Laboratoires IoT avec PlatformIO

SmartComputerLab

Contenu

0. **Introduction** **2**
0.1 ESP32 Soc – une unité avancée pour les architectures IoT 3
0.2 Carte LOLIN D32 3
0.3 IoT DevKit une plate-forme de développement IoT 4
0.4 L'installation de PlatfomIO 5
0.4.1 Installation de VSCode 5
0.4.2 Installer le package PlatformIO IDE pour VSCode 5
0.4.3 Premier démarrage de PlatformIO sur VSCode 6
0.4.4 Le menu PIO 7
0.4.5 Créer un nouveau projet (ESP32, ESP8266, …) 8
0.4.6 Décryptage du fichier platformio.ini 10
0.4.7 Edition du code 11
0.7.8 Premier exemple 12
Laboratoire 1 ..... **15**
1.1 Premier exemple – l'affichage des données 15
A faire: 16
1.2 Deuxième exemple – capture et affichage des valeurs 17
1.2.1 Capture de la température/humidité par SHT21 17
A faire: 18
1.2.2 Capture de la luminosité par BH1750 19
1.2.3 Capture de la luminosité par MAX44009 20
1.2.5 Capture de la pression/température avec capteur BMP180 21
1.2.6 Capture de présence avec un capteur PIR SR602 22
Laboratoire 2 ..... **23**
Communication en WiFi et broker MQTT 23
2.1 Client MQTT – envoi et réception des messages 23
A faire : 25
2.2 Simple broker MQTT avec ESP8266 26
A faire : 28
Laboratoire 3 – WiFi avec WiFiManager et serveur ThingSpeak.com (.fr) **29**
3.1 Introduction 29
3.1.1 Un programme de test – scrutation du réseau WiFi 29
3.2 Mode WiFi – STA, client WEB et serveur ThingSpeak 31
A faire 32
Laboratoire 4 – communication longue distance avec LoRa (*Long Range*) **39**
4.1 Introduction 39
4.1.1 Modulation LoRa 39
4.1.2 Paquets LoRa 40
4.2 Premier exemple – émetteur et récepteur des paquets LoRa 41
A faire : 43
4.3 onReceive() – récepteur des paquets LoRa avec une interruption 44
A faire : 45
Laboratoire 5 - Développement de simples passerelles IoT **46**
5.1 Passerelle LoRa-ThingSpeak 46
5.1.1 Le principe de fonctionnement 46
5.1.2 Les éléments du code 46
5.1.3 Code complet pour une passerelle des paquets en format de base 49
A faire : 50
5.2 Passerelle LoRa – WiFi – broker MQTT 51
5.2.1 L'émetteur des messages MQTT sur LoRa 51
5.2.2 La passerelle des messages MQTT sur LoRa vers WiFi et broker MQTT 52
A faire : 53

Laboratoire 4 – communication longue distance avec `LoRa` (*Long Range*)

4.1 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).

4.1.1 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`**=434MHz, **`sf`**=7, et **`sb`**=125KHz

Notre DevKit peut être complété par une carte d'extension – modem LoRa.

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

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

4.1.1.1 Fréquence LoRa en France

```
LoRa.setFrequency(434E6);
```

définit la **fréquence** du modem sur **434 MHz**.

4.1.1.2 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.

4.1.1.3 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.

4.1.2 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
```

4.2 Premier exemple – émetteur et récepteur des paquets LoRa

Figure 4.1 Un lien LoRa avec un émetteur et un récepteur

Figure 4.2 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.

```
#include <Arduino.h>

#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
union pack
{
uint8_t frame[16]; // trames avec octets
float data[4]; // 4 valeurs en virgule flottante
} rdp ; // paquet de réception

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);
}

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

4.3 `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 :

```
#include <Arduino.h>

#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
```

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 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
}

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?).
4. La solution consiste à utiliser une file de message - **queue**

```
#include <Arduino.h>

..
static QueueHandle_t msg_queue;
union pack
{
uint8_t frame[16]; // trames avec octets
float data[4]; // 4 valeurs en virgule flottante
} rdp ; // paquet de réception

void onReceive(int packetSize)
{
int rssi=0;
uint8_t buff[16];
if (packetSize == 0) return; // if there's no packet, return
int i=0;
if (packetSize==16) {
  while (LoRa.available()) { buff[i]=LoRa.read();i++; }
xQueueReset(msg_queue);
xQueueSend(msg_queue, (void *)buff, 16);
  }
}

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!");
msg_queue = xQueueCreate(4, sizeof(rdp)); // 4 éléments type union rdp
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
}

void loop()
{
xQueueReceive(msg_queue, (void *)rdp.frame, 1000);
Serial.printf("Received packet:%2.2f,%2.2f\n",rdp.data[0],rdp.data[1]);
}
```