Laboratoires IoT de base

Mise en oeuvre des architectures IoT à la base de IoT-DevKit de SmartComputerLab

Contenu

0. **Introduction** **3**
- 0.1 ESP32 Soc – une unité avancée pour les architectures IoT 4
- 0.2 Carte Heltec WiFi LoRa 4
- 0.3 IoT DevKit une plate-forme de développement IoT 5
  - 0.3.1 Cartes d'extension simples – quelques exemples 6
  - 0.3.2 Cartes d'extension multi-capteurs – quelques exemples 7
- 0.4 L'installation de l'Arduino IDE sur un OS Ubuntu 8
  - 0.4.1 Installation des nouvelles cartes ESP32 et ESP8266 8
  - 0.4.2 Préparation d'un code Arduino pour la compilation et chargement 10

Laboratoire 1 ..... **11**
- 1.1 Premier exemple – l'affichage des données sur l'écran OLED 11
  - A faire 11
- 1.2 Deuxième exemple – capture et affichage des valeurs 12
  - 1.2.1 Capture de la température/humidité par SHT21 12
  - 1.2.2 Capture de la température/humidité par HTU21D 13
  - 1.2.3 Capture de la luminosité par BH1750 14
  - 1.2.4 Capture de la luminosité par MAX44009 15
  - 1.2.5 Capture de la pression/température avec capteur BMP180 16
  - A faire 17
  - 1.2.6 Détection de mouvement avec capteur PIR (SR602) 18
  - A faire : 18
  - 1.2.7 Mesure de distance avec capteur VL53L0X 19
  - A faire : 19
  - 1.2.8 Temps réel et positionnement GPS avec NEO-6M 20
  - 1.2.9 Affichage sur l »écran TFT - IL9341 21

Laboratoire 2 – communication en WiFi et serveur ThingSpeak.fr **23**
- 2.1 Introduction 23
  - 2.1.1 Un programme de test – scrutation du réseau WiFi 23
- 2.2 Mode WiFi – STA, client WEB et serveur ThingSpeak 24
  - 2.2.1 Envoi des données sur ThingSpeak 24
  - 2.2.2 Réception des données à partir de ThingSpeak 25
  - 2.2.3 Accès WiFi – votre Phone (PhoneAP) ou un routeur WiFi-4G 26
  - 2.2.4 ThingView - ThingSpeak viewer (Android) 27
  - A faire 28
  - 2.3 Mode WiFi – STA, avec une connexion à eduroam 28
  - 2.3.1 Envoi des données sur ThingSpeak avec point d'accès eduroam 28
  - 2.3.2 Réception des données partir du ThingSpeak avec point d'accès eduroam 29
  - A faire (si vous êtes à l'école ou à l'université – accès eduroam) 31

Laboratoire 3 – MQTT broker et clients **32**
- 3.1 Protocole MQTT 32
  - 3.1.1 Les bases 32
  - 3.1.2 Comment fonctionne MQTT 32
  - 3.4.3 Configuration du Broker 33
- 3.2 Serveur externe de MQTT - test.mosquitto.org 34
  - 3.2.1 Configuration des clients 34
  - 3.2.2 Utilisation de la bibliothèque MQTT 37
  - 3.2.3 Publication et réception des valeurs des capteurs 38
- 3.3 Utilisation d'une connexion sécurisée avec SSL et WiFiClientSecure 40
- 3.4 Utilisation de WiFiMQTTManager 41

3.4.1 Le code....41
3.5 Envoi de messages MQTT au serveur ThingSpeak....43
3.5.1 Le code....43
3.5.2 Mode deepSleep()et les données dans SRAM et EPROM....44
3.6 Android MQTT et Application Broker....45
A faire:....46

Laboratoire 4 - Bluetooth Classic (BT) et Bluetooth Low Energy (BLE)....48
4.1 Bluetooth Classique....48
4.2 Bluetooth Classique avec ESP32....49
4.2.1. Bluetooth simple lecture écriture en série....49
4.2.2 Échange de données à l'aide de Bluetooth Serial et de votre smartphone....50
4.2.3 A faire:....51
4.3 Bluetooth Low Energy (BLE) avec ESP32....52
4.3.1 Qu'est-ce que Bluetooth Low Energy?....52
4.3.2 Server BLE (notifier) et Client BLE....53
4.3.3 GATT....53
4.3.4 Services BLE....53
4.3.5 Caractéristiques BLE....54
4.4 BLE avec ESP32....55
4.4.1 ESP32 : Serveur BLE - notifier....55
4.4.2 Application BLE Scanner....57
4.4.3 ESP32 : Scanner BLE....58
4.4.4 Test d'un client ESP32 BLE avec votre smartphone....59
4.4.5 Serveur BLE avec un capteur– et fonction notify....62
4.5 Développement d'une passerelle BLE-WiFi vers serveur IoT....65
4.6 Résumé....67
Travail à faire (sur une carte DevKit)....68

Laboratoire 5 - Développement de serveurs locaux WEB-IoT....69
5.0 ESP32 – organisation de la mémoire....69
5.1 WiFiManager – Initialisation des identifiants (credentials) WiFi....70
5.2 Serveurs WEB simples en mode station (STA) et en mode point d'accès (AP)....72
5.2.1 Serveur WEB en mode station - STA....72
A faire :....74
5.2.2 Un simple serveur WEB en mode softAP....75
A faire :....76
5.3 Un serveur WEB avec système SPIFFS....77
5.3.1 Système de fichiers SPIFFS....77
5.3.2 Un serveur WEB avec le système de fichiers SPIFFS....78
5.4 Mini serveur WEB avec une carte SD....82
5.4.1 Un programme de test de la carte SD....82
5.3.2 Le programme mini-serveur WEB avec une carte SD....83
A faire :....86

Laboratoire 6 - Programmation OTA (Over The Air)....88
6.1 Introduction....88
6.1.1 Mémoire flash de ESP32....88
6.1.2 Mécanisme OTA....88
6.2 Implémentation d'OTA sur carte ESP32 par OTA de base....89
6.2.1 La mise en œuvre de l'OTA de base....89
6.2.2 Télécharger un nouveau code via WiFi....91
A faire :....93
6.3 OTA sur carte ESP32 avec serveur WEB....94
6.3.1 Le programme de départ avec Webserver....94
6.3.2 Accéder au serveur Web....97
6.3.3 Téléchargez un nouveau programme via WiFi (.bin)....98
6.3.4 Générez un fichier .bin dans l'IDE Arduino....100
6.3.5 Download a new sketch live on the ESP32....101
6.4 Implémentation d'OTA avec la bibliothèque WebOTA....102
6.4.1 Code initial....102
6.4.2 Chargement initial et final....102

Laboratoires IoT de base

Mise en oeuvre des architectures IoT à la base de IoT-DevKit de SmartComputerLab

0. Introduction

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

Le kit de développement contient une carte de base "basecard" pour y accueillir une unité centrale et un ensemble de cartes d'extension pour les capteurs, les actuateurs et les modems supplémentaires.
L'unité centrale est une carte équipée d'un **SoC ESP32** et d'un modem **LoRa** (*Long Range*).

Le premier laboratoire permet de préparer l'environnement de travail et de tester l'utilisation des capteurs connectés sur le bus I2C (température/humidité/luminosité/pression) et d'un afficheur. Pour nos développements et nos expérimentations nous utilisons le **IDE Arduino** comme outils de programmation et de chargement des programmes dans la mémoire flash de l'unité centrale.

Le deuxième laboratoire est consacré à la pris en main du modem WiFi intégré dans la carte principale (**Heltec ESP32-WiFi-LoRa**). La communication WiFi en mode station et un client permet d'envoyer les données des capteurs vers un serveur **WEB**. Dans notre cas nous utilisons le serveur de type **ThingSpeak**; (**`ThingSpeak.fr/com`**). Ces serveurs sont accessibles gratuitement, mais leur utilisation peut être limitée un temps (le cas de **`ThingSpeak.com`**).

Le troisième laboratoire permet de découvrir le protocole **MQTT** et d'expérimenter avec plusieurs exemples de **brokers** et **clients** basés sur différentes bibliothèques. Dans le laboratoire vous pouvez communiquer et échanger vos donnés IoT à distance via les broker externes.

Le quatrième laboratoire est orienté sur la technologie de communication **Bluetooth** – **Bluetooth Classic** (**BT**) et **Bluetooth Low Energy** (**BLE**). Vous allez expérimenter avec cette technologie à l'aide de vos smartphones. Vous allez développer une passerelle **Bluetooth-WiFI** sur votre **DevKit**.

Le cinquième laboratoire cible développement de simples **serveurs WEB** s'exécutant sur notre **DevKit**. Ces serveurs peuvent fonctionner en mode de station (**`STA`**) ou en mode Point d'Accès (**`AP`**). Il peuvent utiliser exclusivement la mémoire interne ou un contenu externe enregistré sur un **carte SD**.

Le sixième laboratoire introduit la programmation **OTA** (**Over The Air**). Ce mode de programmation permet de charger à distance les nouvelles applications via une connexion WiFi.

Après ces laboratoires préparatifs nous vous proposons plusieurs **laboratoires thématiques** qui permettrons de concevoir et développer différentes sortes d'applications IoT sous la forme de **mini-projets**. On vous présentera quelques exemples des architectures IoT dans les domaines d'application différents tels que la messagerie, le sécurité, l'environnement, le commerce,etc.

Vous pouvez vous inspirer de ces laboratoires ou de proposer d'autres de complexité comparable.

Nous vous fournirons (selon les disponibilités) des cartes d'extension et de capteurs/actionneurs nécessaires pour la réalisation de ces projets.

0.1 ESP32 Soc – une unité avancée pour les architectures IoT

ESP32 est une unité de microcontrôleur avancée conçue pour le développement d'architectures IoT. Un ESP32 intègre deux processeurs RISC 32-bit fonctionnant à 240 MHz et plusieurs unités de traitement et de communication supplémentaires, notamment un processeur ULP (**Ultra Low Power**), des modems WiFi / Bluetooth /BLE et un ensemble de contrôleurs E/S pour bus série (UART, I2C, SPI). , ..). Ces blocs fonctionnels sont décrits ci-dessous dans la figure suivante.

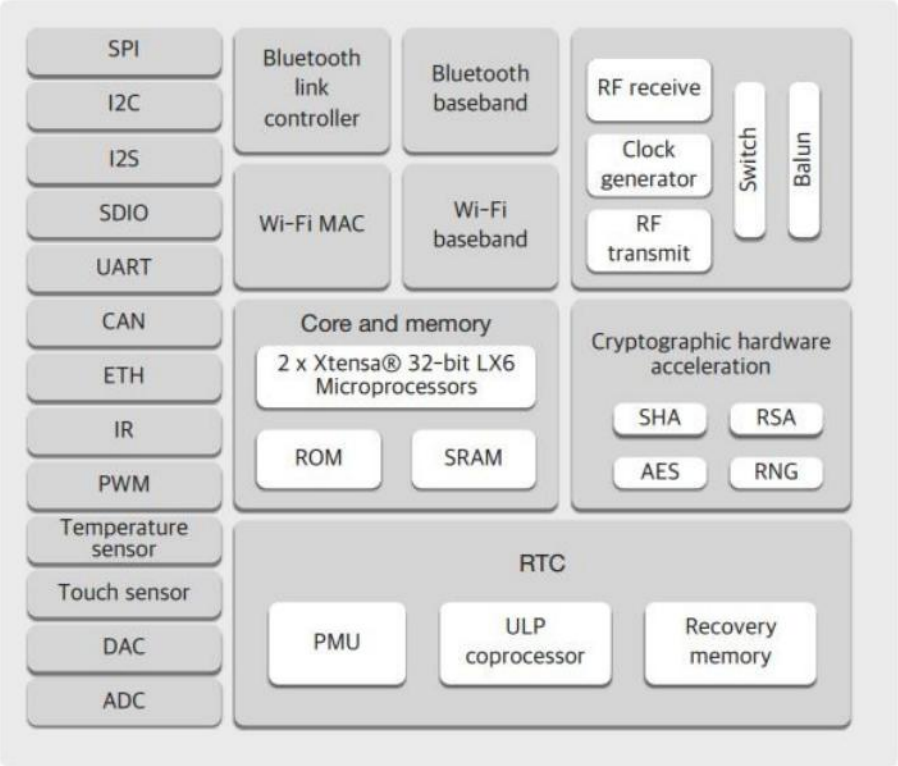


Figure 0.1 ESP32 SoC – architecture interne

0.2 Carte Heltec WiFi LoRa

De nos jours, les SoC ESP32 sont intégrés dans un certain nombre de cartes de développement qui incluent des circuits supplémentaires et des modems de communication. Notre choix est la carte **ESP32-Heltec WiFi-LoRa** qui intègre le modem LoRa **Semtech S1276/78** et (éventuellement) un écran . La figure suivante montre la carte **ESP32-Heltec WiFi-LoRa**

Figure 0.2 Carte MCU **ESP32-Heltec WiFi-LoRa**

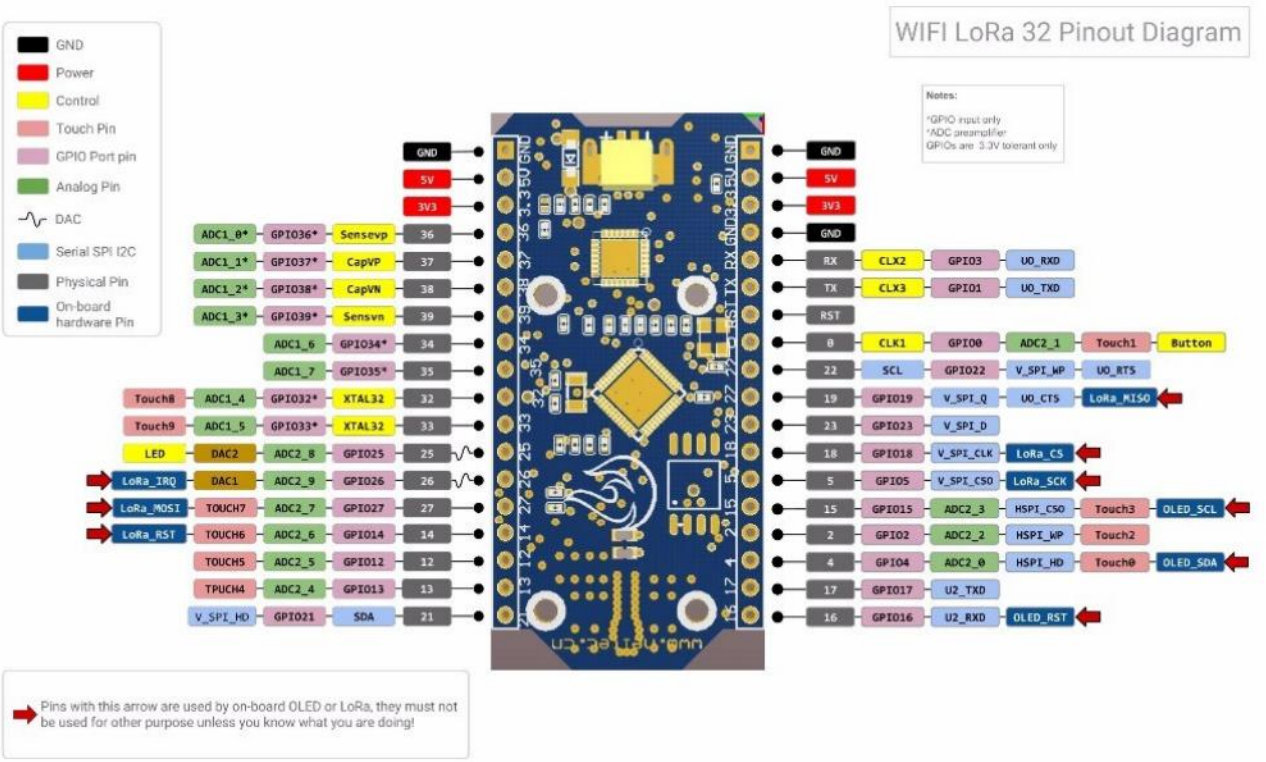


Figure 0.3. Carte **ESP32-Heltec WiFi-LoRa** et son *pin-out*

Comme nous pouvons le voir sur la figure ci-dessus, la carte Heltec expose des broches 2x18. Certaines de ces broches sont utilisées pour connecter le modem LoRa SX1276/8 (SPI) et l'afficheur OLED (I2C), d'autres broches peuvent être utilisées pour connecter des composants externes tels que des capteurs ou des actionneurs.

0.3 IoT DevKit une plate-forme de développement IoT

Une intégration efficace de la carte Heltec sélectionnée dans les architectures IoT nécessite l'utilisation d'une plate-forme de développement telle que IoT DevKit proposée par **SmartComputerLab**.
L'**IoTDevKit** est composé d'une carte de base et d'un grand nombre de cartes d'extension conçues pour l'utilisation efficace des bus de connexion et de tous les types de capteurs et d'actionneurs.

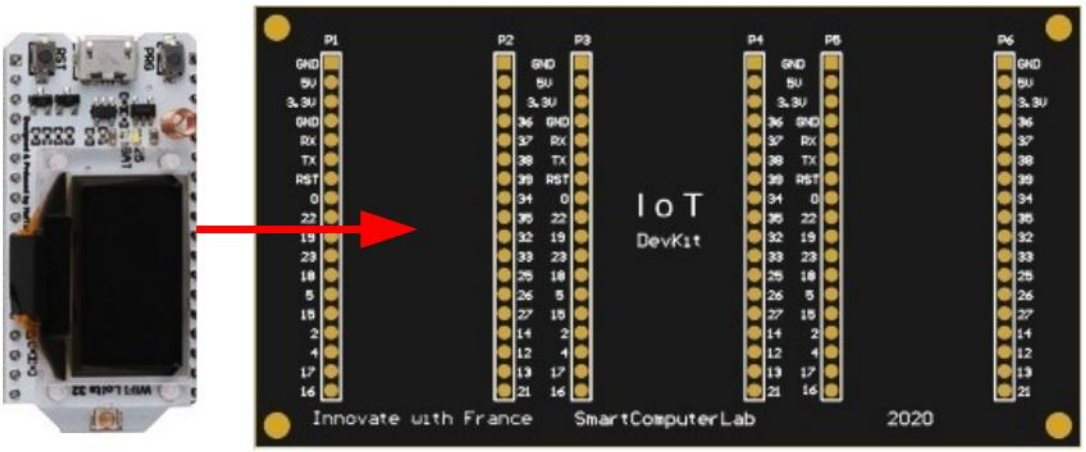
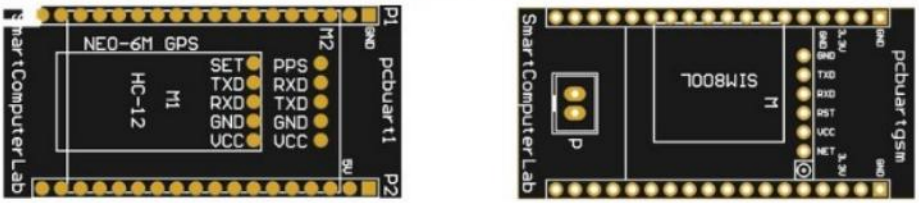


Figure 0.4. **IoT DevKit**: carte de base (type bridge) avec l'unité principale et les emplacements pour les cartes d'extension

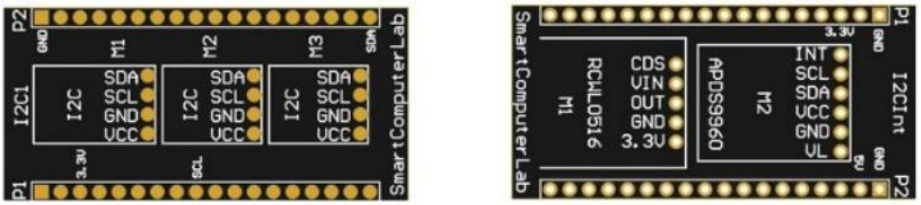
0.3.1 Cartes d'extension simples – quelques exemples

Figure 0.5. **IoT DevKit**: cartes d'extension, exemple d'implantation sur les d'extension **I2C** avec capteurs **BH1750** et **HTU21D** et une carte d'extension **UART** avec module GPS NEO-M6

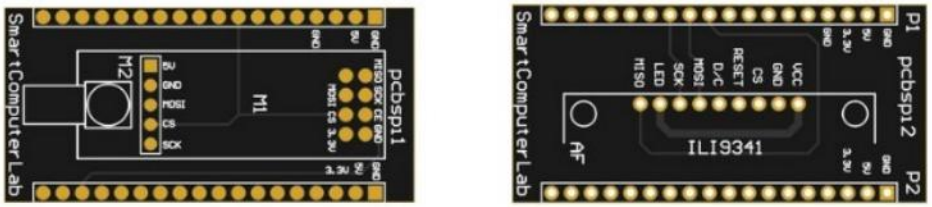
Cartes d'extension pour le bus **UART** (deux exemples):



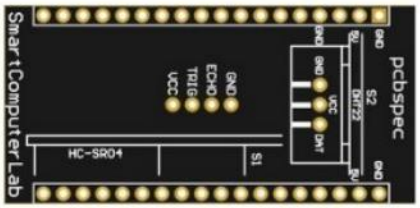
Cartes d'extension pour le bus **I2C** (deux exemples):



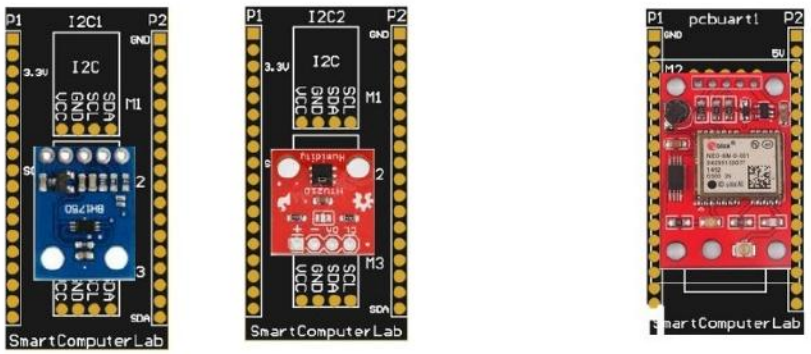
Cartes d'extension pour le bus **SPI** (deux exemples):



Cartes d'extension spécifiques (un exemple):

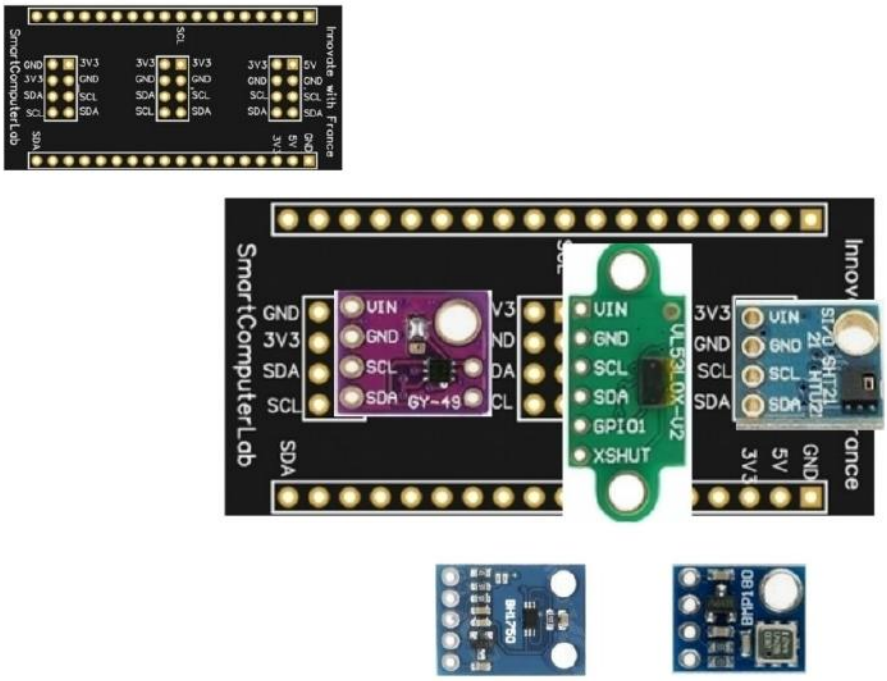


Cartes d'extension **I2C** avec capteurs **BH1750** et **HTU21D** et une carte d'extension **UART** avec module GPS NEO-M6

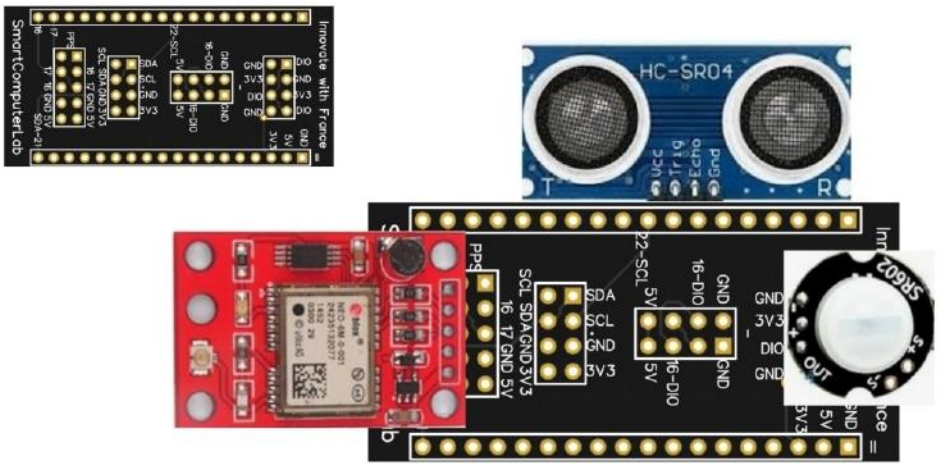


0.3.2 Cartes d'extension multi-capteurs – quelques exemples

Carte **multi-capteurs I2C**



Carte **multi-capteurs** : PIR, Echo, GPS (**UART**), **I2C**



0.4 L'installation de l'Arduino IDE sur un OS Ubuntu

Pour nos développements et nos expérimentations nous utilisons l'**IDE Arduino** comme outils de programmation et de chargement des programmes dans la mémoire flash de l'unité centrale.
Afin de pouvoir développer le code pour les application nous avons besoin d'un environnement de travail comprenant; un PC, un OS type Ubuntu 16.04LTS, l'environnement Arduino IDE, les outils de compilation/chargement pour les cartes ESP32 et les bibliothèque de contrôle pour les capteurs, actuateurs (par exemple **relais**), et les modems de communication.

Pour commencer mettez le système à jour par :

```
sudo apt-upgrade  et  sudo apt update
sudo apt-get install python-serial python3-serial
```

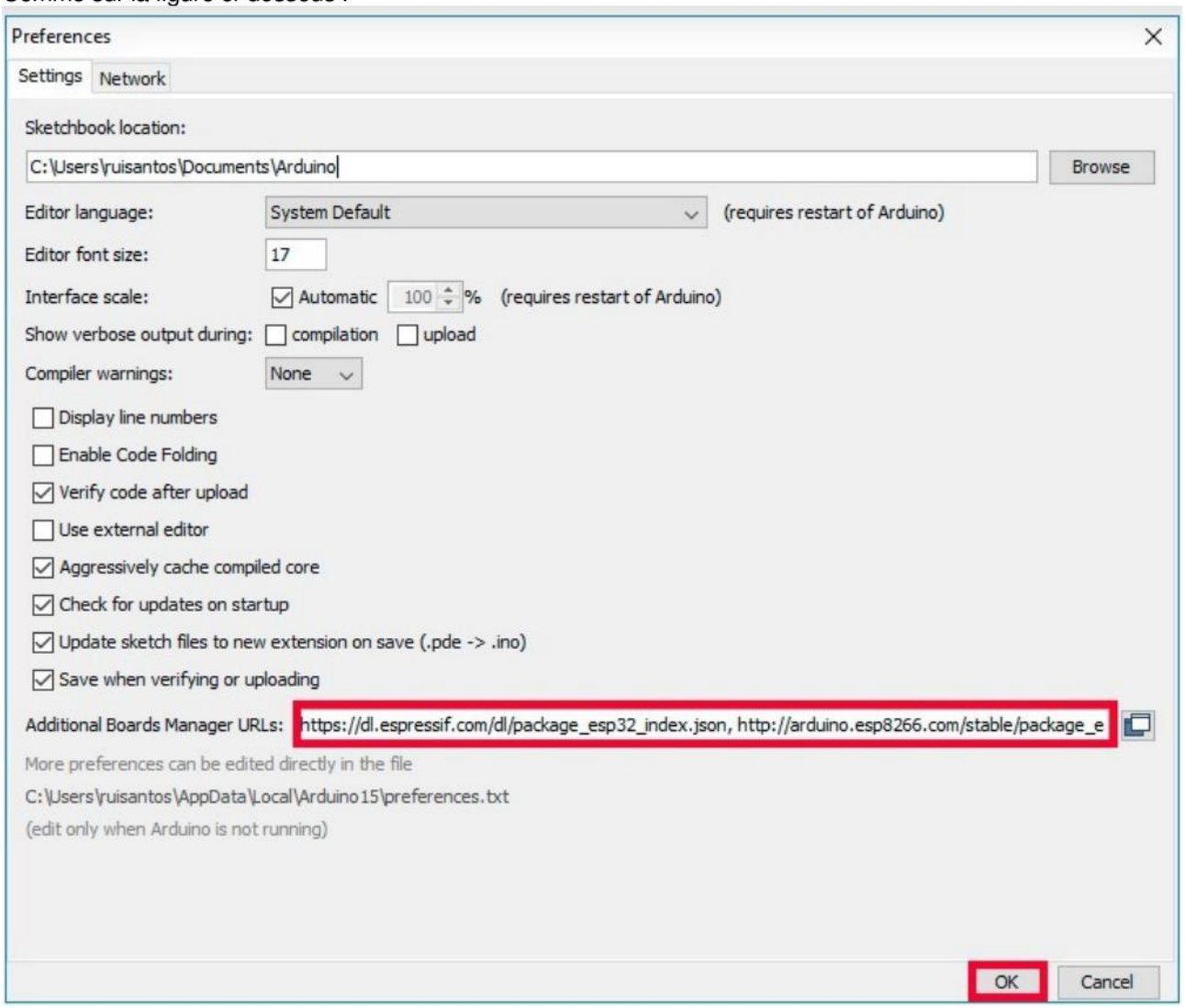
Ensuite il faut installer le dernier Arduino IDE à partir de **`https://www.arduino.cc`**

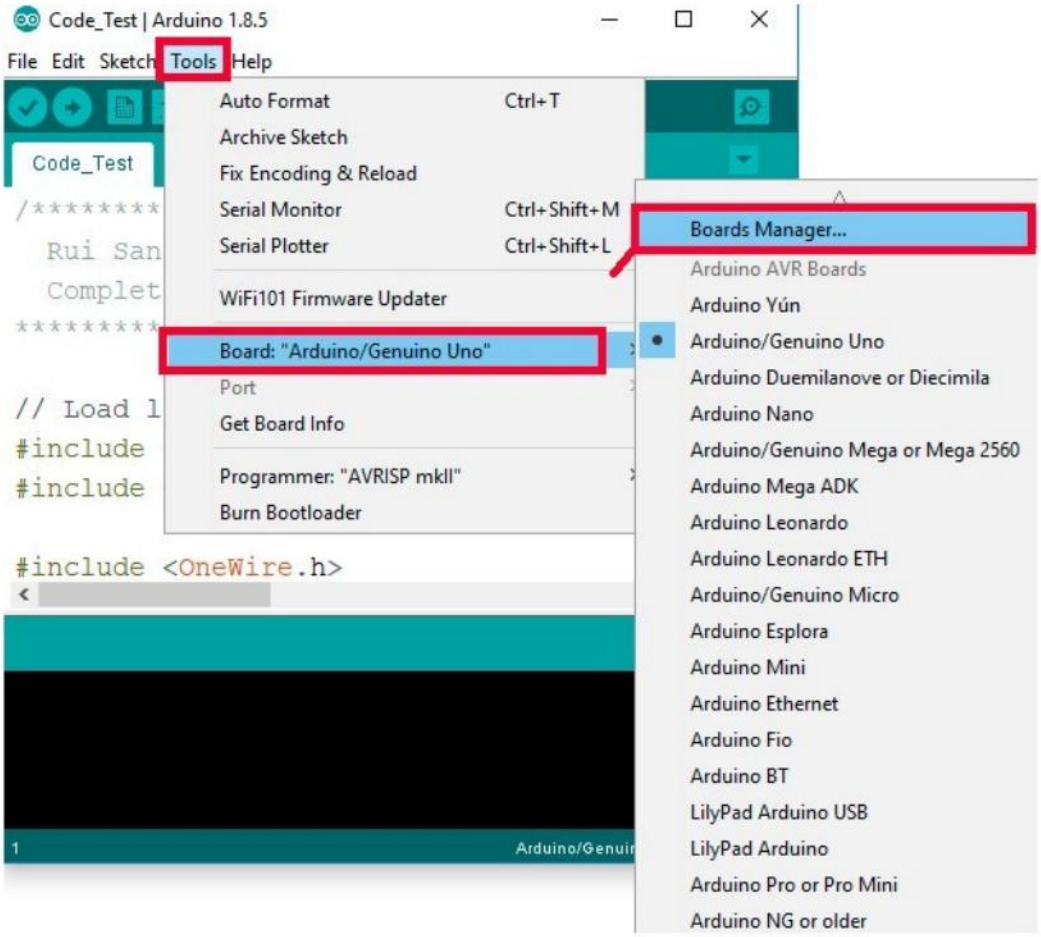
0.4.1 Installation des nouvelles cartes ESP32 et ESP8266

Après son installation allez dans les **Preferences** et ajoutez deux **Boards Manager URLs** (séparées par une virgule) :

`https://dl.espressif.com/dl/package_esp32_index.json`,
`http://arduino.esp8266.com/stable/package_esp8266com_index.json`

Comme sur la figure ci-dessous :





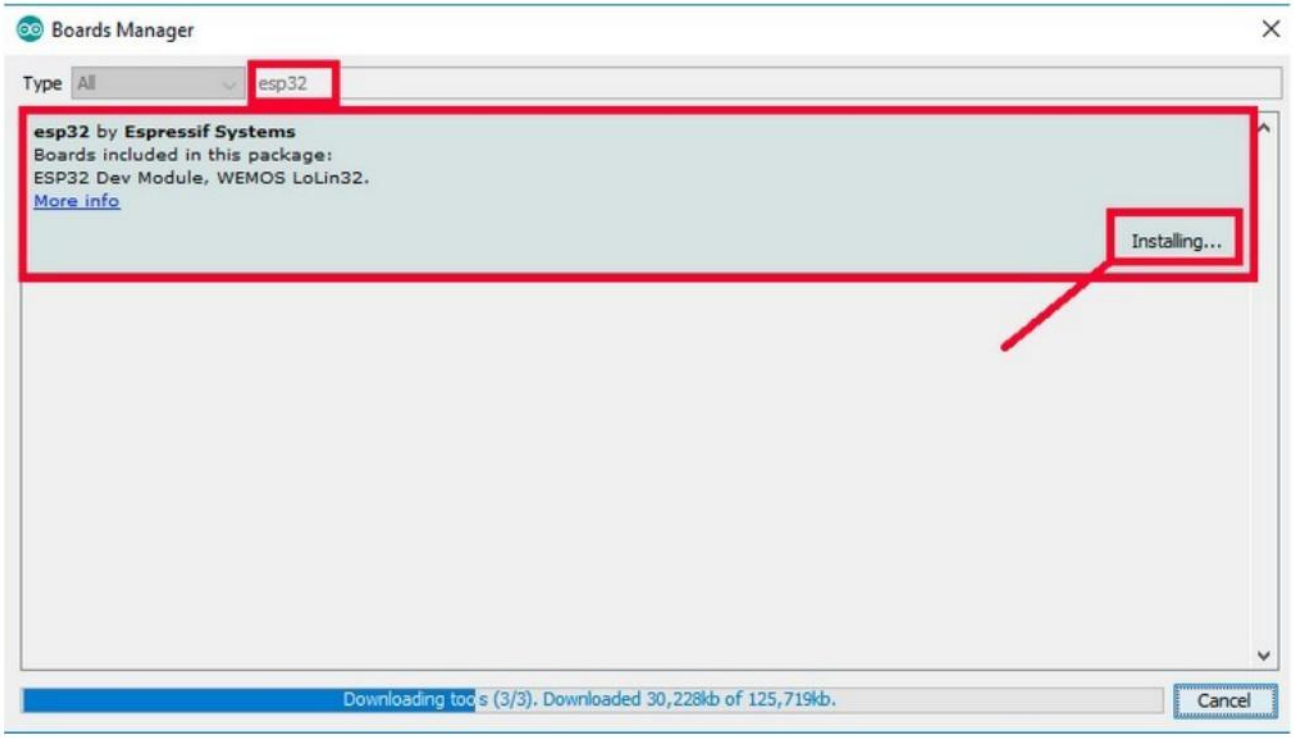


Figure 0.6. L'intégration des cartes **ESP32** et **ESP8266** dans l'environnement IDE Arduino

0.4.2 Préparation d'un code Arduino pour la compilation et chargement

La compilation doit être effectuée sur la carte **Heltec_WIFI_LoRa_32.** Elle doit être sélectionnée dans le menu **`Tools→Board`**

Avant la compilation il faut installer les bibliothèques nécessaires pour piloter différents dispositifs.
Par exemple l'image ci-dessous montre comment installer la bibliothèque **`U8g2`** qui pilote les écrans type **OLED**.

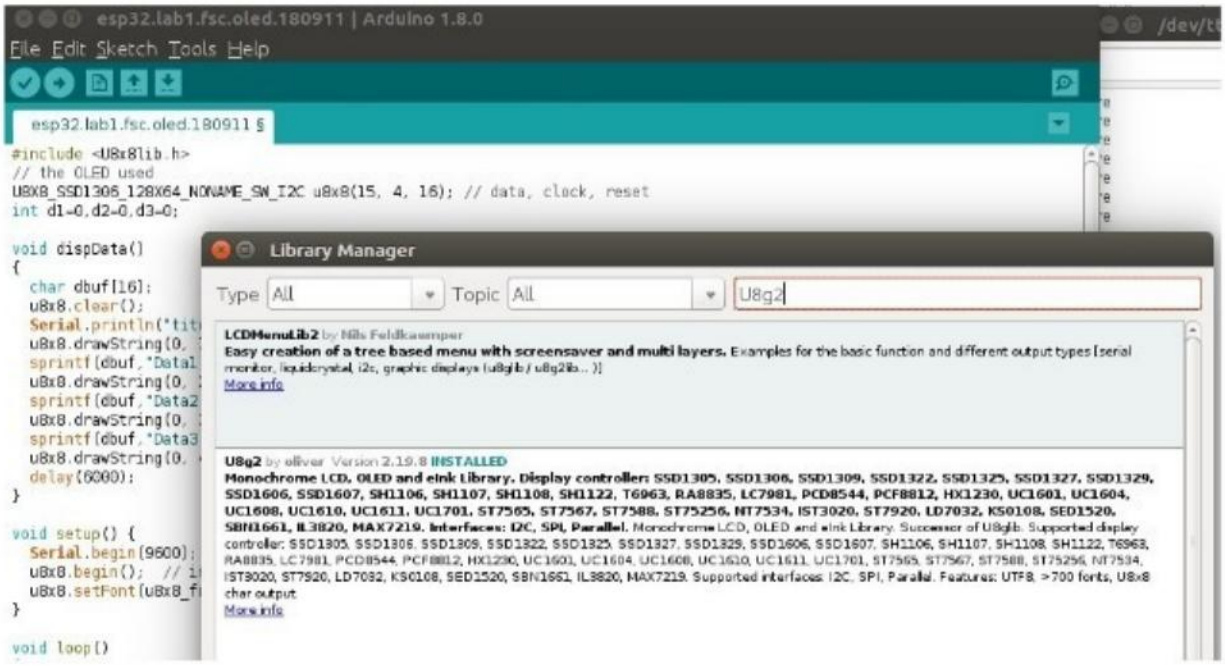


Figure 0.7. Intégration d'une bibliothèque Arduino (**U8g2**)

Laboratoire 1

1.1 Premier exemple – l'affichage des données sur l'écran OLED

Dans cet exercice nous allons simplement afficher un titre et 2 valeurs numériques sur l'écran **OLED** intégré dans la carte ESP32.

Figure 1.1 Ecran **OLED** de la carte ESP32 (Heltec WiFi LoRa 32)

Vous devez installer la bibliothèque **U8g2** (**https://github.com/olikraus/u8g2**). Cela peut être trouvé dans le gestionnaire de bibliothèque IDE Arduino. Ouvrez **Sketch> Include Library> Manage Libraries** et recherchez, puis installez **U8g2**. Notez que l'écran **OLED** est connecté sur un bus I2C avec trois broches : **SCL** – **clock**, **SDA** – **data/address** et **RESET**.

```
#include <U8x8lib.h>  // bibliothèque à charger a partir de
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15,4,16); // clock, data, reset
int d1=0,d2=0,d3=0;

void dispData()
{
  char dbuf[16];
  u8x8.clear();
  Serial.println("titre");
  u8x8.drawString(0,1,"titre"); // 0 – colonne (max 15), 1 – ligne (max 7)
  sprintf(dbuf,"Data1:%d",d1); u8x8.drawString(0,2,dbuf);
  sprintf(dbuf,"Data2:%d",d2); u8x8.drawString(0,3,dbuf);
  sprintf(dbuf,"Data3:%d",d3);u8x8.drawString(0,4,dbuf);
  delay(6000);
}

void setup() {
  Serial.begin(9600);
  u8x8.begin();  // initialize OLED
  u8x8.setFont(u8x8_font_chroma48medium8_r);
}

void loop()
{
d1++;d2+=2;d3+=4 ;
// ici appeler la fonction d'affichage
}
```

A faire

Compléter, compiler, et charger ce programme.

1.2 Deuxième exemple – capture et affichage des valeurs

1.2.1 Capture de la température/humidité par SHT21

Dans cet exercice nous allons lire les valeurs fournies par le capteur Température/Humidité **SHT21** et afficher les 2 valeurs sur l'écran OLED intégré dans la carte ESP32.
Le capteur **SHT21** doit être connecté sur le bus **I2C**, donc il nous faut une carte d'extension I2C avec une configuration des broches identique à celle du capteur (VIN correspond à 3V3).

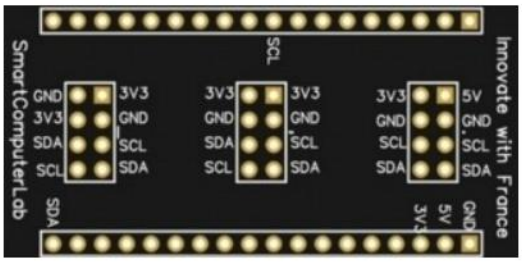


Figure 1.2. **IoT DevKit :** carte d'extension multi **I2C** et un capteur de température/humidité **SHT21**

Code Arduino

Code Arduino

```
// https://github.com/chaveiro/SHT2x-Arduino-Library

#include <Wire.h>
#include "SHT2x.h"

SHT2x SHT2x;

void setup()
{
  Wire.begin(21,22);
  SHT2x.begin();
  Serial.begin(9600);
}

void loop()
{
  uint32_t start = micros();
  Serial.print("Humidity(%RH): ");
  Serial.print(SHT2x.GetHumidity(),1);
  Serial.print("\tTemperature(C): ");
  Serial.print(SHT2x.GetTemperature(),1);

  uint32_t stop = micros();
  Serial.print("\tRead Time: ");
  Serial.println(stop - start);
  delay(1000);
}
```

1.2.2 Capture de la température/humidité par HTU21D

Dans cet exercice nous allons lire les valeurs fournies par le capteur Température/Humidité **HTU21D** et afficher les 2 valeurs sur l'écran OLED intégré dans la carte ESP32.
Le capteur **HTU21D** doit être connecté sur le bus **I2C**, donc il nous faut une carte d'extension **I2C** et l'emplacement avec la configuration des broches identique à celle du capteur.

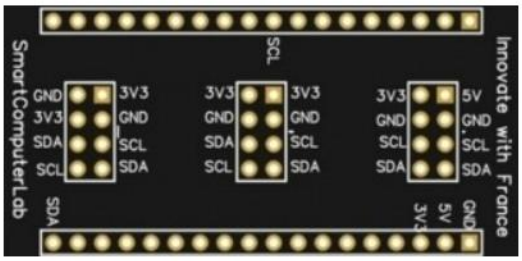


Figure 1.3 IoT DevKit : carte d'extension multi **I2C** et un capteur de température/humidité **HTU21D**

Code Arduino

```
#include <Wire.h>
#include "SparkFunHTU21D.h"
//Create an instance of the object
HTU21D sensor;

void setup()
{
  Serial.begin(9600);
  Serial.println("HTU21D Example!");
  myHumidity.begin();
}

void loop()
{
  float humd = sensor.readHumidity();
  float temp = sensor.readTemperature();
  Serial.print("Time:");
  Serial.print(millis());
  Serial.print(" Temperature:");
  Serial.print(temp, 1);
  Serial.print("C");
  Serial.print(" Humidity:");
  Serial.print(humd, 1);
  Serial.print("%");
  Serial.println();
  delay(1000);
}
```

Attention :

Pour le bon fonctionnement il faut **enlever le capteur SHT21**.

1.2.3 Capture de la luminosité par BH1750

Dans cet exercice utilisez la carte **I2C1** pour le capteur **de la luminosité BH1750**

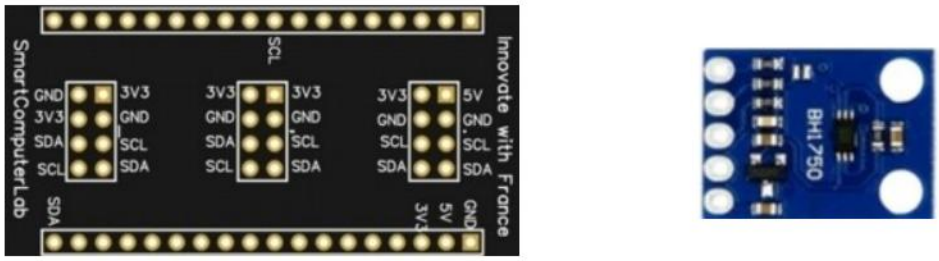


Figure 1.4. **IoT DevKit :** cartes d'extension multi - **I2C** pour le capteur Luminosité **BH1750**

Code Arduino

Attention : Avant la compilation il faut installer la bibliothèque **`BH1750.h`**

https://github.com/claws/BH1750

```
#include <Wire.h>
#include <BH1750.h>
BH1750 lightMeter;

void setup(){
  Wire.begin(21,22);  // carte d'extension I2C1 ou petite carte I2C
  Serial.begin(9600);
  lightMeter.begin();
  Serial.println("Running...");
  delay(1000);
}

void loop() {
  uint16_t lux = lightMeter.readLightLevel();
  delay(1000);
  Serial.print("Light: ");
  Serial.print(lux);
  Serial.println(" lx");
  delay(1000);
}
```

1.2.4 Capture de la luminosité par MAX44009

Dans cet exemple nous utilisons un capteur de luminosité type **MAX44009** (GY-49). Ce capteur est connectée comme le capteur BH1750 sur le bus I2C.
Nous communiquons avec ce capteurs **directement** par les trames **I2C** ce qui permet de mieux comprendre le **fonctionnement de ce bus**.

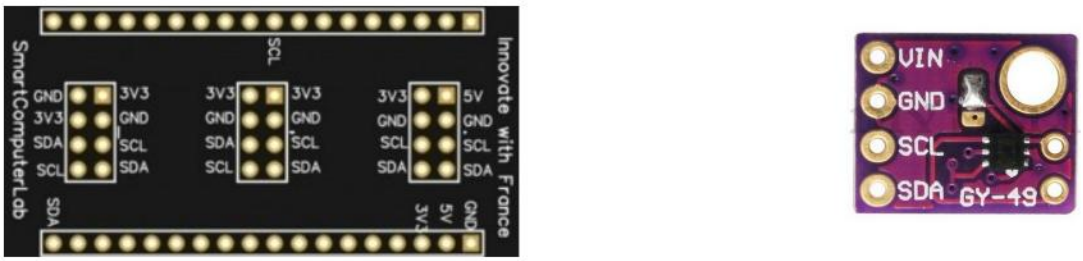


Figure 1.5 IoT DevKit : cartes d'extension **I2C** et un capteur de luminosité **MAX44009 (GY-49)**

Code Arduino :

```
#include<Wire.h>
#define Addr 0x4A

void setup()
{
Wire.begin(21,22);
Serial.begin(9600);
Wire.beginTransmission(Addr);
Wire.write(0x02); Wire.write(0x40);
Wire.endTransmission();
delay(300);
}

void loop()
{
unsigned int data[2];
Wire.beginTransmission(Addr);
Wire.write(0x03);
Wire.endTransmission();
Wire.requestFrom(Addr, 2); // Request 2 bytes of data
// Read 2 bytes of data luminance msb, luminance lsb
if (Wire.available() == 2)
{
data[0] = Wire.read(); data[1] = Wire.read();
}
// Convert the data to lux
int exponent = (data[0] & 0xF0) >> 4;
int mantissa = ((data[0] & 0x0F) << 4) | (data[1] & 0x0F);
float luminance = pow(2, exponent) * mantissa * 0.045;
Serial.print("Ambient Light luminance :"); Serial.print(luminance);
Serial.println(" lux");
delay(500);
}
```

1.2.5 Capture de la pression/température avec capteur BMP180

Le capteur BMP180 permet de capter la pression atmosphérique et la température. Sa précision est seulement de **+/- 100 Pa** et **+/-1.0C**.

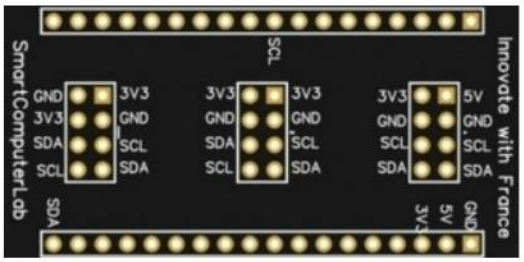


Figure 1.6. **IoT DevKit :** carte d'extension multi **I2C** et un capteur de pression/température **BMP180**

La valeur standard de la pression atmosphérique est :

101 325 Pa = 1,013 25 bar = 1 atm

Code Arduino - BMP180

```
#include <Arduino.h>
#include <Wire.h>
#include <BMP180I2C.h>
#define I2C_ADDRESS 0x77
BMP180I2C bmp180(I2C_ADDRESS);

void setup() {
      Serial.begin(9600);
      Wire.begin();
      if (!bmp180.begin())
      {
            Serial.println("check your BMP180 Interface and I2C Address.");
            while (1);
      }
      bmp180.resetToDefaults();
      //enable ultra high resolution mode for pressure measurements
      bmp180.setSamplingMode(BMP180MI::MODE_UHR);
}

void loop() {
      delay(1000);
      if (!bmp180.measureTemperature())
      {
      Serial.println("could not start temperature measurement");
            return;
      }
      //wait for hasValue() returned true.
      do
      {
            delay(100);
      } while (!bmp180.hasValue());
      Serial.print("Temperature: ");
      Serial.print(bmp180.getTemperature());
      Serial.println(" degC");
```

```
      if (!bmp180.measurePressure())
      {
      Serial.println("could not start perssure measurement");
            return;
      }
      do
      {
            delay(100);
      } while (!bmp180.hasValue());
      Serial.print("Pressure: ");
      Serial.print(bmp180.getPressure());
      Serial.println(" Pa");
}
```

A faire

1. Complétez les programmes ci-dessus afin d'afficher les données de la Luminosité sur l'écran OLED

2. Développer une application avec la capture et l'affichage de l'ensemble de trois données : Température/Humidité et Luminosité.

3. Développer une application avec la capture et l'affichage de l'ensemble de trois données : Température/Humidité et Pression (BMP180).

1.2.6 Détection de mouvement avec capteur PIR (SR602)

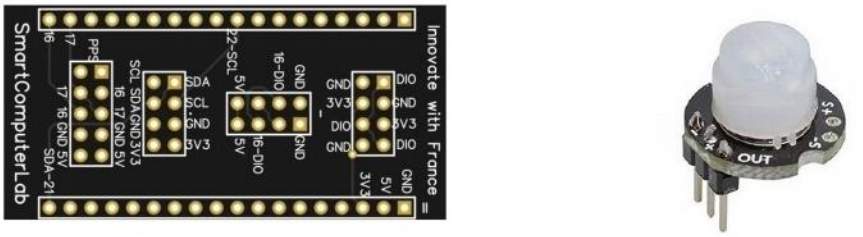


Figure 1.7 Carte **multi-capteurs (I2C, UART, one-wire)** et capteur **PIR (SR602)**

Code Arduino

```
#include <U8x8lib.h>  // bibliothèque à charger a partir de
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15,4,16); // clock, data, reset
int d1=0,d2=0,d3=0;
#define PIR 16  // problème ?
bool MOTION_DETECTED = false;
int counter=0;

void dispData()
{
  char dbuf[16];
  u8x8.begin();  // initialize OLED
  u8x8.setFont(u8x8_font_chroma48medium8_r);u8x8.clear();
  u8x8.drawString(0,1,"Motion detected");
  sprintf(dbuf,"Count:%d",counter); u8x8.drawString(0,3,dbuf);
  delay(100);
}

void pinChanged() { MOTION_DETECTED = true; }

void setup() {
Serial.begin(9600);pinMode(PIR,INPUT);
attachInterrupt(PIR, pinChanged, RISING);
}

void loop()
{
  int i=0;
  if(MOTION_DETECTED){ Serial.println("Motion detected.");
      delay(1000);counter++;
      MOTION_DETECTED = false;
      Serial.println(counter);
      dispData();
      pinMode(PIR,INPUT) attachInterrupt(PIR, pinChanged, RISING);
      }
}
```

A faire :

1. Analyser le programme ci-dessus

Pourquoi **doit on réinitialiser l'écran et l'interruption à chaque pas d'utilisation.**

1.2.7 Mesure de distance avec capteur VL53L0X

Le **VL53L0X** est un module de **télémétrie laser** à **temps de vol** (**ToF**) offrant une mesure de distance précise quelles que soit la cible contrairement aux technologies conventionnelles. Il peut mesurer des distances absolues jusqu'à 2m**. (le capteur fonctionne mieux avec Vcc=5V)**

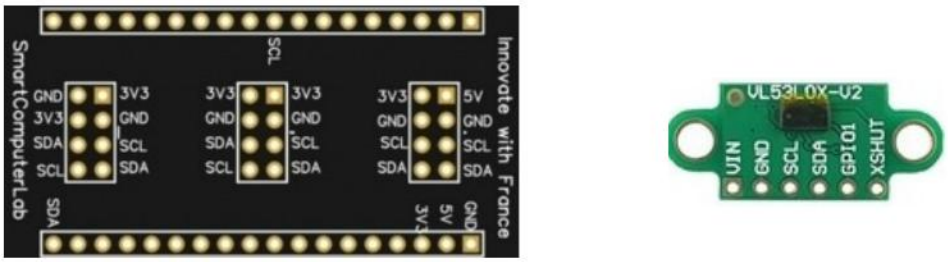


Figure 1.8 Carte multi-capteurs (I2C) et capteur **VL53L0X (laser-ToF)**

Fonctions :

```
setMode(ModeState mode, PrecisionState precision)
```

- `Continuous--->Continuous measurement model`
- `Single------->Single measurement mode`
- `High--------->Accuracy of 0.25 mm`
- `Low---------->Accuracy of 1 mm`

```
void start() - This function is used to enabled VL53L0X
float getDistance()  - This function is used to get the distance
uint16_t getAmbientCount() - This function is used to get the ambient count
uint16_t getSignalCount() - This function is used to get the signal count
uint8_t getStatus(); - This function is used to get the status
void stop() - This function is used to stop measuring
```

Le code

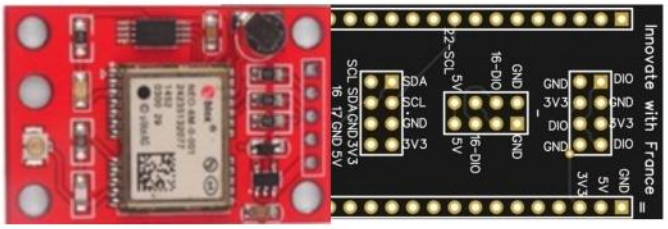
```
#include "Arduino.h"
#include "Wire.h"
#include "DFRobot_VL53L0X.h"

DFRobotVL53L0X sensor;

void setup() {
  Serial.begin(9600);
  Wire.begin(21,22);   // SDA, SCL
  sensor.begin(0x50);//Set I2C sub-device address
  //Set to Back-to-back mode and high precision mode
  sensor.setMode(Continuous,High);
  //Laser rangefinder begins to work
  sensor.start();
}

void loop()
{
  Serial.print("Distance: ");
  Serial.print(sensor.getDistance());Serial.println("mm");

  delay(500); // delay does not affect the measurement accuracy
}
```

A faire :

1. Testez le programme ci-dessus et ajoutez l'affiche sur l'écran OLED

1.2.8 Temps réel et positionnement GPS avec NEO-6M

NEO-6M est un module de **global positioning system** (GPS). Il est très populaire, économique et de hautes performances avec une antenne patch en céramique, une puce de mémoire intégrée et une batterie de secours est intégrée sur le module.

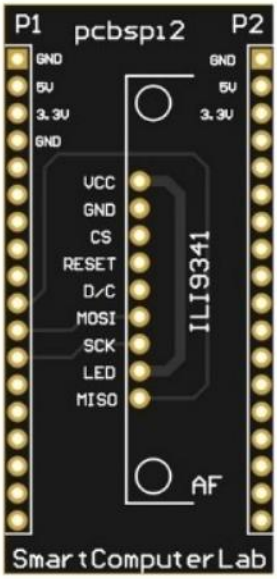


Le code

```
#include <TinyGPS++.h>
HardwareSerial uart(2);
TinyGPSPlus gps;
void setup(){
  Serial.begin(9600);
  uart.begin(9600, SERIAL_8N1, 17, 16);}

  int i=0,ihour;
  char gpst[2048],ttime[16],ftime[16],hour[8],nhour[2];
  char *ptr=NULL;

  void loop(){
    while (uart.available() > 0)
      {
        gps.encode(uart.read());
        if (gps.location.isUpdated())
          {
            Serial.print("Latitude= ");
            Serial.print(gps.location.lat(), 6);
            Serial.print(" Longitude= ");
            Serial.println(gps.location.lng(), 6);
            Serial.print("Date:");
            Serial.print(gps.date.month());
            Serial.print("/");
            Serial.print(gps.date.day());
            Serial.print("/");
            Serial.print(gps.date.year());
            Serial.print("   Time:");
            if (gps.time.hour() < 10)
              Serial.print("0");
            Serial.print(gps.time.hour());
            Serial.print(":");
            if (gps.time.minute() < 10)
              Serial.print("0");
            Serial.print(gps.time.minute());
            Serial.print(":");
            Serial.println(gps.time.second());
          }
        }
      }
```

1.2.9 Affichage sur l »écran TFT - IL9341

L'exemple suivant montre l'utilisation d'une carte pour les écrans **SPI-TFT** type **IL9341**. Nous utilisons ici la bibliothèque **`TFT_eSPI.h`** avec une interface **SPI configurable** dans le fichier **`User_Setup.h`**



Un exemple du code :

Attention : Il faut préparer les connections sur le bus SPI dans le fichier User_Setup.h présent dans votre répertoire **`TFT_eSPI-master`** dans le **`libraries`** de **`Arduino`**.

```
#include <TFT_eSPI.h> // Graphics and font library for ILI9341 driver chip
#include <SPI.h>
// to be defined in User_Setup.h
//#define TFT_MISO 19 // Not connected
//#define TFT_MOSI 23
//#define TFT_SCLK 18
//#define TFT_CS 5 // Chip select control pin
//#define TFT_DC 2 // Data Command control pin
//#define TFT_RST 4 // Reset pin (could connect to RST pin)
// GND tp GND , VCC to 3.3V and LED to 3.3V
#define TFT_GREY 0x5AEB // New colour
TFT_eSPI tft = TFT_eSPI(); // Invoke library
void setup(void) {
Serial.begin(9600);
tft.init();
tft.setRotation(3);
}

int count=0;
char disp[12];

void loop() {
Serial.println("in the loop");
// Fill screen with grey
// tft.fillScreen(TFT_GREY);
tft.fillScreen(TFT_BLACK);
// Set "cursor" at top left corner of display (0,0) and select font 2
// or stay on the line is there is room for the text with tft.print)
tft.setCursor(0, 0, 4);
```

```
// Set the font colour to be white with a black background, set text size
multiplier to 1
if(count>12)tft.setTextColor(TFT_RED,TFT_BLACK);
else tft.setTextColor(TFT_GREEN,TFT_BLACK);
tft.setTextSize(8);
// We can now plot text on screen using the "print" class
sprintf(disp,"%3.3d",count); count++;
tft.println(disp);
tft.setCursor(20, 150, 2);
tft.setTextSize(3);
if(count>12) tft.println("Porte fermee");
else tft.println("Porte ouverte");

tft.setCursor(200, 220, 2);
// Set the font colour to be yellow with no background, set to font 7
tft.setTextColor(TFT_YELLOW); tft.setTextSize(1);
tft.println("SmartComputerLab");
// Set the font colour to be red with black background, set to font 4
//tft.setTextColor(TFT_RED,TFT_BLACK);
//tft.setTextFont(4);
//tft.println(3735928559L, HEX); // Should print DEADBEEF
// Set the font colour to be green with black background, set to font 4
//tft.setTextColor(TFT_GREEN,TFT_BLACK);
//tft.setTextFont(4);
//tft.println("Groop");
//tft.println("I implore thee,");
// Change to font 2
//tft.setTextFont(2);
//tft.println("my foonting turlingdromes.");
//tft.println("And hooptiously drangle me");
//tft.println("with crinkly bindlewurdles,");
// This next line is deliberately made too long for the display width to test
// automatic text wrapping onto the next line
//tft.println("Or I will rend thee in the gobberwarts with my
blurglecruncheon,see if I don't!");
// Test some print formatting functions
//float fnumber = 123.45;
// Set the font colour to be blue with no background, set to font 4
//tft.setTextColor(TFT_BLUE);
//tft.setTextFont(4);
//tft.print("Float = "); tft.println(fnumber);
// Print floating point number
//tft.print("Binary = "); tft.println((int)fnumber, BIN); // Print as integer
value in binary
//tft.print("Hexadecimal = "); tft.println((int)fnumber, HEX); // Print as
integer number in Hexadecimal
delay(3000);
}
```

A faire :

1. Modifier le contenu du programme ci-dessus pour afficher , par exemple:
`IoT - Lab1` sur votre écran TFT
2. Ajouter un **capteur** , par exemple température-humidité sur la carte multi-capteurs **I2C** et afficher les valeurs captées
3. Ajouter le **modem GPS** , la carte multi-capteurs **PIR, UART** et afficher les valeurs captées

Laboratoire 2 – communication en WiFi et serveur `ThingSpeak.fr`

2.1 Introduction

Dans ce laboratoire nous allons nous intéresser aux moyens de la **communication** et de la **présentation** (**stockage**) des résultats. Etant donné que le SoC ESP32 intègre les modems de WiFi et de Bluetooth nous allons étudier la communication par **WiFi**.

Principalement nous avons deux modes de fonctionnement **WiFi** – mode station **STA**, et mode point d'accès **`softAP`**.
Dans le mode **STA** nous pouvons également tester l'état de fonctionnement : connecté ou de-connecté. Dans le mode **AP** nous pouvons déterminer l'adresse IP, le masque du réseau, et le canal de communication WiFi (1-13).
Un troisième mode appelé **ESP-NOW** permet de communiquer entre les cartes directement (sans un point d'accès), donc plus rapidement et à plus grande distance, par le biais de trames physiques **MAC**.

Une fois la communication WiFi est opérationnelle nous pouvons choisir un des multiples protocoles pour transmettre nos données : **UDP**, **TCP**, **HTTP**/**TCP**, ..Par exemple la carte peut fonctionner comme client ou serveur WEB. Dans une application fonctionnant comme un client WEB nous allons contacter un serveur externe type **`ThingSpeak`** pour y envoyer et stocker nos données.
Bien sur nous aurons besoin de bibliothèques relatives à ces protocoles.

2.1.1 Un programme de test – scrutation du réseau WiFi

Pour commencer notre étude de la communication WiFi essayons un exemple permettant de scruter notre environnement dans la recherche de point d'accès visibles par notre modem.

```
#include "WiFi.h"
#include <U8x8lib.h>
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15, 4, 16);

void setup()
{
  WiFi.mode(WIFI_STA);
  WiFi.disconnect();
  delay(100);
  u8x8.begin(); u8x8.setFont(u8x8_font_chroma48medium8_r);
}

void loop()
{
  int n = WiFi.scanNetworks();
  if(n==0) u8x8.drawString(0,0, "Searching !");
  else {
   u8x8.drawString(0, 0, "Networks found: "); u8x8.clear();
   for(int i=0;i<n;++i) {
      char currentSSID[16];
      memset(currentSSID,0x00,16);
      WiFi.SSID(i).toCharArray(currentSSID,16);
      u8x8.drawString(0,i+1,currentSSID);
     }
  }
}
```

2.2 Mode WiFi – STA, client WEB et serveur `ThingSpeak`

Une connexion WiFi permet de créer une application avec un client WEB. Ce client WEB peut envoyer les requêtes HTTP vers un serveur WEB.
Dans nos laboratoires nous allons utiliser un serveur IoT externe type **ThingSpeak**. **ThingSpeak** est un serveur «*open source*» qui peut être installé sur un PC
ThingSpeak est une plate-forme open source «Internet des objets» pour stocker et récupérer des données d'objets en utilisant HTTP sur Internet. Avec **ThingSpeak**, vous pouvez créer des applications de journalisation de capteurs, des applications de suivi de localisation et un réseau social d'objets avec des mises à jour de statut.

2.2.1 Envoi des données sur ThingSpeak

Nous pouvons donc envoyer les requêtes HTTP (par exemple en format **`GET`** et les données attachées à **`URL`**) sur le serveur **`ThingSpeak.com`** de **Matlab** ou **`ThingSpeak.fr`** de **SmartComputerLab**.
La préparation de ces requêtes peut être laborieuse, heureusement il existe une bibliothèque **`ThingSpeak.h`** qui permet de simplifier cette tâche.
Il faut donc installer la bibliothèque **`ThingSpeak.h`**. On peut la trouver à l'adresse suivante :

`https://github.com/mathworks/thingspeak-arduino`

Attention
Si vous voulez travailler avec un serveur **ThingSpeak** autre que **`ThingSpeak.com`** le fichier **`ThingSpeak.h`** doit être modifié pour y mettre l'**adresse IP** et le **numéro de port** correspondant.

Exemple de modification du fichier **`ThingSpeak.h`** pour **`ThingSpeak.fr`** :

```
//#define THINGSPEAK_URL "api.thingspeak.com"
//#define THINGSPEAK_PORT_NUMBER 80
#define THINGSPEAK_URL "90.49.254.215"
#define THINGSPEAK_PORT_NUMBER 443
```

Après l'inscription sur le serveur **ThingSpeak** vous créez un **channel** composé de max 8 **fields**. Ensuite vous pouvez envoyer et lire vos données dans ce **fields** à condition de fournir la clé d'écriture associée au **channel**.
Une écriture/envoi de plusieurs valeurs en virgule flottante est effectué comme suit :

```
ThingSpeak.setField(1, sensor[0]);  // préparation du field1
ThingSpeak.setField(2, sensor[1]);  // préparation du field2
```

puis

```
ThingSpeak.writeFields(myChannelNumber[1], myWriteAPIKey[1]);  // envoi
```

Voici le début d'un tel programme (déclarations et **`setup()`**):

```
#include <WiFi.h>
#include "ThingSpeak.h"
WiFiClient  client;
char ssid[] = "PhoneAP";             //  your network SSID (name)
char pass[] = "smartcomputerlab";                  // your network passw
unsigned long myChannelNumber = 1;  // your channel number
const char * myWriteAPIKey = "HEU64K3PGNWAAAA4";  // your API write key
IPAddress ip;

void setup() {
 Serial.begin(9600);
WiFi.disconnect(true);  // effacer de l'EEPROM WiFi credentials
delay(1000);
WiFi.begin(ssid, pass);
delay(1000);
```

```
    while (WiFi.status() != WL_CONNECTED) {
    delay(500); Serial.print(".");
    }
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");Serial.println(ip);
  Serial.println("WiFi setup ok");
  delay(1000);
  ThingSpeak.begin(client); // connexion (TCP) du client au serveur
  delay(1000);
  Serial.println("ThingSpeak begin");
}
```

Pour simplifier l'exemple dans la fonction de la boucle principale **`loop()`** nous allons envoyer les données d'un compteur.

```
int tout=20000;  // en millisecondes
float luminosity=100.0, temperature=10.0;

void loop()
{
ThingSpeak.setField(1, luminosity);  // préparation du field1
ThingSpeak.setField(2, temperature);  // préparation du field1
    while (WiFi.status() != WL_CONNECTED) {
    delay(500);Serial.print(".");
    }
ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
luminosity++;
temperature++;
delay(tout);
}
```

Attention : pour le serveur **`ThingSpeak.com`** la valeur minimale de **`tout`** est 20 secondes.

2.2.2 Réception des données à partir de ThingSpeak

Les données envoyées vers **ThingSpeak** sont stockées dans sa base **Mysql**. Chaque donnée est marquée par la date de réception et sa valeur en format **JSON**.
Le programme suivant montre comment récupérer les données à partir de ThinSpeak. Dans cet exemple nous lisons simplement les dernières valeurs enregistrées dans 2 champs du canal.
Pour pouvoir récevoir les données privées, il faut associer la clé de lecture (**`myReadAPIKey`**) à chaque lecture des données.

```
#include <WiFi.h>
#include "ThingSpeak.h"
WiFiClient  client;
const char* ssid = "Livebox-08B0";
const char* pass = "G79ji6dtEptVTPWmZP";
unsigned long myChannelNumber =  1088701;  // IoTDevKit1
const char * myWriteAPIKey = "VKRHS9PGD9K1HJ6R";
const char * myReadAPIKey ="OP2YX22NPFCO763Z";
IPAddress ip;

void setup() {
 Serial.begin(9600);
WiFi.disconnect(true);  // effacer de l'EEPROM WiFi credentials
delay(1000);
WiFi.begin(ssid, pass);
delay(1000);
    while (WiFi.status() != WL_CONNECTED) {
```

```
          delay(500); Serial.print(".");
    }
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");
  Serial.println(ip);
  Serial.println("WiFi setup ok");
  delay(1000);
  ThingSpeak.begin(client); // connexion (TCP) du client au serveur
  delay(1000);
  Serial.println("ThingSpeak begin");
}

int tout=20000;  // en millisecondes
float luminosity=0, temperature=0;

void loop()
{
while (WiFi.status() != WL_CONNECTED) {
    delay(500);Serial.print(".");
    }
luminosity = ThingSpeak.readIntField(1088701,1,myReadAPIKey );  // key NULLL for
public channel view
delay(tout);
temperature = ThingSpeak.readIntField(1088701,2,myReadAPIKey );
delay(tout);
Serial.println(luminosity);Serial.println(temperature);
}
```

2.2.3 Accès WiFi – votre Phone (`PhoneAP`) ou un routeur WiFi-4G

La connexion WiFi nécessite la disponibilité d'un point d'accès. Un tel point d'accès peut être créé par votre smarthphone avec une application Hot-Spot mobile ou un routeur dédié type B315 de Huawei

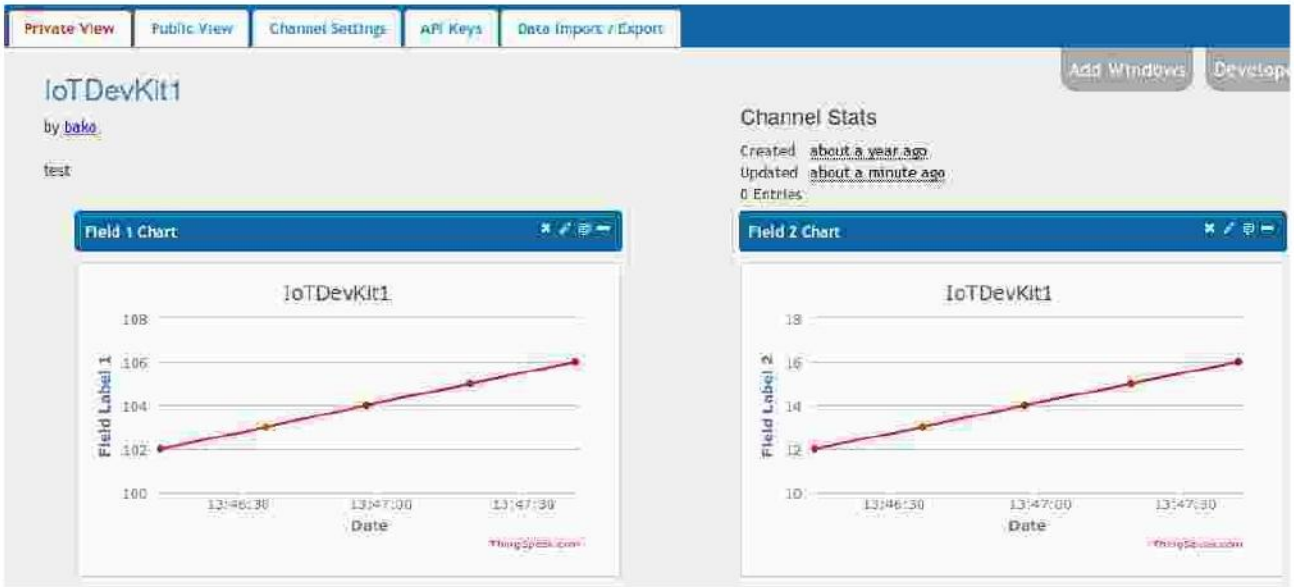


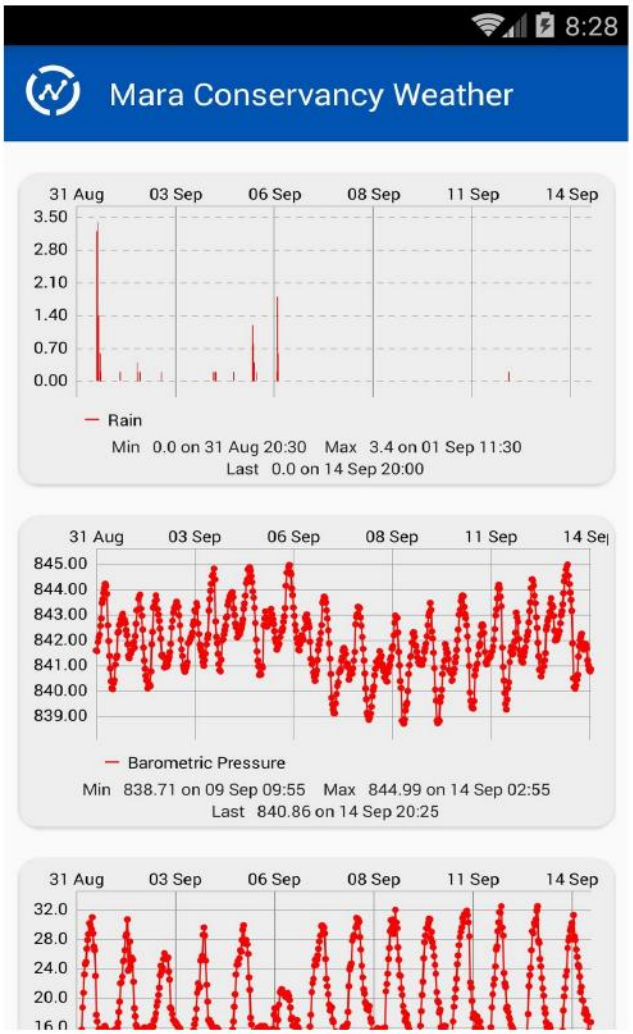
Figure 2.1. **ThingSpeak** : l'affichage des données envoyées sur le serveur type **ThingSpeak**

2.2.4 `ThingView` - `ThingSpeak` viewer (Android)

ThingView vous permet de visualiser vos chaînes ThingSpeak de manière simple, entrez simplement l'ID de chaîne et vous êtes prêt à partir.

Pour les **canaux publiques**, l'application respectera vos paramètres Windows: couleur, échelle de temps, type de graphique et nombre de résultats. La version actuelle prend en charge les graphiques en courbes et en colonnes, les graphiques splines sont affichés sous forme de graphiques en courbes.

Pour les **canaux privés**, les données seront affichées en utilisant les paramètres par défaut, car il n'y a aucun moyen de lire les paramètres de fenêtres privées avec la clé api uniquement.



A faire

1. Créer un canal **`ThingSpeak`** puis récupérer les paramètres du canal : *number* et *write key*.
2. Intégrer l'utilisation des capteurs de Température/Humidité et de la Luminosité.
3. Envoyer les données cycliquement (par exemple toutes les 30 seconde) vers le serveur **`ThingSpeak`**.
4. Recevoir les données cycliquement (par exemple toutes les 30 seconde) à partir du serveur **`ThingSpeak`**. et les afficher sur l'écran **OLED**.

Important : Ce travail peut être effectué à **distance** entre **2 membres du binôme** qui connaissent le même canal ThingSpeak avec ses clés d'écriture et de lecture.

2.3 Mode WiFi – STA, avec une connexion à `eduroam`

La carte ESP32 permet également d'établir un lien de communication WiFi avec un point d'accès type **eduroam** (**enterprise AP**).
Ci-dessous le code permettant d'effectuer une telle connexion. Comme dans l'exemple précédent le programme communique avec un serveur **ThingSpeak**.

2.3.1 Envoi des données sur ThingSpeak avec point d'accès eduroam

```
#include "ThingSpeak.h"
#include "esp_wpa2.h"
#include <WiFi.h>
#define EAP_IDENTITY "bakowski-p@univ-nantes.fr" //eduroam login -->
identity@youruniversity.domain
#define EAP_PASSWORD "..." //your password
String line; //variable for response
const char* ssid = "eduroam"; // Eduroam SSID

WiFiClient  client;
unsigned long myChannelNumber =  1234;  // no de votre canal ThingSpeak
const char * myWriteAPIKey = "9KLC1D8XJUGKHRO6"; // code d'ecriture

float temperature=0.0,humidity=0.0,tem,hum;

void setup() {
  Serial.begin(9600);
  delay(10);Serial.println();
  Serial.print("Connecting to network: ");
  Serial.println(ssid);
  WiFi.disconnect(true);  //disconnect form wifi to set new wifi connection
  WiFi.mode(WIFI_STA);
  esp_wifi_sta_wpa2_ent_set_identity((uint8_t *)EAP_IDENTITY,
strlen(EAP_IDENTITY)); //provide identity
  esp_wifi_sta_wpa2_ent_set_username((uint8_t *)EAP_IDENTITY,
strlen(EAP_IDENTITY)); //provide username
  esp_wifi_sta_wpa2_ent_set_password((uint8_t *)EAP_PASSWORD,
strlen(EAP_PASSWORD)); //provide password
  esp_wpa2_config_t config = WPA2_CONFIG_INIT_DEFAULT();
  esp_wifi_sta_wpa2_ent_enable(&config);

  WiFi.begin(ssid); //connect to Eduroam function
  WiFi.setHostname("ESP32Name"); //set Hostname for your device - not neccesary
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address set: ");
  Serial.println(WiFi.localIP()); //print LAN IP
  ThingSpeak.begin(client);
}

void loop() {
  delay(5000);
  if (WiFi.status() != WL_CONNECTED) { //if we lost connection, retry
    WiFi.begin(ssid);
    delay(500);
}
  Serial.println("Connecting to ThingSpeak.fr or ThingSpeak.com");
  // modify the ThingSpeak.h file to provide correct URL/IP/port
  WiFiClient client;
```

```
  delay(1000);
  ThingSpeak.begin(client);
  delay(1000);
  Serial.println("Fields update");
  ThingSpeak.setField(1, temperature);
  ThingSpeak.setField(2, humidity);
  ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
  delay(6000);
  temperature+=0.1;
  humidity+=0.2;
  tem =ThingSpeak.readFloatField(myChannelNumber,1);
  Serial.print("Last temperature:");
  Serial.println(tem);
  delay(6000);
  hum =ThingSpeak.readFloatField(myChannelNumber,2);
  Serial.print("Last humidity:");
  Serial.println(hum);
  delay(6000);
}
```

2.3.2 Réception des données partir du ThingSpeak avec point d'accès eduroam

Le programme ci-dessous permet de récupérer les données envoyées par notre programme précédent (émetteur) Les données sont lues sur le serveur **`ThingSpeak.fr`**, dans le canal pré-configuré puis elle sont affichées sur l'écran **OLED**. Le temps réel est lu par le biais du protocole **NTP**, dont les données sont portées par le protocole **UDP**.
Malheureusement le port numéro **123** de **UDP** n'est pas (normalement) ouvert sur un lien **`eduroam`**. L'affichage donnée simplement la durée de fonctionnement du récepteur.

```
#include <WiFi.h>
#include "esp_wpa2.h"
#include "ThingSpeak.h"
#include <Wire.h>
#include "OLED.h"
#include <NTPClient.h>
#include <WiFiUdp.h>
OLED display(21, 22);  // SDA-D2,SCL-D1 ESP32 - 22-SCL, 21-SDA
const char* ssid = "eduroam"; // your ssid
#define EAP_ID "bakowski-p"
#define EAP_USERNAME "bakowski-p"
#define EAP_PASSWORD ".." //removed for obvious reasons
int status = WL_IDLE_STATUS;
WiFiClient  client;
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP);
unsigned long myChannelNumber =  2;  // numero du canal
const char * myWriteAPIKey = "WT4X574BN1FGDZFW "; // clé de lecture
float temp=0.0,humi=0.0,inter=0.0;

void mydispfloat(int clr,int line,float field)
{
  char disp[32];
  snprintf(disp,20,"%f",field);
  if(clr) display.clear();
  display.print(disp,line);
}
```

```
void mydispstring(int clr,int line,char *field)
{
  char disp[32];
  snprintf(disp,20,"%s",field);
  if(clr) display.clear();  display.print(disp,line); }
void setup() {
  Serial.begin(9600);delay(1000);
WiFi.disconnect(true);
esp_wifi_sta_wpa2_ent_set_identity((uint8_t *)EAP_ID, strlen(EAP_ID));
esp_wifi_sta_wpa2_ent_set_username((uint8_t *)EAP_USERNAME,
strlen(EAP_USERNAME));
esp_wifi_sta_wpa2_ent_set_password((uint8_t *)EAP_PASSWORD,
strlen(EAP_PASSWORD));
esp_wpa2_config_t config = WPA2_CONFIG_INIT_DEFAULT();
esp_wifi_sta_wpa2_ent_enable(&config);
WiFi.begin(ssid);
while (WiFi.status() != WL_CONNECTED) {
delay(500);
Serial.print(".");
}
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");Serial.println(ip);
  Serial.println("setup ok");
  ThingSpeak.begin(client);
  display.begin();
  timeClient.update();
}

String date;
char disp[24];
void loop() {
    delay(1000);
    Serial.println("in the loop");timeClient.update();
    date=timeClient.getFormattedTime();date.toCharArray(disp,24);
    mydispstring(1,0,"Channel 2");
    temp =ThingSpeak.readFloatField(myChannelNumber,1);
    Serial.print("Last temperature:"); Serial.println(temp);
    mydispstring(0,1,"Last temperature");mydispfloat(0,2,temp);
    delay(5000);
    humi =ThingSpeak.readFloatField(myChannelNumber,2);
    Serial.print("Last humidity:");
    Serial.println(humi); mydispstring(0,3,"Last humidity");
    mydispfloat(0,4,humi); delay(5000);
    inter =ThingSpeak.readFloatField(myChannelNumber,3);
    Serial.print("Last interrupt:");
    Serial.println(humi); mydispstring(0,5,"Last interrupt");
    mydispfloat(0,6,inter);mydispstring(0,7,disp);
    delay(15000);
}
```

A faire (si vous êtes à l'école ou à l'université – accès eduroam)

1. Comme dans le sujets précédant fonctionnant avec un point d'accès maison ou smartphone créer un canal **ThingSpeak** puis récupérer les paramètres du canal : **number** et **write key**.

2. Intégrer l'utilisation des capteurs de Température/Humidité et de la Luminosité.
Envoyer ces données cycliquement (par exemple toutes les 30 seconde) vers le serveur **ThingSpeak**.

3. Récupérer les données cycliquement (par exemple toutes les 30 seconde) à partr du serveur **ThingSpeak**.
Afficher les résultats sur l'écran OLED.

Laboratoire 3 – MQTT broker et clients

3.1 Protocole MQTT

MQTT (**Message Queuing Telemetry Transport**) est un protocole de messagerie type **publication** - **abonnement** qui fonctionne sur la pile de protocoles TCP/IP. La première version du protocole a été développée par Andy Stanford-Clark d'IBM et Arlen Nipper de Cirrus Link en 1999.

Les messages **MQTT** peuvent être aussi petits que **2 octets**, alors que HTTP (cas de messages envoyés vers serveur ThingSpeak dans le Lab précédant) nécessite des en-têtes qui contiennent de nombreuses informations peu importants pour les équipements IoT. Si vous avez plusieurs appareils en attente d'une demande avec HTTP, vous devrez envoyer une action **`POST`** à chaque client.

Avec MQTT, lorsqu'un serveur (**broker**) reçoit des informations d'un client, il les distribuera automatiquement à chacun des clients intéressés (abonnés).

3.1.1 Les bases

Les termes (opérations) utilisés dans un réseau **MQTT**:

- **Broker** (**courtier**) - Le c est le serveur qui distribue les informations aux clients intéressés connectés au serveur.
- **Client** - Le périphérique qui se connecte au broker pour envoyer ou recevoir des informations.
- **Topic (sujet)**- Le nom sur lequel porte le message. Les clients publient, s'abonnent ou font les deux à un topic.
- **Publish (publier)** - Clients qui envoient des informations au broker pour les distribuer aux clients intéressés en fonction du nom du topic.
- **Subscribe (s'abonner**) - Les clients indiquent au broker le ou les topics qui les intéressent. Lorsqu'un client s'abonne à un topic, tout message publié au broker est distribué aux abonnés de ce topic. Les clients peuvent également se désabonner pour ne plus recevoir de messages du broker sur ce topic.

3.1.2 Comment fonctionne MQTT

Comme mentionné précédemment, MQTT est un protocole de messagerie de publication-abonnement.
Les clients se connecteront au réseau. Ils peuvent s'abonner ou publier sur un topic.
Lorsqu'un client publie sur un topic, les données sont envoyées au broker, qui les ensuite distribue à tous les clients abonnés à ce topic.

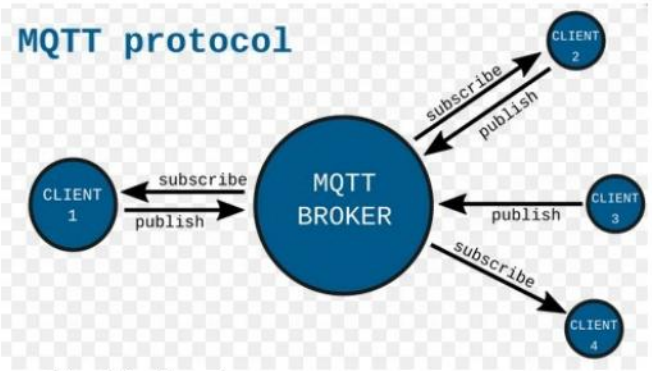


Figure 3.1 Protocole MQTT avec abonnement / publication de messages

Les topics sont organisés dans une structure de type répertoire. Un topic peut être «**LivingRoom**» ou «**LivingRoom/Light**» si vous avez plusieurs clients dans ce **topic parent**. Le client abonné écoutera les messages entrants du topic souscrit et réagira à ce qui a été publié sur ce sujet, par exemple «activé» ou «désactivé».
Les clients peuvent s'abonner à un topic et publier sur un autre également. Si le client s'abonne à «**LivingRoom/Light**», il peut également souhaiter publier sur un autre topic tel que «**LivingRoom/Light/Humidity**» afin que d'autres clients puissent surveiller l'état de cette lumière.

Maintenant que nous comprenons la théorie du fonctionnement de MQTT, construisons un exemple rapide et facile avec notre **IoT DevKit** basé sur ESP32.

3.4.3 Configuration du Broker

Il existe une grande collection de brokers MQTT disponibles qui peuvent fonctionner à partir d'un **serveur distant**, ou **localement**, à la fois **sur votre ordinateur** de bureau ainsi que **sur un SBC** comme Nano-Pi (DevKit IoT sur NanoPi - Linux).
Dans l'exemple utilisé dans ce Lab, nous allons utiliser un Nano Pi connecté à notre réseau local exécutant un broker gratuit et open-source appelé **`mosquitto`**.

Pour les utilisateurs de Windows voici le lien d'installation de **Mosquitto v 1.5.8**
`http://www.steves-internet-guide.com/install-mosquitto-broker/`

Pour installer **`mosquitto`** (Ubuntu):
`sudo apt-get install mosquitto -y`

Une fois installé, nous voudrons nous assurer que notre broker fonctionne correctement en créant un client de test pour écouter un topic. Nous le ferons en installant les clients **`mosquitto`**:

`sudo apt-get install mosquitto mosquitto-clients -y`
Une fois les clients installés, nous nous abonnerons au topic **`test_topic`** en entrant:

`mosquitto_sub -t "test_topic"`

Nous disons à **`mosquitto`** que nous aimerions nous abonner à un topic en entrant **`mosquitto_sub`**, et que nous aimerions nous abonner à un topic désigné par **`-t`** avec le nom **`test_topic`**.
Désormais, chaque fois que nous publions sur **`test_topic`**, le message envoyé apparaîtra dans cette fenêtre.

Parce que notre terminal écoute les messages de notre broker, nous devrons ouvrir une deuxième fenêtre de terminal pour publier les messages. Une fois ouvert, nous publierons sur **`test_topic`** avec la commande suivante:

`mosquitto_pub -t "test_topic" -m "HELLO WORLD!"`
Tout comme avant, nous utilisons **`-t`** pour désigner le topic, mais cette fois, nous ajoutons un message à publier sur le topic en utilisant **`mosquitto_pub`** et en utilisant **`-m`** pour désigner le message que nous aimerions publier.

Une fois que nous avons appuyé sur Entrée, nous devrions voir notre message apparaître sur la fenêtre du terminal de l'abonné, comme indiqué ci-dessous. Vous pouvez remplacer ce texte par n'importe quelle chaîne que vous souhaitez après **`-m`** pour envoyer votre message à tous les clients abonnés à **`test_topic`**.

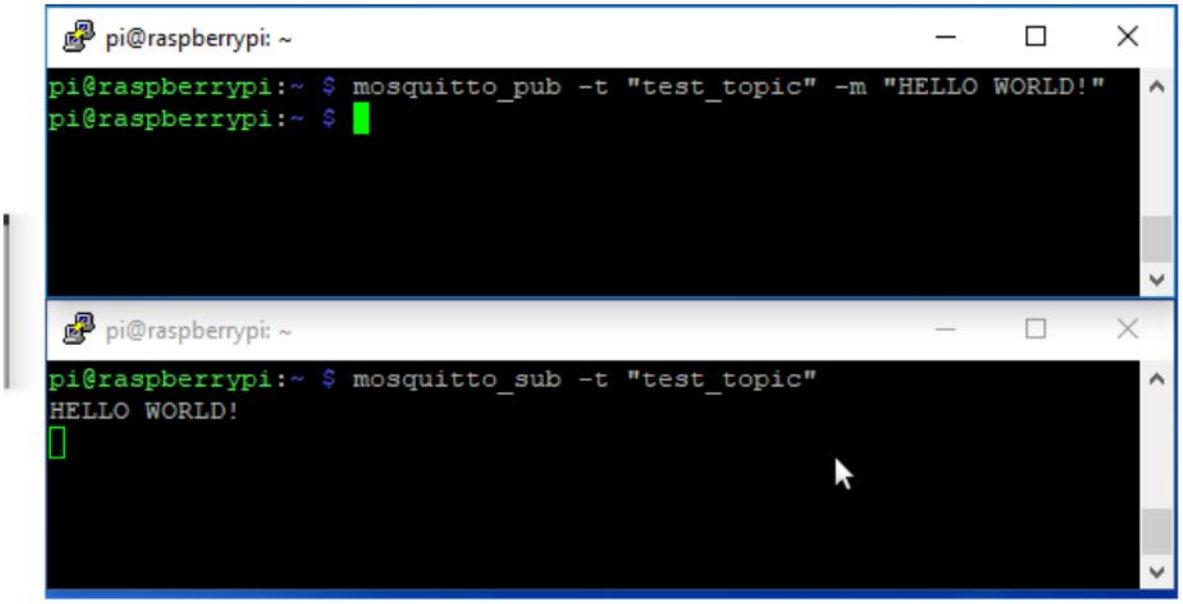


Figure 3.2 Commandes **`mosquitto`** avec abonnement / publication de messages

3.2 Serveur externe de MQTT - `test.mosquitto.org`

Instead of your local **`mosquitto`** server you can as well use the external **`mosquitto`** server available at **`test.mosquitto.org`** .
Au lieu de votre serveur **mosquitto** local, vous pouvez également utiliser le serveur mosquitto externe disponible sur **`test.mosquitto.org`**.
Ce serveur Mosquitto écoute sur les ports suivants:

- **1883**: MQTT, non crypté
- **8883**: MQTT, crypté
- **8884**: MQTT, crypté, certificat client requis
- **8080**: MQTT sur **WebSockets**, non crypté
- **8081**: MQTT sur **WebSockets**, crypté

Les ports cryptés prennent en charge **`TLS v1.3, v1.2`** ou **`v1.1`** avec des certificats **`x509`** et nécessitent le support client pour se connecter. Dans tous les cas, vous devez utiliser le fichier d'autorité de certification (**`mosquitto.org.crt`** (format **PEM**) ou **`mosquitto.org.der`** (format **DER**)) pour vérifier la connexion au serveur.
Le port **8884** oblige les clients à fournir un certificat pour authentifier leur connexion. Il est désormais possible de générer votre propre certificat.
Vous êtes libre d'utiliser **`test.mosquitto.org`** pour n'importe quelle application, mais veuillez ne pas en abuser ni vous y fier pour quoi que ce soit d'important. Vous devez également créer votre client pour faire face au redémarrage du broker.
Si vous avez installé les clients **`mosquitto`**, essayez:

```
mosquitto_sub -h test.mosquitto.org -t "smtr-lab3" -v
```

3.2.1 Configuration des clients

Maintenant que nous savons que notre broker est opérationnel, il est temps d'ajouter nos clients. Nous allons créer un client qui s'abonnera à **`room/light`** et répondra au message en allumant ou éteignant la LED (broche 25 sur notre carte ESP32). Dans l'exemple suivant, nous utilisons la bibliothèque **`PubSubClient.h`** , elle doit être installée précédemment dans votre IDE Arduino.

3.2.1.1 Le client abonné en attente de messages MQTT

L'exemple suivant permet d'envoyer (publier) les messages et les recevoir sur la fonction **`callback()`**.
Nous utilisons ici notre broker local installé sur notre PC ou SBC.
Attention
Si vous copiez cet exemple directement il faut que vous **personnalisiez le ID – le nom de votre `device`**.

```
#include <WiFi.h>
#include <PubSubClient.h>
#define ssid "PhoneAP"
#define pass "smartcomputerlab"
const byte LIGHT_PIN = 25;            // Pin to control the light with
const char *ID = "My_Device...";     // Name of our device, must be unique
const char *TOPIC = "smtr-lab3";     // Topic to subcribe to
const char *STATE_TOPIC = "smtr-lab3/state";
                          // Topic to publish the light state to
IPAddress broker(192,168,43,159); // IP address of your MQTT broker
WiFiClient wclient;
PubSubClient client(wclient); // Setup MQTT client
// Handle incomming messages from the broker
void callback(char* topic, byte* payload, unsigned int length) {
  String response;
  for (int i = 0; i < length; i++) {
    response += (char)payload[i];
  }
  Serial.print("Message arrived [");
  Serial.print(topic); Serial.print("] ");
  Serial.println(response);
```

```
  if(response == "on")  // Turn the light on
  {
    digitalWrite(LIGHT_PIN, HIGH);
  }
  else if(response == "off")  // Turn the light off
  {
    digitalWrite(LIGHT_PIN, LOW);
  }
}
// Connect to WiFi network
void setup_wifi() {
  Serial.print("\nConnecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, pass); // Connect to network
  while (WiFi.status() != WL_CONNECTED) { // Wait for connection
    delay(500); Serial.print(".");
  }
  Serial.println();
  Serial.println("WiFi connected");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
}
// Reconnect to client
void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if(client.connect(ID)) {
      client.subscribe(TOPIC);
      Serial.println("connected");
      Serial.print("Subcribed to: ");
      Serial.println(TOPIC);
      Serial.println('\n');
    } else {
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(9600); // Start serial communication at 115200 baud
  pinMode(LIGHT_PIN, OUTPUT); // Configure LIGHT_PIN as an output
  delay(100);
  setup_wifi(); // Connect to network
  client.setServer(broker, 1883);
  client.setCallback(callback);// Initialize the callback routine
}

void loop() {
  if (!client.connected())  // Reconnect if connection is lost
  {
    reconnect();
  }
  client.loop();
}
```

Une fois que le deuxième ESP32 se connecte au réseau, il s'abonnera automatiquement au topic **room/light**, la LED intégrée connectée à la broche **GPIO 25** devrait répondre et s'allumer et s'éteindre.

L'affichage de publication sur **mosquitto**:

```
pi@nano-pi:~$ mosquitto_pub m "on" smtr-lab3
pi@nano-pi:~$ mosquitto_pub -m "off" -t smtr-lab3
```

L'affichage sur le terminal Arduino IDE :

```
Connecting to PhoneAP
..
WiFi connected
IP address: 192.168.43.185
Attempting MQTT connection...connected
Subcribed to: room/light

Message arrived [room/light] on
Message arrived [room/light] off
```

3.2.1.2 Le client publiant des messages MQTT

```
#include <WiFi.h>
#include <PubSubClient.h>
#define ssid "PhoneAP"
#define pass "smartcomputerlab"
const char* mqttServer = "5.196.95.208";  // test.mosquitto.org

WiFiClient espClient;
PubSubClient client(espClient);

void setup() {
  Serial.begin(9600);
  WiFi.begin(ssid, pass);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("Connected to the WiFi network");
  client.setServer(mqttServer,1883);
  while (!client.connected()) {
    Serial.println("Connecting to MQTT...");
    if (client.connect("ESP32Client")) {
      Serial.println("connected");
    } else {
      Serial.print("failed with state ");
      Serial.print(client.state());
      delay(2000);
    }
  }
  client.publish("esp/test", "Hello from ESP32");
  Serial.println("published");
}

void loop() {
  client.publish("esp/test", "Hello from ESP32");
  Serial.println("published");
  delay(5000);
}
```

L'affichage sur le terminal Arduino IDE :

```
.........................Connected to the WiFi network
Connecting to MQTT...
connected
published
published
published
```

L'affichage sur **mosquitto** subscriber:

```
bako@bako:~$ mosquitto_sub -h test.mosquitto.org -t "esp/test"
Hello from ESP32
Hello from ESP32
Hello from ESP32
Hello from ESP32
```

3.2.2 Utilisation de la bibliothèque MQTT

Les exemples suivants montrent comment exploiter la bibliothèque Arduino **MQTT.h** qui semble plus simple à utiliser que la bibliothèque **PubSubClient.h**.

3.2.2.1 Publication et réception de messages de test

L'exemple suivant publie et reçoit simples messages (« **coucou** »)MQTT sur le topic **/hello** . Nous utilisons le broker gratuit :

```
#include <WiFi.h>
#include <MQTT.h>
const char ssid[] = "PhoneAP";
const char pass[] = "smartcomputerlab";
WiFiClient net;
MQTTClient client;
unsigned long lastMillis = 0;

void connect() {
  Serial.print("checking wifi...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print("."); delay(1000);
  }
  Serial.print("\nconnecting...");
  while (!client.connect("IoT.DevKit")) {
    Serial.print("."); delay(1000);
  }
  Serial.println("\nconnected!");
  client.subscribe("/hello");
}

void messageReceived(String &topic, String &payload) {
  Serial.println("incoming: " + topic + " - " + payload);
}

void setup() {
  Serial.begin(9600);
  WiFi.begin(ssid,pass);
  client.begin("5.196.95.208", net);
  client.onMessage(messageReceived);
  connect();
}
```

```
void loop() {
  client.loop();
  delay(10);  // <- fixes some issues with WiFi stability
  if (!client.connected()) { connect(); }
  if (millis() - lastMillis > 1000) {   // publish a message every second.
    lastMillis = millis();
    client.publish("/hello", "coucou");
  }
}
```

3.2.3 Publication et réception des valeurs des capteurs

Le deuxième exemple avec la bibliothèque **`MQTT.h`** inclut l'utilisation d'un simple capteur **DHT22**. La carte envoie (et reçoit) les messages MQTT et affiche le contenu sur l'écran **OLED**.
Nous utilisons, comme dans l'exemple précédant le broker :**`test.mosquitto.org - "5.196.95.208"`**

```
#include <WiFi.h>
#include <MQTT.h>
#include "DHT.h"
#define DHTTYPE DHT22
DHT dht(17, DHTTYPE);
#include <U8x8lib.h>
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15,4,16); // data, clock,
int dhtPin = 17;
const char ssid[] = "PhoneAP";
const char pass[] = "smartcomputerlab";
WiFiClient net;
MQTTClient client;
unsigned long lastMillis = 0;

void connect() {
  Serial.print("checking wifi...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print("."); delay(1000);
  }
  Serial.print("\nconnecting...");
  u8x8.drawString(0,5,"connecting to");
  u8x8.drawString(0,6,"5.196.95.208");
  while (!client.connect("IoT.DevKit")) {
    Serial.print("."); delay(1000);
  }
  Serial.println("\nconnected !");
  u8x8.drawString(0,7,"connected !");
  client.subscribe("/dht22");
}

void messageReceived(String &topic, String &payload) {
  char dbuff[32], btemp[16]; int len=32;
  Serial.println("incoming: " + topic + " - " + payload);
  u8x8.clear();u8x8.drawString(0,0,"5.196.95.208");
  u8x8.drawString(0,1,"IoT.DevKit");u8x8.drawString(0,2,"MQTT topic");
  topic.toCharArray(dbuff,len);u8x8.drawString(0,3,dbuff);
  u8x8.drawString(0,4,"Payload");payload.toCharArray(dbuff,len);
  memset(btemp,0x00,16);
  memcpy(btemp,dbuff,10);
  u8x8.drawString(0,5,btemp);u8x8.drawString(0,6,dbuff+12);
}
byte mac[6];
char dbuff[32];
```

```
void setup() {
  Serial.begin(9600);
  pinMode(dhtPin,INPUT);
  dht.begin();delay(100);
  WiFi.begin(ssid, pass);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);Serial.print(".");
  };
  IPAddress ip = WiFi.localIP();
  Serial.println(ip);
  WiFi.macAddress(mac);
  Serial.print("MAC: ");Serial.print(mac[5],HEX);Serial.print(":");
  Serial.print(mac[4],HEX);Serial.print(":");
  Serial.print(mac[3],HEX);Serial.print(":");
  Serial.print(mac[2],HEX);Serial.print(":");Serial.print(mac[1],HEX);
  Serial.print(":"); Serial.println(mac[0],HEX);
  u8x8.begin();  // initialize OLED
  u8x8.setFont(u8x8_font_chroma48medium8_r);
  u8x8.clear();
  u8x8.drawString(0,0,"IP address");
  sprintf(dbuff,"%d.%d.%d.%d",ip[0],ip[1],ip[2],ip[3]);
  u8x8.drawString(0,1,dbuff);
  u8x8.drawString(0,2,"MAC address");
  sprintf(dbuff,"%2.2x%2.2x%2.2x%2.2x%2.2x%2.2x",
       mac[0],mac[1],mac[2],mac[3],mac[4],mac[5]);
  u8x8.drawString(0,3,dbuff);
  client.begin("5.196.95.208", net);
  client.onMessage(messageReceived);
  connect();
}

float temperature,humidity;

void loop() {
  char cbuf[32];
  client.loop();
  delay(10);  // <- fixes some issues with WiFi stability
  humidity = dht.readHumidity();          // Read humidity (percent)
  temperature = dht.readTemperature();     // Read temperature as Celsius
  if (!client.connected()) {
    connect(); }
  sprintf(cbuf,"temp:%2.2f, humi:%2.2f",temperature,humidity);
  if (millis() - lastMillis > 20000) {
    lastMillis = millis();
    client.publish("/dht22", cbuf);
  }
}
```

Ce qui suit est le résultat affiché sur le terminal Arduino IDE:

```
192.168.43.185
MAC: D8:3F:A9:BF:71:3C
checking wifi...
connecting...
connected !
incoming: /dht22 - temp:26.10, humi:55.40
incoming: /dht22 - temp:26.10, humi:54.30
incoming: /dht22 - temp:26.10, humi:54.10
incoming: /dht22 - temp:26.10, humi:54.00
```

3.3 Utilisation d'une connexion sécurisée avec `SSL` et `WiFiClientSecure`

Les messages MQTT envoyés au numéro de port **1883** ne sont pas cryptés. Afin d'utiliser une connexion sécurisée, nous avons besoin du protocole **SLL** sur le port numéro **8883**.
Ce qui suit est un exemple simple d'envoi de messages texte sécurisés au broker MQTT-**broker.shift.io**.

```
#include <WiFiClientSecure.h>
#include <MQTT.h>
const char ssid[] = "PhoneAP";
const char pass[] = "smartcomputerlab";
WiFiClientSecure net;
MQTTClient client;
unsigned long lastMillis1 = 0, lastMillis2=0;

void connect() {
  Serial.print("checking wifi...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");  delay(1000);
  }
  Serial.print("\nWiFi connected...");
  Serial.print("\nconnecting to MQTT server...");
  while (!client.connect("esp32sec", "try", "try")) {
    Serial.print("."); delay(1000);
  }
  Serial.println("\nconnected!");
  client.subscribe("/esp32hello");
}

void messageReceived(String &topic, String &payload)
{
  Serial.println(topic + " - " + payload);
}

void setup() {
  Serial.begin(9600);
  WiFi.begin(ssid, pass);
  // MQTT brokers usually use port 8883 for secure connections.
  client.begin("broker.shiftr.io", 8883, net);
  client.onMessage(messageReceived);
  // Name:  broker.shiftr.io - Address: 34.76.6.166
  connect();
}

void loop() {
  client.loop();
  delay(10);  // <- fixes some issues with WiFi stability
  if (!client.connected()) { connect();}
  // publish a message roughly every second.
  if (millis() - lastMillis1 > 6000) {
    lastMillis1 = millis();
    client.publish("/esp32hello", "coucou");
  }
    if (millis() - lastMillis2 > 7000) {
    lastMillis2 = millis();
    client.publish("/esp32hello", "bonjour");
  }
}
```

L'affichage sur le terminal Arduino IDE:

```
checking wifi.....................................checking wifi....
WiFi connected...
connecting to MQTT server....
connected!
/esp32hello - coucou
/esp32hello - bonjour
/esp32hello - coucou
/esp32hello - bonjour
/esp32hello - coucou
```

3.4 Utilisation de `WiFiMQTTManager`

`WiFiMQTTManager` est une bibliothèque ESP qui étend les bibliothèques client **`WiFiManager`** et **`PubSub`** et ajoute la possibilité d'enregistrer les paramètres de connexion **SSID**, mot de **passe WiFi** et **MQTT** afin qu'ils n'aient pas à être codés en dur dans vos sketch.
Le code de base lance le **`WiFiManager`** qui bascule éventuellement en mode point d'accès pour afficher une page web (**`192.168.4.1`**) qui permet de récupérer les identificateurs WiFi et ceux du serveur **MQTT**.
Dans le fichier associé : secrets.h il faut mettre votre mot de passe (default est **`CHANGEME`**).

3.4.1 Le code

```
#include "secrets.h"
#include <WiFiMQTTManager.h>
// Button that will put device into Access Point mode to allow for re-entering
WiFi and MQTT settings
#define RESET_BUTTON 0
WiFiMQTTManager wmm(RESET_BUTTON, AP_PASSWORD); // defined in secrets.h file

void setup() {
  Serial.begin(9600);
  Serial.println("WiFiMQTTManager Basic Example");
  // set debug to true to get verbose logging
  // wm.wm.setDebugOutput(true);
  // most likely need to format FS but only on first use
  // wmm.formatFS = true;
  // optional - define the function that will subscribe to topics if needed
  wmm.subscribeTo = subscribeTo;
  // required - allow WiFiMQTTManager to do it's setup
  wmm.setup(__SKETCH_NAME__);
  // optional - define a callback to handle incoming messages from MQTT
  wmm.client->setCallback(subscriptionCallback);
}
void loop() {
  // required - allow WiFiMQTTManager to check for new MQTT messages,
  // check for reset button push, and reconnect to MQTT if necessary
  wmm.loop();
  // publishing to MQTT a sensor reading once every 10 seconds
  long now = millis();
  if (now - wmm.lastMsg > 10000) {
    wmm.lastMsg = now;
    float temperature = 23.54; // read sensor here
    Serial.print("Temperature: ");
    Serial.println(temperature);
    char topic[100];
    //snprintf(topic, sizeof(topic), "%s%s%s", "sensor/", wmm.deviceId,
"/temperature");
    snprintf(topic, sizeof(topic), "%s", "smtr-lab3");
```

```
    wmm.client->publish(topic, String(temperature).c_str(), true);
  }
}

void subscribeTo() {
  Serial.println("subscribing to some topics...");
  char topic[100];
  wmm.client->subscribe("smtr-lab3"); // subscribe to some topic(s)
}

void subscriptionCallback(char* topic, byte* message, unsigned int length) {
  Serial.print("Message arrived on topic: ");
  Serial.print(topic);Serial.print(". Message: ");
  String messageTemp;
  for (int i = 0; i < length; i++) {
    Serial.print((char)message[i]);
    messageTemp += (char)message[i];
  }
  // do something
}
```

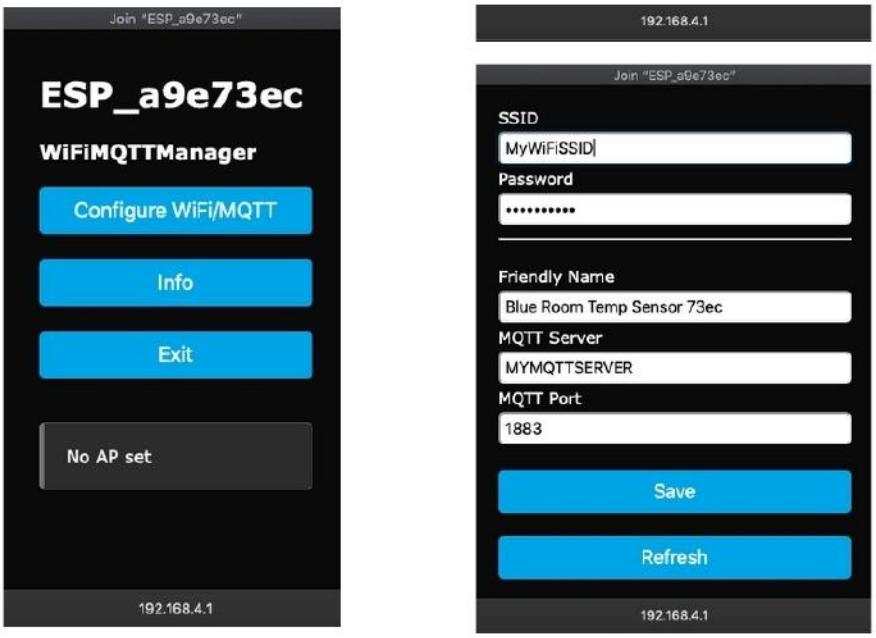


Figure 3.3 Pages WEB de **`WiFiMQTTManager`** pour la configuration du mode de passe et de l'adresse IP du serveur **MQTT**.

3.5 Envoi de messages MQTT au serveur `ThingSpeak`

Le serveur IoT **`ThingSpeak`** est également capable de recevoir les messages MQTT et d'afficher leur contenus sur les chronogrammes. L'exemple suivant montre l'utilisation de la bibliothèque **`PubSubClient.h`**

3.5.1 Le code

Dans la partie initiale du code, vous devez définir et initialiser les variables.
Assurez-vous de modifier les « **credentials** » WiFi , l'ID de canal **`ThingSpeak`** et les clés **API**.
Trouvez votre identifiant de chaîne en haut de la page principale de votre chaîne.

Chaque **champ** (*field*) du canal est considéré comme **topic**.
Pour écrire (publier) dans un topic, vous devez fournir une ligne comme:

```
String temperature_topic =
"channels/1226765/publish/fields/field1/U4TY3T09ZSMVV5SR"
```

où:

`1226765` est le numéro de canal et
`U4TY3T09ZSMVV5SR` est la clé API d'écriture

```
#include <WiFi.h>
#include <PubSubClient.h> // MQTT

const char* ssid = "PhoneAP";
const char* password = "smartcomputerlab";
const char* mqtt_server = "mqtt.thingspeak.com";
String temperature_topic =
"channels/1226765/publish/fields/field1/U4TY3T09ZSMVV5SR";
String humidity_topic =
"channels/1226765/publish/fields/field2/U4TY3T09ZSMVV5SR";

WiFiClient espClient;
PubSubClient client(espClient);
long lastMsg = 0;
char msg[50];
int value = 0;
byte temperature=23;
byte humidity=66;

void setup() {
  Serial.begin(9600);
  WiFi.begin(ssid, password);
  int wifiTryConnectCounter = 0;
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);Serial.print(".");
  }
  Serial.println("WiFi connected");
  //MQTT
  client.setServer(mqtt_server, 1883);
  // Create a random client ID
  String clientId = "ESPClient-";
  clientId += String(random(0xffff), HEX);
  // Attempt to connect
  if (client.connect(clientId.c_str())) {
   Serial.println("MQTT server connected");
  } else {
    ESP.deepSleep(sleepTimeS * 1000000);
  }
```

```
  //Sensor
  temperature=23; humidity=66;
  //Send
  sprintf(msg,"%ld", (int)temperature);
  //client.publish(temperature_topic.c_str(), msg);
  sprintf(msg,"%ld", (int)humidity);
  client.publish(humidity_topic.c_str(), msg);
  //Sleep and repeat
  client.loop(); // MQTT work
  ESP.deepSleep(20 * 1000000);  // sleeping 20 seconds
}

void loop() {
  // Will not run
}
```

3.5.2 Mode `deepSleep()` et les données dans SRAM et EPROM

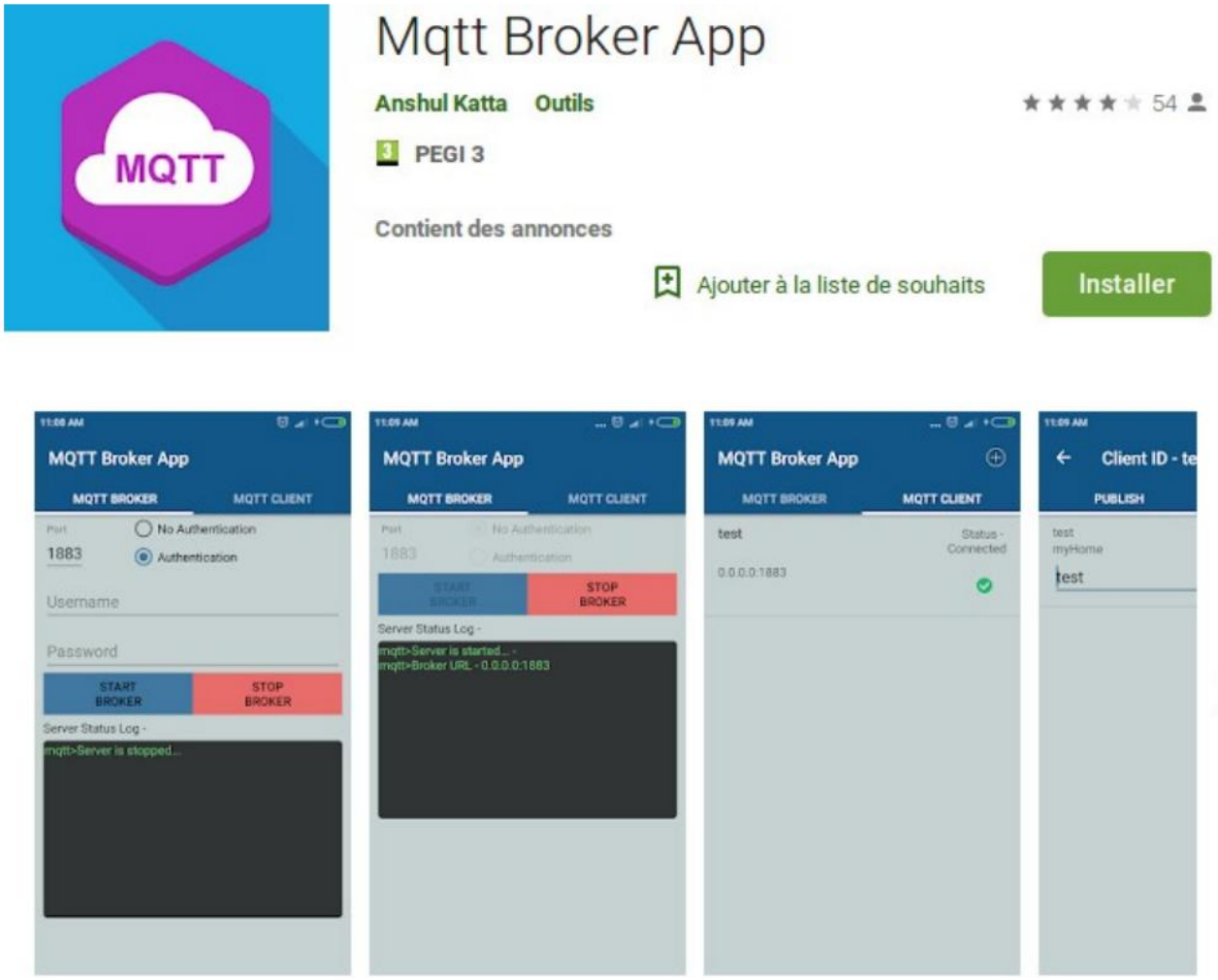
Les programme ci-dessus utilise la fonction **`ESP.deepSleep(20*1000000)`.**
Cette fonction arrête l'exécution de l'application en activant un signal de **time-out** après 20 secondes.
Pendant le **`deepSleep()`** le programme perd toutes les valeurs de données dans la mémoire **SRAM**.
La boucle **`loop()`** (tâche de fond) n'est jamais exécutée !
Pour garder les données du programme il faut les enregistrer dans la mémoire **EEPROM** juste avant le lancement de **`deepSleep()`** et les restaurer au début de **`setup()`.**

Ci dessous vous trouvez les éléments du code nécessaires pour réaliser la **sauvegarde** et la **restauration** des donnés par le biais de la **EEPROM**.

```
#include "EEPROM.h"
#define EEPROM_SIZE 128
uint8_t  mbuff[128];
// in setup()
  if(!EEPROM.begin(EEPROM_SIZE))
  {
    Serial.println("failed to initialise EEPROM");
  }
// to read
  for (i = 0; i < EEPROM_SIZE; i++)
  {
    mbuff[i]=EEPROM.read(i);
  }
// write and commit
for (i = 0; i < EEPROM_SIZE; i++)
      {
      EEPROM.write(i,rbuff[i]);
      }
    EEPROM.commit();
```

3.6 Android MQTT et Application Broker

L'application Android **Mqtt Broker** nous permet de fonctionner en mode broker et client.



Vous pouvez le tester avec les exemples présentés ci-dessus.

A faire:

1. Installez le broker et le client **`mosquitto`** sur votre ordinateur et testez le.

2. Utilisez votre DevKit pour implémenter l'exemple **3.2.1.1** fonctionnant avec votre broker (il faut que votre ordinateur et votre carte soient connectés au même réseau local).

3. Etudiez et testez sur votre DevKit l'exemple **3.2.1.2** avec un broker externe - **`test.mosquitto.org`**

4. Testez l'exemple **3.2.2.1** (bibliothèque **`MQTT.h`**)

5. Sujet pour deux DevKits d'un binôme communiquant par le biais du serveur **`test.mosquitto.org`**
   a. Testez 'exemple **3.2.3** , puis modifiez le capteurs en utilisant les capteurs et SHT21/BH1750 et trois topics TEMP/HUMI/LUMI
   b. Décomposez le programme de **3.2.3** en deux parties (**une par le membre du binôme**) **une publiant** seulement (exemple) :

```
client.publish("/TEMP", Tcbuf);
client.publish("/HUMI", Hcbuf);
client.publish("/LUMI", Lcbuf);
..
```

   et l'autre avec la fonction :

```
void messageReceived(String &topic, String &payload){ … }
```

   qui affiche le message reçu sur son écran **OLED**.

6. Mettez en ouvre le même exemple (**3.2.3**) avec un mode **MQTT sécurisé** avec **`WiFiClientSecure.h`** sur le serveur **`broker.shift.io`**

Sujets supplémentaires :

1. Testez le sujet 3.4 (le code 3.4.1)
2. Mettez en œuvre le mode deepSleep() avec la mémoire EEPROM pour publier dans les exécutions alternativement :

```
client.publish(temperature_topic.c_str(), msg);
```

et

```
client.publish(humidity_topic.c_str(), msg);
```

3. Essayez l'application Android **Mqtt Broker** en mode client et broker pour communiquer avec votre IoT DevKit.