Laboratoires IoT de base

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

Table des contenus

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 (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 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 le serveur **ThingSpeak**/**Thinger.io**.

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 plusieurs **laboratoires thèmatiques** 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/actuateurs 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.

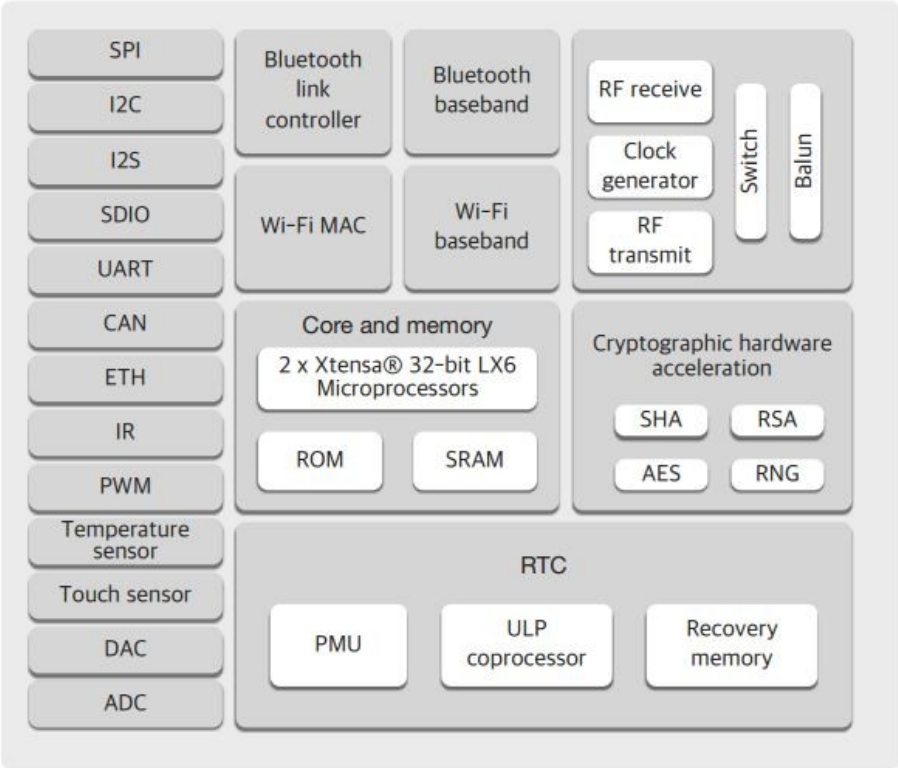


Figure 0.1
ESP32 SoC
– architecture interne

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 S127 /78** et (éventuellement) un écran OLED. La figure suivante montre la carte **ESP32-Heltec WiFi-LoRa**

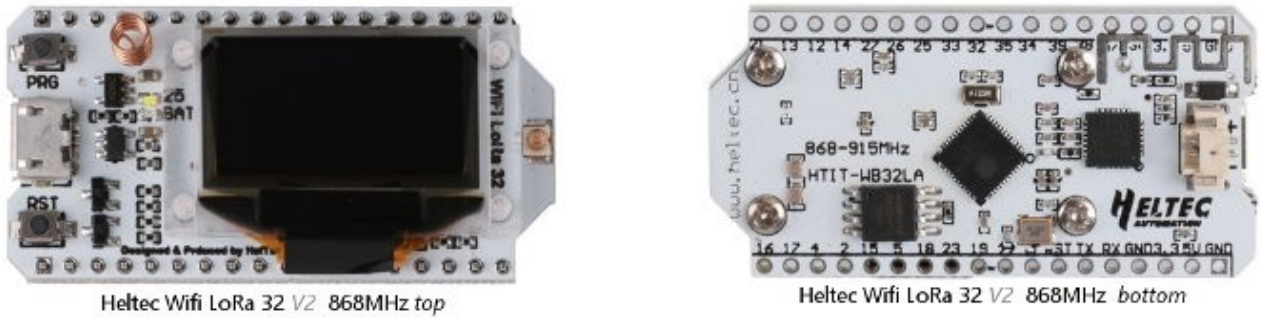


Figure 0.2 Carte MCU **ESP32-Heltec WiFi-LoRa**

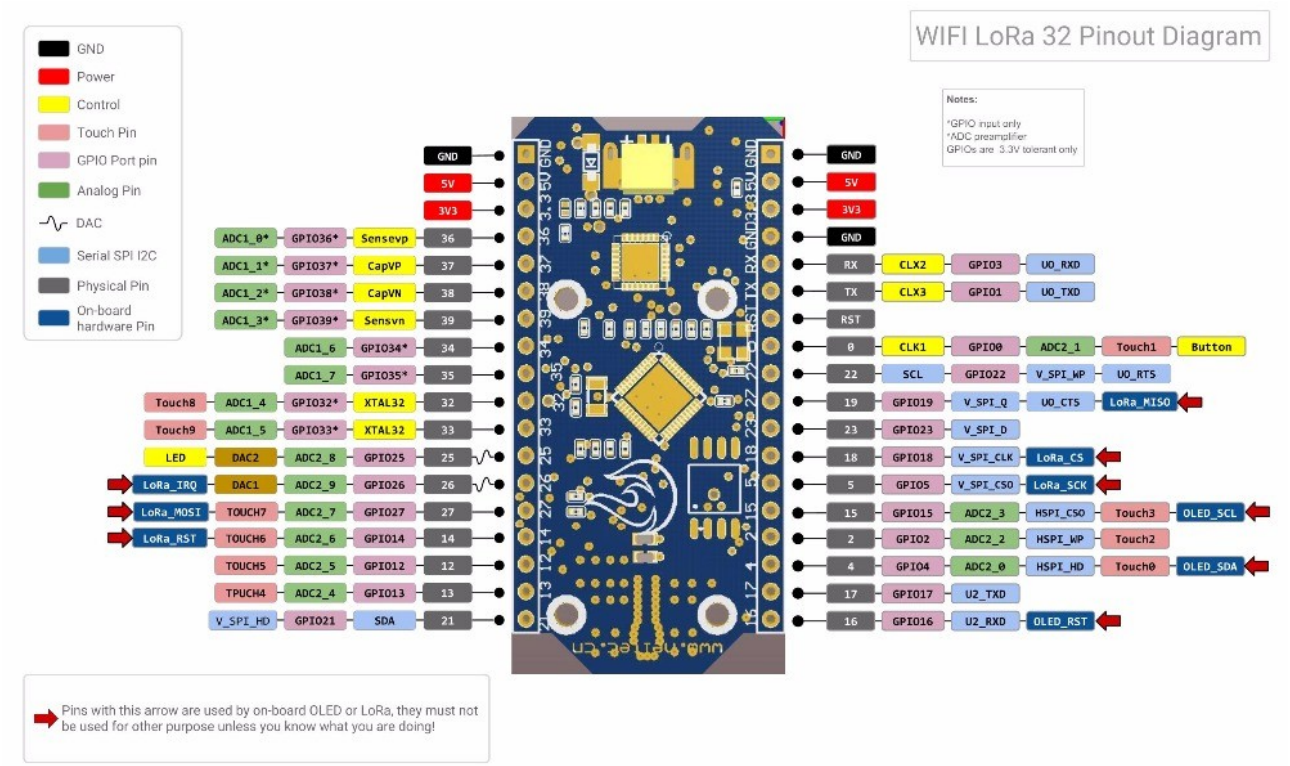


Figure 0.3. Carte **ESP32-Heltec WiFi-LoRa** et son *pin-out*

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.

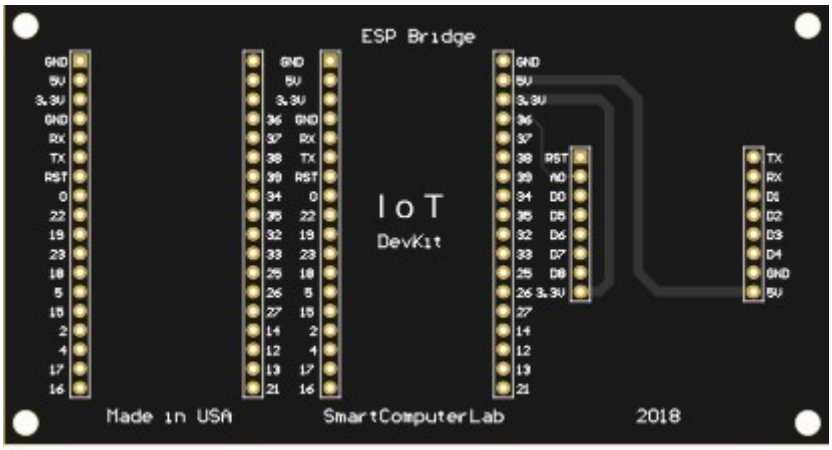
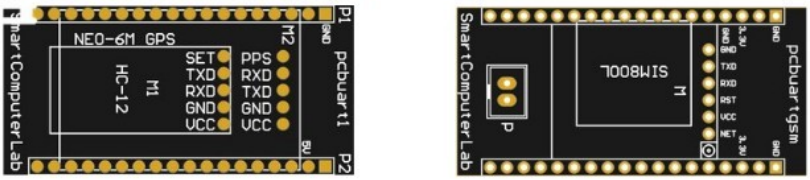


Figure 0.4. **IoT DevKit**: carte de base (type bridge) avec l'unité principale et les emplacements pour les cartes d'extension

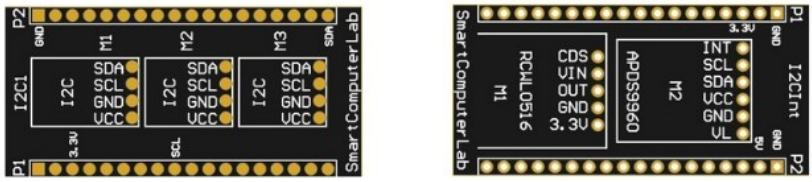
0.3.1 Cartes d'extension – quelques exemples

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

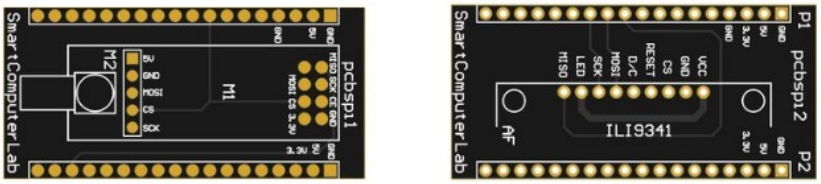
Cartes d'extension pour le bus **UART** (deux exemples):



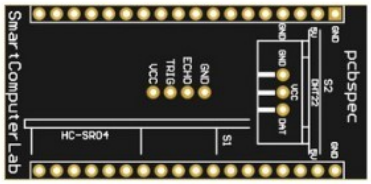
Cartes d'extension pour le bus **I2C** (deux exemples):



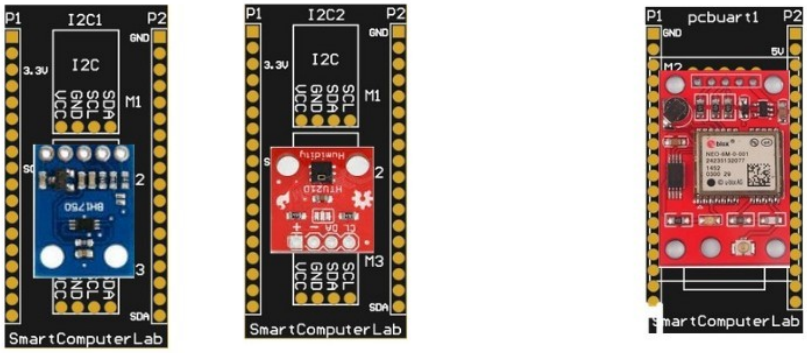
Cartes d'extension pour le bus **SPI** (deux exemples):



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



0.4 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 :

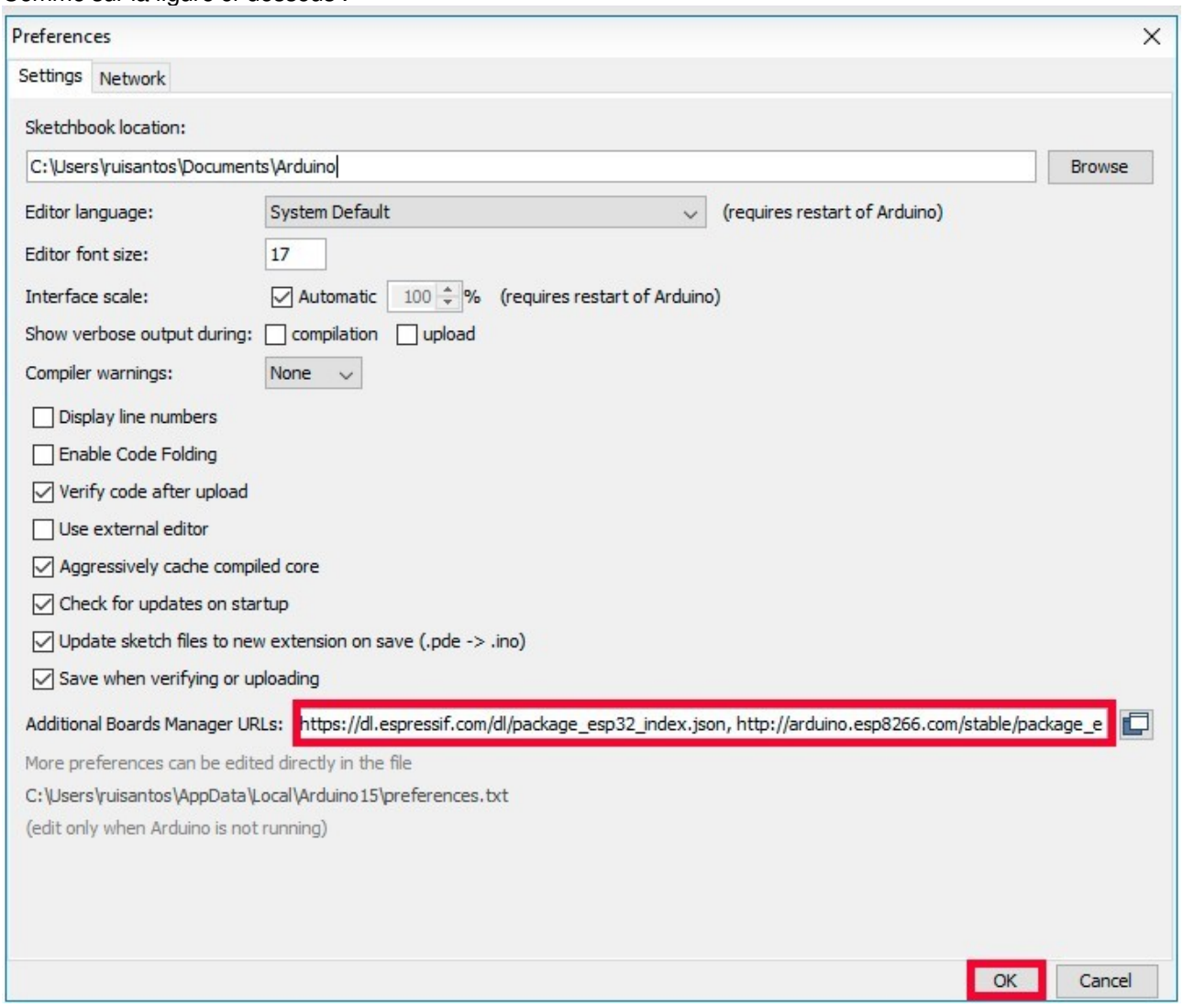
sudo apt-upgrade et **sudo apt update**

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 :



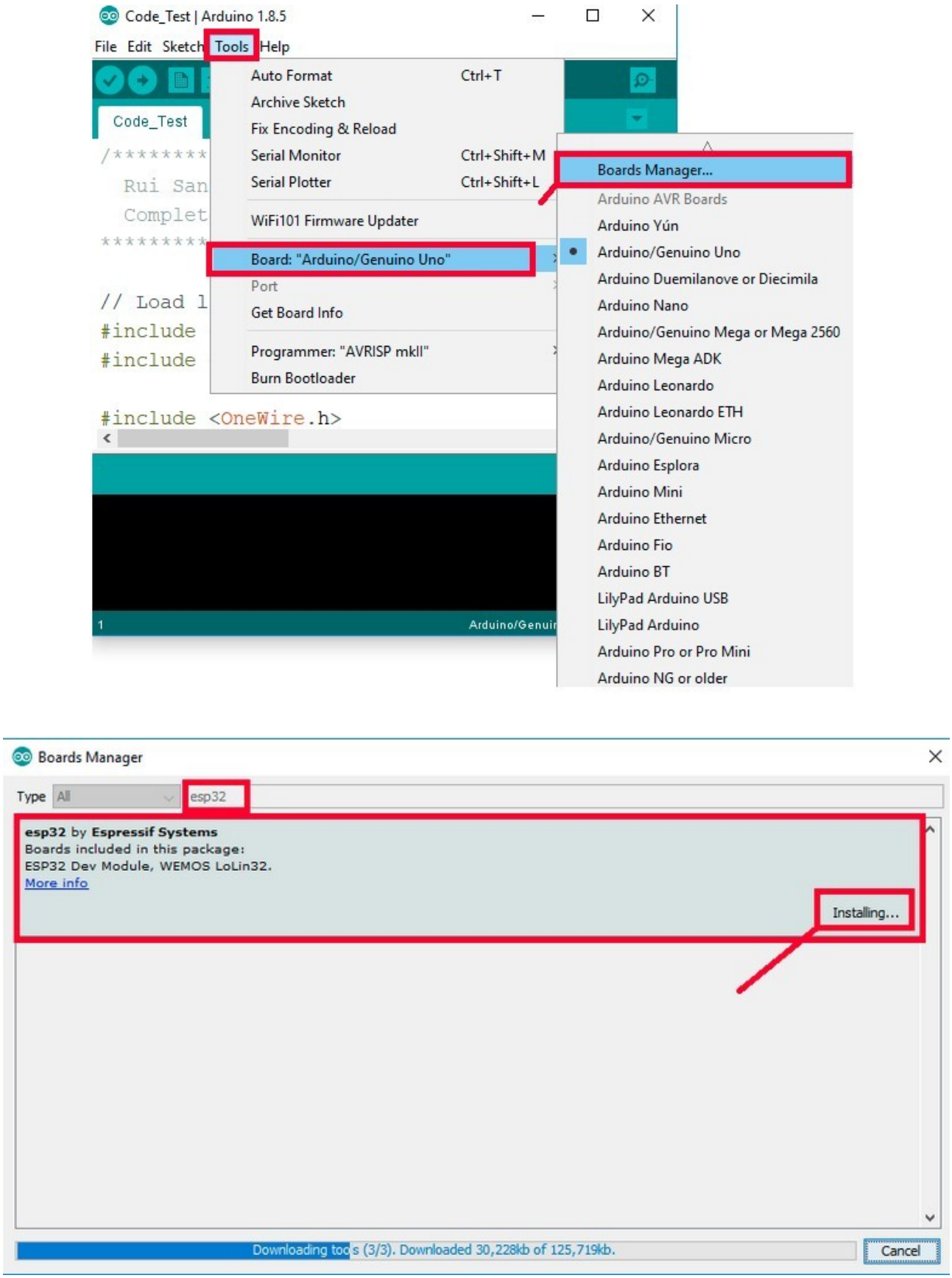


Figure 0.6. L'intégration des cartes **ESP32** et **ESP8266** dans l'environnement IDE Arduino

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**.

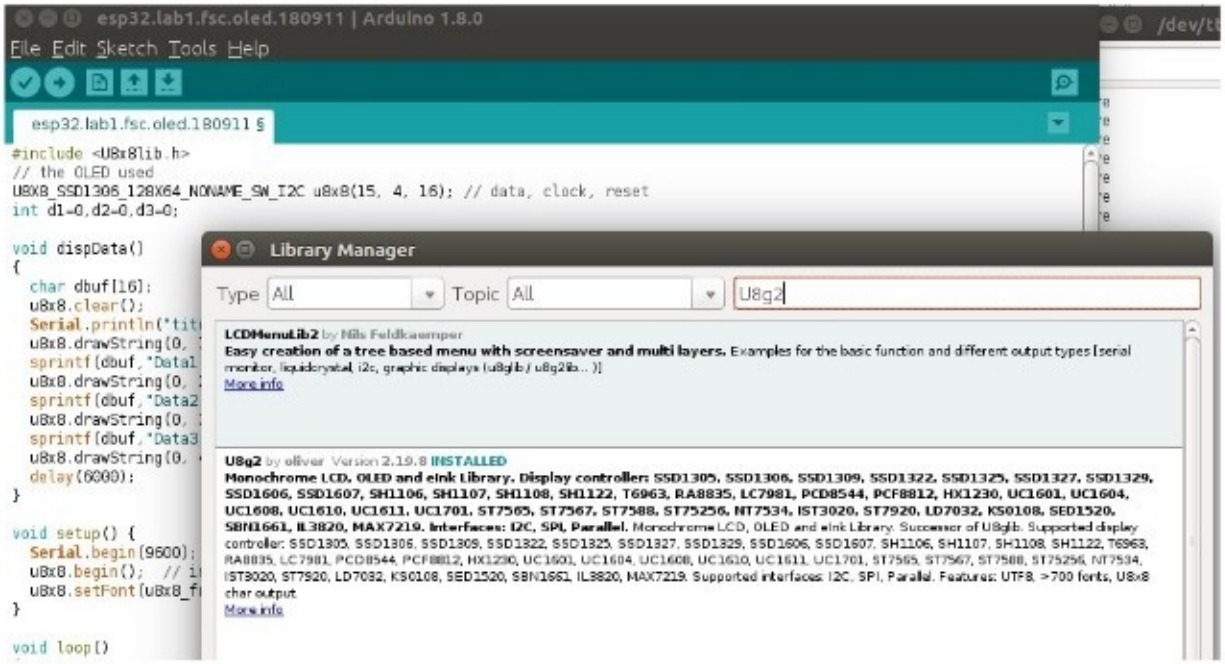


Figure 0.7. Intégration d'une bibliothèque Arduino (**U8g2**)

Laboratoire 1

1.1 Premier exemple – l'affichage des données

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**.

```
#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 la configuration des broches identique à celle du capteur, nous prenons donc la carte **`I2C2`** présentée ci-dessous.

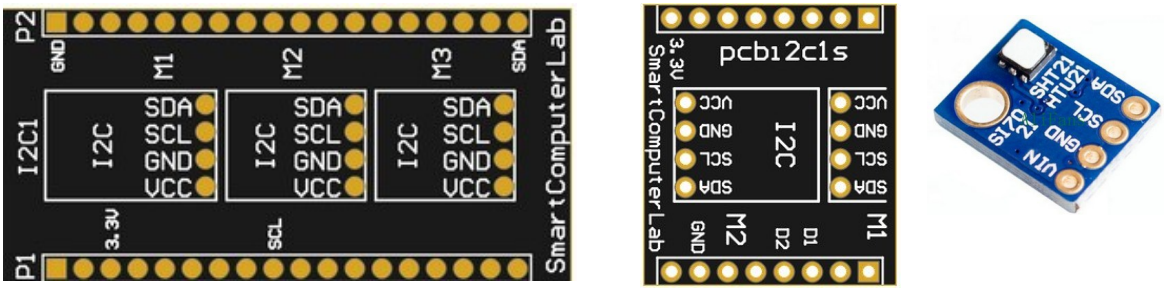


Figure 1.5. **IoT DevKit :** cartes d'extension **`I2C1`** **et etet u**n capteur de température/humidité **SHT21**

Code Arduino

```
#include <Wire.h>
#include <Sodaq_SHT2x.h>

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

void loop()
{
  Serial.print("Humidity(%RH): ");
  Serial.print(SHT2x.GetHumidity());
  Serial.print("     Temperature(C): ");
  Serial.println(SHT2x.GetTemperature());
  Serial.print("     Dewpoint(C): ");
  Serial.println(SHT2x.GetDewPoint());

  delay(1000);
}
```

1.2.2 Capture de la luminosité par BH1750

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

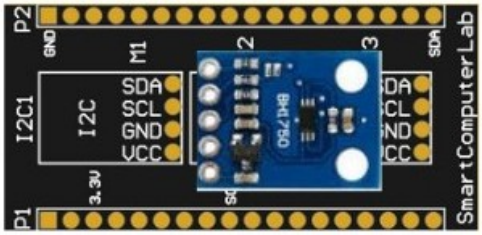


Figure 1.3. **IoT DevKit :** cartes d'extension **`I2C1`** (longue et courte) pour le capteur Luminosité **BH1750**
Attention : Avant la compilation il faut installer la bibliothèque **`BH1750.h`**

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

void setup(){
  Wire.begin(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);
}
```

1.2.3 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.

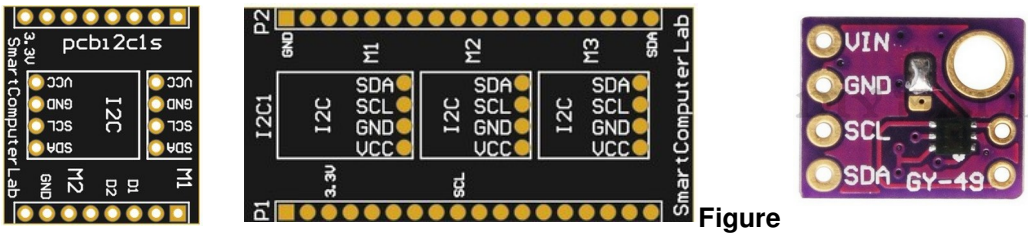


Figure 1.4. **IoT DevKit :** cartes d'extension **I2C1** et un capteur de luminosité **MAX44009**

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);
}
```

1.2.4 Capture de la température ambiante et d'objet avec capteur MLX906 (GY-906)

Le capteur **MLX906** permet d'obtenir la température ambiante et la température d'une surface (distance < 1m)..

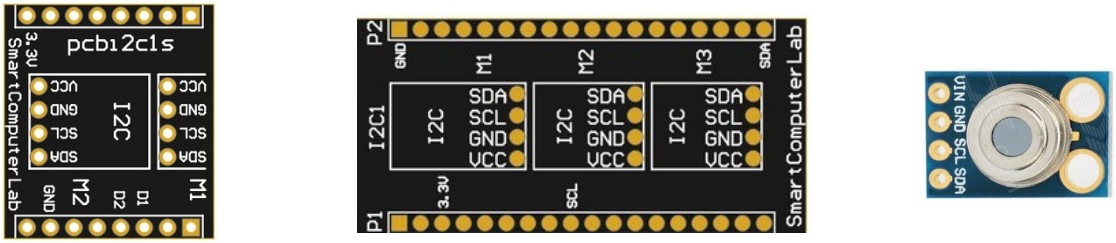


Figure 1.5. **IoT DevKit :** cartes d'extension **I2C1** et un capteur de température ambiante et d'une surface proche.

```
#include <Wire.h>
#include <Adafruit_MLX90614.h>

Adafruit_MLX90614 mlx = Adafruit_MLX90614();

void setup() {
  Serial.begin(9600);
  Wire.begin(21,22);
  Serial.println("Adafruit MLX90614 test");
  mlx.begin();
}

void loop() {
  Serial.print("Ambient = "); Serial.print(mlx.readAmbientTempC());
  Serial.print("*C\tObject = "); Serial.print(mlx.readObjectTempC());
Serial.println("*C");
  Serial.print("Ambient = "); Serial.print(mlx.readAmbientTempF());
  Serial.print("*F\tObject = "); Serial.print(mlx.readObjectTempF());
Serial.println("*F");

  Serial.println();
  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**.

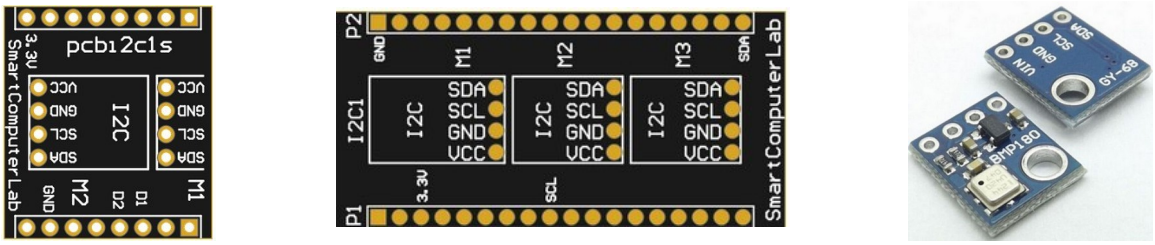


Figure 1.6. **IoT DevKit :** cartes d'extension **I2C1** et un capteur de pression/température **BMP180**

La valeur standard de la pression atmosphérique est :

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

Code Arduino :

```
#include <Wire.h>
#include <BMP180.h>
BMP180 myBMP(BMP180_ULTRAHIGHRES);

void setup()
{
  Serial.begin(9600);
  Wire.begin(21,22);
  while (myBMP.begin() != true)
  {
    Serial.println("BMP180/BMP085 is not connected or fail to read calibration
coefficients");
    delay(5000);
  }
  Serial.println(« BMP180/BMP085 sensor is OK");
}

void loop()
{
  Serial.print("Temperature.......: ");
  Serial.print(myBMP.getTemperature(), 1); Serial.println(F(" +-1.0C"));
  Serial.print("Pressure..........: "); Serial.print(myBMP.getPressure());
  Serial.println(F(" +-100Pa"));
  Serial.print("See level pressure: ");
  Serial.print(myBMP.getSeaLevelPressure(115)); Serial.println(" Pa");
  Serial.print("Starts over again in 10 sec.");
  delay(10000);
}
```

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.

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 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
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://www.arduinolibraries.info/libraries/thing-speak`

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 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 :

```
#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`** la **valeur minimale** de **`tout`** est 20 secondes.

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

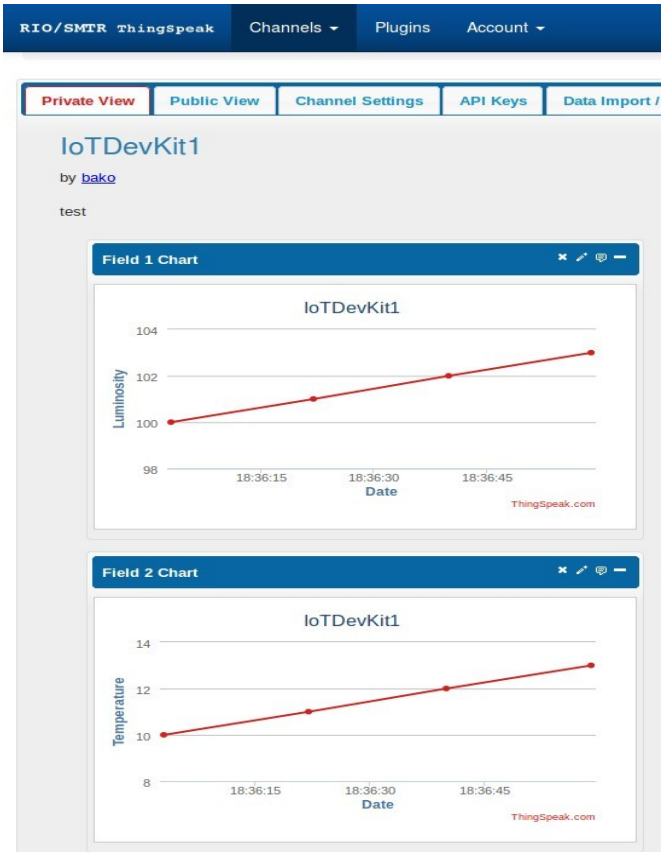


Figure 2.1. **ThingSpeak** : l'affichage des données envoyées sur le serveur type **ThingSpeak**

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

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 Arduino permettant d'effectuer une telle connexion. Comme dans l'exemple précédent le programme communique avec un serveur **ThingSpeak**.

```
#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);
}
```

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**.

2.4 Le programme de réception des données à partir d'un serveur **ThingSpeak**

Le programme ci-dessous permet de récupérer les données envoyées par notre programme précédent (émetteur) Les données sont lues sur le serveur **ThingSpeak.com (.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

1. Modifier (simplifier) ce programme pour une utilisation avec un Point d'Accès direct (votre PhoneAP par exemple)
2. Appliquer le code à votre canal du 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 modulation **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 – 100Kbit).

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 veleurs 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

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 libraries
#include <LoRa.h>
#define SCK     5    // GPIO5  -- SX127x's SCK
#define MISO    19   // GPIO19 -- SX127x's MISO
#define MOSI    27   // GPIO27 -- SX127x's MOSI
#define SS      18   // GPIO18 -- SX127x's CS
#define RST     14   // GPIO14 -- SX127x's RESET
#define DI0     26   // GPIO26 -- SX127x's IRQ(Interrupt Request)
#define freq    868E6
#define sf 8
#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);
  pinMode(DI0, INPUT);  // signal interrupt
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
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=12.0, d2=321.54 ;

void loop()  // la boucle de l'emetteur
{
Serial.println("New Packet") ;
  LoRa.beginPacket();                   // start packet
  sdp.data[0]=d1;
```

```
  sdp.data[1]=d2;
  LoRa.write(sdp.frame,16);
  LoRa.endPacket();
d1++; d2+=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,d2;
int i=0;

void loop()  // la boucle de l'emetteur
{
int packetLen;
packetLen=LoRa.parsePacket();
if(packetLen==16)
  {
    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.println(d1);  Serial.println(d2);
  Serial.println(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=869E6`**, **`sf=10`**, **`sb=250E3`** et tester le résultats de transmission.
3. Afficher les données 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()
{

pinMode(DI0,INPUT);       // pour activer le pin d'interruption
..

LoRa.onReceive(onReceive);  // pour indiquer la fonction ISR
LoRa.receive();  // pour activer l'interruption (une fois)
}
```

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

```
void onReceive(int packetSize)
{
int i=0,rssi=0;
uint8_t rdbuff[24], rbbuff[8];
  if (packetSize == 0) return;   // if there's no packet, return
  i=0;
  if (packetSize==24)
    {
    while (LoRa.available()) {
      rdbuff[i]=LoRa.read();i++;
      }
      rssi=LoRa.packetRssi();
      // affichage des données réçues
    }
}

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

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 le serveur **ThingSpeak.fr**.

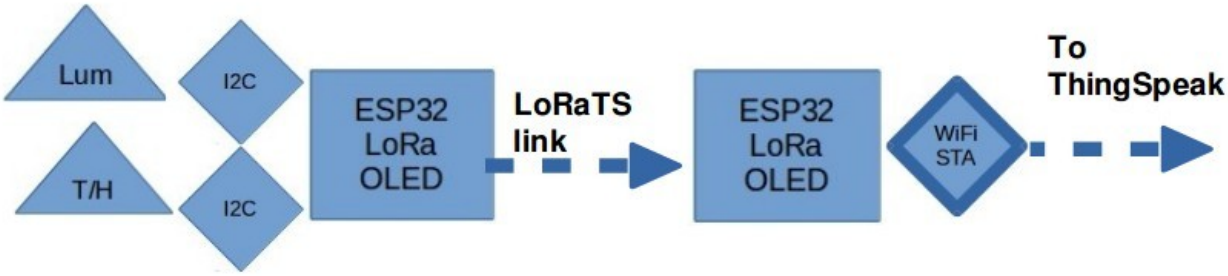


Figure 4.1 Une simple architecture IoT avec un terminal à 2 capteurs et une passerelle **LoRa-WiFi**

4.1.1 Le principe de fonctionnement

La passerelle attend les messages LoRa envoyés dans le format **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 message. La tâche principale, dans la boucle l**`oop()`** récupère ce message dans la file par la fonction non destructive **`xQueuePeek()`** puis l'envoie sur le serveur **`ThingSpeak.fr`**.

Le message en format **LoRaTS** est transformé en un ou plusieurs fonctions :
`ThingSpeak.setField(fn,value);`

et la fonction

`ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);`

4.1.2 Le code

Ci-dessous nous présentons les éléments du code de la passerelle. Les messages LoRa sont envoyés dans un format simplifié de **LoRaTS**.

```
union tspack
  {
    uint8_t frame[40]; // ou [72]
    struct pack
      {
        uint8_t head[4];
        uint16_t num;
        uint16_t tout;
        float sensor[4];
      } rp;
  } rf;
```

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 – par exemple le masque **`0xC0`** signifie **`field1`** et **`field2`**

`head[3]` – longueur de la donnée – par exemple **`0x10`** ou **`0x20`**

```
QueueHandle_t dqueue;  // queues for data frames
```
à déclarer au début du code

```
dqueue = xQueueCreate(4, 24); // queue for 4 LoRaTS data frames
```
à initialiser dans le **setup()**

```
void onReceive(int packetSize)
{
int i=0;
uint8_t rdbuff[24];
  if (packetSize == 0) return;   // if there's no packet, return
  i=0;
  if (packetSize==24)
    {
    while (LoRa.available()) {
      rdbuff[i]=LoRa.read();i++;
      }
      xQueueReset(dqueue); // to keep only the last element
      xQueueSend(dqueue, rdbuff, portMAX_DELAY);
    }
  delay(200);
}
```

La fonction de la tâche en **loop()** :

```
loop(){
xQueueReceive(dqueue, rdf.frame, 6000); // timeout - 6s, default: portMAX_DELAY
    if(rdf.pack.head[1]==0x01 || rdf.pack.head[1]==0x02)
    {
        if(rdf.pack.head[2] & 0x80) ThingSpeak.setField(1, rdf.pack.sensor[0]);
        if(rdf.pack.head[2] & 0x40) ThingSpeak.setField(2, rdf.pack.sensor[1]);
        if(rdf.pack.head[2] & 0x20) ThingSpeak.setField(3, rdf.pack.sensor[2]);
        if(rdf.pack.head[2] & 0x10) ThingSpeak.setField(4, rdf.pack.sensor[3]);
        while (WiFi.status() != WL_CONNECTED) {
          delay(500);
          }
        ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
    }
    LoRa.receive();
  }
```

A faire :

1. Ecrire le code complet et le tester avec un puis avec 2 terminaux.

2. Chaque terminal génère les donnes sur différents champs (*fields*) ; par exemple terminal `0x01` utilise les *field1* et *field2*, tandis que le terminal `0x02` exploite *field3* et *field4*.

Table of Contents