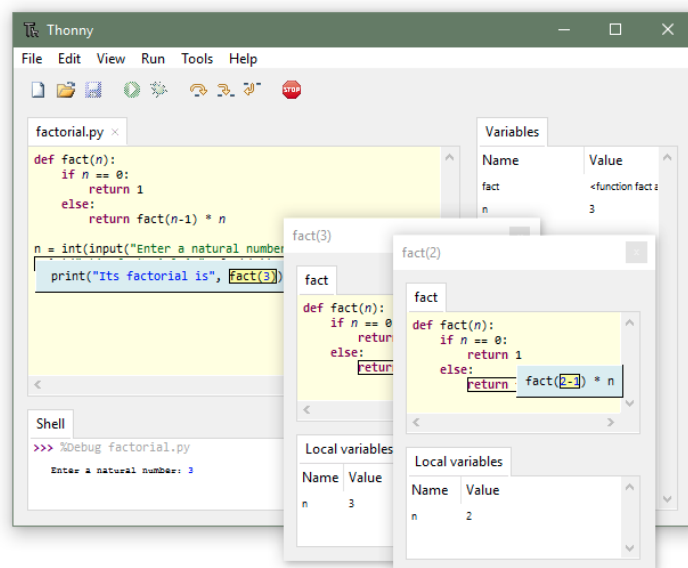
With the help of Thonny, it becomes very easy to code in MicroPython as it has an inbuilt debugger which helps to find any error in the program by debugging the script line by line.

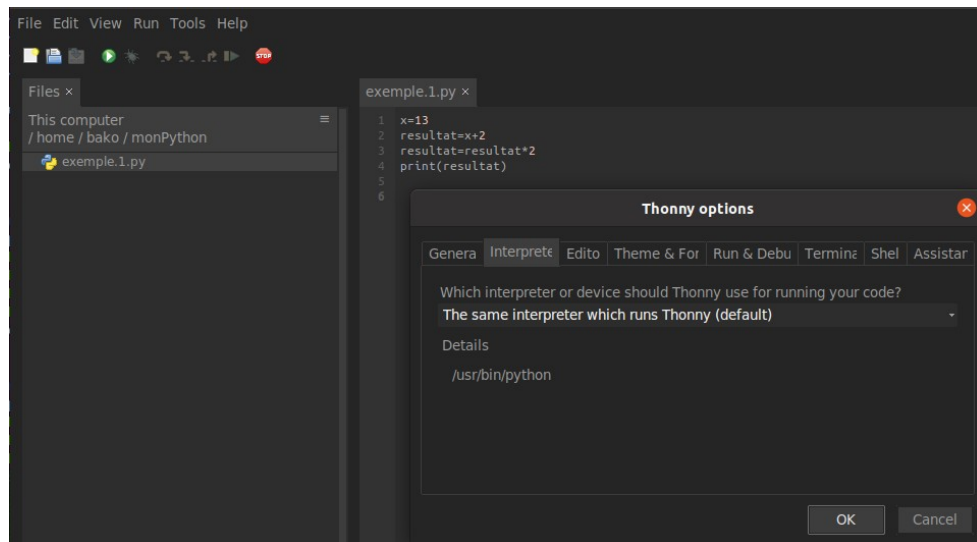
Here is the installation page of the Thonny IDE. You follow the instructions.



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

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





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

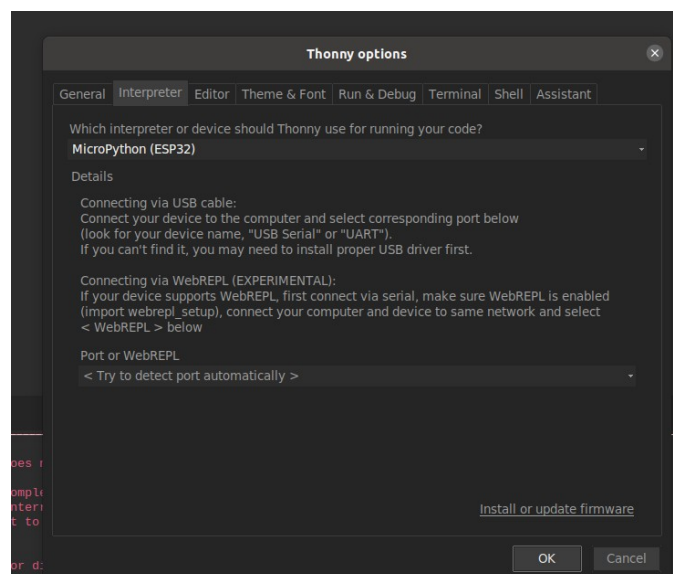
[github.com/smartcomputerlab/Smart-IoT-Labs](https://github.com/smartcomputerlab/Smart-IoT-Labs)

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**.

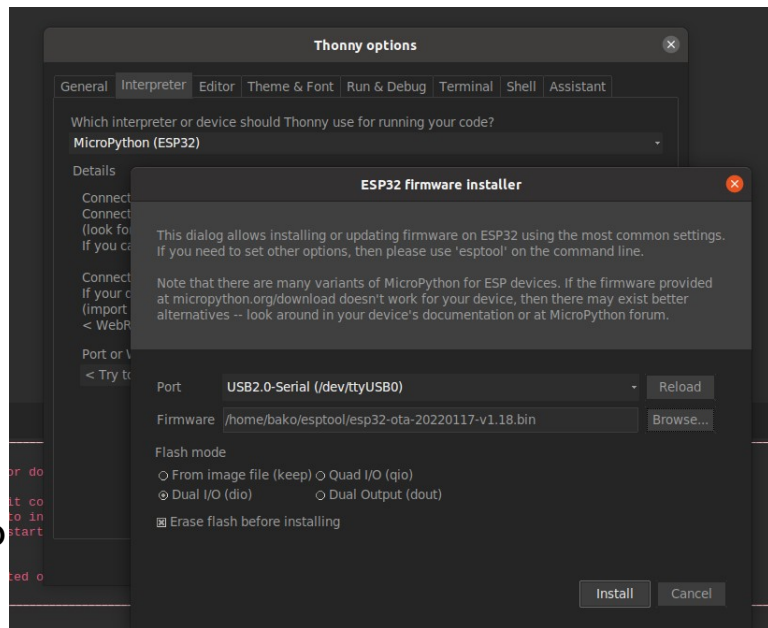


In the interpreter installation phase, you must **connect** your card to the PC and choose the **USB interface**. Then you have to **download the binary code** of the interpreter on the page:

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

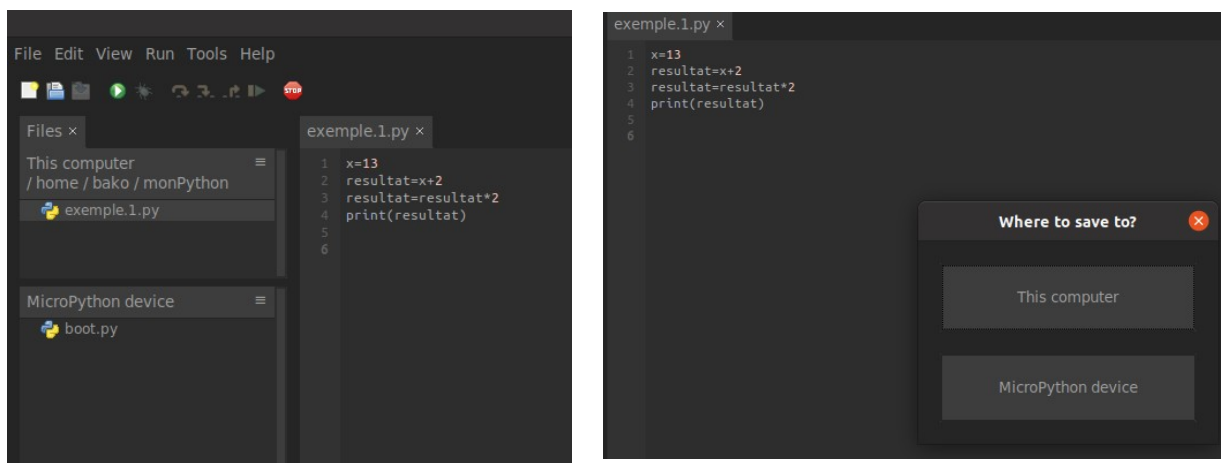
The installation must be preceded by **Erase flash** before installing.

**Attention:** Flash mode must be **Dual I/O (dio)**.  
Then we click on **Install**.



After loading the MicroPython interpreter on the ESP32 board we can connect our board with the USB cable to our PC and launch Thonny IDE again.

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

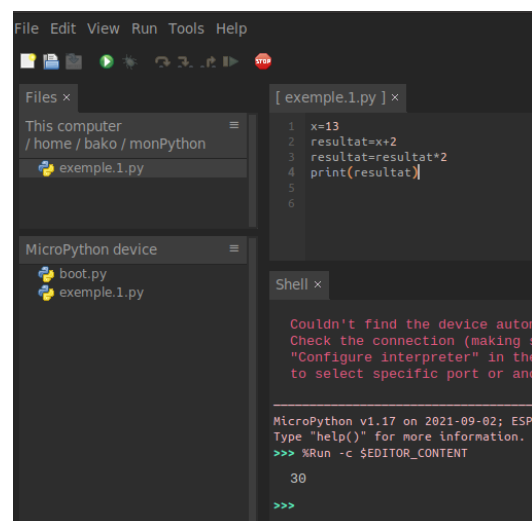


Let's see the available files, **View->Files**.

Our newly "flashed" map only contains the **boot.py** file. We are going to add our **exemple.1.py** program to it. It is possible to save the program on the PC (**This computer**) or on the card (**MicroPython device**).

Let's do both.

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





#### 0.4.4 First example – `x.led.blink.py`

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

The following figure shows the working windows in the Thonny IDE. On the left at the top we have the contents of the `/home/bako/monPython` directory on our PC. At the bottom left we have the list of programs recorded on the card.

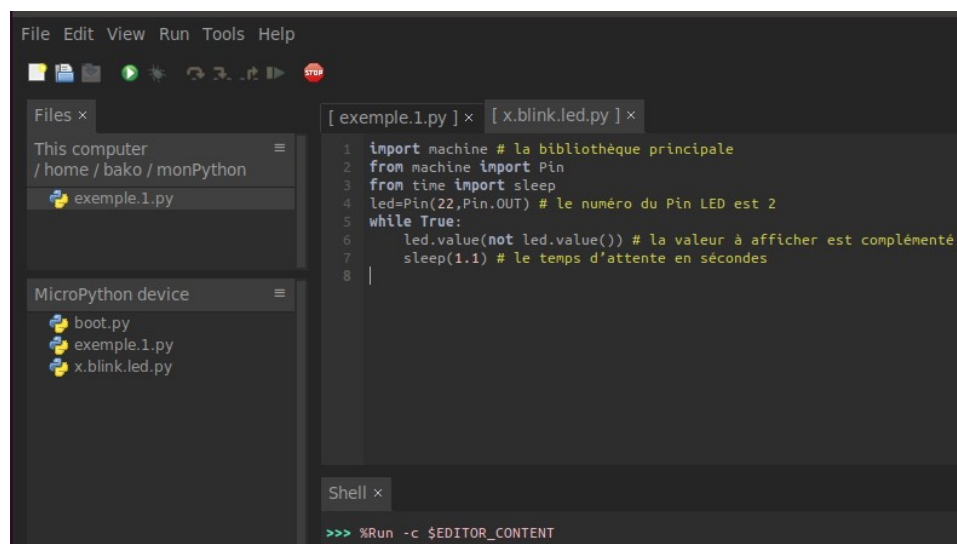
When the board boots first time there is only `boot.py`; other programs are loaded later.

In the main window we display the content of the last edited program, here `x.led.blink.py`.

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



**To do:**

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

# Lab 1

## Sensor reading and data display (i2c)

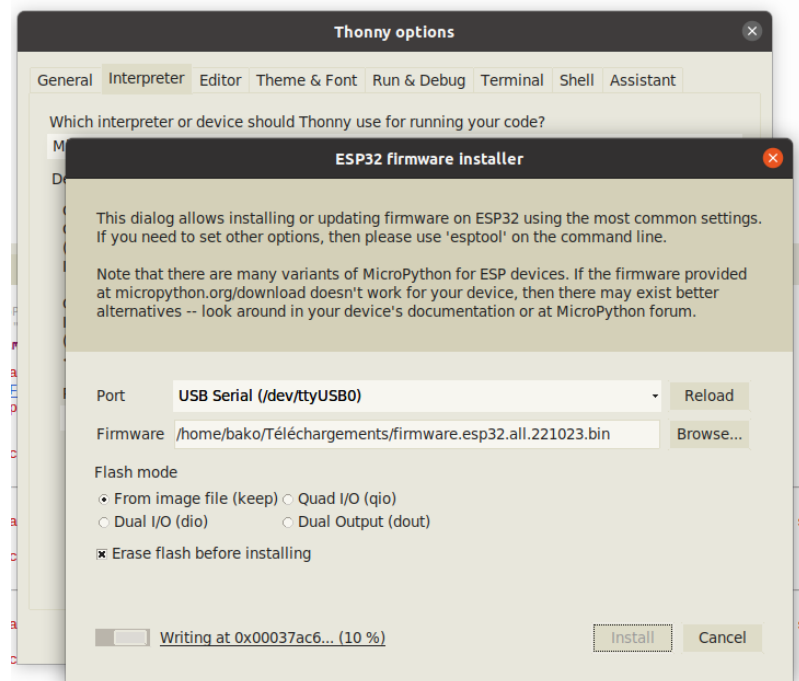
### 1.0 Introduction

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

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

**I2C bus consists of 2 lines** (signals or wires); **SCL-14** which carries the **CLock** signal and **SDA-12** which carries the information (Data, Address).

**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 **a11**)



**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
_owewire        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         mfr522         uasyncio/funcs  usocket
bmp180         micropyGPS     uasyncio/lock   ssl
bmp280         micropython    uasyncio/stream struct
btree         mip            ubinascii       sys
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
```

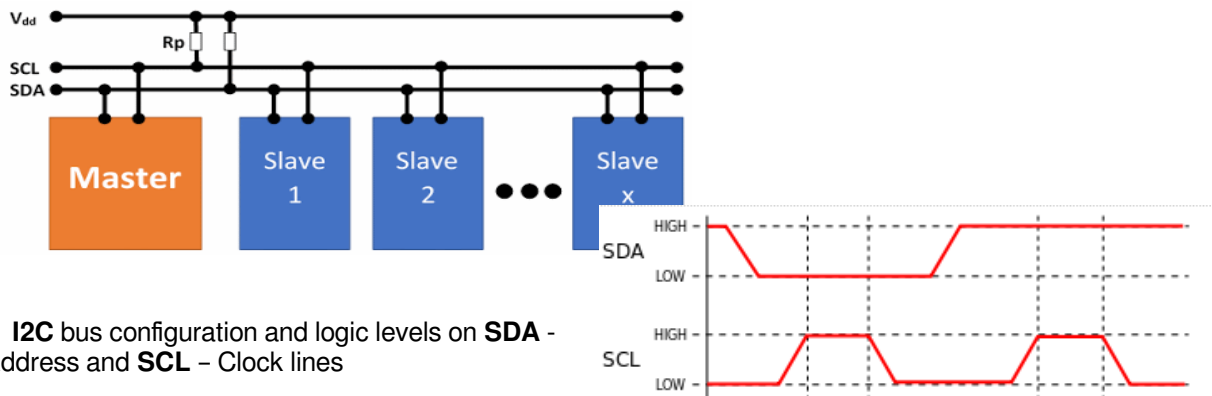
## 1.1 First example – data display on OLED screen

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 (**Data, Address**).



**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

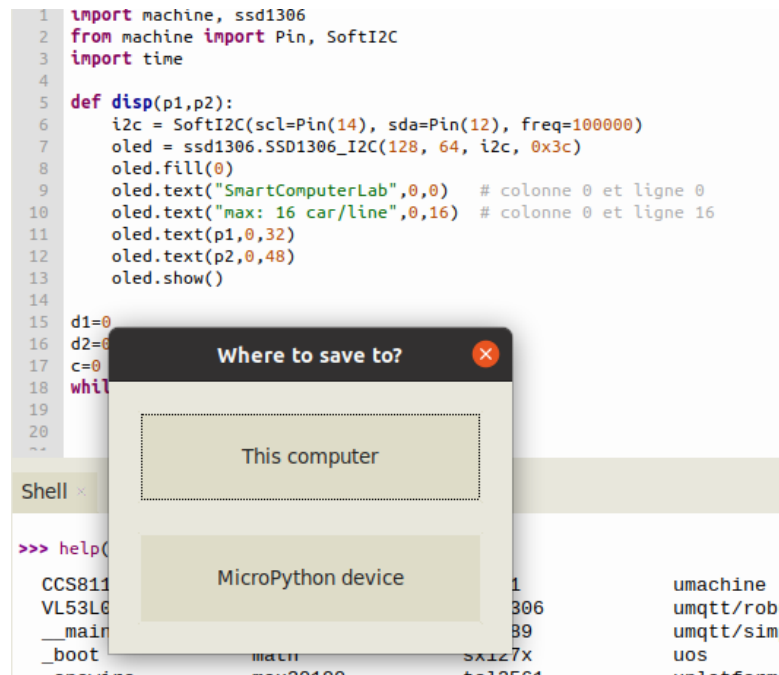
def disp(p1,p2):
    i2c = SoftI2C(scl=Pin(14), sda=Pin(12), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab",0,0) # colonne 0 et ligne 0
    oled.text("max: 16 car/line",0,16) # colonne 0 et ligne 16
    oled.text(p1,0,32)
    oled.text(p2,0,48)
    oled.show()
```

```
d1=0
d2=0
```

```

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

```



**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:

```

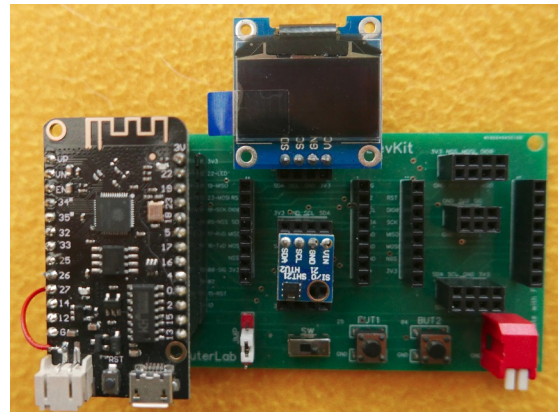
while c<10:

```

**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

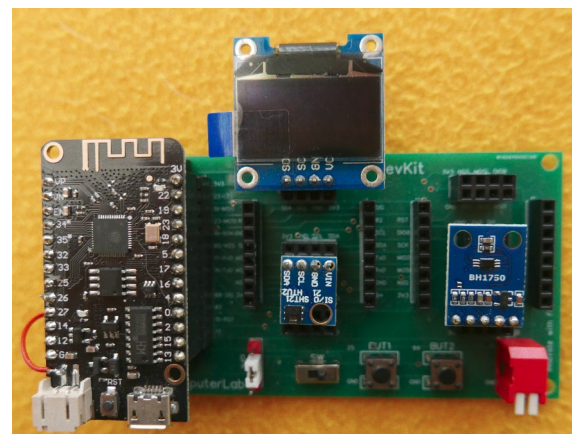
### 1.2.1 Preparing the code

```
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)

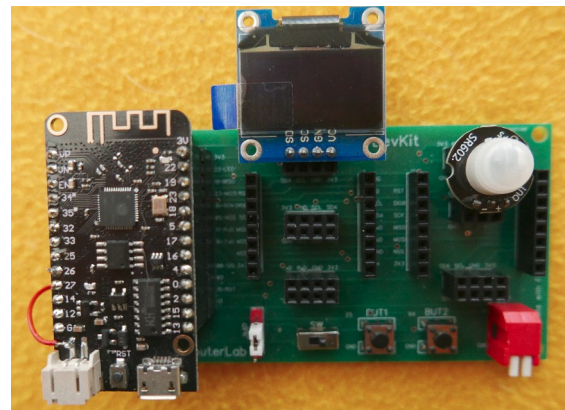
%Run -c $EDITOR_CONTENT
lum
496
```

#### 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)**.

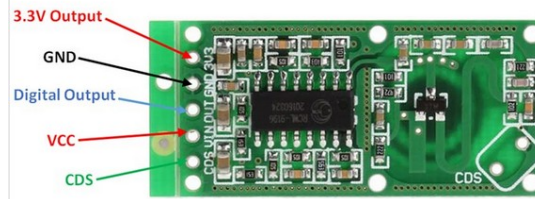
```
from machine import Pin
import time
ldr = Pin(0, Pin.IN) # create input pin on GPIO2
while True:
    if ldr.value():
        print('OBJECT DETECTED')
    else:
        print('ALL CLEAR')
    time.sleep(1)

>>> %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:

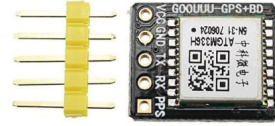
1. Study and test the program.
2. Modify the code to change the intensity of colors.
3. Write a simple **counter** (modulo) 12 and activate in turn the corresponding leds.

## 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 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:{}:03d'{}:02.4f".format(gps.latitude[2],gps.latitude[0],gps.latitude[1]),0,y)
        y += dy
        dsp.text("Lo:{}:03d'{}: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

* VAO-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.

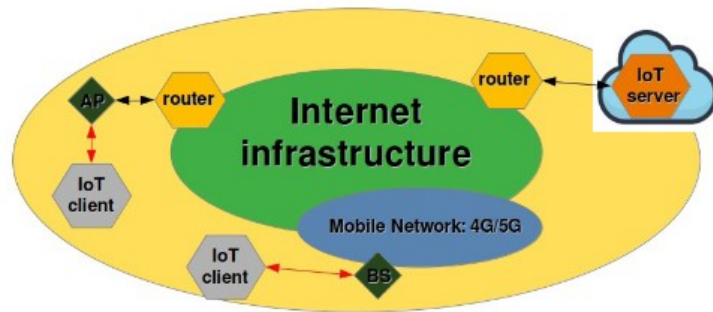
```
from machine import Pin
from time import sleep

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,month,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,month,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**.

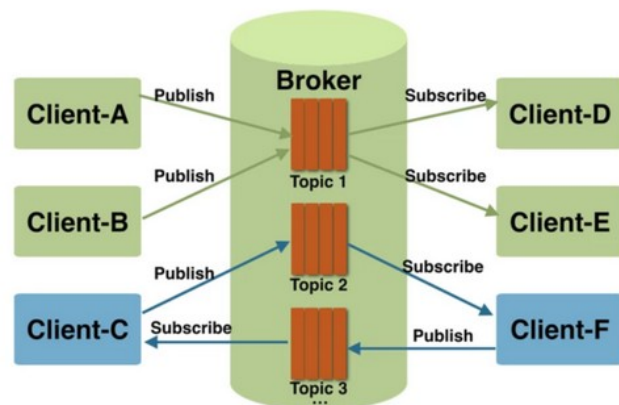
### 3.1 MQTT Protocol and MQTT Client

**MQTT**, that stands for 'Message Queuing Telemetry Transport', is a **publish/subscribe messaging protocol** based on the **TCP/IP** protocol. A client, called publisher, first establishes a 'publish' type connection with the MQTT server, called broker.

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

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

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



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

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

#### 3.1.1 MQTT client – the code

In this example, we will connect our 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 , 'YOX31MOEDKO0JATK', [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)
```

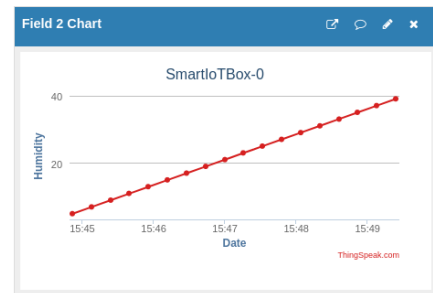
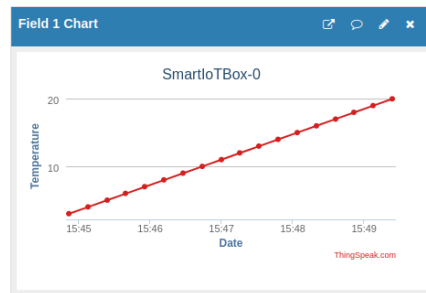


## Channel Stats

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

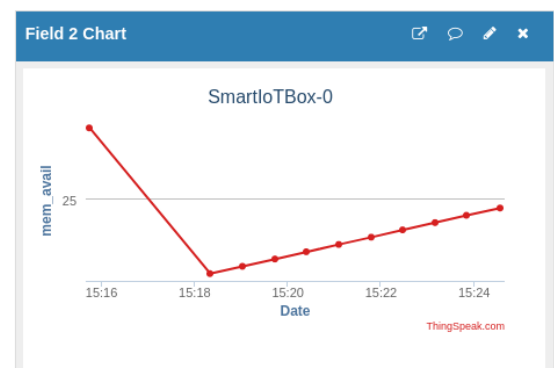
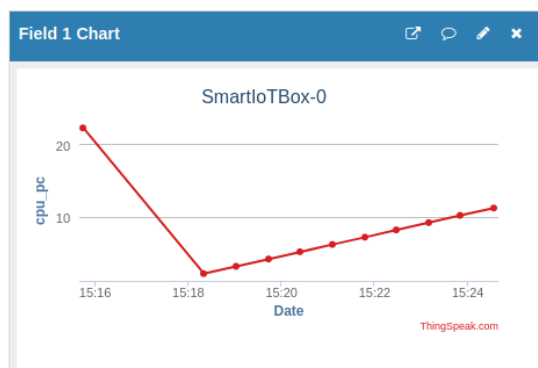
def http_get(url):
    import socket
    _, _, host, path = url.split('/', 3)
    print(path)
    print(host)
    addr = socket.getaddrinfo(host, 80)[0][-1]
    s = socket.socket()
    s.connect(addr)
    s.send(bytes('GET /%s HTTP/1.0\r\nHost: %s\r\n\r\n' % (path, host), 'utf8'))
    while True:
        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: b42bfff93-34e7-4416-94fb-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

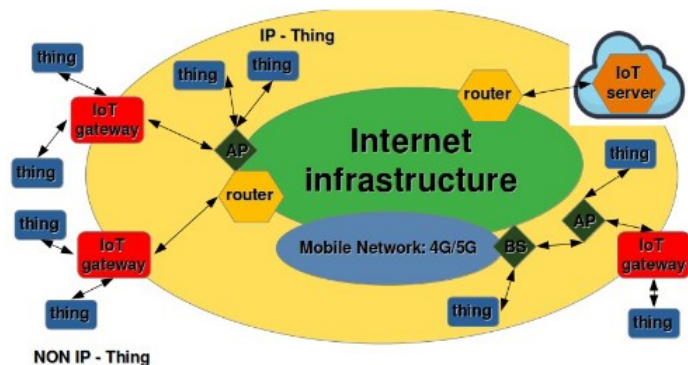
# LoRa technology for Long Range communication

## 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

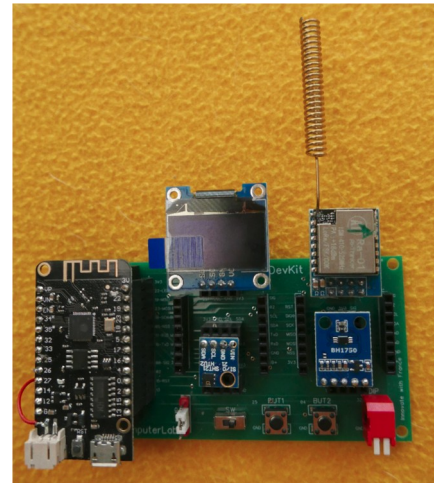
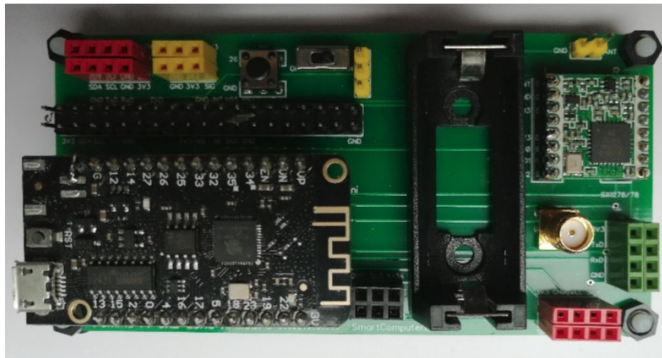
Here are some excerpts from the `sx127x.py` library

```
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** (Received Signal Strength Indicator) value corresponding to the received LoRa messages.

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

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

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

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

#### To do:

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

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

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

Here is the code:

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

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

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



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

### To do:

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

## Lab 5

# Development of simple IoT gateways

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

### 5.1 LoRa-WiFi Gateway (MQTT)

Our first example illustrates the construction of a **LoRa-WiFi 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 = "Y0X31M0EDK00JATK"

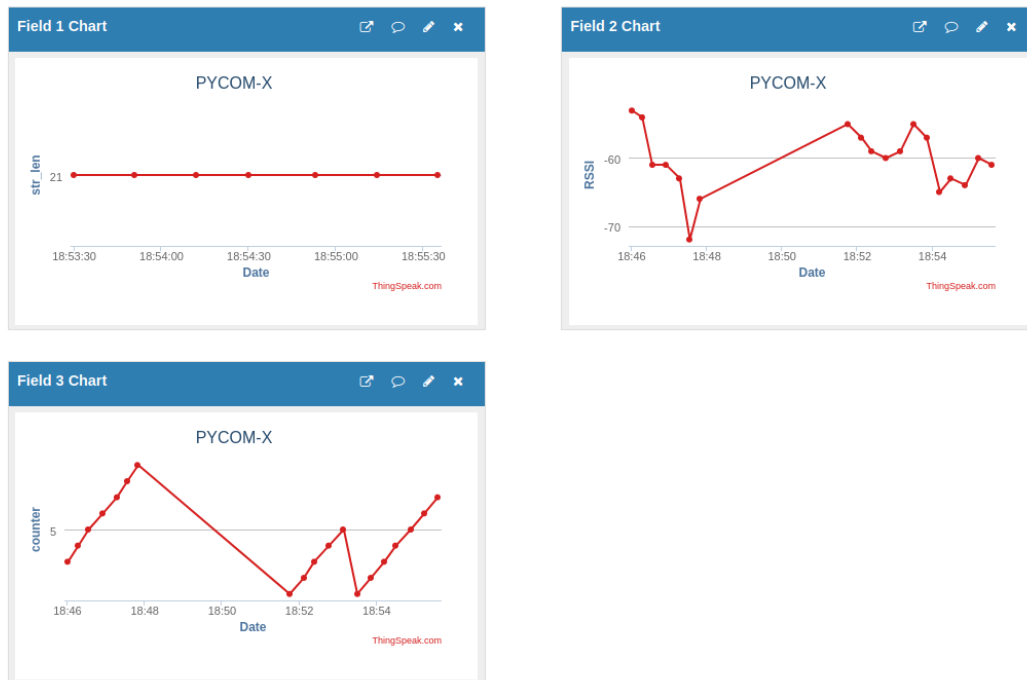
def disp(p):
    i2c = SoftI2C(scl=Pin(14), sda=Pin(12), freq=100000)
    oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text("LoRa receiver", 0, 16)
    oled.text("Packet Nr:", 0, 32)
    oled.text("{}".format(p), 0, 48)
    oled.show()

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, 'YOX31MOEDKO0JATK', [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, 'YOX31MOEDKO0JATK', [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)



## Appendix:

### More on sensors

In this appendix we present some additional sensors that are not covered in Lab 1.

#### A.1 MPU6050 – accelerometer, position and temperature sensor

##### A.1.1 File mpu6050.py

```
import machine

class accel():
    def __init__(self, i2c, addr=0x68):
        self.iic = i2c
        self.addr = addr
        self.iic.start()
        self.iic.writeto(self.addr, bytearray([107, 0]))
        self.iic.stop()

    def get_raw_values(self):
        self.iic.start()
        a = self.iic.readfrom_mem(self.addr, 0x3B, 14)
        self.iic.stop()
        return a

    def get_ints(self):
        b = self.get_raw_values()
        c = []
        for i in b:
            c.append(i)
        return c

    def bytes_toint(self, firstbyte, secondbyte):
        if not firstbyte & 0x80:
            return firstbyte << 8 | secondbyte
        return - (((firstbyte ^ 255) << 8) | (secondbyte ^ 255) + 1)

    def get_values(self):
        raw_ints = self.get_raw_values()
        vals = {}
        vals["AcX"] = self.bytes_toint(raw_ints[0], raw_ints[1])
        vals["AcY"] = self.bytes_toint(raw_ints[2], raw_ints[3])
        vals["AcZ"] = self.bytes_toint(raw_ints[4], raw_ints[5])
        vals["Tmp"] = self.bytes_toint(raw_ints[6], raw_ints[7]) / 340.00 + 36.53
        vals["GyX"] = self.bytes_toint(raw_ints[8], raw_ints[9])
        vals["GyY"] = self.bytes_toint(raw_ints[10], raw_ints[11])
        vals["GyZ"] = self.bytes_toint(raw_ints[12], raw_ints[13])
        return vals # returned in range of Int16
        # -32768 to 32767

    def val_test(self): # ONLY FOR TESTING! Also, fast reading sometimes crashes IIC
        from time import sleep
        while 1:
            print(self.get_values())
            sleep(0.05)
```

##### A.1.2 Test mpu6050.py

```
from machine import I2C
from machine import Pin
from machine import sleep
import mpu6050
i2c = I2C(scl=Pin(0), sda=Pin(10)) #initializing the I2C method for PYCOM-V

mpu= mpu6050.accel(i2c)
c=0
while c<20:
    mpu.get_values()
    print(mpu.get_values())
    sleep(500)
    c=c+1
```

## A.2 DHT driver and test

This example shows the use of DHT22 sensor operating over **single line**.

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

p0 = Pin(0, Pin.IN)
#p0.init(p0.IN,p0.PULL_DOWN)
d = dht.DHT22(p0) # or dht.DHT11(p0)

while True:
    d.measure()
    t=d.temperature() # eg. 23 (°C)
    h=d.humidity()    # eg. 41 (% RH)
    print(t)
    print(h)
    time.sleep(3)
```

## A.3 BMP180 driver and test

### A.3.1 Module for the BMP180 pressure sensor : bmp180.py

```
from ustruct import unpack as unp
from machine import I2C, Pin
import math
import time

# BMP180 class
class BMP180():
    _bmp_addr = 119                # adress of BMP180 is hardcoded on the sensor

    # init
    def __init__(self, i2c_bus):
        # create i2c object
        _bmp_addr = self._bmp_addr
        self._bmp_i2c = i2c_bus
        self._bmp_i2c.start()
        self.chip_id = self._bmp_i2c.readfrom_mem(_bmp_addr, 0xD0, 2)
        # read calibration data from EEPROM
        self._AC1 = unp('>h', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xAA, 2))[0]
        self._AC2 = unp('>h', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xAC, 2))[0]
        self._AC3 = unp('>h', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xAE, 2))[0]
        self._AC4 = unp('>H', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xB0, 2))[0]
        self._AC5 = unp('>H', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xB2, 2))[0]
        self._AC6 = unp('>H', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xB4, 2))[0]
        self._B1 = unp('>h', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xB6, 2))[0]
        self._B2 = unp('>h', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xB8, 2))[0]
        self._MB = unp('>h', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xBA, 2))[0]
        self._MC = unp('>h', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xBC, 2))[0]
        self._MD = unp('>h', self._bmp_i2c.readfrom_mem(_bmp_addr, 0xBE, 2))[0]
        # settings to be adjusted by user
        self.oversample_setting = 3
        self.baseline = 101325.0
        # output raw
        self.UT_raw = None
        self.B5_raw = None
        self.MSB_raw = None
        self.LSB_raw = None
        self.XLSB_raw = None
        self.gauge = self.makegauge() # Generator instance
        for _ in range(128):
            next(self.gauge)
            time.sleep_ms(1)
```

```

def compvaldump(self):
    '''
    Returns a list of all compensation values
    '''
    return [self._AC1, self._AC2, self._AC3, self._AC4, self._AC5, self._AC6,
            self._B1, self._B2, self._MB, self._MC, self._MD, self.oversample_setting]

# gauge raw
def makegauge(self):
    '''
    Generator refreshing the raw measurments.
    '''
    delays = (5, 8, 14, 25)
    while True:
        self._bmp_i2c.writeto_mem(self._bmp_addr, 0xF4, bytearray([0x2E]))
        t_start = time.time()
        while (time.time() - t_start) <= 5: # 5mS delay
            yield None
        try:
            self.UT_raw = self._bmp_i2c.readfrom_mem(self._bmp_addr, 0xF6, 2)
        except:
            yield None
        self._bmp_i2c.writeto_mem(self._bmp_addr, 0xF4, bytearray([0x34+(self.oversample_setting
<< 6)]))

        t_pressure_ready = delays[self.oversample_setting]
        t_start = time.time()
        while (time.time() - t_start) <= t_pressure_ready:
            yield None
        try:
            self.MSB_raw = self._bmp_i2c.readfrom_mem(self._bmp_addr, 0xF6, 1)
            self.LSB_raw = self._bmp_i2c.readfrom_mem(self._bmp_addr, 0xF7, 1)
            self.XLSB_raw = self._bmp_i2c.readfrom_mem(self._bmp_addr, 0xF8, 1)
        except:
            yield None
        yield True

def blocking_read(self):
    if next(self.gauge) is not None: # Discard old data
        pass
    while next(self.gauge) is None:
        pass

@property
def oversample_sett(self):
    return self.oversample_setting

@oversample_sett.setter
def oversample_sett(self, value):
    if value in range(4):
        self.oversample_setting = value
    else:
        print('oversample_sett can only be 0, 1, 2 or 3, using 3 instead')
        self.oversample_setting = 3

@property
def temperature(self):
    '''
    Temperature in degree C.
    '''
    next(self.gauge)
    try:
        UT = unp('>H', self.UT_raw)[0]
    except:
        return 0.0
    X1 = (UT-self._AC6)*self._AC5/2**15
    X2 = self._MC*2**11/(X1+self._MD)
    self.B5_raw = X1+X2
    return ((X1+X2)+8)/2**4/10

@property
def pressure(self):
    '''
    Pressure in mbar.

```

```

'''
next(self.gauge)
self.temperature # Populate self.B5_raw
try:
    MSB = unp('B', self.MSB_raw)[0]
    LSB = unp('B', self.LSB_raw)[0]
    XLSB = unp('B', self.XLSB_raw)[0]
except:
    return 0.0
UP = ((MSB << 16)+(LSB << 8)+XLSB) >> (8-self.oversample_setting)
B6 = self.B5_raw-4000
X1 = (self._B2*(B6**2/2**12))/2**11
X2 = self._AC2*B6/2**11
X3 = X1+X2
B3 = ((int((self._AC1*4+X3)) << self.oversample_setting)+2)/4
X1 = self._AC3*B6/2**13
X2 = (self._B1*(B6**2/2**12))/2**16
X3 = ((X1+X2)+2)/2**2
B4 = abs(self._AC4)*(X3+32768)/2**15
B7 = (abs(UP)-B3) * (50000 >> self.oversample_setting)
if B7 < 0x80000000:
    pressure = (B7*2)/B4
else:
    pressure = (B7/B4)*2
X1 = (pressure/2**8)**2
X1 = (X1*3038)/2**16
X2 = (-7357*pressure)/2**16
return pressure+(X1+X2+3791)/2**4

@property
def altitude(self):
    '''
    Altitude in m.
    '''
    try:
        p = -7990.0*math.log(self.pressure/self.baseline)
    except:
        p = 0.0
    return p

```

and test file: `test.bmp180.py`

```

from bmp180 import BMP180
from machine import I2C, Pin # create an I2C bus object accordingly to the port you are using
bus = I2C(scl=Pin(0), sda=Pin(10), freq=100000) #
bmp180 = BMP180(bus)
bmp180.oversample_sett = 2
bmp180.baseline = 101325

temp = bmp180.temperature
p = bmp180.pressure
altitude = bmp180.altitude
print(temp, p, altitude)

```

## A.4 ADC converter – ADS1115

### A.4.1 File ADS1115.py

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

# CONVERSION DELAY (in mS)
ADS1115_CONVERSIONDELAY = 8

# POINTER REGISTER
REG_MASK = 0x03
REG_CONVERT = 0x00
REG_CONFIG = 0x01
REG_LOWTHRESH = 0x02
REG_HITHRESH = 0x03

# REG_CONFIG REGISTER
OS_REG_MASK = 0x8000
OS_SINGLE = 0x8000 # Write: Set to start a single-conversion
OS_BUSY = 0x0000 # Read: Bit = 0 when conversion is in progress
OS_NOTBUSY = 0x8000 # Read: Bit = 1 when device is not performing a conversion

MUX_REG_MASK = 0x7000
MUX_DIFF_0_1 = 0x0000 # Differential P = AIN0, N = AIN1 (default)
MUX_DIFF_0_3 = 0x1000 # Differential P = AIN0, N = AIN3
MUX_DIFF_1_3 = 0x2000 # Differential P = AIN1, N = AIN3
MUX_DIFF_2_3 = 0x3000 # Differential P = AIN2, N = AIN3
MUX_SINGLE_0 = 0x4000 # Single-ended AIN0
MUX_SINGLE_1 = 0x5000 # Single-ended AIN1
MUX_SINGLE_2 = 0x6000 # Single-ended AIN2
MUX_SINGLE_3 = 0x7000 # Single-ended AIN3

PGA_REG_MASK = 0x0E00
PGA_6_144V = 0x0000 # +/-6.144V range = Gain 2/3
PGA_4_096V = 0x0200 # +/-4.096V range = Gain 1
PGA_2_048V = 0x0400 # +/-2.048V range = Gain 2 (default)
PGA_1_024V = 0x0600 # +/-1.024V range = Gain 4
PGA_0_512V = 0x0800 # +/-0.512V range = Gain 8
PGA_0_256V = 0x0A00 # +/-0.256V range = Gain 16

MODE_REG_MASK = 0x0100
MODE_CONTIN = 0x0000 # Continuous conversion mode
MODE_SINGLE = 0x0100 # Power-down single-shot mode (default)

DR_REG_MASK = 0x00E0
DR_8SPS = 0x0000 # 8 samples per second
DR_16SPS = 0x0020 # 16 samples per second
DR_32SPS = 0x0040 # 32 samples per second
DR_64SPS = 0x0060 # 64 samples per second
DR_128SPS = 0x0080 # 128 samples per second (default)
DR_250SPS = 0x00A0 # 250 samples per second
DR_475SPS = 0x00C0 # 475 samples per second
DR_860SPS = 0x00E0 # 860 samples per second

CMODE_REG_MASK = 0x0010
CMODE_TRAD = 0x0000 # Traditional comparator with hysteresis (default)
CMODE_WINDOW = 0x0010 # Window comparator

CPOL_REG_MASK = 0x0008
CPOL_ACTVLOW = 0x0000 # ALERT/RDY pin is low when active (default)
CPOL_ACTVHI = 0x0008 # ALERT/RDY pin is high when active

CLAT_REG_MASK = 0x0004 # Determines if ALERT/RDY pin latches once asserted
CLAT_NONLAT = 0x0000 # Non-latching comparator (default)
CLAT_LATCH = 0x0004 # Latching comparator

CQUE_REG_MASK = 0x0003
CQUE_1CONV = 0x0000 # Assert ALERT/RDY after one conversions
CQUE_2CONV = 0x0001 # Assert ALERT/RDY after two conversions
CQUE_4CONV = 0x0002 # Assert ALERT/RDY after four conversions
CQUE_NONE = 0x0003 # Disable the comparator and put ALERT/RDY in high state (default)
```

```

GAIN_TWOTHIRDS    = PGA_6_144V
GAIN_ONE          = PGA_4_096V
GAIN_TWO          = PGA_2_048V
GAIN_FOUR         = PGA_1_024V
GAIN_EIGHT        = PGA_0_512V
GAIN_SIXTEEN      = PGA_0_256V

DEVID = 0x48

class ADS1115(object):

    def __init__(self, i2c_devid=DEVID):
        self._addr = i2c_devid
        self.i2c = I2C(scl=Pin(0), sda=Pin(10), freq=400000)
        self.cbuffer = bytearray(2)
        self._conversionDelay = ADS1115_CONVERSIONDELAY
        self._bitShift = 0
        self._gain = GAIN_TWOTHIRDS # +/- 6.144V range (limited to VDD +0.3V max!)

    def write_register(self, reg, value):
        """Writes 16-bits to the specified destination register"""
        # print ('reg {0:4x}'.format(reg))
        # print ('value {0:4x}'.format(value))
        buf = bytearray(2)
        msb = value >> 8
        lsb = value & 0xFF
        buf[0] = msb
        buf[1] = lsb
        self.i2c.writeto_mem(self._addr, reg, buf) # write 2 bytes to self._addr, register

    def read_register(self, reg):
        """Reads 16-bits from the specified destination register"""
        self.cbuffer = self.i2c.readfrom_mem(self._addr, reg, 2) # read 2 bytes from
self._addr, register
        value = (self.cbuffer[0]<<8) +self.cbuffer[1]
        return value

    def set_gain(self, gain):
        """Sets the gain and input voltage range"""
        self._gain = gain

    def get_gain(self):
        """Gets a gain and input voltage range"""
        return self._gain

    def read_adc(self, channel, type = 'single'):
        """Gets a single-ended, or differential ADC reading from the specified channel"""
        if type == 'single':
            if channel > 3:
                return 0
            else:
                if channel >1:
                    return 0

        # Start with default values
        config = CQUE_NONE # Disable the comparator (default val)
        config |= CLAT_NONLAT # Non-latching (default val)
        config |= CPOL_ACTVLOW # Alert/Rdy active low (default val)
        config |= CMODE_TRAD # Traditional comparator (default val)
        config |= DR_128SPS # 1600 samples per second (default)
        config |= MODE_SINGLE # Single-shot mode (default)

        # Set PGA/voltage range
        config |= self._gain

        if type == 'single':
            # Set single-ended input channel
            if channel == 0:
                config |= MUX_SINGLE_0
            elif channel == 1:
                config |= MUX_SINGLE_1

```

```

        elif channel == 2:
            config |= MUX_SINGLE_2
        elif channel == 3:
            config |= MUX_SINGLE_3
    else:
        # Differential
        if channel == 0:
            config |= MUX_DIFF_0_1          # AIN0 = P, AIN1 = N
        elif channel == 1:
            config |= MUX_DIFF_2_3          # AIN2 = P, AIN3 = N

    # Set 'start single-conversion' bit
    config |= OS_SINGLE

    # Write config register to the ADC
    self.write_register(REG_CONFIG, config)

    # Wait for the conversion to complete
    sleep_ms(self._conversionDelay)

    # Read the conversion results
    return self.read_register(REG_CONVERT)

def adc_continuous(self, channel, type = 'single'):
    """Starts a continuous single-ended, or differential ADC reading from the specified
    channel"""
    if type == 'single':
        if channel > 3:
            return 0
    else:
        if channel > 1:
            return 0

    # Start with default values
    config = CQUE_NONE          # Disable the comparator (default val)
    config |= CLAT_NONLAT       # Non-latching (default val)
    config |= CPOL_ACTVLOW      # Alert/Rdy active low (default val)
    config |= CMODE_TRAD        # Traditional comparator (default val)
    config |= DR_128SPS         # 1600 samples per second (default)
    config |= MODE_CONTIN       # Continuous mode

    # Set PGA/voltage range
    config |= self._gain

    if type == 'single':
        # Set single-ended input channel
        if channel == 0:
            config |= MUX_SINGLE_0
        elif channel == 1:
            config |= MUX_SINGLE_1
        elif channel == 2:
            config |= MUX_SINGLE_2
        elif channel == 3:
            config |= MUX_SINGLE_3
    else:
        # Differential
        if channel == 0:
            config |= MUX_DIFF_0_1          # AIN0 = P, AIN1 = N
        elif channel == 1:
            config |= MUX_DIFF_2_3          # AIN2 = P, AIN3 = N

    # Write config register to the ADC
    self.write_register(REG_CONFIG, config)

def start_comparator_single_ended(self, channel, threshold):
    """Sets up the comparator to operate in basic mode, causing the
    ALERT/RDY pin to assert (go from high to low) when the ADC
    value exceeds the specified threshold.
    This will also set the ADC in continuous conversion mode."""

    # Start with default values
    config = CQUE_1CONV         # Comparator enabled and asserts on 1 match
    config |= CLAT_LATCH        # Latching mode

```

```

        config |= CPOL_ACTVLOW # Alert/Rdy active low (default val)
        config |= CMODE_TRAD # Traditional comparator (default val)
        config |= DR_128SPS # 128 samples per second (default)
        config |= MODE_CONTIN # Continuous conversion mode

    # Set PGA/voltage range
    config |= self._gain

    # Set single-ended input channel
    if channel == 0:
        config |= MUX_SINGLE_0
    elif channel == 1:
        config |= MUX_SINGLE_1
    elif channel == 2:
        config |= MUX_SINGLE_2
    elif channel == 3:
        config |= MUX_SINGLE_3

    # Set the high threshold register
    # Shift 12-bit results left 4 bits for the ADS1015
    self.write_register(REG_HITHRESH, threshold)

    # Write config register to the ADC
    write_register(REG_CONFIG, config)

def get_last_conversion_results(self):
    """In order to clear the comparator, we need to read the
    conversion results. This function reads the last conversion
    results without changing the config value."""

    # Wait for the conversion to complete
    # sleep_ms(self._conversionDelay)

    # Read the conversion results
    return self.read_register(REG_CONVERT)

ads=ADS1115()
val=ads.read_adc(2)

```

## A.5 Micro Laser – VL53L0X

### A.5.1 File VL53L0X.py

```

from machine import I2C
from machine import Pin
from micropython import const
import ustruct
import utime
from machine import Timer
import time

_IO_TIMEOUT = 1000
_SYSRANGE_START = const(0x00)
_EXTSUP_HV = const(0x89)
_MSRC_CONFIG = const(0x60)
_FINAL_RATE_RTN_LIMIT = const(0x44)
_SYSTEM_SEQUENCE = const(0x01)
_SPAD_REF_START = const(0x4f)
_SPAD_ENABLES = const(0xb0)
_REF_EN_START_SELECT = const(0xb6)
_SPAD_NUM_REQUESTED = const(0x4e)
_INTERRUPT_GPIO = const(0x0a)
_INTERRUPT_CLEAR = const(0x0b)
_GPIO_MUX_ACTIVE_HIGH = const(0x84)
_RESULT_INTERRUPT_STATUS = const(0x13)
_RESULT_RANGE_STATUS = const(0x14)
_OSC_CALIBRATE = const(0xf8)
_MEASURE_PERIOD = const(0x04)

SYSRANGE_START = 0x00

```



```

SYSTEM_THRESH_HIGH = 0x0C
SYSTEM_THRESH_LOW = 0x0E

SYSTEM_SEQUENCE_CONFIG = 0x01
SYSTEM_RANGE_CONFIG = 0x09
SYSTEM_INTERMEASUREMENT_PERIOD = 0x04

SYSTEM_INTERRUPT_CONFIG_GPIO = 0x0A

GPIO_HV_MUX_ACTIVE_HIGH = 0x84

SYSTEM_INTERRUPT_CLEAR = 0x0B

RESULT_INTERRUPT_STATUS = 0x13
RESULT_RANGE_STATUS = 0x14

RESULT_CORE_AMBIENT_WINDOW_EVENTS_RTN = 0xBC
RESULT_CORE_RANGING_TOTAL_EVENTS_RTN = 0xC0
RESULT_CORE_AMBIENT_WINDOW_EVENTS_REF = 0xD0
RESULT_CORE_RANGING_TOTAL_EVENTS_REF = 0xD4
RESULT_PEAK_SIGNAL_RATE_REF = 0xB6

ALGO_PART_TO_PART_RANGE_OFFSET_MM = 0x28

I2C_SLAVE_DEVICE_ADDRESS = 0x8A

MSRC_CONFIG_CONTROL = 0x60

PRE_RANGE_CONFIG_MIN_SNR = 0x27
PRE_RANGE_CONFIG_VALID_PHASE_LOW = 0x56
PRE_RANGE_CONFIG_VALID_PHASE_HIGH = 0x57
PRE_RANGE_MIN_COUNT_RATE_RTN_LIMIT = 0x64

FINAL_RANGE_CONFIG_MIN_SNR = 0x67
FINAL_RANGE_CONFIG_VALID_PHASE_LOW = 0x47
FINAL_RANGE_CONFIG_VALID_PHASE_HIGH = 0x48
FINAL_RANGE_CONFIG_MIN_COUNT_RATE_RTN_LIMIT = 0x44

PRE_RANGE_CONFIG_SIGMA_THRESH_HI = 0x61
PRE_RANGE_CONFIG_SIGMA_THRESH_LO = 0x62

PRE_RANGE_CONFIG_VCSEL_PERIOD = 0x50
PRE_RANGE_CONFIG_TIMEOUT_MACROP_HI = 0x51
PRE_RANGE_CONFIG_TIMEOUT_MACROP_LO = 0x52

SYSTEM_HISTOGRAM_BIN = 0x81
HISTOGRAM_CONFIG_INITIAL_PHASE_SELECT = 0x33
HISTOGRAM_CONFIG_READOUT_CTRL = 0x55

FINAL_RANGE_CONFIG_VCSEL_PERIOD = 0x70
FINAL_RANGE_CONFIG_TIMEOUT_MACROP_HI = 0x71
FINAL_RANGE_CONFIG_TIMEOUT_MACROP_LO = 0x72
CROSSTALK_COMPENSATION_PEAK_RATE_MCPS = 0x20

MSRC_CONFIG_TIMEOUT_MACROP = 0x46

SOFT_RESET_GO2_SOFT_RESET_N = 0xBF
IDENTIFICATION_MODEL_ID = 0xC0
IDENTIFICATION_REVISION_ID = 0xC2

OSC_CALIBRATE_VAL = 0xF8

GLOBAL_CONFIG_VCSEL_WIDTH = 0x32
GLOBAL_CONFIG_SPAD_ENABLES_REF_0 = 0xB0
GLOBAL_CONFIG_SPAD_ENABLES_REF_1 = 0xB1
GLOBAL_CONFIG_SPAD_ENABLES_REF_2 = 0xB2
GLOBAL_CONFIG_SPAD_ENABLES_REF_3 = 0xB3
GLOBAL_CONFIG_SPAD_ENABLES_REF_4 = 0xB4
GLOBAL_CONFIG_SPAD_ENABLES_REF_5 = 0xB5

GLOBAL_CONFIG_REF_EN_START_SELECT = 0xB6
DYNAMIC_SPAD_NUM_REQUESTED_REF_SPAD = 0x4E
DYNAMIC_SPAD_REF_EN_START_OFFSET = 0x4F

```

```

POWER_MANAGEMENT_GO1_POWER_FORCE = 0x80

VHV_CONFIG_PAD_SCL_SDA__EXTSUP_HV = 0x89

ALGO_PHASECAL_LIM = 0x30
ALGO_PHASECAL_CONFIG_TIMEOUT = 0x30

class TimeoutError(RuntimeError):
    pass

class VL53L0X:
    def __init__(self, i2c, address=0x29):
        self.i2c = i2c
        self.address = address
        self.init()
        self._started = False
        self.measurement_timing_budget_us = 0
        self.set_measurement_timing_budget(self.measurement_timing_budget_us)
        self.enables = {"tcc": 0,
                        "dss": 0,
                        "msrc": 0,
                        "pre_range": 0,
                        "final_range": 0}
        self.timeouts = {"pre_range_vcsel_period_pclks": 0,
                         "msrc_dss_tcc_mclks": 0,
                         "msrc_dss_tcc_us": 0,
                         "pre_range_mclks": 0,
                         "pre_range_us": 0,
                         "final_range_vcsel_period_pclks": 0,
                         "final_range_mclks": 0,
                         "final_range_us": 0}
        self.vcsel_period_type = ["VcselPeriodPreRange", "VcselPeriodFinalRange"]

    def _registers(self, register, values=None, struct='B'):
        if values is None:
            size = ustruct.calcsize(struct)
            data = self.i2c.readfrom_mem(self.address, register, size)
            values = ustruct.unpack(struct, data)
            return values
        data = ustruct.pack(struct, *values)
        self.i2c.writeto_mem(self.address, register, data)

    def _register(self, register, value=None, struct='B'):
        if value is None:
            return self._registers(register, struct=struct)[0]
        self._registers(register, (value,), struct=struct)

    def _flag(self, register=0x00, bit=0, value=None):
        data = self._register(register)
        mask = 1 << bit
        if value is None:
            return bool(data & mask)
        elif value:
            data |= mask
        else:
            data &= ~mask
        self._register(register, data)

    def _config(self, *config):
        for register, value in config:
            self._register(register, value)

    def init(self, power2v8=True):
        self._flag(_EXTSUP_HV, 0, power2v8)

        # I2C standard mode
        self._config(
            (0x88, 0x00),
            (0x80, 0x01),

```

```

        (0xff, 0x01),
        (0x00, 0x00),
    )
    self._stop_variable = self._register(0x91)
    self._config(
        (0x00, 0x01),
        (0xff, 0x00),
        (0x80, 0x00),
    )

    # disable signal_rate_msrc and signal_rate_pre_range limit checks
    self._flag(_MSRC_CONFIG, 1, True)
    self._flag(_MSRC_CONFIG, 4, True)

    # rate_limit = 0.25
    self._register(_FINAL_RATE_RTN_LIMIT, int(0.1 * (1 << 7)),
        struct='>H')
    self._register(_SYSTEM_SEQUENCE, 0xff)
    spad_count, is_aperture = self._spad_info()
    spad_map = bytearray(self._registers(_SPAD_ENABLES, struct='6B'))
    # set reference spads
    self._config(
        (0xff, 0x01),
        (_SPAD_REF_START, 0x00),
        (_SPAD_NUM_REQUESTED, 0x2c),
        (0xff, 0x00),
        (_REF_EN_START_SELECT, 0xb4),
    )

    spads_enabled = 0
    for i in range(48):
        if i < 12 and is_aperture or spads_enabled >= spad_count:
            spad_map[i // 8] &= ~(1 << (i >> 2))
        elif spad_map[i // 8] & (1 << (i >> 2)):
            spads_enabled += 1

    self._registers(_SPAD_ENABLES, spad_map, struct='6B')
    self._config(
        (0xff, 0x01),
        (0x00, 0x00),
        (0xff, 0x00),
        (0x09, 0x00),
        (0x10, 0x00),
        (0x11, 0x00),
        (0x24, 0x01),
        (0x25, 0xFF),
        (0x75, 0x00),
        (0xFF, 0x01),
        (0x4E, 0x2C),
        (0x48, 0x00),
        (0x30, 0x20),
        (0xFF, 0x00),
        (0x30, 0x09),
        (0x54, 0x00),
        (0x31, 0x04),
        (0x32, 0x03),
        (0x40, 0x83),
        (0x46, 0x25),
        (0x60, 0x00),
        (0x27, 0x00),
        (0x50, 0x06),
        (0x51, 0x00),
        (0x52, 0x96),
        (0x56, 0x08),
        (0x57, 0x30),
        (0x61, 0x00),
        (0x62, 0x00),
        (0x64, 0x00),
        (0x65, 0x00),
        (0x66, 0xA0),
        (0xFF, 0x01),
        (0x22, 0x32),
        (0x47, 0x14),
    )

```

```

        (0x49, 0xFF),
        (0x4A, 0x00),
        (0xFF, 0x00),
        (0x7A, 0x0A),
        (0x7B, 0x00),
        (0x78, 0x21),
        (0xFF, 0x01),
        (0x23, 0x34),
        (0x42, 0x00),
        (0x44, 0xFF),
        (0x45, 0x26),
        (0x46, 0x05),
        (0x40, 0x40),
        (0x0E, 0x06),
        (0x20, 0x1A),
        (0x43, 0x40),
        (0xFF, 0x00),
        (0x34, 0x03),
        (0x35, 0x44),
        (0xFF, 0x01),
        (0x31, 0x04),
        (0x4B, 0x09),
        (0x4C, 0x05),
        (0x4D, 0x04),

        (0xFF, 0x00),
        (0x44, 0x00),
        (0x45, 0x20),
        (0x47, 0x08),
        (0x48, 0x28),
        (0x67, 0x00),
        (0x70, 0x04),
        (0x71, 0x01),
        (0x72, 0xFE),
        (0x76, 0x00),
        (0x77, 0x00),

        (0xFF, 0x01),
        (0x0D, 0x01),

        (0xFF, 0x00),
        (0x80, 0x01),
        (0x01, 0xF8),

        (0xFF, 0x01),
        (0x8E, 0x01),
        (0x00, 0x01),
        (0xFF, 0x00),
        (0x80, 0x00),
    )

    self._register(_INTERRUPT_GPIO, 0x04)
    self._flag(_GPIO_MUX_ACTIVE_HIGH, 4, False)
    self._register(_INTERRUPT_CLEAR, 0x01)

    # XXX Need to implement this.
    # budget = self._timing_budget()
    # self._register(_SYSTEM_SEQUENCE, 0xe8)
    # self._timing_budget(budget)

    self._register(_SYSTEM_SEQUENCE, 0x01)
    self._calibrate(0x40)
    self._register(_SYSTEM_SEQUENCE, 0x02)
    self._calibrate(0x00)

    self._register(_SYSTEM_SEQUENCE, 0xe8)

def _spad_info(self):
    self._config(
        (0x80, 0x01),
        (0xFF, 0x01),
        (0x00, 0x00),

```

```

        (0xff, 0x06),
    )
    self._flag(0x83, 3, True)
    self._config(
        (0xff, 0x07),
        (0x81, 0x01),

        (0x80, 0x01),

        (0x94, 0x6b),
        (0x83, 0x00),
    )
    for timeout in range(_IO_TIMEOUT):
        if self._register(0x83):
            break
        utime.sleep_ms(1)
    else:
        raise TimeoutError()
    self._config(
        (0x83, 0x01),
    )
    value = self._register(0x92)
    self._config(
        (0x81, 0x00),
        (0xff, 0x06),
    )
    self._flag(0x83, 3, False)
    self._config(
        (0xff, 0x01),
        (0x00, 0x01),

        (0xff, 0x00),
        (0x80, 0x00),
    )
    count = value & 0x7f
    is_aperture = bool(value & 0b10000000)
    return count, is_aperture

def _calibrate(self, vhw_init_byte):
    self._register(_SYSRANGE_START, 0x01 | vhw_init_byte)
    for timeout in range(_IO_TIMEOUT):
        if self._register(_RESULT_INTERRUPT_STATUS) & 0x07:
            break
        utime.sleep_ms(1)
    else:
        raise TimeoutError()
    self._register(_INTERRUPT_CLEAR, 0x01)
    self._register(_SYSRANGE_START, 0x00)

def start(self, period=0):
    self._config(
        (0x80, 0x01),
        (0xFF, 0x01),
        (0x00, 0x00),
        (0x91, self._stop_variable),
        (0x00, 0x01),
        (0xFF, 0x00),
        (0x80, 0x00),
    )
    if period:
        oscillator = self._register(_OSC_CALIBRATE, struct='>H')
        if oscillator:
            period *= oscillator
        self._register(_MEASURE_PERIOD, period, struct='>H')
        self._register(_SYSRANGE_START, 0x04)
    else:
        self._register(_SYSRANGE_START, 0x02)
    self._started = True

def stop(self):
    self._register(_SYSRANGE_START, 0x01)
    self._config(
        (0xFF, 0x01),

```

```

        (0x00, 0x00),
        (0x91, self._stop_variable),
        (0x00, 0x01),
        (0xFF, 0x00),
    )
    self._started = False

def read(self):
    if not self._started:
        self._config(
            (0x80, 0x01),
            (0xFF, 0x01),
            (0x00, 0x00),
            (0x91, self._stop_variable),
            (0x00, 0x01),
            (0xFF, 0x00),
            (0x80, 0x00),
            (_SYSRANGE_START, 0x01),
        )
        for timeout in range(_IO_TIMEOUT):
            if not self._register(_SYSRANGE_START) & 0x01:
                break
            utime.sleep_ms(1)
        else:
            raise TimeoutError()
    for timeout in range(_IO_TIMEOUT):
        if self._register(_RESULT_INTERRUPT_STATUS) & 0x07:
            break
        utime.sleep_ms(1)
    else:
        raise TimeoutError()
    value = self._register(_RESULT_RANGE_STATUS + 10, struct='>H')
    self._register(_INTERRUPT_CLEAR, 0x01)
    return value

def set_signal_rate_limit(self, limit_Mcps):
    if limit_Mcps < 0 or limit_Mcps > 511.99:
        return False
    self._register(0x44, limit_Mcps * (1 << 7))
    return True

def decode_Vcsel_period(self, reg_val):
    return (((reg_val) + 1) << 1)

def encode_Vcsel_period(self, period_pclks):
    return (((period_pclks) >> 1) - 1)

def set_Vcsel_pulse_period(self, type, period_pclks):
    vcsel_period_reg = self.encode_Vcsel_period(period_pclks)

    self.get_sequence_step_enables()
    self.get_sequence_step_timeouts()

    if type == self.vcsel_period_type[0]:
        if period_pclks == 12:
            self._register(PRE_RANGE_CONFIG_VALID_PHASE_HIGH, 0x18)
        elif period_pclks == 14:
            self._register(PRE_RANGE_CONFIG_VALID_PHASE_HIGH, 0x30)
        elif period_pclks == 16:
            self._register(PRE_RANGE_CONFIG_VALID_PHASE_HIGH, 0x40)
        elif period_pclks == 18:
            self._register(PRE_RANGE_CONFIG_VALID_PHASE_HIGH, 0x50)
        else:
            return False

        self._register(PRE_RANGE_CONFIG_VALID_PHASE_LOW, 0x08)
        self._register(PRE_RANGE_CONFIG_VCSEL_PERIOD, vcsel_period_reg)

        new_pre_range_timeout_mclks =
self.timeout_microseconds_to_Mclks(self.timeouts["pre_range_us"],
                                     period_pclks)

        self._register(PRE_RANGE_CONFIG_TIMEOUT_MACROP_HI,
self.encode_timeout(new_pre_range_timeout_mclks))

```

```

        new_msrc_timeout_mclks =
self.timeout_microseconds_to_Mclks(self.timeouts["msrc_dss_tcc_us"],
                                     period_pclks)

        self._register(MSRC_CONFIG_TIMEOUT_MACROP, 255 if new_msrc_timeout_mclks > 256 else
(new_msrc_timeout_mclks - 1))
        elif type == self.vcsel_period_type[1]:
            if period_pclks == 8:
                self._register(FINAL_RANGE_CONFIG_VALID_PHASE_HIGH, 0x10)
                self._register(FINAL_RANGE_CONFIG_VALID_PHASE_LOW, 0x08)
                self._register(GLOBAL_CONFIG_VCSEL_WIDTH, 0x02)
                self._register(ALGO_PHASECAL_CONFIG_TIMEOUT, 0x0C)
                self._register(0xFF, 0x01)
                self._register(ALGO_PHASECAL_LIM, 0x30)
                self._register(0xFF, 0x00)
            elif period_pclks == 10:
                self._register(FINAL_RANGE_CONFIG_VALID_PHASE_HIGH, 0x28)
                self._register(FINAL_RANGE_CONFIG_VALID_PHASE_LOW, 0x08)
                self._register(GLOBAL_CONFIG_VCSEL_WIDTH, 0x03)
                self._register(ALGO_PHASECAL_CONFIG_TIMEOUT, 0x09)
                self._register(0xFF, 0x01)
                self._register(ALGO_PHASECAL_LIM, 0x20)
                self._register(0xFF, 0x00)
            elif period_pclks == 12:
                self._register(FINAL_RANGE_CONFIG_VALID_PHASE_HIGH, 0x38)
                self._register(FINAL_RANGE_CONFIG_VALID_PHASE_LOW, 0x08)
                self._register(GLOBAL_CONFIG_VCSEL_WIDTH, 0x03)
                self._register(ALGO_PHASECAL_CONFIG_TIMEOUT, 0x08)
                self._register(0xFF, 0x01)
                self._register(ALGO_PHASECAL_LIM, 0x20)
                self._register(0xFF, 0x00)
            elif period_pclks == 14:
                self._register(FINAL_RANGE_CONFIG_VALID_PHASE_HIGH, 0x48)
                self._register(FINAL_RANGE_CONFIG_VALID_PHASE_LOW, 0x08)
                self._register(GLOBAL_CONFIG_VCSEL_WIDTH, 0x03)
                self._register(ALGO_PHASECAL_CONFIG_TIMEOUT, 0x07)
                self._register(0xFF, 0x01)
                self._register(ALGO_PHASECAL_LIM, 0x20)
                self._register(0xFF, 0x00)
            else:
                return False

        self._register(FINAL_RANGE_CONFIG_VCSEL_PERIOD, vcsel_period_reg)

        new_final_range_timeout_mclks =
self.timeout_microseconds_to_Mclks(self.timeouts["final_range_us"], period_pclks)

        if self.enables["pre_range"]:
            new_final_range_timeout_mclks += 1
            self._register(FINAL_RANGE_CONFIG_TIMEOUT_MACROP_HI,
self.encode_timeout(new_final_range_timeout_mclks))
        else:
            return False
        self.set_measurement_timing_budget(self.measurement_timing_budget_us)
        sequence_config = self._register(SYSTEM_SEQUENCE_CONFIG)
        self._register(SYSTEM_SEQUENCE_CONFIG, 0x02)
        self.perform_single_ref_calibration(0x0)
        self._register(SYSTEM_SEQUENCE_CONFIG, sequence_config)

        return True

def get_sequence_step_enables(self):
    sequence_config = self._register(0x01)

    self.enables["tcc"] = (sequence_config >> 4) & 0x1
    self.enables["dss"] = (sequence_config >> 3) & 0x1
    self.enables["msrc"] = (sequence_config >> 2) & 0x1
    self.enables["pre_range"] = (sequence_config >> 6) & 0x1
    self.enables["final_range"] = (sequence_config >> 7) & 0x1

def get_vcsel_pulse_period(self, type):
    if type == self.vcsel_period_type[0]:
        return self.decode_Vcsel_period(0x50)

```

```

elif type == self.vcsel_period_type[1]:
    return self.decode_Vcsel_period(0x70)
else:
    return 255

def get_sequence_step_timeouts(self):
    self.timeouts["pre_range_vcsel_period_pclks"] =
self.get_vcsel_pulse_period(self.vcsel_period_type[0])
    self.timeouts["msrc_dss_tcc_mclks"] = int(self._register(MSRC_CONFIG_TIMEOUT_MACROP)) + 1
    self.timeouts["msrc_dss_tcc_us"] =
self.timeout_Mclks_to_microseconds(self.timeouts["msrc_dss_tcc_mclks"],
                                     self.timeouts[

"pre_range_vcsel_period_pclks"])
    self.timeouts["pre_range_mclks"] = self.decode_timeout(PRE_RANGE_CONFIG_TIMEOUT_MACROP_HI)
    self.timeouts["pre_range_us"] =
self.timeout_Mclks_to_microseconds(self.timeouts["pre_range_mclks"],
                                     self.timeouts[

"pre_range_vcsel_period_pclks"])
    self.timeouts["final_range_vcsel_period_pclks"] =
self.get_vcsel_pulse_period(self.vcsel_period_type[1])
    self.timeouts["final_range_mclks"] =
self.decode_timeout(self._register(FINAL_RANGE_CONFIG_TIMEOUT_MACROP_HI))

    if self.enables["pre_range"]:
        self.timeouts["final_range_mclks"] -= self.timeouts["pre_range_mclks"]
        self.timeouts["final_range_us"] =
self.timeout_Mclks_to_microseconds(self.timeouts["final_range_mclks"],
                                     self.timeouts[

"final_range_vcsel_period_pclks"])

def timeout_Mclks_to_microseconds(self, timeout_period_mclks, vcsel_period_pclks):
    macro_period_ns = self.calc_macro_period(vcsel_period_pclks)
    return ((timeout_period_mclks * macro_period_ns) + (macro_period_ns / 2)) / 1000

def timeout_microseconds_to_Mclks(self, timeout_period_us, vcsel_period_pclks):
    macro_period_ns = self.calc_macro_period(vcsel_period_pclks)
    return (((timeout_period_us * 1000) + (macro_period_ns / 2)) / macro_period_ns)

def calc_macro_period(self, vcsel_period_pclks):
    return (((2304 * (vcsel_period_pclks) * 1655) + 500) / 1000)

def decode_timeout(self, reg_val):
    return ((reg_val & 0x00FF) << ((reg_val & 0xFF00) >> 8)) + 1

def encode_timeout(self, timeout_mclks):
    timeout_mclks = int(timeout_mclks)
    ls_byte = 0
    ms_byte = 0

    if timeout_mclks > 0:
        ls_byte = timeout_mclks - 1

        while (ls_byte & 0xFFFFFFF0) > 0:
            ls_byte >>= 1
            ms_byte += 1
        return (ms_byte << 8) or (ls_byte & 0xFF)
    else:
        return 0

def set_measurement_timing_budget(self, budget_us):
    start_overhead = 1320
    end_overhead = 960
    msrc_overhead = 660
    tcc_overhead = 590
    dss_overhead = 690
    pre_range_overhead = 660
    final_range_overhead = 550

    min_timing_budget = 20000

```



```

    if budget_us < min_timing_budget:
        return False
    used_budget_us = start_overhead + end_overhead

    self.get_sequence_step_enables()
    self.get_sequence_step_timeouts()

    if self.enables["tcc"]:
        used_budget_us += self.timeouts["msrc_dss_tcc_us"] + tcc_overhead
    if self.enables["dss"]:
        used_budget_us += 2* self.timeouts["msrc_dss_tcc_us"] + dss_overhead
    if self.enables["msrc"]:
        used_budget_us += self.timeouts["msrc_dss_tcc_us"] + msrc_overhead
    if self.enables["pre_range"]:
        used_budget_us += self.timeouts["pre_range_us"] + pre_range_overhead
    if self.enables["final_range"]:
        used_budget_us += final_range_overhead

    if used_budget_us > budget_us:
        return False
    final_range_timeout_us = budget_us - used_budget_us
    final_range_timeout_mclks = self.timeout_microseconds_to_Mclks(final_range_timeout_us,
self.timeouts["final_range_vcsel_period_pclks"])

    if self.enables["pre_range"]:
        final_range_timeout_mclks += self.timeouts["pre_range_mclks"]
    self._register(FINAL_RANGE_CONFIG_TIMEOUT_MACROP_HI,
self.encode_timeout(final_range_timeout_mclks))
    self.measurement_timing_budget_us = budget_us
    return True

def perform_single_ref_calibration(self, vhw_init_byte):
#     chrono = Timer.init(mode=Timer.ONE_SHOT, period=1000, callback=mycallback)
#     self._register(SYSRANGE_START, 0x01|vhw_init_byte)
#     chrono.start()
#     while self._register((RESULT_INTERRUPT_STATUS & 0x07) == 0):
#         time_elapsed = chrono.read_ms()
#         if time_elapsed > _IO_TIMEOUT:
#             return False
#     self._register(SYSTEM_INTERRUPT_CLEAR, 0x01)
#     self._register(SYSRANGE_START, 0x00)
    return True

i2c = I2C(scl=Pin(0), sda=Pin(10))      #initializing the I2C method for PYCOM-V
tof = VL53L0X(i2c)

tof.set_Vcsel_pulse_period(tof.vcsel_period_type[0], 18)

tof.set_Vcsel_pulse_period(tof.vcsel_period_type[1], 14)

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

```

# Table of Contents

SmartComputerLab.....	1
SmartComputerLab.....	2
0. Introduction.....	2
0.1 ESP32 (S1) Soc – an advanced unit for IoT architectures.....	3
0.2 ESP32 LOLIN32 (D32) board.....	3
0.3 IoT DevKits with ESP32S1 (D32) main board.....	4
0.4 Software – Thonny IDE.....	5
0.4.1 Installing Thonny IDE - thonny.org.....	5
0.4.2 Preparing the ESP32 LOLIN32 board.....	6
0.4.4 First example – x.led.blink.py.....	8
Lab 1.....	9
Sensor reading and data display (i2c).....	9
1.0 Introduction.....	9
1.1 First example – data display on OLED screen.....	10
1.2 Second example – sensor reading (T/H): SHT21.....	12
1.2.1 Preparing the code.....	12
1.3 Third example – reading a luminosity sensor (L) - BH1750.....	12
1.4 Fourth example – reading a PIR sensor – SR602.....	13
1.5 Fifth example – using RGB ring (neopixel).....	14
1.6 Sixth example – GPS module: NEO-6M (UART).....	15
Lab 2.....	16
WiFi communication and WEB servers.....	16
2.1 Network scan.....	16
2.2 Connection to the WiFi network, station mode – STA.....	17
2.3 Reading a WEB page.....	18
2.4 Getting time from NTP server.....	19
To do:.....	19
2.5 Simple WEB server – reading a variable.....	21
2.6 Simple WEB server – sending an order.....	22
2.4.3 Mini WEB server with Access Point – RGB LED management.....	23
Lab 3.....	25
MQTT Broker and ThingSpeak Server.....	25
3.1 MQTT Protocol and MQTT Client.....	25
3.1.1 MQTT client – the code.....	25
3.1.2 Broker MQTT on a PC.....	26
3.2 ThingSpeak server.....	27
3.2.1 Preparation for sending data with thingspeak.py library.....	27
3.2.2 Preparation for sending data as MQTT messages.....	28
3.2.2 Preparation for sending data as simple HTTP requests.....	29
Lab 4.....	31
LoRa technology for Long Range communication.....	31
4.0 Introduction.....	31
4.1 LoRa Modulation.....	31
4.2 sx127x.py driver library.....	31
4.3 Main program.....	33
4.4 LoRa functional modules.....	34
4.4.1 Transmitter – sender() (LoRaSender.py).....	34
4.4.2 Receiver – receive (LoRaReceiver.py).....	35
4.4.3 Receiver – onReceive (LoRaReceiverCallback.py).....	35
Lab 5.....	37
Development of simple IoT gateways.....	37
5.1 LoRa-WiFi Gateway (MQTT).....	37
5.2 LoRa-WiFi gateway (ThingSpeak) and MQTT.....	40
5.3 LoRa-WiFi gateway (ThingSpeak) and thingspeak.py library.....	42
Appendix:.....	43
More on sensors.....	43
A.1 MPU6050 – accelerometer, position and temperature sensor.....	43
A.1.1 File mpu6050.py.....	43
A.1.2 Test mpu6050.py.....	43
A.2 DHT driver and test.....	44
A.3 BMP180 driver and test.....	44
A.3.1 Module for the BMP180 pressure sensor : bmp180.py.....	44

A.4 ADC converter – ADS1115.....	47
A.4.1 File ADS1115.py.....	47
A.5 Micro Laser – VL53L0X.....	50
A.5.1 File VL53L0X.py.....	50