Laboratoire 11 :

Bluetooth Classic (BT) et Bluetooth Low Energy (BLE)

L'ESP32 est livré avec Wi-Fi, **Bluetooth Low Energy** et **Bluetooth Classic**. Dans ce laboratoire, vous apprendrez comment utiliser ESP32 Bluetooth Classic et Bluetooth Low Energy (BLE) avec Arduino IDE pour échanger des données entre un ESP32 et un smartphone Android.

Pour commencer nous contrôlerons un ESP32 et enverrons des relevés de capteur à un smartphone Android à l'aide de Bluetooth Classic.

11.1 Bluetooth Classic avec ESP32

À l'heure actuelle, l'utilisation de Bluetooth Classic est beaucoup plus simple que Bluetooth Low Energy. Si vous avez déjà programmé un Arduino avec un module Bluetooth comme le **HC-06**, c'est très similaire. Il utilise le protocole et les fonctions **série standard**.

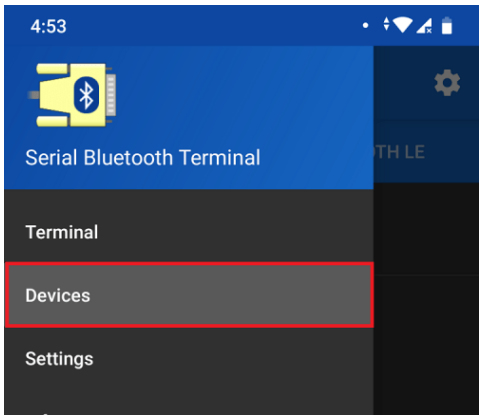
Dans cette partie du laboratoire, nous allons commencer par utiliser un exemple fourni avec l'IDE Arduino. Ensuite, nous allons construire un projet simple pour échanger des données entre l'ESP32 et votre smartphone Android.

Cherchez le premier exemple avec : **`File`** > **`Examples`** > **`BluetoothSerial`** > **`SerialtoSerialBT.`**

11.1.1. Bluetooth simple lecture écriture en série

Le code suivant démarre la communication Bluetooth et échange les données via les fonctions **`SerialBT.read ()`** et **`SerialBT.write ()`**.

```
BluetoothSerial SerialBT;

void setup() {
  Serial.begin(9600);
  SerialBT.begin("ESP32test"); //Bluetooth device name
  Serial.println("The device started, now you can pair it with bluetooth!");
}

void loop() {
  if (Serial.available()) {
    SerialBT.write(Serial.read());
  }
  if (SerialBT.available()) {
    Serial.write(SerialBT.read());
  }
  delay(20);
}
```

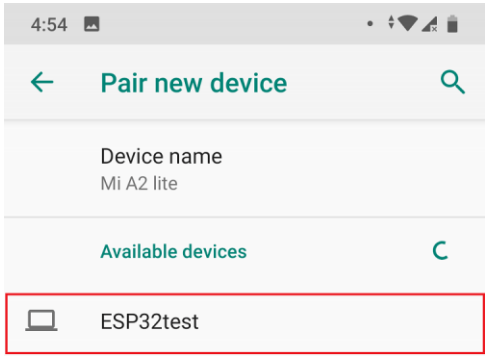


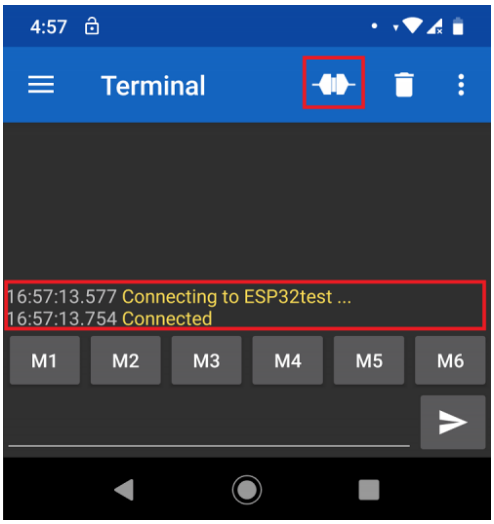
Accédez à votre smartphone et ouvrez l'application **Serial Bluetooth Terminal**. Assurez-vous d'avoir activé le Bluetooth de votre smartphone. Pour vous connecter à l'ESP32 pour la première fois, vous devez coupler un nouvel appareil.

Accédez à **Devices**.

Click the settings icon, and select **Pair new device**. You should get a list with the available Bluetooth devices, including the **ESP32test**. Pair with the **ESP32test**.

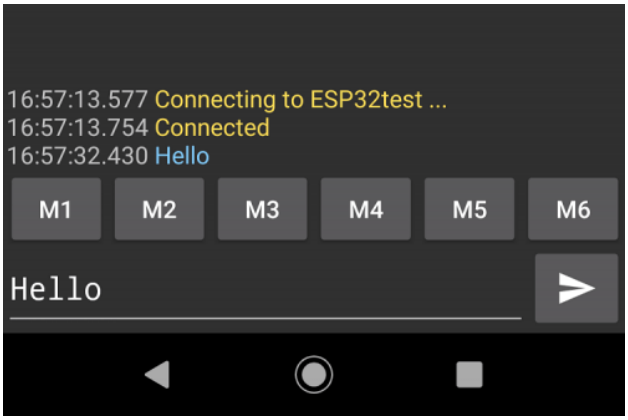
Cliquez sur l'icône des paramètres et sélectionnez **Pair new device**. Vous devriez obtenir une liste des appareils Bluetooth disponibles, y compris **l'`ESP32test`**. Associez-le à l'**`ESP32test`**.





Revenez ensuite au terminal Bluetooth série. Cliquez sur l'icône en haut pour vous connecter à l'ESP32. Vous devriez recevoir un message **Connected**.

Après cela, tapez quelque chose dans l'application **Serial Bluetooth Terminal**. Par exemple, «**Hello**».



11.2 Exchange data using Bluetooth Serial and your smartphone

Now that you know how to exchange data using Bluetooth Serial, you can modify the previous sketch to make something useful. For example, control the ESP32 outputs when you receive a certain message, or send data to your smartphone like sensor readings.

The project we'll build sends temperature readings every 10 seconds to your smartphone. We'll be using the **SHT21** temperature/humidity sensor.

Through the Android app, we'll send messages to control an ESP32 output. When the ESP32 receives the **led_on** message, we'll turn the GPIO on, when it receives the **led_off** message, we'll turn the LED **GPIO** off.

Before proceeding with this project, assemble the circuit by following the next schematic diagram.

11.2 Échange de données à l'aide de Bluetooth Serial et de votre smartphone

Maintenant que vous savez comment échanger des données à l'aide de Bluetooth Serial, vous pouvez modifier l'esquisse précédente pour en faire quelque chose d'utile. Par exemple, contrôlez les sorties ESP32 lorsque vous recevez un certain message ou envoyez des données à votre smartphone comme des lectures de capteur.

Le projet que nous allons créer envoie des relevés de température toutes les 10 secondes à votre smartphone. Nous utiliserons le capteur de température/humidité **SHT21**.

Grâce à l'application Android, nous enverrons des messages pour contrôler une sortie ESP32. Lorsque l'ESP32 reçoit le message **`led_on`**, nous allumons le GPIO, lorsqu'il reçoit le message **`led_off`**, nous éteignons le LED GPIO (pin **25**).

11.2.1 Le code

```
#include "BluetoothSerial.h"
#include <Wire.h>
#include <Sodaq_SHT2x.h>
#include <U8x8lib.h>
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15, 4, 16);
// Check if Bluetooth configs are enabled
#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

BluetoothSerial SerialBT;  // Bluetooth Serial object

const int ledPin =  25;   // GPIO where LED is connected to

String message = "";
char incomingChar;
String temperatureString = "";
// Timer: auxiliar variables
unsigned long previousMillis = 0;// Stores last time temperature was published
const long interval = 10000;   // interval at which to publish sensor readings

void setup() {
  pinMode(ledPin, OUTPUT);
  Wire.begin(21,22);
  Serial.begin(9600);
  SerialBT.begin("ESP32");
  u8x8.begin();  // initialize OLED
  u8x8.setFont(u8x8_font_chroma48medium8_r);
  u8x8.clear();
  u8x8.drawString(0,0,"Start BT:ESP32");
  Serial.println("The device started, now you can pair it with bluetooth!");
}

void loop() {
  char cbuff[40]; int i=0;
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval){
    previousMillis = currentMillis;
    temperatureString = String(SHT2x.GetTemperature()) + "C";
    SerialBT.println(temperatureString);
  }
  // Read received messages (LED control command)
  i=0;
  if (SerialBT.available()){
    char incomingChar = SerialBT.read();
    if (incomingChar != '\n'){
      message += String(incomingChar);
    }
    else{
      message = "";
    }
    Serial.write(incomingChar); message.toCharArray(cbuff,40);
```

```
  }
  // Check received message and control output accordingly
  u8x8.drawString(0, 3,"Received");
  u8x8.drawString(0, 5, cbuff);
  if (message =="led_on"){
    digitalWrite(ledPin, HIGH); Serial.println("LED-ON");
  }
  else if (message =="led_off"){
    digitalWrite(ledPin, LOW);Serial.println("LED-OFF");
  }
  delay(20);
}
```

A faire:

Installez l'application et testez le programme ci-dessus.

11.2 Bluetooth Low Energy (BLE) avec ESP32

Cette partie du laboratoire commence par une introduction rapide au BLE avec l'ESP32. Tout d'abord, nous explorerons ce qu'est le BLE et à quoi il peut être utilisé, puis nous examinerons quelques exemples avec l'ESP32 utilisant Arduino IDE. Pour une introduction simple, nous allons créer un serveur ESP32 BLE et un scanner ESP32 BLE pour trouver ce serveur.

11.2.1 Qu'est-ce que Bluetooth Low Energy?

Bluetooth Low Energy, BLE pour faire court, est une variante d'économie d'énergie de Bluetooth. L'application principale de BLE est la transmission à courte distance de petites quantités de données (faible bande passante). Contrairement à Bluetooth qui est toujours activé, BLE reste **constamment en mode veille**, sauf lorsqu'une connexion est établie. Cela lui fait consommer très peu d'énergie. Le BLE consomme environ 100 fois moins d'énergie que le Bluetooth. C'est pourquoi c'est le choix parfait pour la communication à courte distance avec et entre les appareils IoT.

De plus, BLE prend en charge non seulement la communication point à point, mais également le mode de diffusion et le réseau maillé (**mesh**).

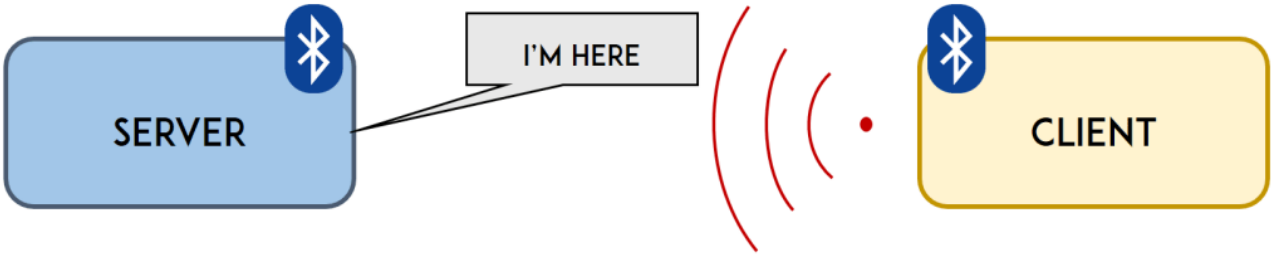
Le tableau ci-dessous compare BLE et Bluetooth Classic plus en détail.

	Bluetooth Low Energy (LE)	Bluetooth Basic Rate/ Enhanced Data Rate (BR/EDR)
Optimized For...	Short burst data transmission	Continuous data streaming
Frequency Band	2.4GHz ISM Band (2.402 – 2.480 GHz Utilized)	2.4GHz ISM Band (2.402 – 2.480 GHz Utilized)
Channels	40 channels with 2 MHz spacing (3 advertising channels/37 data channels)	79 channels with 1 MHz spacing
Channel Usage	Frequency-Hopping Spread Spectrum (FHSS)	Frequency-Hopping Spread Spectrum (FHSS)
Modulation	GFSK	GFSK, π/4 DQPSK, 8DPSK
Power Consumption	~0.01x to 0.5x of reference (depending on use case)	1 (reference value)
Data Rate	LE 2M PHY: 2 Mb/s LE 1M PHY: 1 Mb/s LE Coded PHY (S=2): 500 Kb/s LE Coded PHY (S=8): 125 Kb/s	EDR PHY (8DPSK): 3 Mb/s EDR PHY (π/4 DQPSK): 2 Mb/s BR PHY (GFSK): 1 Mb/s
Max Tx Power*	Class 1: 100 mW (+20 dBm) Class 1.5: 10 mW (+10 dbm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)	Class 1: 100 mW (+20 dBm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)
Network Topologies	Point-to-Point (including piconet) Broadcast Mesh	Point-to-Point (including piconet)

En raison de ses propriétés, BLE convient aux applications qui ont besoin d'échanger de petites quantités de données s'exécutant périodiquement sur une pile bouton. Par exemple, BLE est d'une grande utilité dans les secteurs de la santé, du fitness, du *tracking*, des balises (*beacons*), de la sécurité et de la domotique.

11.2.2 Server BLE (notifier) et Client BLE

Avec Bluetooth Low Energy, il existe deux types d'appareils: le **serveur** et le **client**. L'ESP32 peut faire office de client ou de serveur.



Le serveur annonce son existence, il peut donc être trouvé par d'autres appareils et il contient les données que le client peut lire. Le client analyse les périphériques à proximité et lorsqu'il trouve le serveur qu'il recherche, il établit une connexion et écoute les données entrantes. C'est ce qu'on appelle la communication **point à point**.

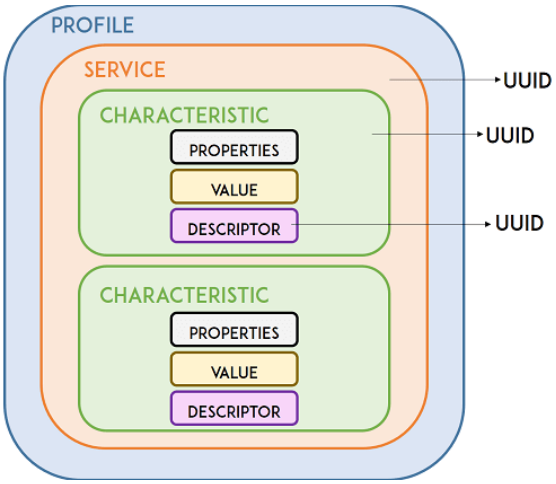
Comme mentionné précédemment, BLE prend également en charge le mode de **diffusion** et le réseau maillé:

- Mode diffusion: le serveur transmet des données à de nombreux clients connectés;
- Réseau maillé: tous les appareils sont connectés, il s'agit d'une connexion **plusieurs à plusieurs**.

Même si les configurations de réseau de diffusion et de maillage sont possibles à implémenter, elles ont été développées très récemment, donc il n'y a pas beaucoup d'exemples implémentés pour l'ESP32 en ce moment.

11.2.3 GATT

GATT signifie **Generic Attributes** et définit une structure de données hiérarchique qui est exposée aux périphériques BLE connectés. Cela signifie que GATT définit la façon dont deux appareils BLE envoient et reçoivent des messages standard. La compréhension de cette hiérarchie est importante, car elle facilitera la compréhension de l'utilisation du BLE et de l'écriture de vos applications.



11.2.4 Services BLE

Le niveau supérieur de la hiérarchie est un profil, qui est composé d'un ou plusieurs services. Habituellement, un appareil BLE contient plusieurs services.

Chaque service contient au moins une caractéristique et peut également référencer d'autres services. Un **service** est simplement une collection d'informations, comme les **lectures de capteurs**, par exemple.

Il existe des **services prédéfinis** pour plusieurs types de données définies par le SIG (Bluetooth **Special Interest Group**) comme: le niveau de la batterie, la pression artérielle, la fréquence cardiaque, l'échelle de poids, etc. Vous pouvez consulter ici d'autres services définis.

Name	Uniform Type Identifier	Assigned Number	Specification
Generic Access	org.bluetooth.service.generic_access	0x1800	GSS
Alert Notification Service	org.bluetooth.service.alert_notification	0x1811	GSS
Automation IO	org.bluetooth.service.automation_io	0x1815	GSS
Battery Service	org.bluetooth.service.battery_service	0x180F	GSS
Blood Pressure	org.bluetooth.service.blood_pressure	0x1810	GSS
Body Composition	org.bluetooth.service.body_composition	0x181B	GSS
Bond Management Service	org.bluetooth.service.bond_management	0x181E	GSS
Continuous Glucose Monitoring	org.bluetooth.service.continuous_glucose_monitoring	0x181F	GSS
Current Time Service	org.bluetooth.service.current_time	0x1805	GSS
Cycling Power	org.bluetooth.service.cycling_power	0x1818	GSS
Cycling Speed and Cadence	org.bluetooth.service.cycling_speed_and_cadence	0x1816	GSS
Device Information	org.bluetooth.service.device_information	0x180A	GSS
Environmental Sensing	org.bluetooth.service.environmental_sensing	0x181A	GSS
Fitness Machine	org.bluetooth.service.fitness_machine	0x1826	GSS
Generic Attribute	org.bluetooth.service.generic_attribute	0x1801	GSS

11.2.5 Caractéristiques BLE

La caractéristique appartient toujours à un service et c'est là que les données réelles sont contenues dans la hiérarchie (valeur). La caractéristique a toujours deux attributs: la déclaration de caractéristique (qui fournit des métadonnées sur les données) et la valeur de caractéristique.

De plus, la valeur de caractéristique peut être suivie de descripteurs, qui développent davantage les métadonnées contenues dans la déclaration de caractéristique.

Les propriétés décrivent comment peut interagir avec la valeur de caractéristique. Fondamentalement, ils contiennent les opérations et procédures pouvant être utilisées avec la caractéristique:

- Diffusion
- Lecture
- Écriture sans réponse
- Écriture
- Notification
- Écritures authentifiées et signées
- Propriétés étendues

11.2.5.1 UUID

Chaque service, caractéristique et descripteur possède un **UUID** (**Universally Unique Identifier**). Un **UUID** est un numéro unique de **128 bits** (16 octets).

Par exemple:

55072829-bc9e-4c53-938a-74a6d4c78776

Il existe des UUID raccourcis pour tous les types, services et profils spécifiés dans le SIG (Bluetooth **Special Interest Group**). Mais si votre application a besoin de son **propre UUID**, vous pouvez le générer à l'aide de ce site Web de générateur d'UUID.

https://www.uuidgenerator.net/

En résumé, l'UUID est utilisé pour identifier de manière unique les informations. Par exemple, il peut identifier un service particulier fourni par un appareil Bluetooth.

11.6 BLE avec ESP32

L'ESP32 peut agir comme un serveur BLE ou comme un client BLE. Il existe plusieurs exemples BLE pour l'ESP32 dans la bibliothèque ESP32 BLE pour Arduino IDE. Cette bibliothèque est installée par défaut lorsque vous installez l'ESP32 sur l'IDE Arduino.

Dans votre Arduino IDE, vous pouvez aller dans **File** > **Examples** > **ESP32 BLE Arduino** et explorer les exemples fournis avec la bibliothèque BLE.

Remarque: pour voir les exemples ESP32, vous devez avoir la carte ESP32 sélectionnée dans **Tools** > **Board ,** par exemple : **(Heltec LoRa WiFi)**.

Pour une brève introduction à l'ESP32 avec BLE sur l'IDE Arduino, nous allons créer un serveur ESP32 BLE, puis un scanner ESP32 BLE pour trouver ce serveur. Nous utiliserons et expliquerons les exemples fournis avec la bibliothèque BLE.

11.6.1 ESP32 : Server BLE - notifier

Pour créer un serveur ESP32 BLE, ouvrez votre Arduino IDE et allez dans **File** > **Examples ESP32 BLE** Arduino et sélectionnez l'exemple **BLE_server**. Le code suivant devrait se charger:

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"

void setup() {
  Serial.begin(9600);
  Serial.println("Starting BLE work!");

  BLEDevice::init("Long name works now");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService(SERVICE_UUID);
  BLECharacteristic *pCharacteristic = pService->createCharacteristic(
                                         CHARACTERISTIC_UUID,
                                         BLECharacteristic::PROPERTY_READ |
                                         BLECharacteristic::PROPERTY_WRITE
                                       );

  pCharacteristic->setValue("Hello World says Smartcomputerlab");
  pService->start();
  // BLEAdvertising *pAdvertising = pServer->getAdvertising();  // this still is
working for backward compatibility
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(true);
  pAdvertising->setMinPreferred(0x06);  // functions that help with iPhone
connections issue
```

```
  pAdvertising->setMinPreferred(0x12);
  BLEDevice::startAdvertising();
  Serial.println("Characteristic defined! Now you can read it in your phone!");
}

void loop() {
  // put your main code here, to run repeatedly:
  delay(2000);
}
```

Comment fonctionne le code

Voyons rapidement comment fonctionne l'exemple de code du serveur BLE. Il commence par importer les bibliothèques nécessaires pour les capacités BLE.

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
```

Ensuite, vous devez définir un **UUID** pour le service et la caractéristique (**Service** , **Characteristic**) .

```
#define SERVICE_UUID "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"
```

Vous pouvez laisser les UUID par défaut ou vous rendre sur **uuidgenerator.net** pour créer des **UUID** aléatoires pour vos services et caractéristiques.

Dans la fonction **setup()**, il démarre la communication série à un débit de 9600 bauds.

```
Serial.begin(9600);
```

Ensuite, vous créez un périphérique **BLE** appelé "**MyESP32**". Vous pouvez changer ce nom en ce que vous voulez.

```
// Create the BLE Device
BLEDevice::init("MyESP32");
```

Dans la ligne suivante, vous définissez le périphérique **BLE** en tant que **serveur**.

```
BLEServer *pServer = BLEDevice::createServer();
```

Après cela, vous créez un service pour le serveur BLE avec l'UUID défini précédemment.

```
BLEService *pService = pServer->createService(SERVICE_UUID);
```

Ensuite, vous définissez la caractéristique de ce service. Comme vous pouvez le voir, vous utilisez également l'UUID défini précédemment et vous devez passer comme **arguments** les propriétés de la caractéristique. Dans ce cas, c'est: **READ** et **WRITE**.

```
BLECharacteristic *pCharacteristic = pService->createCharacteristic(
                                     CHARACTERISTIC_UUID,
                                     BLECharacteristic::PROPERTY_READ |
                                     BLECharacteristic::PROPERTY_WRITE
                                     );
```

Après avoir créé la caractéristique, vous pouvez définir sa **valeur** avec la méthode **setValue ()**.

```
pCharacteristic->setValue("Hello World from Smartcomputerlab");
```

Dans ce cas, nous définissons la valeur sur le texte **`Hello World from Smartcomputerlab`**. Vous pouvez changer ce texte comme bon vous semble. Dans les projets futurs, ce texte peut être une lecture de capteur, ou l'état d'une lampe, par exemple.

Enfin, vous pouvez démarrer le service et la publicité (*advertising*), afin que d'autres appareils BLE puissent numériser et trouver cet appareil BLE.

```
BLEAdvertising *pAdvertising = pServer->getAdvertising();
pAdvertising->start();
```

Ceci est juste un exemple simple sur la façon de créer un serveur BLE. Dans ce code, rien n'est fait dans la fonction (tache) **`loop()`**, mais vous pouvez ajouter une action lorsqu'un nouveau client se connecte.

11.6.2 ESP32 : Scanner BLE

La création d'un scanner ESP32 BLE est simple. Prenez un autre ESP32 (pendant que l'autre exécute l'esquisse du serveur BLE). Dans votre Arduino IDE, allez dans **`File>Examples`**>**`ESP32 BLE Arduino`** et sélectionnez l'exemple BLE_scan. Le code suivant devrait se charger.

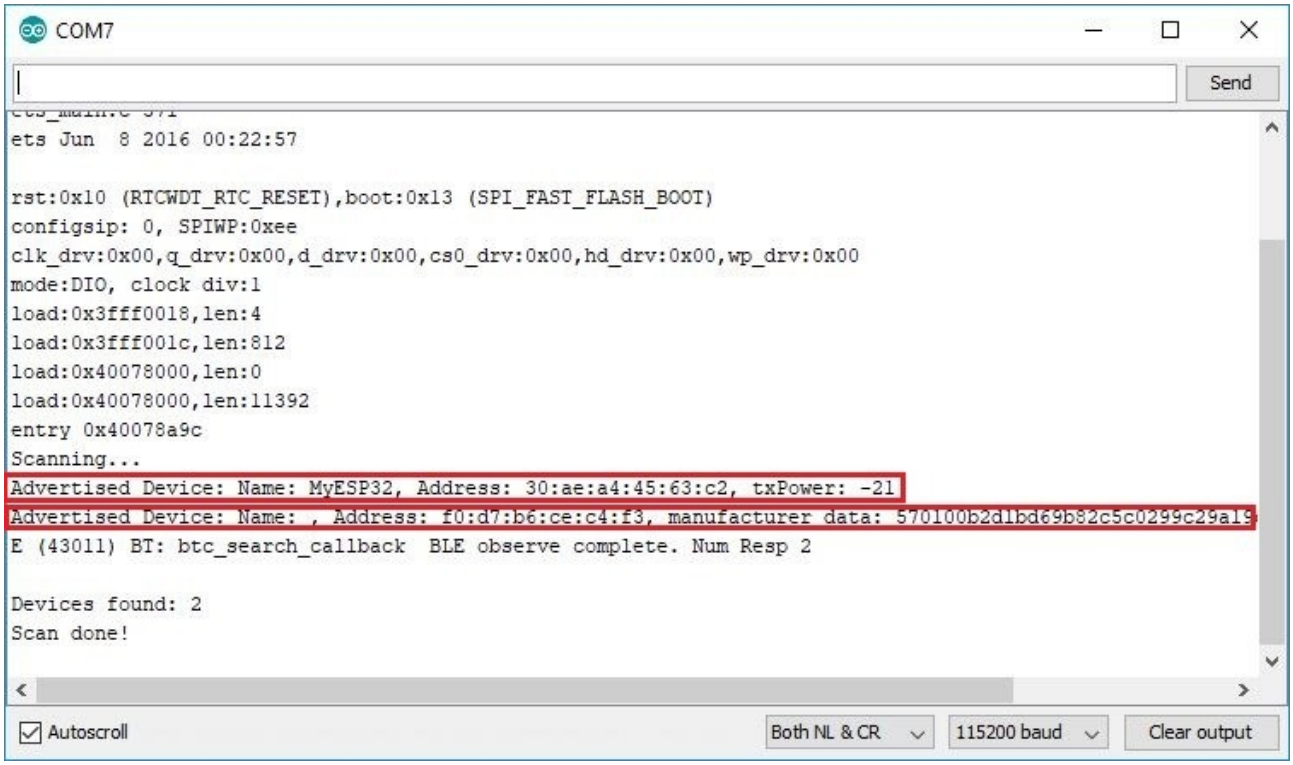
```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEScan.h>
#include <BLEAdvertisedDevice.h>
int scanTime = 5; //In seconds
BLEScan* pBLEScan;

class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {
    void onResult(BLEAdvertisedDevice advertisedDevice) {
      Serial.printf("Advertised Device: %s \n",
advertisedDevice.toString().c_str());
    }
};

void setup() {
  Serial.begin(9600);
  Serial.println("Scanning...");

  BLEDevice::init("");
  pBLEScan = BLEDevice::getScan(); //create new scan
  pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
  pBLEScan->setActiveScan(true); //active scan uses more power, but get results
faster
  pBLEScan->setInterval(100);
  pBLEScan->setWindow(99);  // less or equal setInterval value
}

void loop() {
  // put your main code here, to run repeatedly:
  BLEScanResults foundDevices = pBLEScan->start(scanTime, false);
  Serial.print("Devices found: ");
  Serial.println(foundDevices.getCount());
  Serial.println("Scan done!");
  pBLEScan->clearResults();   // delete results fromBLEScan buffer to release
memory
  delay(2000);
}
```

Ce code initialise l'ESP32 en tant que périphérique BLE et recherche les périphériques à proximité. Téléchargez ce code sur votre ESP32. Vous pouvez déconnecter l'autre ESP32 de votre ordinateur et l'alimenter par la batterie de DevKit, vous êtes donc sûr de télécharger le code sur la bonne carte ESP32.

Une fois le code téléchargé et vous devriez avoir les deux cartes ESP32 sous tension:

- Un ESP32 avec l'esquisse **`BLE_server`**;
- Autre avec esquisse ESP32 **`BLE_scan`**.

Accédez au moniteur série avec l'ESP32 exécutant l'exemple «**`BLE_scan`**», appuyez sur le bouton ACTIVER ESP32 (avec l'esquisse «**`BLE_scan`**») pour redémarrer et attendez quelques secondes pendant la numérisation.

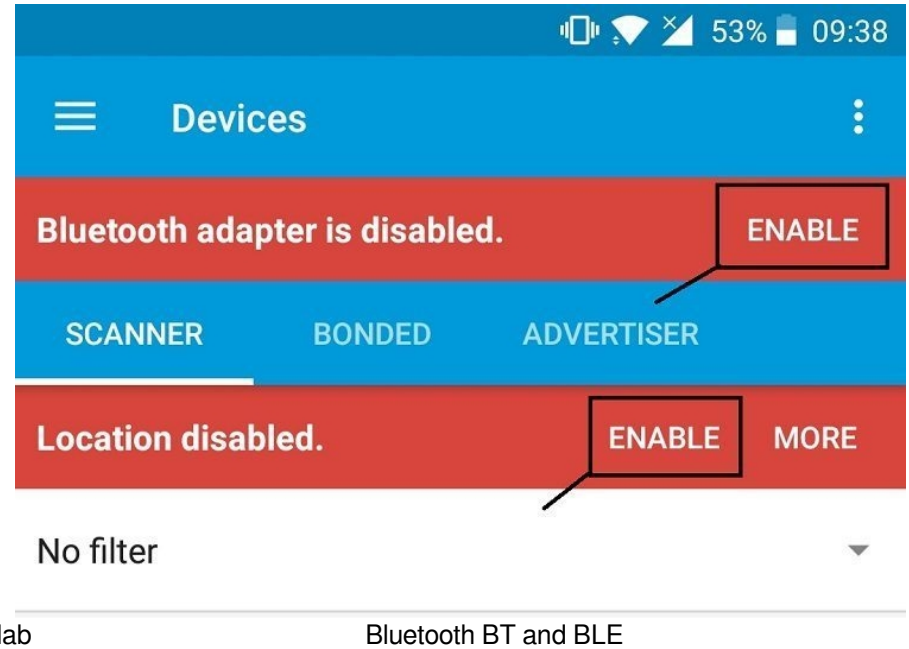


11.6.3 Test du serveur ESP32 BLE avec votre smartphone

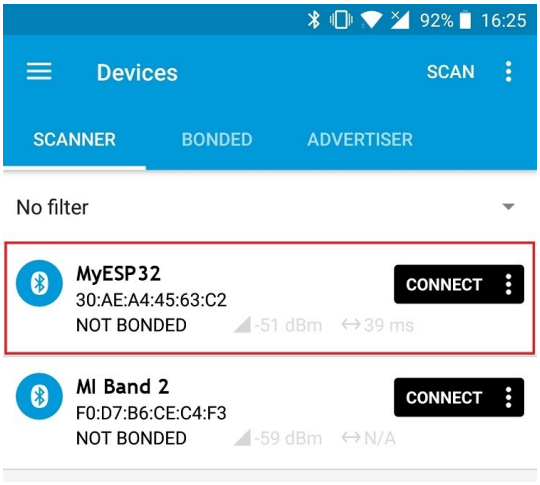
La plupart des smartphones modernes devraient avoir des capacités BLE. Vous pouvez scanner votre serveur ESP32 BLE avec votre smartphone et voir ses services et caractéristiques. Pour cela, nous utiliserons une application gratuite appelée **nRF Connect** pour mobile de **Nordic**, elle fonctionne sur **Android** (Google Play Store) et **iOS** (App Store).

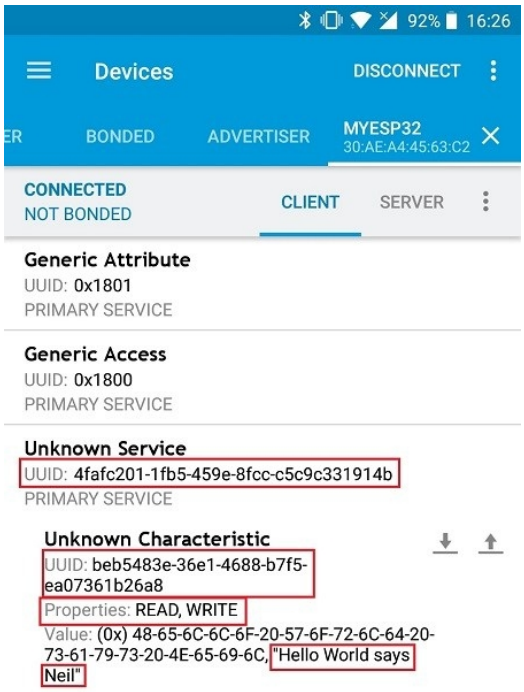
Accédez à **Google Play Store** ou App Store et recherchez «**`nRF Connect for Mobile`**». Installez l'application et ouvrez-la.

N'oubliez pas d'aller dans les paramètres Bluetooth et d'**activer l'adaptateur Bluetooth** sur votre smartphone. Vous pouvez également vouloir le rendre visible aux autres appareils pour tester d'autres croquis plus tard.



Une fois que tout est prêt dans votre smartphone et que l'ESP32 exécute l'esquisse du serveur BLE, dans l'application, appuyez sur le bouton de **SCANNER** pour rechercher les appareils à proximité. Vous devriez trouver un ESP32 avec le nom "**`MyESP32`**".





Cliquez sur le bouton **Connect**.

Comme vous pouvez le voir dans la figure ci-dessous, l'ESP32 dispose d'un **service** avec **l'UUID** que vous avez défini précédemment. Si vous appuyez sur le service, il élargit le menu et affiche la **caractéristique** avec **l'UUID** que vous avez également défini.

La caractéristique a les propriétés **`READ`** et **`WRITE`**, et la valeur est celle que vous avez précédemment définie dans l'esquisse du serveur BLE. Donc, tout fonctionne bien.

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"
#include <U8x8lib.h>  // bibliothèque à charger a partir de
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15, 4, 16);
char texte[40];
bool received = false;
char car;
int var=0;

class MyCallbacks:
    public BLECharacteristicCallbacks {
    void onWrite(BLECharacteristic *pCharacteristic) {
      std::string value = pCharacteristic->getValue();
      int i;
      if (value.length() > 0) {
        Serial.println("*********");
        Serial.print("New value: ");
        for (int i = 0; i < value.length(); i++)
          {
            Serial.print(value[i]);
            }
        delay(1000);
        Serial.println("test");
        //received = true;
        for(i=0;i< value.length();i++) texte[i]=(char)value[i];
      }
      else
        {
```

```
        Serial.println();
        Serial.println("*********");
      }
    }
};

void setup() {
  Serial.begin(9600);
  Serial.println("1- Download and install an BLE scanner app in your phone");
  Serial.println("2- Scan for BLE devices in the app");
  Serial.println("3- Connect to MyESP32");
  Serial.println("4- Go to CUSTOM CHARACTERISTIC in CUSTOM SERVICE and write
something");
  Serial.println("5- See the magic =)");
  u8x8.begin();  // initialize OLED
  u8x8.setFont(u8x8_font_chroma48medium8_r);
  u8x8.clear();
  u8x8.drawString(0,0,"Start BLE");
  BLEDevice::init("MyESP32");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService(SERVICE_UUID);
  BLECharacteristic *pCharacteristic = pService->createCharacteristic(
                                         CHARACTERISTIC_UUID,
                                         BLECharacteristic::PROPERTY_READ |
                                         BLECharacteristic::PROPERTY_WRITE
                                       );
  pCharacteristic->setCallbacks(new MyCallbacks());
  pCharacteristic->setValue("Hello World");
  pService->start();
  BLEAdvertising *pAdvertising = pServer->getAdvertising();
  pAdvertising->start();
}

void loop() {
  //xQueueReceive( xQueue, texte, 1000 );
  if(texte[0])  { Serial.println("received"); Serial.println(texte);}
  u8x8.clear();
  u8x8.drawString(0, 2,"Received");
  u8x8.drawString(0, 4,texte);
  delay(5000);
}
```

11.6.4 Serveur BLE avec un capteur– et fonction `notify`

L'exemple suivant active les fonctions du serveur puis il attend la notification du client. Il se connecte ensuite au client et envoie les données capturées à partir d'un capteur type **SHT21**.

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <U8x8lib.h>  // bibliothèque à charger a partir de
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15, 4, 16);
#include <Wire.h>
#include <Sodaq_SHT2x.h>

BLEServer* pServer = NULL;
BLECharacteristic* pCharacteristic = NULL;
bool deviceConnected = false;
bool oldDeviceConnected = false;
uint32_t value[4];

#define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"

class MyServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
      deviceConnected = true;
    };
```

```
    void onDisconnect(BLEServer* pServer) {
      deviceConnected = false;
    }
};

void setup() {
  Serial.begin(9600);
  Wire.begin(21,22);
  u8x8.begin();  // initialize OLED
  u8x8.setFont(u8x8_font_chroma48medium8_r);
  u8x8.clear();
  u8x8.drawString(0,0,"Start BLE");
  // Create the BLE Device
  BLEDevice::init("ESP32");
  // Create the BLE Server
  pServer = BLEDevice::createServer();
  pServer->setCallbacks(new MyServerCallbacks());
  // Create the BLE Service
  BLEService *pService = pServer->createService(SERVICE_UUID);
  // Create a BLE Characteristic
  pCharacteristic = pService->createCharacteristic(
                      CHARACTERISTIC_UUID,
                      BLECharacteristic::PROPERTY_READ   |
                      BLECharacteristic::PROPERTY_WRITE  |
                      BLECharacteristic::PROPERTY_NOTIFY |
                      BLECharacteristic::PROPERTY_INDICATE
                    );
  // Create a BLE Descriptor
  pCharacteristic->addDescriptor(new BLE2902());
  // Start the service
  pService->start();
  // Start advertising
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(false);
  pAdvertising->setMinPreferred(0x0);  // 0x00 to not advertise this parameter
  BLEDevice::startAdvertising();
  Serial.println("Waiting a client connection to notify...");
  u8x8.drawString(0,1,"Waiting for ");
  u8x8.drawString(0,2,"client to notify");
}

union {
  uint8_t frame[16];
  float sensor[4];
} sfr;

void loop() {
  char cbuff[40];
    // notify changed value
    if (deviceConnected) {
        pCharacteristic->setValue((uint8_t*)&value, 16);
        pCharacteristic->notify();
        memcpy(&value,sfr.frame,16);
        sfr.sensor[0]= SHT2x.GetTemperature();
        u8x8.clear();
        u8x8.drawString(0, 1,"Send");
        sprintf(cbuff,"%2.2f",sfr.sensor[0]);
        u8x8.drawString(0, 3,cbuff);
        sprintf(cbuff,"%2.2f",sfr.sensor[1]);
        u8x8.drawString(0, 4,cbuff);
        sprintf(cbuff,"%2.2f",sfr.sensor[2]);
        u8x8.drawString(0, 5,cbuff);
        sprintf(cbuff,"%2.2f",sfr.sensor[3]);
        u8x8.drawString(0, 6,cbuff);
        delay(10000); // 10 seconds between the messages
    }
    // disconnecting
    if (!deviceConnected && oldDeviceConnected) {
        delay(2000); // give the bluetooth stack the chance to get things ready
```

```
        pServer->startAdvertising(); // restart advertising
        Serial.println("start advertising");
        oldDeviceConnected = deviceConnected;
    }
    // connecting
    if (deviceConnected && !oldDeviceConnected) {
        // do stuff here on connecting
        oldDeviceConnected = deviceConnected;
    }
}
```

11.6.5 Client BLE – réception et affichage des données sur l'écran OLED

Le code suivant fonctionne avec le serveur BLE - notifier. Il notifie sa présence et attend les données. Les données reçues sont affichées sur l'écran OLED de la carte ESP32.

```
#include "BLEDevice.h"
#include <U8x8lib.h>  // bibliothèque à charger a partir de
U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15, 4, 16);

// The remote service we wish to connect to.
static BLEUUID serviceUUID("4fafc201-1fb5-459e-8fcc-c5c9c331914b");
// The characteristic of the remote service we are interested in.
static BLEUUID    charUUID("beb5483e-36e1-4688-b7f5-ea07361b26a8");

static boolean doConnect = false;
static boolean connected = false;
static boolean doScan = false;
static BLERemoteCharacteristic* pRemoteCharacteristic;
static BLEAdvertisedDevice* myDevice;

  union {
        uint8_t value[16];
        float sensor[4];
  } sfr;

static void notifyCallback(
  BLERemoteCharacteristic* pBLERemoteCharacteristic, uint8_t* pData, size_t
length,bool isNotify)
  {
    char cbuff0[18];char cbuff1[18];char cbuff2[18];char cbuff3[18];
    Serial.print("Notify callback for characteristic ");
    Serial.print(pBLERemoteCharacteristic->getUUID().toString().c_str());
    Serial.print(" of data length ");
    Serial.println(length);
    memcpy(sfr.value,pData,length);
    Serial.println(sfr.sensor[0]);Serial.println(sfr.sensor[1]);
    Serial.println(sfr.sensor[2]);Serial.println(sfr.sensor[3]);
    u8x8.clear();
    u8x8.drawString(0, 1,"Received");
    sprintf(cbuff0,"Temp:%2.2f%",sfr.sensor[0]);
    u8x8.drawString(0, 3,cbuff0);
    sprintf(cbuff1,"Humi:%2.2f%",sfr.sensor[1]);
    u8x8.drawString(0, 4,cbuff1);
    sprintf(cbuff2,"Lumi:%2.2f%",sfr.sensor[2]);
    u8x8.drawString(0, 5,cbuff2);
    sprintf(cbuff3,"CO2 :%2.2f%",sfr.sensor[3]);
    u8x8.drawString(0, 6,cbuff3);
}

class MyClientCallback : public BLEClientCallbacks {
  void onConnect(BLEClient* pclient) {
  }

  void onDisconnect(BLEClient* pclient) {
    connected = false;
    Serial.println("onDisconnect");
  }
};
```

```
bool connectToServer() {
    Serial.print("Forming a connection to ");
    Serial.println(myDevice->getAddress().toString().c_str());

    BLEClient*  pClient  = BLEDevice::createClient();
    Serial.println(" - Created client");
    pClient->setClientCallbacks(new MyClientCallback());
    // Connect to the remove BLE Server.
    pClient->connect(myDevice);
   // if you pass BLEAdvertisedDevice instead of address,
   // it will be recognized type of peer device address (public or private)
    Serial.println(" - Connected to server");
    // Obtain a reference to the service we are after in the remote BLE server.
    BLERemoteService* pRemoteService = pClient->getService(serviceUUID);
    if (pRemoteService == nullptr) {
      Serial.print("Failed to find our service UUID: ");
      Serial.println(serviceUUID.toString().c_str());
      pClient->disconnect();
      return false;
    }
    Serial.println(" - Found our service");

    // Obtain a reference to the characteristic in the service of the remote BLE
server.
    pRemoteCharacteristic = pRemoteService->getCharacteristic(charUUID);
    if (pRemoteCharacteristic == nullptr) {
      Serial.print("Failed to find our characteristic UUID: ");
      Serial.println(charUUID.toString().c_str());
      pClient->disconnect();
      return false;
    }
    Serial.println(" - Found our characteristic");
    // Read the value of the characteristic.
    if(pRemoteCharacteristic->canRead()) {
      std::string value = pRemoteCharacteristic->readValue();
      Serial.print("The characteristic value was: ");
      Serial.println(value.c_str());
    }
    if(pRemoteCharacteristic->canNotify())
      pRemoteCharacteristic->registerForNotify(notifyCallback);
    connected = true;
    return true;
}
//Scan for BLE servers and find the first one that advertises
// the service we are looking for.
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {
//Called for each advertising BLE server.

  void onResult(BLEAdvertisedDevice advertisedDevice) {
    Serial.print("BLE Advertised Device found: ");
    Serial.println(advertisedDevice.toString().c_str());
    // We have found a device
    // let us now see if it contains the service we are looking for.
    if (advertisedDevice.haveServiceUUID() &&
advertisedDevice.isAdvertisingService(serviceUUID)) {
      BLEDevice::getScan()->stop();
      myDevice = new BLEAdvertisedDevice(advertisedDevice);
      doConnect = true;
      doScan = true;
    } // Found our server
  } // onResult
}; // MyAdvertisedDeviceCallbacks

void setup() {
  Serial.begin(9600);
  Serial.println("Starting Arduino BLE Client application...");
  u8x8.begin();  // initialize OLED
  u8x8.setFont(u8x8_font_chroma48medium8_r);
  u8x8.clear();
```

```
  u8x8.drawString(0,0,"Start BLE");
  BLEDevice::init("");

// Retrieve a Scanner and set the callback we want to use to be informed when we
// have detected a new device.Specify that we want active scanning and start the
  // scan to run for 5 seconds.
  BLEScan* pBLEScan = BLEDevice::getScan();
  pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
  pBLEScan->setInterval(1349);
  pBLEScan->setWindow(449);
  pBLEScan->setActiveScan(true);
  pBLEScan->start(5, false);
} // End of setup.

// This is the Arduino main loop function.
void loop() {
// If "doConnect" is true then we have scanned for and found the desired
// BLE Server with which we wish to connect.  Now we connect to it. Once we are
// connected we set the connected flag to be true.
  if (doConnect == true) {
    if (connectToServer()) {
      Serial.println("We are now connected to the BLE Server.");
    } else {
      Serial.println("Failed to connect to the server;nothing more to do.");
    }
    doConnect = false;
  }
  // If we are connected to a peer BLE Server,
  // update the characteristic each time we are reached
  // with the current time since boot.
  if (connected) {
    String newValue = "Time since boot: " + String(millis()/1000);
    Serial.println("Setting new characteristic value to \"" + newValue + "\"");
//characteristic's value to be the array of bytes that is actually a string.
    pRemoteCharacteristic->writeValue(newValue.c_str(), newValue.length());
  }
  else if(doScan){
    BLEDevice::getScan()->start(0);
 // this is just example to start scan after disconnect,
  }
  delay(1000); // Delay a second between loops.
} // End of loop
```

11.7 Développement d'une passerelle BLE-WiFi vers serveur IoT

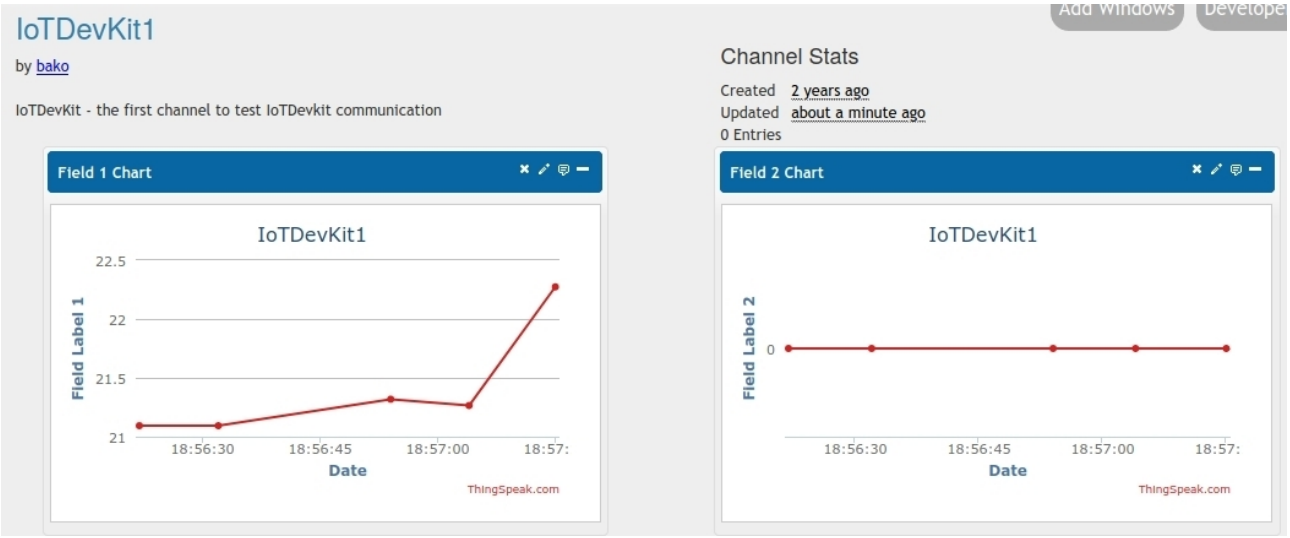
Le client BLE présenté dans l'exemple précédant reçoit les messages avec les données du capteur et les affiche sur l'écran OLED. Dans l'exemple suivant, nous montrons comment créer une simple passerelle BLE-WiFi qui envoie les données reçues au serveur **ThingSpeak**.

Notez que les messages envoyés par le Server-Notifier doivent être séparés d'au moins 10 secondes.

```
#include "BLEDevice.h"
#include <U8x8lib.h>
#include <WiFi.h>
#include "ThingSpeak.h"
//Attention this file should be modified if you use ThgingSpeak.fr server
// by putting updated IP address value

char ssid[] = "PhoneAP";          //  your network SSID (name)
char pass[] = "smartcomputerlab";                 // your network passw
unsigned long myChannelNumber = 1;
const char * myWriteAPIKey="HEU64K3PGNWG36C4" ;
WiFiClient  client;

U8X8_SSD1306_128X64_NONAME_SW_I2C u8x8(15, 4, 16);
// The remote service we wish to connect to.
static BLEUUID serviceUUID("4fafc201-1fb5-459e-8fcc-c5c9c331914b");
// The characteristic of the remote service we are interested in.
static BLEUUID    charUUID("beb5483e-36e1-4688-b7f5-ea07361b26a8");

static boolean doConnect = false;
static boolean connected = false;
static boolean doScan = false;
static BLERemoteCharacteristic* pRemoteCharacteristic;
static BLEAdvertisedDevice* myDevice;
  union {
        uint8_t value[16];
        float sensor[4];
  } sfr;

static void notifyCallback(
  //BLERemoteCharacteristic* pBLERemoteCharacteristic,uint8_t* pData,size_t
length,bool isNotify)
  BLERemoteCharacteristic* pBLERemoteCharacteristic, uint8_t* pData, size_t
length,bool isNotify)
  {
    char cbuff0[18];char cbuff1[18];char cbuff2[18];char cbuff3[18];
    Serial.print("Notify callback for characteristic ");
    Serial.print(pBLERemoteCharacteristic->getUUID().toString().c_str());
    Serial.print(" of data length ");
    Serial.println(length);
    memcpy(sfr.value,pData,length);
    Serial.println(sfr.sensor[0]);Serial.println(sfr.sensor[1]);
    Serial.println(sfr.sensor[2]);Serial.println(sfr.sensor[3]);
    u8x8.clear();
    u8x8.drawString(0, 1,"Received");
    sprintf(cbuff0,"Temp:%2.2f%",sfr.sensor[0]);
    u8x8.drawString(0, 3,cbuff0);
    sprintf(cbuff1,"Humi:%2.2f%",sfr.sensor[1]);
    u8x8.drawString(0, 4,cbuff1);
    sprintf(cbuff2,"Lumi:%2.2f%",sfr.sensor[2]);
    u8x8.drawString(0, 5,cbuff2);
    sprintf(cbuff3,"CO2 :%2.2f%",sfr.sensor[3]);
    u8x8.drawString(0, 6,cbuff3);
    if (WiFi.status() != WL_CONNECTED)
      { //if we lost connection, retry
      WiFi.begin(ssid); delay(500);
      }
    Serial.println("Connecting to ThingSpeak.fr or ThingSpeak.com");
    WiFiClient client;
    ThingSpeak.begin(client);
  Serial.println("Fields update");
  ThingSpeak.setField(1, sfr.sensor[0]);
```

```
  ThingSpeak.setField(2, sfr.sensor[1]);
  ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
}

class MyClientCallback : public BLEClientCallbacks {
  void onConnect(BLEClient* pclient) {
  }

  void onDisconnect(BLEClient* pclient) {
    connected = false;
    Serial.println("onDisconnect");
  }
};

bool connectToServer() {
    Serial.print("Forming a connection to ");
    Serial.println(myDevice->getAddress().toString().c_str());

    BLEClient*  pClient  = BLEDevice::createClient();
    Serial.println(" - Created client");
    pClient->setClientCallbacks(new MyClientCallback());
    // Connect to the remove BLE Server.
    pClient->connect(myDevice);
    // if you pass BLEAdvertisedDevice instead of address,
    // it will be recognized type of peer device address (public or private)
    Serial.println(" - Connected to server");
    // Obtain a reference to the service we are after in the remote BLE server.
    BLERemoteService* pRemoteService = pClient->getService(serviceUUID);
    if (pRemoteService == nullptr) {
      Serial.print("Failed to find our service UUID: ");
      Serial.println(serviceUUID.toString().c_str());
      pClient->disconnect();
      return false;
    }
    Serial.println(" - Found our service");
    // Obtain a reference to the characteristic in the service
    // of the remote BLE server.
    pRemoteCharacteristic = pRemoteService->getCharacteristic(charUUID);
    if (pRemoteCharacteristic == nullptr) {
      Serial.print("Failed to find our characteristic UUID: ");
      Serial.println(charUUID.toString().c_str());
      pClient->disconnect();
      return false;
    }
    Serial.println(" - Found our characteristic");
    // Read the value of the characteristic.
    if(pRemoteCharacteristic->canRead()) {
      std::string value = pRemoteCharacteristic->readValue();
      Serial.print("The characteristic value was: ");
      Serial.println(value.c_str());
    }
    if(pRemoteCharacteristic->canNotify())
      pRemoteCharacteristic->registerForNotify(notifyCallback);
    connected = true;
    return true;
}
//Scan for BLE servers and find the first one that advertises
// the service we are looking for.
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {
 // Called for each advertising BLE server.
  void onResult(BLEAdvertisedDevice advertisedDevice) {
    Serial.print("BLE Advertised Device found: ");
    Serial.println(advertisedDevice.toString().c_str());
    // We have found a device, let us now see if it contains
    // the service we are looking for.
    if (advertisedDevice.haveServiceUUID() &&
advertisedDevice.isAdvertisingService(serviceUUID)) {
      BLEDevice::getScan()->stop();
      myDevice = new BLEAdvertisedDevice(advertisedDevice);
      doConnect = true;
```

```
      doScan = true;
    } // Found our server
  } // onResult
}; // MyAdvertisedDeviceCallbacks

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");
    WiFiClient client;

  Serial.println("Starting Arduino BLE Client application...");
  u8x8.begin();  // initialize OLED
  u8x8.setFont(u8x8_font_chroma48medium8_r);
  u8x8.clear();
  u8x8.drawString(0,0,"Start BLE");
  BLEDevice::init("");
  // Retrieve a Scanner and set the callback we want to use to be informed
  // when we have detected a new device.
  // Specify that we want active scanning and start the scan to run for 5 sec
  BLEScan* pBLEScan = BLEDevice::getScan();
  pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
  pBLEScan->setInterval(1349);
  pBLEScan->setWindow(449);
  pBLEScan->setActiveScan(true);
  pBLEScan->start(5, false);
} // End of setup.

void loop() {
  // If the flag "doConnect" is true then we have scanned for and
  // found the desired  BLE Server with which we wish to connect.
  // Now we connect to it.  Once we are
  // connected we set the connected flag to be true.
  if (doConnect == true) {
    if (connectToServer()) {
      Serial.println("We are now connected to the BLE Server.");
    } else {
      Serial.println("Failed to connect to the server; nothing more to do.");
    }
    doConnect = false;
  }
  // If we are connected to a peer BLE Server, update the characteristic
  // each time we are reached with the current time since boot.
  if (connected) {
    String newValue = "Time since boot: " + String(millis()/1000);
    Serial.println("Setting new characteristic value to \"" + newValue + "\"");

    // characteristic's value - the array of bytes that is actually a string.
    pRemoteCharacteristic->writeValue(newValue.c_str(), newValue.length());
  }
  else if(doScan){
    BLEDevice::getScan()->start(0);
   // this is just example to start scan after disconnect
  }
  delay(1000); // Delay a second between loops.
} // End of loop
```



11.8 Résumé

Dans ce laboratoire, nous vous avons montré les principes de base du **Bluetooth Classic** et du **Bluetooth Low Energy** et vous avons montré quelques exemples avec l'ESP32. Nous avons exploré l'esquisse du serveur BLE et l'esquisse du client BLE.

Le dernier exemple donné est une **passerelle BLE-WiFi** qui montre comment nous pouvons exploiter ces deux modules radio pour construire des architectures IoT.

Travail à faire