Bluetooth Low Energy avec ESP32-C3 et ESP32

Il n'y a pas que le wifi dans la vie…

Mathias Claußen (Elektor)

Contrairement à l'ESP8266, l'ESP32-C3 est équipé d'une liaison de communication RF Bluetooth Low Energy. Si vous n'avez besoin d'envoyer que de petites quantités de données sur de courtes distances, cette norme est une alternative économe en énergie au Wi-Fi. Nous le démontrons ici avec un petit projet : un capteur de température/humidité avec un ESP32-C3 transmet ses données à un ESP32 équipé d'un petit écran OLED.

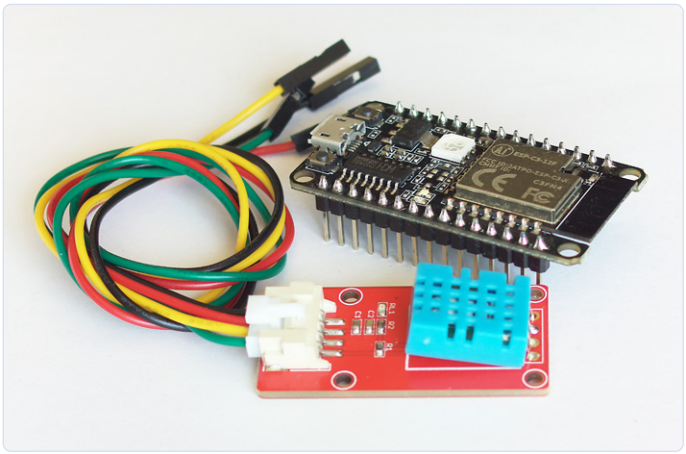

Figure 1. Tout ce dont vous avez besoin pour créer le nœud capteur …

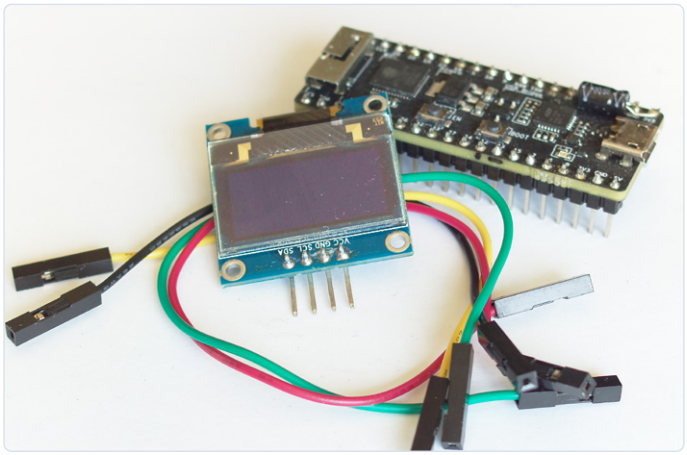

Figure 2. … et pour l'affichage.

L'ESP32-C3 [1] avec son cœur RISC-V et son rapport qualité-prix particulièrement bon peut être considéré comme le successeur du microcontrôleur ESP8266 d'Espressif. L'un des avantages que cette nouvelle puce apporte est sa capacité de communication à faible consommation Bluetooth (Bluetooth Low Energy BLE) intégrée. Le BLE permet l'échange de données entre des appareils sur une courte distance et est très économe en énergie. Cette norme de communication est idéale pour une large gamme d'applications, où de petites quantités de données doivent être envoyées sur une courte distance. Les écouteurs, les microphones, les casques ou même les montres utilisent le BLE pour se connecter à divers appareils (principalement des smartphones).

Mais, pourquoi le BLE et pas seulement le wifi ? En ce qui concerne la transmission périodique de données sur de courtes distances, le wifi est assez gourmand en énergie. De plus, le wifi est conçu pour fonctionner avec un point d'accès au sein d'un réseau Ethernet. En comparaison, un petit appareil ESP32-C3 alimenté par batterie communiquant par BLE atteindra une durée de vie de la batterie beaucoup plus longue.

Ce projet vous guidera à travers les premières étapes de l'utilisation de la communication BLE. La configuration utilise un capteur d'humidité et de température connecté à un ESP32-C3 qui envoie les valeurs mesurées à un module ESP32 où elles sont affichées sur un petit écran OLED.

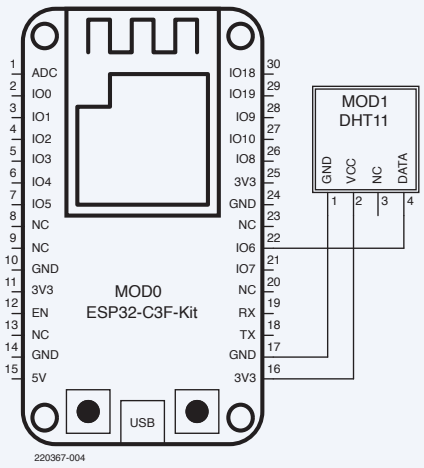

Figure 3. Schéma du circuit du nœud capteur d'humidité.

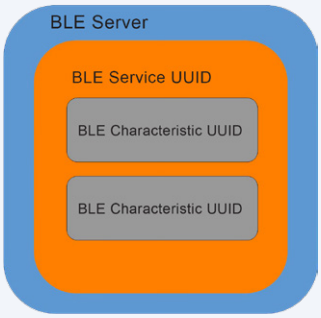

Figure 4: Structure des serveurs BLE et des UUID.

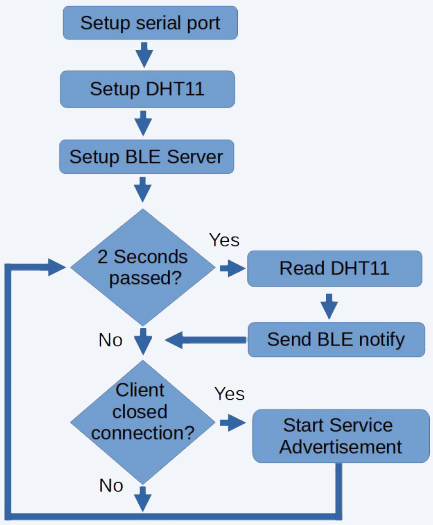

Figure 5. Organigramme du nœud capteur.

Composants

Pour ce projet, nous avons utilisé des composants standards qui peuvent être commandés sur l'e-choppe Elektor. Ils n'ont rien de spécial, vous les avez peut-être même déjà dans votre boîte de composants. Le nœud capteur contient uniquement un capteur DHT11 qui fournit à la fois les relevés de température et d'humidité. Il est contenu, avec de nombreux autres périphériques utiles, dans le coffret d'expérimentation du Raspberry Pi Pico. Le contrôleur ESP32-C3 se présente sous la forme de la carte de développement du kit ESP32-C3-12F (tous deux sont disponibles dans l'e-choppe Elektor — voir l'encadré). Tous les composants utilisés sont illustrés à la **figure 1**.

Pour l'affichage, nous utilisons un contrôleur ESP32-PICO-Kit V4 et un petit écran OLED de 0,96 pouce (voir l'encadré des produits) avec seulement quatre fils de pontage pour le câblage. Il est également possible d'utiliser un WeMos Lolin ESP32 avec un écran OLED intégré, mais les affectations des broches de l'afficheur devront être modifiées. Les pièces de l'unité d'affichage sont visibles sur la **figure 2**.

Paquets de données BLE

Le protocole de communication utilisé pour envoyer des données via BLE n'est pas compatible avec l'ancien protocole Bluetooth Classic. Avec BLE, il y a essentiellement des serveurs et des clients, qui sont tous deux capables d'échanger des paquets de données à l'aide des protocoles d'attributs (ATT) et des profils d'attributs génériques (GATT). Le GATT fournit une liste de services et de caractéristiques contenant des procédures et des attributs. Par exemple, un attribut peut représenter une valeur de capteur. Chaque attribut est adressé par un identifiant unique nommé UUID, qui peut être attribué par le développeur. Les attributs sont à leur tour regroupés en services, un ou plusieurs par serveur, qui à leur tour ont également un UUID. Un exemple de service serait le provisionnement d'un jeu de données contenant des valeurs de capteurs (température, humidité, etc.).

Avec le GATT, l'autorisation d'accès se fait par connexion, c'est-à-dire qu'aucune distinction n'est faite quant au dispositif qui établit la connexion, tant que les paramètres et les clés permettent d'établir la connexion.

Cette représentation très simplifiée du GATT devrait suffire pour ce projet. Après tout, ceci est uniquement destiné à être une introduction à l'utilisation du BLE, donc aucun mécanisme de sécurité n'est implémenté ici. Vous trouverez plus d'informations sur le BLE sur la page Bluetooth SIG [2] ou en regardant le webinaire d'Elektor sur les applications Android BLE [3]. Nous pouvons maintenant passer à la configuration de notre serveur BLE et de notre client BLE.

Serveur BLE

Avant de commencer avec le logiciel, regardons de plus près le matériel. La connexion du DHT11 à l'ESP32-C3 est illustrée à la **figure 3**. VCC se connecte à 3,3 V, GND à la masse et le signal DATA du capteur à la broche IO06 de l'ESP32-C3.

La structure du serveur BLE est visible à la **figure 4**. Il est structuré en couches, le serveur lui-même formant la couche externe. Viennent ensuite les services, qui dans ce cas n'en sont qu'un, avec l'UUID *91bad492-b950-4226-aa2b-4ede9fa42f59*. Le Service contient les caractéristiques qui seront fournies. L'une d'elles possède l'UUID *cba1d466-344c-4be3-ab3f-189f80dd7518* pour la température en degrés Celsius (°C) et l'autre l'UUID *ca73b3ba-39f6-4ab3-91ae-186dc9577d99* pour les valeurs d'humidité. Chacune de ces caractéristiques a une valeur et une description. Cette description a également un UUID et est définie ici à *0x2902* — où cette valeur indique qu'il s'agit d'une description de caractéristique. Si vous souhaitez en savoir plus sur ce sujet, reportez-vous au document « *Bluetooth low energy characteristics, a beginner's tutorial* » de Nordic Semiconductor. [4]

Dans notre logiciel, toutes les parties du serveur BLE sont d'abord configurées. Une nouvelle valeur est ensuite mesurée par le DHT11 toutes les deux secondes et envoyée aux appareils connectés sous forme de notification BLE (*Notify*). Il s'agit d'un message *push* ; le client

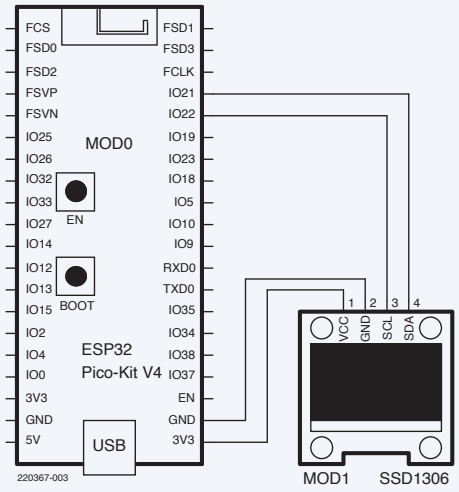

Figure 6. Le schéma du circuit d'affichage.

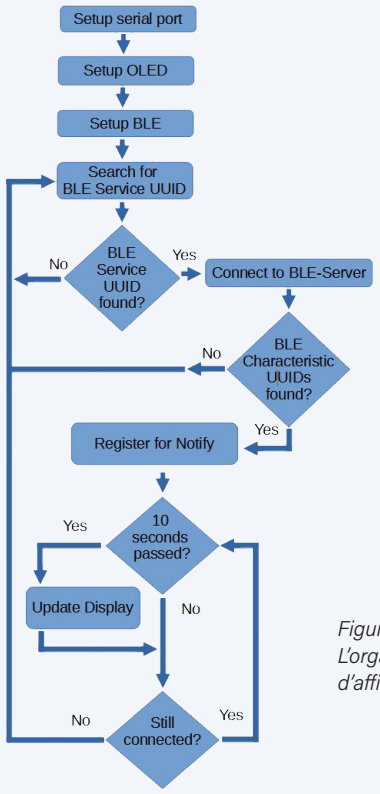

Figure 7. L'organigramme d'affichage.

n'a pas besoin de confirmer que ce dernier est arrivé. La séquence logicielle peut être vue à la **figure 5**.
Une chose qui se démarque dans le code, ce sont les appels à `delay(5);` après les appels à `notify();`. Par exemple :

```
dht11HumidityCharacteristics.setValue(String(event.
   relative_humidity).c_str());
dht11HumidityCharacteristics.notify();
delay(5);
```

Cela permet d'éviter la congestion des paquets dans la pile BLE. Malgré cela et selon la version de l'environnement Arduino pour l'ESP32 et l'ESP32-C3, il arrive parfois que la pile BLE cesse de fonctionner. Même après un `Disconnect`, nous invoquons un délai de 500 ms pour le traitement de la pile BLE. Incidemment, cela affecte (malheureusement) à la fois l'ESP32 et l'ESP32-C3, car les deux utilisent la même pile BLE par défaut.
Une alternative à la pile par défaut du Bluetooth Classic et du BLE (pile basée sur BlueDroid) [5] est « l'Apache MyNewt NimBLE » [6] qui ne peut être utilisé que pour des applications purement BLE (et non Bluetooth Classic) dans le framework Arduino [7]. Nous allons maintenant utiliser cette pile pour l'unité client, qui dans notre cas est composée d'un ESP32 avec un écran OLED.

Client BLE

Comme pour le serveur, nous allons d'abord examiner le matériel client, puis le logiciel. Ici, nous avons utilisé un ESP32-PICO-Kit, sur lequel un module ESP32 est monté pour fournir la puissance de traitement. Même si leurs noms sont similaires, ces deux modules n'ont pas le même processeur. L'ESP32 possède deux cœurs de processeur Xtensa LX6 [8], tandis que l'ESP32-C3 utilise un seul cœur basé sur RISC-V (RV32IMC). Seuls quatre fils sont nécessaires pour connecter l'écran OLED au ESP32-PICO-Kit. Le VCC de l'écran est connecté à 3,3 V, et le GND de l'écran au GND du ESP32-PICO-Kit. Ensuite, nous connectons les broches SDA et SCL pour la connexion I²C. SDA se connecte au GPIO21 et SCL au GPIO22 du ESP32-PICO-Kit. Le schéma du circuit peut être vu sur la **figure 6**.
Un organigramme logiciel du processus complet est illustré à la **figure 7**. Après le démarrage de l'ESP32, l'interface série et l'OLED sont initialisés, suivis de la pile BLE. Commence alors la recherche de nouveaux serveurs BLE pendant cinq secondes.
Si un serveur est trouvé, la fonction `onResult` de la classe `Configured_AdvertisedDeviceCallbacks` est appelée en réponse. Ici, on essaye de déterminer si le serveur offre un service avec l'UUID *91bad492-b950-4226-aa2b-4ede9fa42f59*. Si c'est le cas, la recherche d'un nouveau serveur BLE se termine et une connexion au serveur BLE trouvé est établie. Ensuite, on vérifie si le service du serveur propose les UUID pour les deux caractéristiques *cba1d466-344c-4be3-ab3f-189f80dd7518* et *ca73b3ba-39f6-4ab3-91ae-186dc9577d99* utilisés par le serveur pour identifier les lectures de température et d'humidité. Si les caractéristiques sont toutes les deux disponibles, la connexion est maintenue. Si aucun serveur ou service BLE approprié n'est trouvé, une nouvelle recherche sera lancée.
Tout d'abord, le client vérifie si les deux UUID pour la température et l'humidité peuvent également envoyer des notifications (*Notifies*). Le code suivant est utilisé à cet effet (l'exemple ici est juste pour les lectures d'humidité) :

```
pRemoteHumCharacteristic =
pRemoteService->getCharacteristic(humUUID);
…
    if (pRemoteHumCharacteristic != nullptr) {
      if(true==pRemoteHumCharacteristic->canNotify()){
        pRemoteHumCharacteristic->
        registerForNotify(NewHumNotify);
      }
…
}
```

Si des notifications peuvent être envoyées, un rappel est mis en place pour celles-ci. Chaque fois qu'une nouvelle notification arrive, la fonction `NewHumNotify` est alors appelée pour l'humidité. Une notification peut

contenir jusqu'à 20 octets de données, qui peuvent librement être attribuées et de n'importe quel format choisi. Dans notre logiciel, le serveur BLE envoie la valeur de notification sous forme de chaînes lisibles entièrement formatées. Pour les afficher à l'écran, il suffit de les préparer de manière appropriée.
Dès qu'il y a une connexion au serveur BLE, l'affichage OLED est simplement mis à jour toutes les 10 s pour afficher les dernières valeurs au fur et à mesure de leur réception. Le serveur BLE et le client BLE terminés sont visibles dans la **figure 8.**

BLE et l'ESP32/ESP32-C3

L'exemple d'application BLE décrit ici n'est pas le plus sophistiqué. L'intention est de démontrer les principes impliqués afin que vous puissiez vous familiariser avec les bases et gagner en confiance pour approfondir le sujet. Le BLE offre une option d'économie d'énergie pour le transport de données à courte portée qui ne nécessite pas l'infrastructure d'un réseau wifi.

220267-04

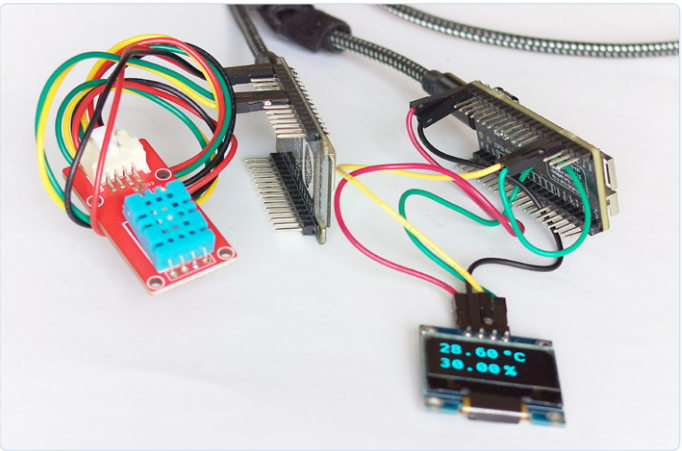

Figure 8. Le serveur et le client BLE fonctionnent ensemble.

Des questions, des commentaires ?

Envoyez un courriel à l'auteur (mathias.claussen@elektor.com) ou contactez Elektor (redaction@elektor.fr).

PRODUITS

- **Kit d'expérimentation Raspberry Pi Pico**
  www.elektor.fr/19834
- **Kit ESP32 PICO V4**
  www.elektor.fr/18423
- **Module d'affichage OLED WeMos Lolin ESP32 pour Arduino (SKU 18575)**
  www.elektor.fr/18575
- **Carte de développement ESP-C3-12F-Kit avec 4 Mo de mémoire Flash intégrée (SKU 19855)**
  www.elektor.fr/19855
- **Capteur de température et d'humidité Grove DHT11 de Seeed Studio (SKU 20020)**
  www.elektor.fr/20020
- **Écran OLED 0,96 pouce (bleu, I²C, 4 broches) (SKU 18747)**
  www.elektor.fr/18747
- ***Livre en anglais « Develop your own Bluetooth Low Energy Applications »* (SKU 20200)**
  www.elektor.fr/20200
- ***« Develop your own Bluetooth Low Energy Applications » (livre numérique, SKU 20200)***
  www.elektor.fr/20201

LIENS

[1] Mathias Claußen, « Prise en main du microcontrôleur ESP32-C3 RISC-V » : www.elektormagazine.fr/news/prise-en-main-du-microcontroleur-esp32-c3-risc-v
[2] Bluetooth SIG: « Intro to Bluetooth GAP (GATT) »: www.bluetooth.com/bluetooth-resources/intro-to-bluetooth-gap-gatt/
[3] C. Valens, « Rapid Prototyping Bluetooth Low Energy Android Apps Using MIT App Inventor », Elektor.TV, June 2021: www.youtube.com/watch?v=Jxv9h0nHIBA&t=2930s
[4] Nordic Semiconductor, « Bluetooth low energy Characteristics, a beginner's tutorial »: https://bit.ly/3sCEVzR
[5] Pile Bluetooth Classic et BLE de l'ESP32: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/bluetooth/index.html
[6] Pile ESP32 NimBLE: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/bluetooth/nimble/index.html
[7] NimBLE-Arduino: https://github.com/h2zero/NimBLE-Arduino
[8] ESP32 LX6 Core: https://en.wikipedia.org/wiki/Tensilica