Laboratoires IoT de base

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

Contenu

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 Thinger.io**. 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 (ESP32-LOLIN). 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 d'expérimenter avec les liens à longue distance (1Km). Il s'agit de la technologie (et modulation) LoRa. Le modem LoRa est intégré sur la carte centrale. Nous allons envoyer les données d'un capteur géré par un terminal sur un lien LoRa vers une passerelle **LoRa-WiFi**.
La passerelle retransmettra ces données vers un serveur **`ThingSpeak`**.

Le quatrième laboratoire permettra d'intégrer l'ensemble de liens WiFi et LoRa pour créer une application complète avec les terminaux, le *gateway* LoRa-WiFi et les serveurs **`ThingSpeak.`**

Après ces laboratoires préparatifs nous vous proposons **2 séries de laboratoires IoT** qui permettrons d’approfondir les connaissances et le savoir faire dans le domaine des Architectures IoT.

La première série de 6 laboratoires est dédié aux architectures IoT centrées sur la communication WiFi , Bluetooth, BLE et le modes de programmation possibles.

La deuxième série de 6 laboratoires introduit les architectures plus complexes avec l’intégration de LoRa, programmation faible consommation, programmation temps-réel et le développement des protocoles LoRa-WiFi pour les serveurs ThingSpeak et brokers MQTT.

Les multiples cartes d’extension avec les capteurs/actionneurs, modems, afficheurs permettrons le développement des applications sur mesure.

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.

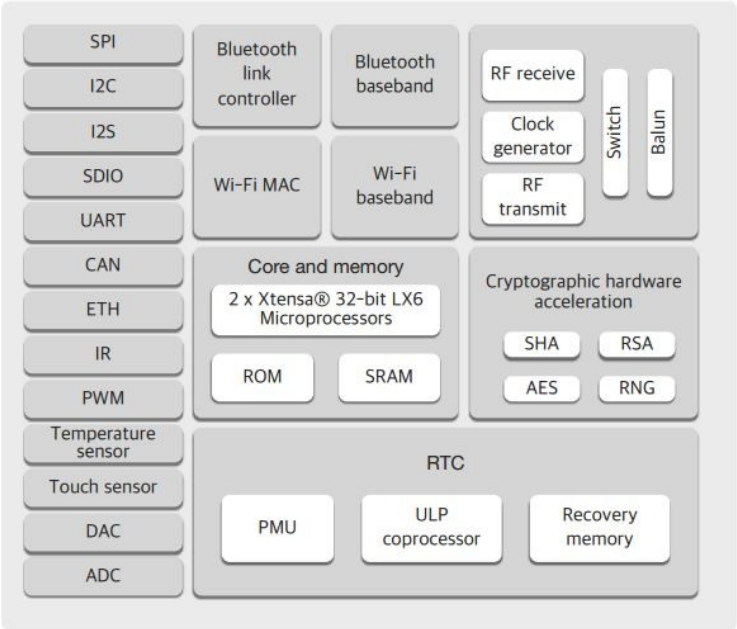


Figure 0.1 ESP32 SoC – architecture interne

0.2 Carte LOLIN D32

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-LOLIN D32** qui intègre une interface avec les batteries LiPo (3V7)

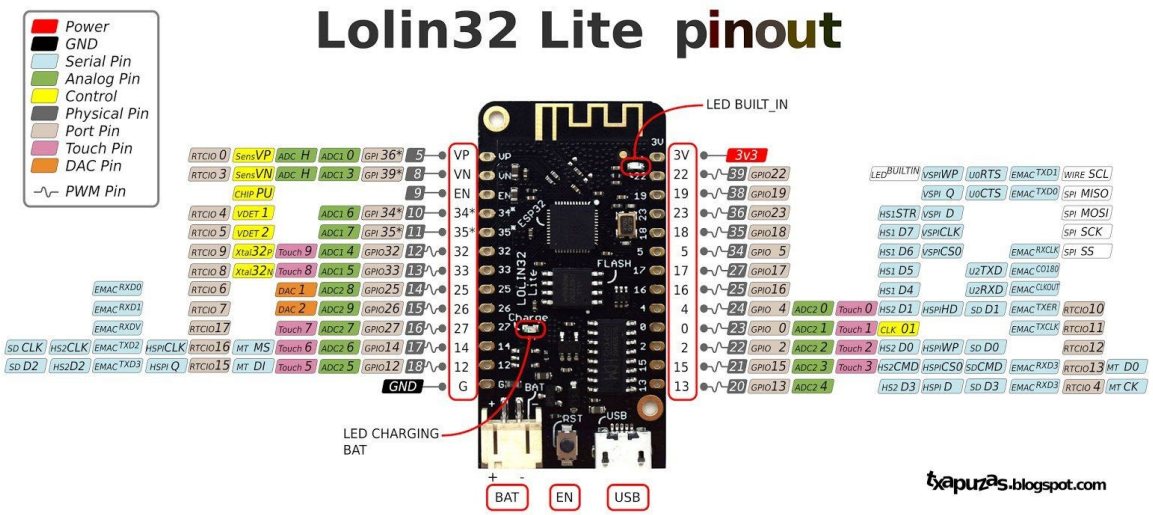


Figure 0.2 Carte MCU **Lolin32 Lite (LOLIN D32) et son *pinout***

Comme nous pouvons le voir sur la figure ci-dessus, la carte Heltec expose 2x13 broches 2x18. Ces broches portent les bus **I2C** (14,12), **SPI** (18,19,23), **UART** (16,17) plus les signaux de contrôle ((NSS,RST,INT,..)

0.3 IoT DevKit une plate-forme de développement IoT

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

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

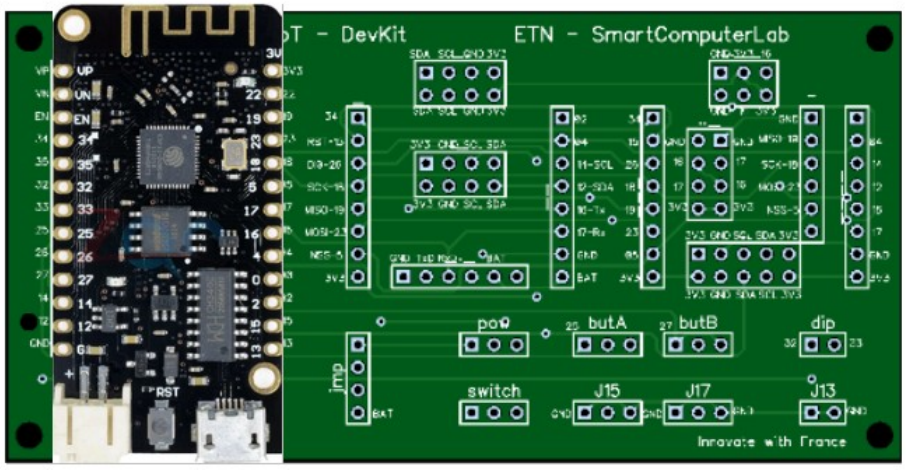


Figure 0.3 Carte de base pour **Lolin32 Lite (LOLIN D32)** et ses interfaces intégrées

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

Ci dessous quelques exemples de cartes d'extension.

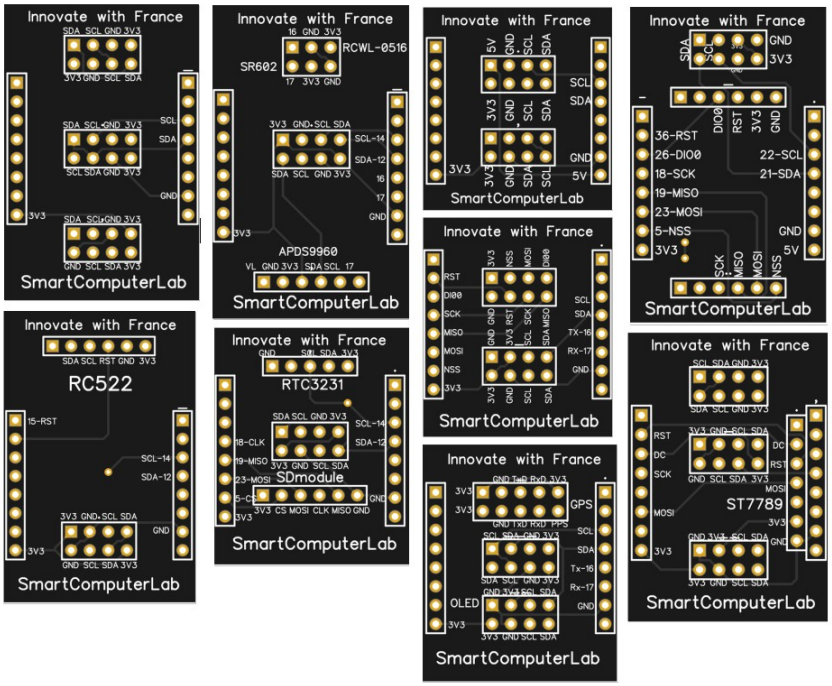


Figure 0.4 Cartes d'extension pour les différents composants IoT: capteurs, afficheurs, modems, ..

0.4 L'installation de 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 20.04 LTS, l'environnement Arduino IDE, les outils de compilation/chargement pour les cartes ESP32 et les bibliothèque de contrôle pour les capteurs, actionneurs (par exemple **relais**), et les modems de communication.

Pour commencer mettez le système à jour par :
`sudo apt-upgrade` et **`sudo apt update`**

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 :

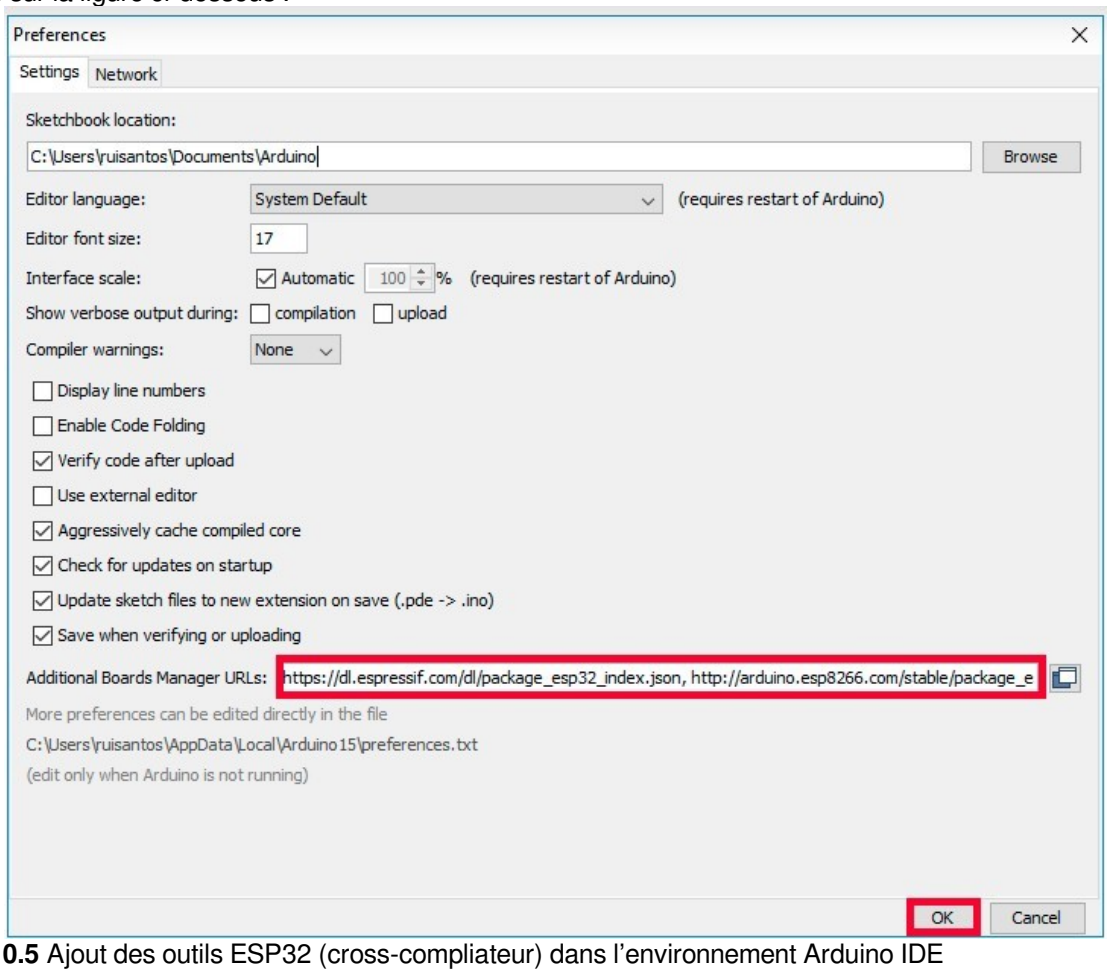


Figure 0.5 Ajout des outils ESP32 (cross-compliateur) dans l'environnement Arduino IDE

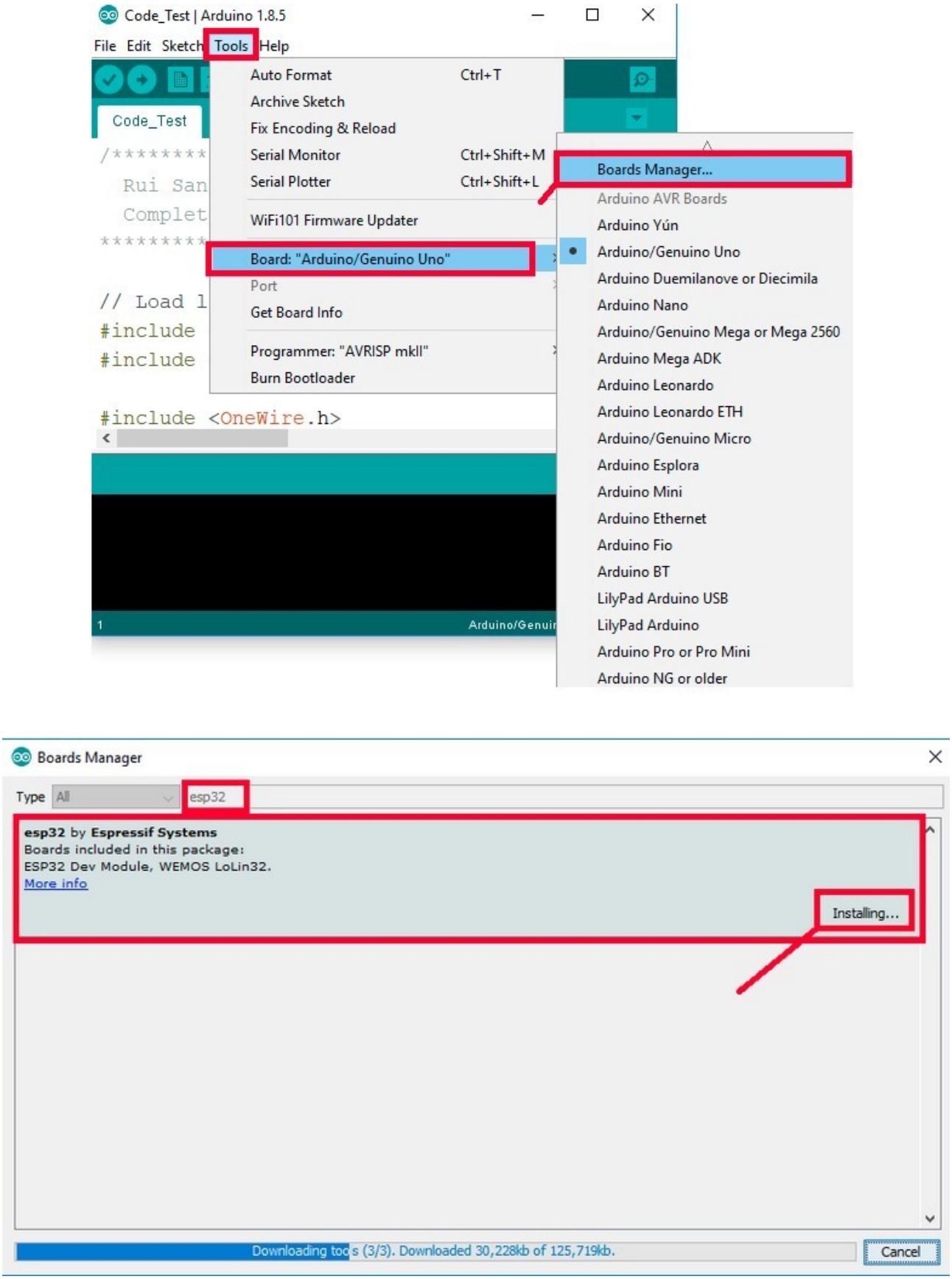


Figure 0.6. L'intégration des cartes **ESP32** dans l'environnement IDE Arduino

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

La compilation doit être effectuée sur la carte **LOLIN D32.** Elle doit être sélectionnée dans le menu **`Tools→Board.`** Pour le chargement (upload) du code sur la carte utilisez la vitesse de **46 0800** .

Avant la compilation il faut installer les bibliothèques nécessaires pour piloter différents dispositifs.

Par exemple l'image ci-dessous montre comment installer la bibliothèque **`SSD1306Wire`** qui pilote les écrans type **OLED**.

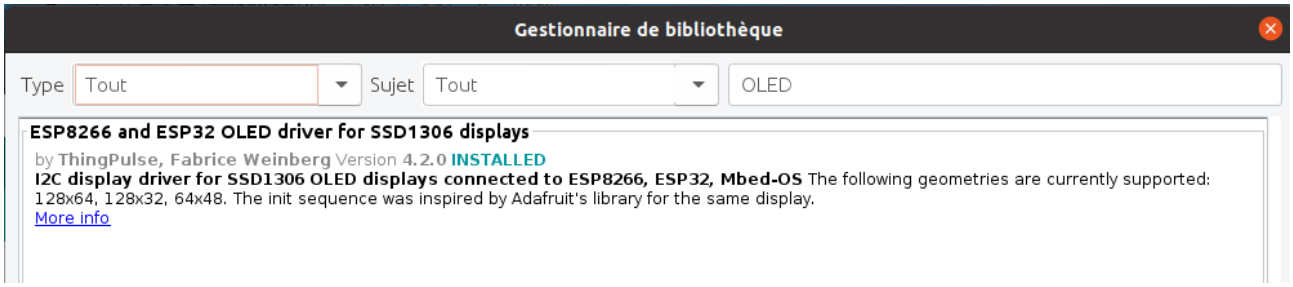


Figure 0.7. Intégration d'une bibliothèque Arduino (**`SSD1306Wire.h`**)

Laboratoire 1

1.1 Premier exemple – l'affichage des données

Dans cet exercice nous allons simplement afficher un titre et 2 valeurs numériques sur l'écran OLED ajouté à notre ESP32.

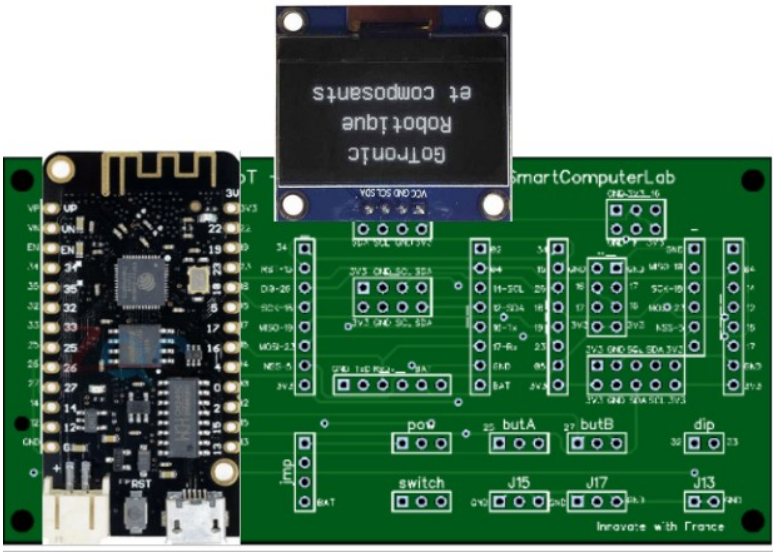
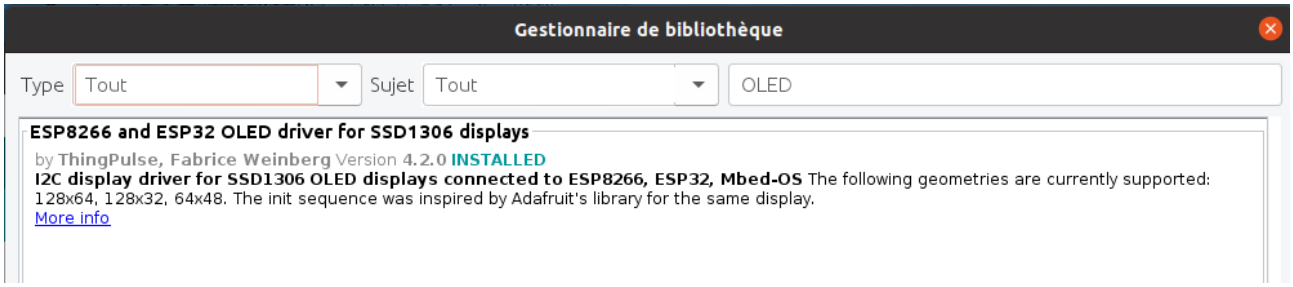


Figure 1.1 Ecran **OLED** ajouté à la carte ESP32

Vous devez installer la bibliothèque **`OLED`** compatible avec ESP32. Cela peut être trouvée dans le gestionnaire de bibliothèque IDE Arduino.



Le code

```
#include <Wire.h>
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c, 12, 14);

void display_SSD1306(float d1,float d2,float d3,float d4)
{
  char buff[64];
  display.init();
  //display.flipScreenVertically();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_16);
  display.drawString(0, 0, "ETN – IoT DevKit");
```

```
  display.setFont(ArialMT_Plain_10);
  sprintf(buff,"d1: %2.2f, d2: %2.2f\nd3: %2.2f, d4: %2.2f",d1,d2,d3,d4);
  display.drawString(0, 22, buff);
  display.drawString(20, 52, "SmartComputerLab");
  display.display();
}

void setup() {
  Serial.begin(9600);
  Wire.begin(12,14);
  Serial.println("Wire started");
}

float v1=0.0,v2=0.0,v3=0.0,v4=0.0;

void loop()
{
  Serial.println("in the loop");
  display_SSD1306(v1,v2,v3,v4);
  v1=v1+0.1;v2=v2+0.2;v3=v3+0.3;v4=v4+0.4;
  delay(2000);
}
```

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 ajouté sur la carte ESP32. Le capteur **SHT21** doit également être connecté sur le bus **I2C**.

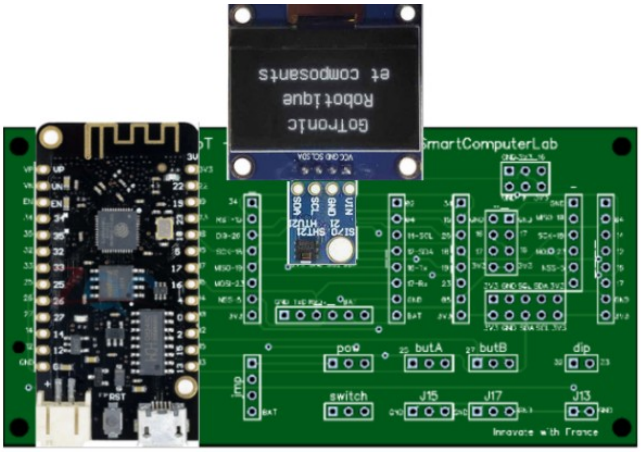


Figure 1.5. **IoT DevKit** avec un capteur de température/humidité **SHT21** et l'écran **OLED.**

Code :

Avant de commencer le codage il faut télécharger et installer la bibliothèque **`SHT21.h`** compatible avec notre carte. Elle se trouve dans le **github**: **e-radionicacom/SHT21-Arduino-Library**

```
#include <Wire.h>
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c, 12, 14);
#include "SHT21.h"
SHT21 SHT21;

float stab[4]= {0.0,0.0,0.0,0.0};

void get_SHT21()
{
    SHT21.begin();
    delay(1000);
    stab[0]=SHT21.getTemperature();
    delay(100);
    stab[1]=SHT21.getHumidity();
    Serial.printf("T:%2.2f, H:%2.2f\n",stab[0],stab[1]);
}

void display_SSD1306(float d1,float d2,float d3,float d4)
{
  char buff[64];
  display.init();
  //display.flipScreenVertically();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_16);
  display.drawString(0, 0, "ETN - IoT DevKit");
  display.setFont(ArialMT_Plain_10);
  sprintf(buff,"T: %2.2f, H: %2.2f\nd3: %2.2f, d4: %2.2f",d1,d2,d3,d4);
  display.drawString(0, 22, buff);
  display.drawString(20, 52, "SmartComputerLab");
```

```
  display.display();
}

void setup() {
  Serial.begin(9600);
  Wire.begin(12,14);
  Serial.println();
}

void loop()
{
  Serial.println("in the loop");
// à completer
  delay(2000);
}
```

A faire:

1. Compléter, compiler, et charger ce programme.
2. Tester le programme suivant (**`I2CScan`**) pour vérifier que les deux dispositifs (l'écran **SSD1306** et le capteur **SHT21**) sont visibles avec leurs adresses correspondantes (**`0x3C`** et **`0x40`**).

```
#include <Wire.h>

void I2Cscan()
{
  byte address;  int nDevices;  delay(200);
  Serial.println("Scanning...");
  nDevices = 0;
  for(address = 1; address < 127; address++ )
    {
    Wire.beginTransmission(address);
    if(! Wire.endTransmission())  // active address found
      {
      Serial.print("I2C device found at address 0x");
      if (address<16) Serial.print("0");
      Serial.print(address,HEX);  Serial.println("  !");
      nDevices++;
      }
    }
  if (nDevices == 0)
    Serial.println("No I2C devices found\n");
  else
    Serial.println("done\n");
}

void setup()
{
  Serial.begin(9600);
  Wire.begin(12,14,400000); // SDA, SCL on ESP32, 400 kHz rate
  delay(1000); Serial.println();Serial.println();
  I2Cscan(); delay(1000);
}

void loop(){  }
```

1.2.2 Capture de la luminosité par BH1750

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

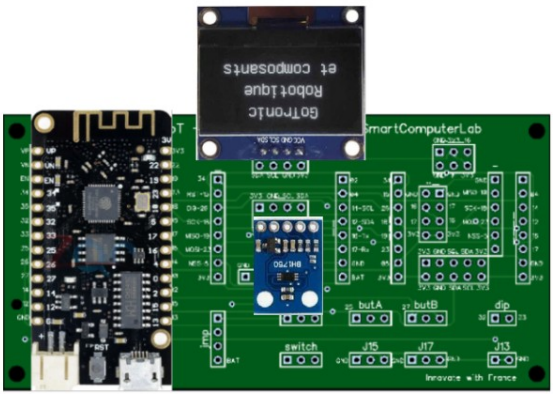
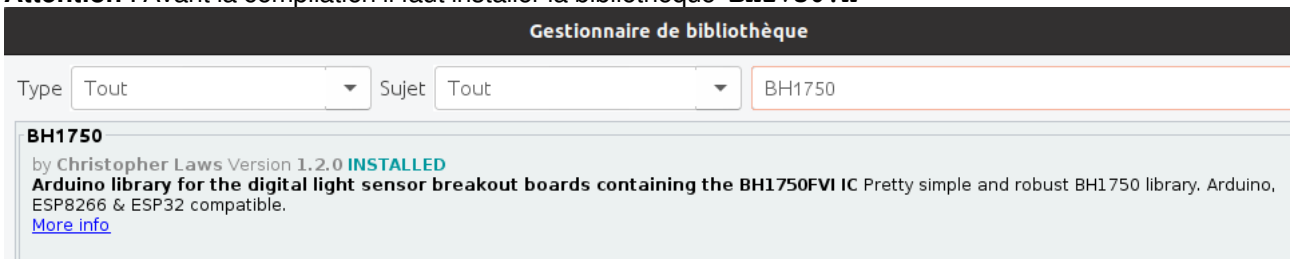


Figure 1.3. **IoT DevKit** avec le capteur Luminosité **BH1750**

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




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

void setup(){
  Wire.begin(12,14);
  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);
}
```

A faire :

1. Tester le programme
2. Ajouter l'affichage sur l'écran OLED.

1.2.3 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 1.4. Capteur de luminosité **MAX44009**

Code Arduino :

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

void setup()
{
Wire.begin(12,14);
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();
// Request 2 bytes of data
Wire.requestFrom(Addr, 2);
// 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);
}
```

A faire :

3. Tester le programme
4. Ajouter l'affichage sur l'écran OLED.

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**.

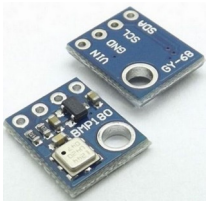


Figure 1.6. **C**apteur de pression/température **BMP180**

La valeur standard de la pression atmosphérique est :

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

Code Arduino :


```
#include <Wire.h>
#include <Adafruit_BMP085.h>
Adafruit_BMP085 bmp;

void setup() {
  Serial.begin(9600);
  Wire.begin(12,14);
  bmp.begin();
}

void loop() {
    Serial.print("Temperature = ");
    Serial.print(bmp.readTemperature()); Serial.println(" *C");
    Serial.print("Pressure = "); Serial.print(bmp.readPressure());
    Serial.println(" Pa");
    // Calculate altitude assuming 'standard' barometric
    // pressure of 1013.25 millibar = 101325 Pascal
    Serial.print("Altitude = ");
    Serial.print(bmp.readAltitude());
    Serial.println(" meters");
    // you can get a more precise measurement of altitude
    // if you know the current sea level pressure which will
    // vary with weather and such. If it is 1015 millibars
    // that is equal to 101500 Pascals.
    Serial.print("Real altitude = ");
    Serial.print(bmp.readAltitude(101500));  Serial.println(" meters");
    Serial.println();
    delay(500);
}
```

A faire :

1. Complétez le programme ci-dessus afin d'afficher les données de la température et pression atmosphérique sur l'écran OLED

1.2.6 Capture de présence avec un capteur PIR SR602

Dans l'exemple suivant nous utilisons un capteur de type PIR - **SR602** qui permet de détecter la présence d'une personne en mouvement. Il s'agit d'un capteur simple qui signale l'événement par le basculement de son signal de sortie.

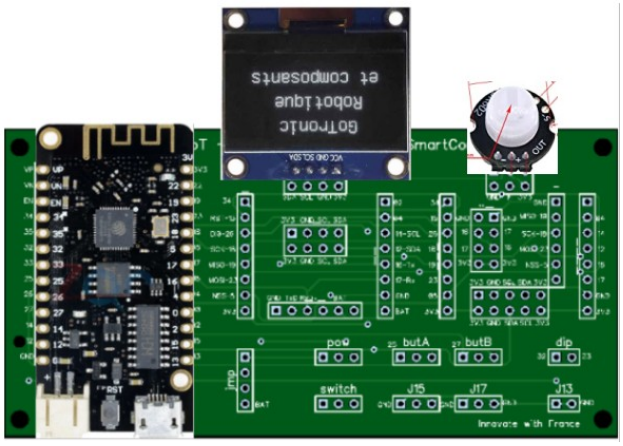


Figure 1.7. **C**apteur de mouvement – **SR602**

Dans le code ci-dessous le changement de la valeur du signal de sortie provoque une interruption captée sur la broche associée à la fonction **`pinChanged()`** qui s'exécute de la façon asynchrone par rapport au déroulement du programme.

```
#define PIR 17
bool MOTION_DETECTED = false;

void pinChanged()
{
  MOTION_DETECTED = true;
}
void setup()
{
  Serial.begin(9600);
  attachInterrupt(PIR, pinChanged, RISING);
}

int counter=0;

void loop()
{
  int i=0;
  if(MOTION_DETECTED)
  {
    Serial.println("Motion detected.");
    delay(1000);counter++;
    MOTION_DETECTED = false;
    Serial.println(counter);
  }
}
```

A faire :

1. Complétez le programme ci-dessus afin d'afficher la valeur du compteur de détection des mouvements sur l'écran OLED

Laboratoire 2 – communication en WiFi et serveur **ThingSpeak.com (.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 connecter notre carte à un point d'accès WiFi (qui peut être notre smartphone). Dans le mode **AP** nous créons un point d'accès sur la carte ESP32. Cet point d'accès peut être utilisé pour effectuer une configuration de la carte, par exemple en lui fournissant le nom du point d'accès (**ssid**) et le mot de passe pour ce point d'accès.
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 des 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 des 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 <Wire.h>
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c, 12, 14);

void display_SSD1306(char *text)
{
  char buff[256];
  strcpy(buff,text);
  display.init();
  display.flipScreenVertically();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_10);
  display.drawString(0, 0, "ETN – WiFi SCAN");
  display.drawString(0, 14, buff);
  display.display();
}

void setup()
{
  char buf[16];
  Serial.begin(9600);
  Wire.begin(12,14);
  Serial.println();
  WiFi.mode(WIFI_STA);
  WiFi.disconnect();
  delay(2000);
  Serial.println("Setup done");
}
```

```
void loop()
{
  int apmax=8, num;
  char buf[16];
  char tab[256];

  Serial.println("scan start");
  Serial.println("WiFi scan");
  // WiFi.scanNetworks will return the number of networks found
  int n = WiFi.scanNetworks();
  Serial.println("WiFi scan");
    if (n == 0) {
        Serial.println("no networks found");
    } else {
        //Serial.print(n);
        sprintf(buf,"WiFi AP#%d\n",n);
        Serial.println(buf);
        delay(2000);
        if(n<apmax) num=n; else num=apmax;
        //Serial.println(" networks found");
        for (int i = 0; i < num; ++i) {
            // Print SSID and RSSI for each network found
            Serial.print(i + 1);
            Serial.print(": ");
            WiFi.SSID(i).toCharArray(buf,16);
            Serial.print(buf);
            if(!i) strcpy(tab,buf);else strcat(tab,buf);strcat(tab,"\n");
            display_SSD1306(tab);
            //Serial.println(WiFi.SSID(i));
            Serial.print(" (");
            Serial.print(WiFi.RSSI(i));
            Serial.print(")");
            Serial.println((WiFi.encryptionType(i) == WIFI_AUTH_OPEN)?" ":"*");
            delay(10);
        }
    }
    Serial.println("");
    delay(5000);
}
```

A faire

Tester et analyser le programme ci-dessus.

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 ou sur une carte SBC.
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.**

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 pour **ThingSpeak.fr**
Attention: l'adresse IP actuelle peut être différente

```
//#define THINGSPEAK_URL "api.thingspeak.com"
//#define THINGSPEAK_URL "86.217.8.171:3000"
#define THINGSPEAK_URL "http://90.49.255.63:443"
//#define THINGSPEAK_IPADDRESS IPAddress(82,217,11,191)
#define THINGSPEAK_PORT_NUMBER 443
//#define THINGSPEAK_IPADDRESS IPAddress(184,106,153,149)
#define THINGSPEAK_IPADDRESS IPAddress(90,49,255,63)
//#define THINGSPEAK_PORT_NUMBER 80
```

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 effectuée 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 :

```
#include <WiFi.h>
#include "ThingSpeak.h"
char ssid[] = "PhoneAP";                 //  your network SSID (name)
char pass[] = "smartcomputerlab";        // your network passw
unsigned long myChannelNumber = 1;
const char * myWriteAPIKey="MEH7A0FHAMNWJE8P" ;
WiFiClient  client;
```

```
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=10000;  // 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`** utilisé gratuitement (max 1 an) la **valeur minimale** de **`tout`** est 20 secondes.

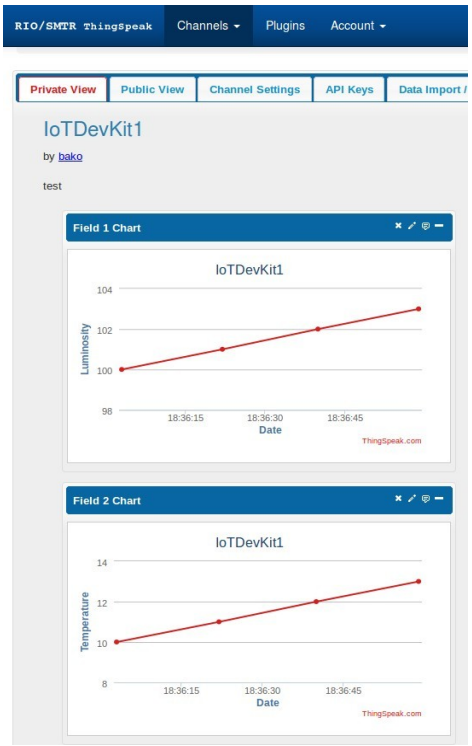


Figure 2.1. **ThingSpeak** : l'affichage des données envoyées sur le serveur type **ThingSpeak**

2.2.1 Accès WiFi – votre Phone 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 Smartphone avec une application Hot-Spot mobile ou un routeur dédié type B315 de Huwaei

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 ces données cycliquement (par exemple toutes les 30 seconde) vers le serveur **`ThingSpeak`**.

2.3 Mode WiFi – `STA` et `WiFiManager`

La carte ESP32 permet de fonctionner en mode Point d'Accès qui permet de déployer un simple serveur WEB avec une page de configuration des paramètres sur la carte ESP32.
Parmi ces paramètres il y a la possibilité de proposer (et enregistrer) le nom d'un point d'accès at son mot de passe.

`WiFiManager` est une fonction qui réalise ce type d'opérations.

Voici notre programme de test pour l'utilisation du **`WiFiManager`**.

```
#include <WiFiManager.h> // https://github.com/tzapu/WiFiManager

void setup() {
    WiFi.mode(WIFI_STA); // set mode to STA+AP
    Serial.begin(9600);
    WiFiManager wm;
    //reset settings - wipe credentials for testing
    wm.resetSettings();  // à tester , puis à commenter
    // Automatically connect using saved credentials,
    bool res;
    res = wm.autoConnect("ESP32AP",NULL); // no password
    if(!res) {
        Serial.println("Failed to connect");
        // ESP.restart();
    }
    else {
        //if you get here you have connected to the WiFi
        Serial.println("connected...yeey :)");
    }
}

void loop() { }
```

Sur votre terminal IDE **`WiFiManager`** affiche ses paramètres et l'état de fonctionnement.

Vous devrez vous connecter avec votre smartphone sur son pont d'accès (ici) ESP32AP et aller avec votre navigateur sur **`192.168.4.1`** pour accéder à la page d'ac**cue**il , puis choisir votre point d'accès local et fournir le mot de passe.

Le point d'accès et le mot de passe seront enregistrés sur votre carte.

N'oubliez pas der **commenter** la ligne :

```
wm.resetSettings();
```

pour pouvoir garder ces valeurs en mémoire **EEPROM** du ESP32.

Voici le déroulement d'affichage de **`WiFiManager`** dans le cas de connexion sur un nouveau point d'accès.

```
*WM: [3] WIFI station disconnect
*WM: [3] WiFi station enable
*WM: [2] Disabling STA
*WM: [2] Enabling AP
*WM: [1] StartAP with SSID:  ESP32AP
*WM: [2] AP has anonymous access!
*WM: [1] SoftAP Configuration
*WM: [1] --------------------
*WM: [1] ssid:              ESP32AP
*WM: [1] password:
*WM: [1] ssid_len:          7
*WM: [1] channel:           1
*WM: [1] authmode:
*WM: [1] ssid_hidden:
*WM: [1] max_connection:    4
*WM: [1] country:           CN
*WM: [1] beacon_interval:   100(ms)
*WM: [1] --------------------
*WM: [1] AP IP address: 192.168.4.1
*WM: [3] setupConfigPortal
*WM: [1] Starting Web Portal
*WM: [3] dns server started with ip:  192.168.4.1
*WM: [2] HTTP server started
*WM: [2] WiFi Scan completed in 5509 ms
*WM: [2] Config Portal Running, blocking, waiting for clients...

*WM: [3] WIFI station disconnect
*WM: [3] WiFi station enable
*WM: [2] Disabling STA
*WM: [2] Enabling AP
*WM: [1] StartAP with SSID:  ESP32AP
*WM: [2] AP has anonymous access!
*WM: [1] SoftAP Configuration
*WM: [1] --------------------
*WM: [1] ssid:              ESP32AP
*WM: [1] password:
*WM: [1] ssid_len:          7
*WM: [1] channel:           1
*WM: [1] authmode:
*WM: [1] ssid_hidden:
*WM: [1] max_connection:    4
*WM: [1] country:           CN
*WM: [1] beacon_interval:   100(ms)
*WM: [1] --------------------
*WM: [1] AP IP address: 192.168.4.1
*WM: [3] setupConfigPortal
*WM: [1] Starting Web Portal
*WM: [3] dns server started with ip:  192.168.4.1
*WM: [2] HTTP server started
*WM: [2] WiFi Scan completed in 5509 ms
*WM: [2] Config Portal Running, blocking, waiting for clients...

WM: [2] NUM CLIENTS: 0
*WM: [3] -> connectivitycheck.platform.hicloud.com
*WM: [2] <- Request redirected to captive portal
*WM: [2] NUM CLIENTS: 1
*WM: [2] <- HTTP Root
*WM: [3] -> 192.168.4.1
*WM: [3] lastconxresulttmp: WL_IDLE_STATUS
*WM: [3] lastconxresult: WL_DISCONNECTED
*WM: [2] WiFi Scan completed in 5106 ms
*WM: [2] <- HTTP Wifi
*WM: [2] Scan is cached 7 ms ago
*WM: [1] 18 networks found
*WM: [2] DUP AP: FreeWifi_secure
*WM: [2] DUP AP: SFR WiFi Mobile
```

```
*WM: [2] DUP AP: Livebox-C82A
*WM: [2] AP:  -52 DIRECT-G8M2070 Series
*WM: [2] AP:  -60 orange
*WM: [2] AP:  -62 Livebox-08B0
*WM: [2] AP:  -63 VAIO-MQ35AL
*WM: [2] AP:  -75 Livebox-08B0_Ext
*WM: [2] AP:  -84 FreeWifi_secure
*WM: [2] AP:  -85 Livebox-3D36
*WM: [2] AP:  -88 SFR WiFi Mobile
*WM: [2] AP:  -90 SFR WiFi FON
*WM: [2] AP:  -93 Livebox-F936
*WM: [2] AP:  -93 Livebox-C82A
*WM: [2] AP:  -94 SFR_E0B0
*WM: [2] AP:  -95 SFR_A0A0
*WM: [2] AP:  -95 freebox_ODTMTH
*WM: [2] AP:  -96 Bbox-EA1EB632
*WM: [3] lastconxresulttmp: WL_IDLE_STATUS
*WM: [3] lastconxresult: WL_DISCONNECTED
*WM: [3] Sent config page
*WM: [3] -> connectivitycheck.platform.hicloud.com
*WM: [2] <- Request redirected to captive portal
*WM: [2] <- HTTP WiFi save
*WM: [3] Method: POST
*WM: [3] Sent wifi save page
*WM: [2] processing save
*WM: [2] Connecting as wifi client...
*WM: [3] STA static IP:
*WM: [2] setSTAConfig static ip not set, skipping
*WM: [1] CONNECTED:
*WM: [1] Connecting to NEW AP: Livebox-08B0
*WM: [3] Using Password: G79ji6dtEptVTPWmZP
*WM: [3] WiFi station enable
*WM: [1] connectTimeout not set, ESP waitForConnectResult...
*WM: [2] Connection result: WL_CONNECTED
*WM: [3] lastconxresult: WL_CONNECTED
*WM: [1] Connect to new AP [SUCCESS]
*WM: [1] Got IP Address:
*WM: [1] 192.168.1.20
*WM: [2] disconnect configportal
dhcps: send_nak>>udp_sendto result 0
*WM: [2] restoring usermode STA
*WM: [2] wifi status: WL_CONNECTED
*WM: [2] wifi mode: STA
*WM: [1] config portal exiting
connected...yeey :)
*WM: [3] unloading
```

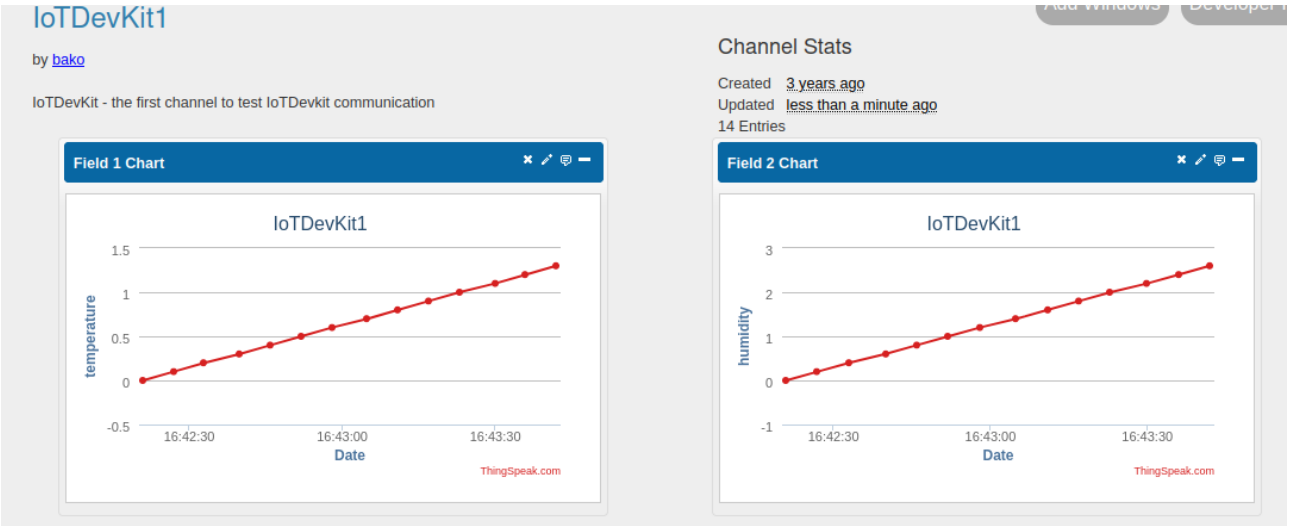
2.4 Envoi des données sur ThingSpeak avec WiFiManager

```
#include <WiFiManager.h>
#include "ThingSpeak.h"
unsigned long myChannelNumber = 1;
const char *myWriteAPIKey="HEU64K3PGNWG36C4" ;
WiFiClient  client;

void setup() {
    WiFi.mode(WIFI_STA);
    Serial.begin(9600);
    WiFiManager wm;
    // wm.resetSettings();
    bool res;
    res = wm.autoConnect("ESP32AP",NULL); // password protected ap
    if(!res) {
        Serial.println("Failed to connect");
        // ESP.restart();
    }
    else {
        //if you get here you have connected to the WiFi
        Serial.println("connected...yeey :)");
    }
  ThingSpeak.begin(client); // connexion (TCP) du client au serveur
  delay(1000);
  Serial.println("ThingSpeak begin");
}

float temperature=0.0,humidity=0.0;

void loop() {
  Serial.println("Fields update");
  ThingSpeak.setField(1, temperature);
  ThingSpeak.setField(2, humidity);
  ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
  delay(6000);
  temperature+=0.1;
  humidity+=0.2;
}
```



2.5 Réception des données de ThingSpeak

```
#include <WiFiManager.h>
#include "ThingSpeak.h"
unsigned long myChannelNumber = 1;
const char * myWriteAPIKey="HEU64K3PGNWG36C4" ;
WiFiClient  client;

void setup() {
    WiFi.mode(WIFI_STA);
    Serial.begin(9600);
    WiFiManager wm;
    //reset settings - wipe credentials for testing
    // wm.resetSettings();
    bool res;
    res = wm.autoConnect("ESP32AP",NULL); // password protected ap
    if(!res) {
        Serial.println("Failed to connect");
        // ESP.restart();
    }
    else {
        //if you get here you have connected to the WiFi
        Serial.println("connected...yeey :)");
    }
  ThingSpeak.begin(client); // connexion (TCP) du client au serveur
  delay(1000);
  Serial.println("ThingSpeak begin");
}

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

void loop() {
  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);
// if channel is private you need to add the reda key: AVG5MID7I5SPHIK8
// tem =ThingSpeak.readFloatField(myChannelNumber,1,"AVG5MID7I5SPHIK8");
  Serial.print("Last temperature:");
  Serial.println(tem);
  delay(6000);
  hum =ThingSpeak.readFloatField(myChannelNumber,2);
  Serial.print("Last humidity:");
  Serial.println(hum);
  delay(6000);
}
```

Voici l'affiche correspondant à l'exécution de ces programme :

```
*WM: [3] STA static IP:
*WM: [2] setSTAConfig static ip not set, skipping
*WM: [1] Connecting to SAVED AP: Livebox-08B0
*WM: [3] Using Password: G79ji6dtEptVTPWmZP
*WM: [3] WiFi station enable
*WM: [1] connectTimeout not set, ESP waitForConnectResult...
*WM: [2] Connection result: WL_CONNECTED
*WM: [3] lastconxresult: WL_CONNECTED
*WM: [1] AutoConnect: SUCCESS
*WM: [1] STA IP Address: 192.168.1.20
connected...yeey :)
ThingSpeak begin
*WM: [3] unloading
Fields update
Last temperature:0.00
Last humidity:0.00
Fields update
Last temperature:0.10
Last humidity:0.20
Fields update
Last temperature:0.20
Last humidity:0.40
Fields update
Last temperature:0.30
Last humidity:0.60
Fields update
Last temperature:0.40
```

A faire (comme dans l'exemple précédent) :

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.

Envoyer ces données cycliquement (par exemple toutes les 30 seconde) vers le serveur **`ThingSpeak`**.

Laboratoire 3 – communication longue distance avec `LoRa` (*Long Range*)

3.1 Introduction

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

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

3.1.1 Modulation LoRa

Modulation LoRa a trois paramètres :

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

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

Notre carte Heltec Wifi Lora 32 intègre un modem LoRa. Il est connecté avec le SoC ESP32 par le biais d'un bus **SPI**.

La paramétrisation du modem LoRa est facilité par la bibliothèque **`LoRa.h`** à inclure dans vos programmes.

```
#include <LoRa.h>
```

3.1.1.1 Fréquence LoRa en France

```
LoRa.setFrequency(868E6);
```

définit la **fréquence** du modem sur **868,0 MHz**.

3.1.1.2 Facteur d'étalement en puissance de 2

Le **facteur d'étalement** (**`sf`**) pour la modulation LoRa varie de **2^7** à **2^12** Il indique combien de **chirps** sont utilisés pour transporter un bit d'information.

```
LoRa.setSpreadingFactor(8);
```

définit le facteur d'étalement à 10 ou 1024 signaux par bit.

3.1.1.3 Bande passante

La **largeur de bande de signal** (**`sb`**) pour la modulation LoRa varie de **31,25 kHz** à **500 kHz** (31,25, 62,5, 125,0, 250,0, 500,0). Plus la bande passante du signal est élevée, plus le débit est élevé.

```
LoRa.setSignalBandwidth (125E3);
```

définit la largeur de bande du signal à 125 KHz.

3.1.2 Paquets LoRa

La **classe LoRa** est activée avec le constructeur **`begin()`** qui prend éventuellement un argument, la fréquence.

```
int begin (longue freq);
```

Pour créer un paquet LoRa, nous commençons par:

```
int beginPacket();
```

pour terminer la construction du paquet que nous utilisons

```
int endPacket ();
```

Les données sont **envoyées** par les fonctions:

```
virtual size_t write (uint8_t byte);
virtual size_t write (const uint8_t * buffer, taille_taille);
```

Dans notre cas, nous utilisons fréquemment la deuxième fonction avec le tampon contenant 4 données en virgule flottante.

La réception **en continue** des paquets LoRa peut être effectuée via la méthode **`parsePacket().`**

```
int parsePacket();
```

Si le paquet est présent dans le tampon de réception du modem LoRa, cette méthode renvoie une valeur entière égale à la taille du paquet (charge utile de trame).

Pour lire efficacement le paquet du tampon, nous utilisons les méthodes :

`LoRa.available()` et **`LoRa.read ().`**

Les paquets LoRa sont envoyés comme chaînes des octets. Ces chaînes doivent porter différents types de données.
Afin d'accommoder ces différents formats nous utilisons les **`union`**.
Par exemple pour formater un paquet avec 4 valeurs un virgule flottante nous définissons une union type :

```
union pack
{
      uint8_t frame[16]; // trames avec octets
      float  data[4];   // 4 valeurs en virgule flottante
} sdp ;  // paquet d'émission
```

Pour les paquet plus évolués nous avons besoin d'ajouter les en-têtes . Voici un exemple d'une union avec les paquets structurés.

```
union tspack
  {
    uint8_t frame[48];
    struct packet
      {
        uint8_t head[4]; uint16_t num; uint16_t tout; float sensor[4];
      } pack;
  } sdf,sbf,rdf,rbf;  // data frame and beacon frame
```

3.2 Premier exemple – émetteur et récepteur des paquets LoRa

Figure 3.1 Un lien LoRa avec un émetteur et un récepteur

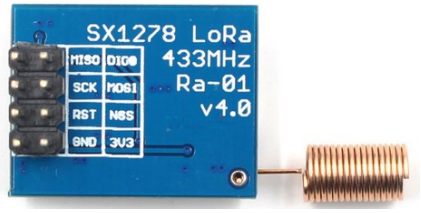


Figure 3.2 Module LoRa avec une connexion sur bus SPI

La partie initiale du code est la même pour l'émetteur et pour le récepteur. Dans cette partie nous insérons les bibliothèque de connexion par le bus SPI (**`SPI.h`**) et de communication avec le modem LoRa (**`LoRa.h`**). Nous proposons les paramètres par défaut pour le lien radio LoRa.

```
#include <SPI.h>
#include <LoRa.h>
#define SCK     18   // GPIO18 -- SX127x's SCK
#define MISO    19   // GPIO19 -- SX127x's MISO
#define MOSI    23   // GPIO23 -- SX127x's MOSI
#define SS       5   // GPIO05 -- SX127x's CS
#define RST     15   // GPIO15 -- SX127x's RESET
#define DI0     26   // GPIO26 -- SX127x's IRQ(Interrupt Request)
#define freq    434E6
#define sf 7
#define sb 125E3
```

Nous définissons également le format d'un paquet dans la trame LoRa avec 4 champs **`float`** pour les données des capteurs.

```
union pack
{
      uint8_t frame[16]; // trames avec octets
      float  data[4];   // 4 valeurs en virgule flottante
} sdp ;  // paquet d'émission
```

Dans la fonction **`setup()`** nous initialisons les connexions avec le modem LoRa et les paramètres radio.

```
void setup() {
  Serial.begin(9600);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
```

```
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
}
```

Enfin dans la fonction **loop()** nous préparons les données à envoyer dans le paquet LoRa de façon cyclique , une fois toutes les 2 secondes.

```
float d1=0.0, d2=0.0 ;

void loop()  // la boucle de l'emetteur
{
Serial.print("New Packet:") ;
  LoRa.beginPacket();                   // start packet
  sdp.data[0]=d1;
  sdp.data[1]=d2;
  LoRa.write(sdp.frame,16);
  LoRa.endPacket();
  Serial.printf("%2.2f,%2.2f\n",d1,d2);
  d1=d1+0.1; d2=d2+0.2;
  delay(2000);
}
```

Le programme du récepteur contient les mêmes déclarations et la fonction de **setup()** que le programme de l'émetteur. Le code ci dessous illustre seulement la partie différente.

```
float d1=0.0, d2=0.0 ;
int rssi;

void loop()
{
int packetLen;
packetLen=LoRa.parsePacket();
if(packetLen==16)
  {
  int i=0;
  while (LoRa.available()) {
    rdp.frame[i]=LoRa.read();i++;
    }
  d1=rdp.data[0];d2=rdp.data[1];
  rssi=LoRa.packetRssi();  // force du signal en réception en dB
  Serial.printf("Received packet:%2.2f,%2.2f\n",d1,d2);
  Serial.printf("RSSI=%d\n",rssi);
  }
}
```

A faire :

1. Tester le code ci-dessus et analyser la force du signal en réception. Par exemple **-60 dB** signifie que le signal reçu est 10^6 plus faible que le signal d'émission.
2. Modifier les paramètres LoRa par exemple **freq=435E6**, **sf=10**, **sb=250E3** et tester le résultats de transmission.
3. Afficher les données envoyées/reçues sur l'écran OLED

3.3 `onReceive()` – récepteur des paquets LoRa avec une interruption

Les interruptions permettent de ne pas exécuter en boucle l'opération d'attente d'une nouvelle trame dans le tampon du récepteur. Une fonction-tâche séparée est réveillée automatiquement après l'arrivée d'une nouvelle trame.
Cette fonction, souvent appelée **`onReceive()`** doit être marquée dans le **`setup()`** par :

```
void setup() {
  Serial.begin(9600);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
LoRa.onReceive(onReceive);  // pour indiquer la fonction ISR
LoRa.receive();             // pour activer l'interruption (une fois)
// puis après chaque émission
}
```

La fonction **ISR** (*Interruption Service Routine*) ci-dessous permet de **capter** une nouvelle trame.

```
void onReceive(int packetSize)
{
int rssi=0;
union pack
  {
  uint8_t frame[16]; // trames avec octets
  float  data[4];    // 4 valeurs en virgule flottante
  } rdp ;  // paquet de réception
  if (packetSize == 0) return;   // if there's no packet, return
  int i=0;
  if (packetSize==16)
    {
    while (LoRa.available())
      {
      rdp.frame[i]=LoRa.read();i++;
      }
      rssi=LoRa.packetRssi();
      Serial.printf("Received packet:%2.2f,%2.2f\n",rdp.data[0],rdp.data[1]);
      Serial.printf("RSSI=%d\n",rssi);
    }
}

void loop()
{
Serial.println("in the loop") ;
delay(5000) ;
}
```

L'affichage pendant l'exécution :

```
Received packet:90.90,181.80
```

```
RSSI=-52
Received packet:91.00,182.00
RSSI=-52
Received packet:91.10,182.20
RSSI=-53
in the loop
Received packet:91.20,182.40
RSSI=-53
Received packet:91.30,182.60
RSSI=-53
in the loop
```

A faire :

1. Tester le code ci-dessus.
2. Afficher les données reçues sur l'écran OLED
3. Transférer les données reçues dans les variables externes (problème?).

Laboratoire 4 - Développement d'une simple passerelle IoT

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

4.1 Passerelle `LoRa-ThingSpeak`

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

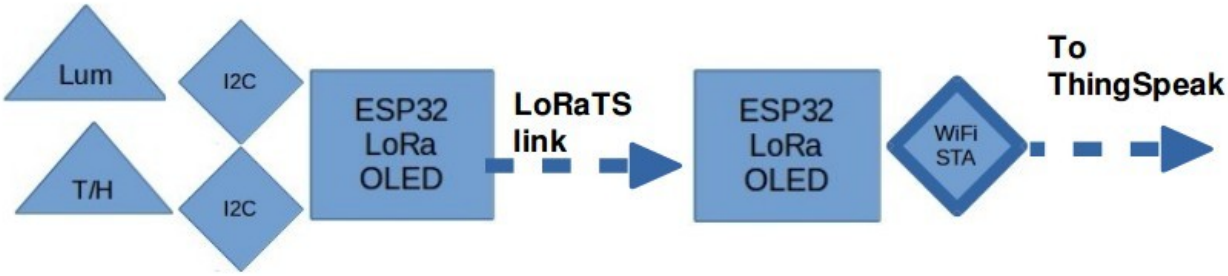


Figure 4.1 Une simple architecture IoT avec un terminal à 2 capteurs et une passerelle **LoRa-WiFi**

4.1.1 Le principe de fonctionnement

La passerelle attend les messages LoRa envoyés dans le format prédéfinie **LoRaTS** sur une interruption I/O redirigée vers la fonction (ISR) **`onReceive()`**. Elle les stocke dans une file de messages (*queue*) en gardant seulement le dernier paquet. La tâche principale, dans la boucle **`loop()`** récupère ce paquet dans la file par la fonction **`xQueueReceive()`** puis l'envoie sur le serveur **`ThingSpeak`**. Le contenu d'un paquet en format **LoRaTS** est transformé en un ou plusieurs fonctions :

`ThingSpeak.setField(fn,value);`

et la fonction

`ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);`

4.1.2 Les éléments du code

Ci-dessous nous présentons les éléments du code de la passerelle. Les messages LoRa sont envoyés dans un **format simplifié** de **LoRaTS**. Nous avons définie le type de l'union/structure d'un paquet (**`pack_t`**) de la façon suivante. Le format simplifié correspond au transfert des paquets LoRa dans le laboratoire précédent.

```
typedef union      // format simplifie
{
  uint8_t frame[16]; // trames avec octets
  float  data[4];    // 4 valeurs en virgule flottante
} pack_t ;  // paquet d'émission

typedef union // format de base
  {
    uint8_t frame[40];
    struct
      {
        uint8_t head[4];   // packet header
        int chnum;         // channel number
        char key[16];      // write or read key
        float sensor[4];
      } pack;
  } pack_t;
```

Pour le format de base dans l'en-tête **head[4]** nous indiquons :
head[0] – adresse de destination, **0x00** – passerelle, **0xff** – diffusion
head[1] – adresse de source
head[2] – type du message – data **write**/**read**, **control**, ..
head[3] – le masque **0xC0** signifie **field1** et **field2**

Les paquets qui arrivent par le lien LoRa sont captés par l'ISR **onReceive()** et déposés dans une file d'attente **FreeRTOS**. Cette file doit être déclarée par :

```
QueueHandle_t dqueue;  // queues for data packets
```

Puis, dans le **setup()** on instancie la file en lui fournissant le nombre d'éléments et la taille d'un élément:

```
dqueue = xQueueCreate(4,16); // queue for 4 data packets in simple format

void onReceive(int packetSize)
{
pack_t rbuff;  // receive buffer with pack_t format
int i=0;
  if (packetSize == 0) return;   // if there's no packet, return
  i=0;
  if (packetSize==16)   // 16 pour le format simplifie , 40 format de base
    {
    while (LoRa.available()) {
      rbuff.frame=LoRa.read();i++;
      }
      xQueueReset(dqueue); // to keep only the last element
      xQueueSend(dqueue, rbuff, portMAX_DELAY);
    }
  delay(200);
}
```

La fonction de **setup()** :

```
void setup()
{
  Serial.begin(9600);
  WiFi.mode(WIFI_STA);
  WiFiManager wm;
  // wm.resetSettings();
  bool res;
  res = wm.autoConnect("ESP32AP",NULL); // password protected ap
  if(!res)
    {
    Serial.println("Failed to connect");// ESP.restart();
    }
  else
    {
    //if you get here you have connected to the WiFi
    Serial.println("connected...yeey :)");
    }

  ThingSpeak.begin(client); // connexion (TCP) du client au serveur
  delay(1000);
  Serial.println("ThingSpeak begin");
  Serial.println("Start Lora");
```

```
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
dqueue = xQueueCreate(4,16); // queue for 4 simple data packets
LoRa.onReceive(onReceive);  // pour indiquer la fonction ISR
LoRa.receive();             // pour activer l'interruption (une fois)
}
```

La fonction de la tâche en `loop()` :

Pour le format simplifié :

```
void loop()
{
pack_t rp;    // packet elements to send
  xQueueReceive(dqueue,rp.frame,portMAX_DELAY); // 6s,default:portMAX_DELAY
  ThingSpeak.setField(1,rp.data[0]);
  ThingSpeak.setField(2,rp.data[1]);
  ThingSpeak.setField(3,rp.data[2]);
  ThingSpeak.setField(4,rp.data[3]);
  Serial.printf("d1=%2.2f,d2=%2.2f,d3=%2.2f,d4=%2.2f\n",
rp.data[0],rp.data[1],rp.data[2],rp.data[3]);
  while (WiFi.status() != WL_CONNECTED) {  delay(500);  }
  ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
  delay(mindel); // mindel is the min waiting time before sending to ThingSpeak
 //LoRa.receive();
}
```

Pour le format de base :

```
loop()
{
pack_t sb;    // packet elements to send
xQueueReceive(dqueue,sb.frame, portMAX_DELAY); // 6s,default:portMAX_DELAY
    if(sb.pack.head[1]==0x01 || sb.pack.head[1]==0x00)
    {
        if(sb.pack.head[2]&0x80) ThingSpeak.setField(1,sb.pack.sensor[0]);
        if(sb.pack.head[2]&0x40) ThingSpeak.setField(2,sb.pack.sensor[1]);
        if(sb.pack.head[2]&0x20) ThingSpeak.setField(3,sb.pack.sensor[2]);
        if(sb.pack.head[2]&0x10) ThingSpeak.setField(4,sb.pack.sensor[3]);
        while (WiFi.status() != WL_CONNECTED) {  delay(500);  }
        ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
    }
    LoRa.receive();
  }
```

4.1.3 Code complet pour une passerelle des paquets en format de base

```
#include <SPI.h>
#include <LoRa.h>
#include <WiFiManager.h>
#include "ThingSpeak.h"
unsigned long myChannelNumber = 1;
const char *myWriteAPIKey="HEU64K3PGNWG36C4";
const char *myReadAPIKey="AVG5MID7I5SPHIK8";
WiFiClient  client;
#define SCK     18   // GPIO18 -- SX127x's SCK
#define MISO    19   // GPIO19 -- SX127x's MISO
#define MOSI    23   // GPIO23 -- SX127x's MOSI
#define SS       5   // GPIO05 -- SX127x's CS
#define RST     15   // GPIO15 -- SX127x's RESET
#define DI0     26   // GPIO26 -- SX127x's IRQ(Interrupt Request)
#define freq    434E6
#define sf      7
#define sb      125E3

typedef union
{
  uint8_t frame[16]; // trames avec octets
  float  data[4];    // 4 valeurs en virgule flottante
} pack_t ;  // paquet d'émission

QueueHandle_t dqueue;  // queues for data packets

void onReceive(int packetSize)
{
pack_t rdp;  int i=0;
if (packetSize == 0) return;   // if there's no packet, return
if (packetSize==16)
  {
  while (LoRa.available())  {  rdp.frame[i]=LoRa.read();i++; }
  xQueueReset(dqueue); // to keep only the last element
  xQueueSend(dqueue, &rdp, portMAX_DELAY);
  }
}

void setup()
{
  Serial.begin(9600);
  WiFi.mode(WIFI_STA);
  WiFiManager wm;
  // wm.resetSettings();
  bool res;
  res = wm.autoConnect("ESP32AP",NULL); // password protected ap
  if(!res)
    {
    Serial.println("Failed to connect");// ESP.restart();
    }
  else
    {
    //if you get here you have connected to the WiFi
    Serial.println("connected...yeey :)");   }
```

```
  ThingSpeak.begin(client); // connexion (TCP) du client au serveur
  delay(1000);
  Serial.println("ThingSpeak begin");
  Serial.println("Start Lora");
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
dqueue = xQueueCreate(4,16); // queue for 4 data packets
LoRa.onReceive(onReceive);  // pour indiquer la fonction ISR
LoRa.receive();             // pour activer l'interruption (une fois)
}

uint32_t mindel=10000;  // 10 seconds


void loop()
{
pack_t rp;    // packet elements to send
  xQueueReceive(dqueue,rp.frame,portMAX_DELAY); // 6s,default:portMAX_DELAY
  ThingSpeak.setField(1,rp.data[0]);
  ThingSpeak.setField(2,rp.data[1]);
  ThingSpeak.setField(3,rp.data[2]);
  ThingSpeak.setField(4,rp.data[3]);
  Serial.printf("d1=%2.2f,d2=%2.2f,d3=%2.2f,d4=%2.2f\n",rp.data[0],rp.data[1],
                rp.data[2],rp.data[3]);
  while (WiFi.status() != WL_CONNECTED) {  delay(500);  }
  ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
  delay(mindel); // mindel – min waiting time before sending to ThingSpeak
 //LoRa.receive();
}
```

A faire :

1. Tester le code de base
2. Ecrire le code complet pour la version de base avec le transfert du numéro du canal et de clé d'écriture dans les paquets LoRa. Tester ce nouveau programme de passerelle avec un puis avec 2 terminaux.
3. Utiliser les capteurs SHT21 et BH1750 pour envoyer des données « réelles »