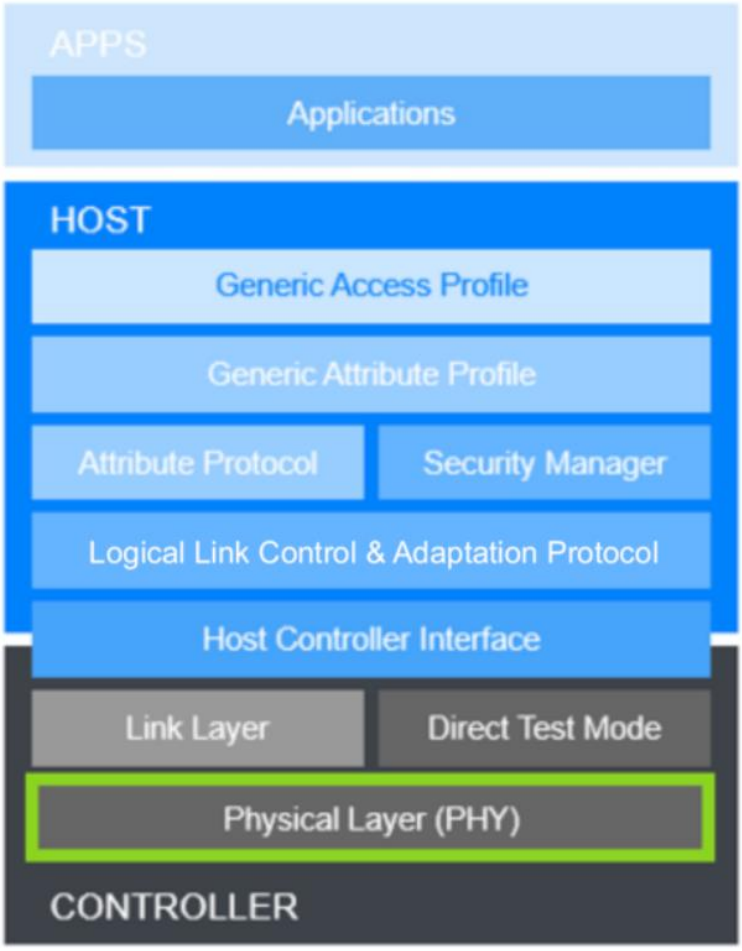
Responsable du document : Nathan BRIENT

1. PRESENTATION DU BLE

Bluetooth : norme communications permettant l'échange bidirectionnel de données à très courte distance en utilisant des ondes radio UHF sur une bande de fréquence de 2,4 GHz.

BLE (Bluetooth Low Energy): technique basé sur du bluetooth, complète mais ne le remplace pas. Permet un même débit que le bluetooth avec consommation plus faible.

Le BLE repose sur un certains nombre de couches :



SIG Bluetooth (Special Interest Group) est l'organisme qui supervise l'élaboration de normes Bluetooth et octroie les licences de la marque et de la technologie Bluetooth aux fabricants.

La couche GAP (Generic Access Profile) est responsable de l'établissement du lien et de la supervisation de connexion entre deux appareils. Il permettra aux dispositifs BLE de signaler leur présence via des advertising (notifications).

La couche GATT (Generic Attribute Profile) quant à elle apporte un ensemble de sous-procédures basées sur la couche ATT pour permettre d'orchestrer toute la gestion des données supportées par un appareil. La structure définit les éléments de base, tels que les services et les caractéristiques, utilisés dans un profil.

La couche ATT (Attribute Protocol) repose sur une relation client/serveur. Le serveur dispose des informations (telles que les valeurs des capteurs). Le client est celui qui veut accèder à ces informations. ATT permet à un serveur d'exposer à un client un ensemble d'attributs. Un attribut est une valeur associée aux trois propriétés suivantes: un type d'attribut défini par un UUID, un descripteur d'attribut et un ensemble d'autorisations.

Remarque : La couche LL (Link Layer) introduit les deux modes maître (master) et esclave (slave).

Plus d'infos sur le GAP (Generic Access Profile) :

2 types de communications avec 4 rôles possibles :

- mode advertising avec :
  - Advertiser/broadcaster : L'objet ne fait qu'émettre des données (31 octets de données utiles au maximum) sans se connecter à un autre objet. Nos balises.
  - Scanner/observer : L'objet ne fait que recevoir des données, sans se connecter à un autre objet. Il est à l'écoute des messages d'éventuels objets qui tiendraient un rôle d'advertiser. Notre MP1.

L'objet en mode scanner reste passif, il écoute simplement les émissions à intervalle régulier (advertising interval) de l'objet en mode advertiser.

- mode connecté avec :
  - Peripheral (esclave) : Objet disposant de peu de ressources (ex. capteur connecté) qui renverra des informations à un central.
  - Central (maître) : Objet disposant de ressources plus étendues (ex: smartphone) auquel se connectent les périphériques.

Une fois qu'une connexion est établie, l'advertising prend généralement fin et le protocole GATT entre en action. Les échanges sont alors limités entre le central et l'unique périphérique auquel il est connecté.

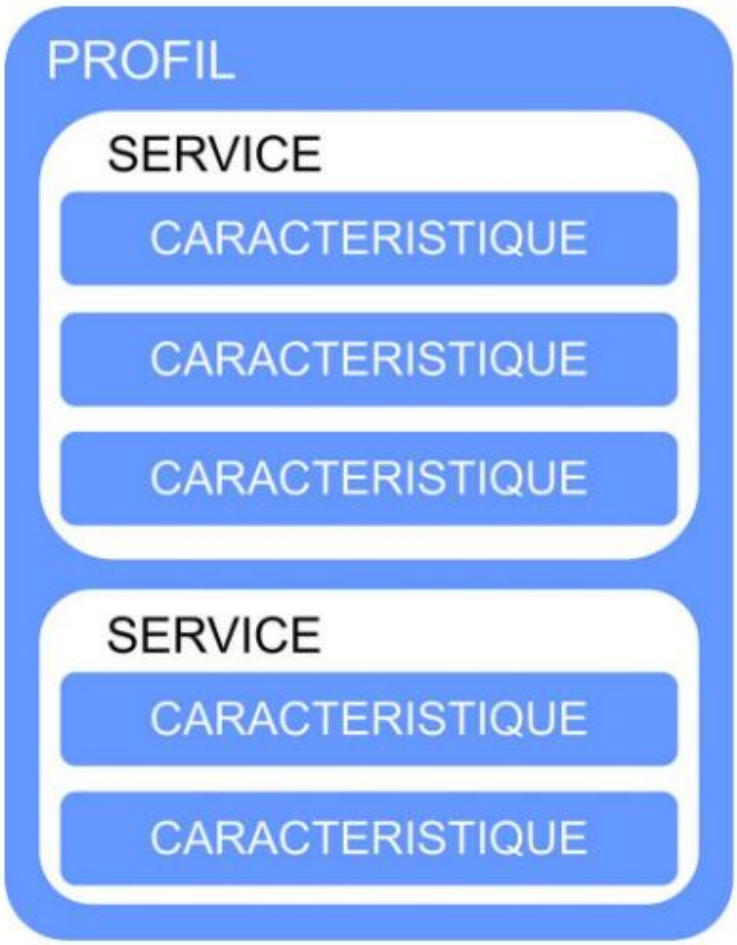
Le Generic Attributes Profile (GATT) :

Ce protocole définit comment deux objets échangent des données à l'aide de services et de caractéristiques. GATT entre en jeu après GAP, une fois que deux objets sont connectés.

L'un des objets, en général qui était un advertiser/broadcaster avant connexion, adoptera le rôle de peripheral et se comportera comme un serveur. L'autre objet, qui était un scanner/observer avant connexion, deviendra un central, et se comportera comme un client.

Il est important de noté concernant GATT, que les connexions son exclusives : un périphérique BLE ne peut pas être connecté à plus d'un central à un moment donné. Mais un central peut être connecté à plusieurs périphériques simultanément.

Le protocole GATT repose sur un autre protocole d'échange de données bidirectionnel et structuré nommé Attribute Protocol (ATT) qui met en oeuvre les concepts de Profils, de Services et de Caractéristiques, que l'on peut imaginer comme des conteneurs imbriqués selon la figure suivante :



- **Les profils** n'existent pas sur les périphériques, ce sont simplement des collections prédéfinies de services qui ont été compilées soit par le Bluetooth SIG, soit par le concepteur du périphérique.

Le profil "Heart Rate", par exemple, rassemble le service "Heart Rate" et le service "Device Information". La liste complète des profils GATT officiellement adoptée est disponible ici.

- **Les services** segmentent les données en unités logiques appelées caractéristiques. Un service peut contenir une ou plusieurs caractéristiques et chaque service est identifié de façon unique par un entier appelé UUID pour "Universally Unique IDentifier". L'UUID est codé sur 16 bits pour les services BLE officiels et sur 128 bits pour les services BLE personnalisés. Une liste complète des services BLE officiels peut être consultée ici.

Par exemple, le service "Heart Rate" a comme UUID la valeur 0x180D et contient jusqu'à trois caractéristiques, seule la première étant obligatoire : Heart Rate Measurement, Body Sensor Location et Heart Rate Control Point.

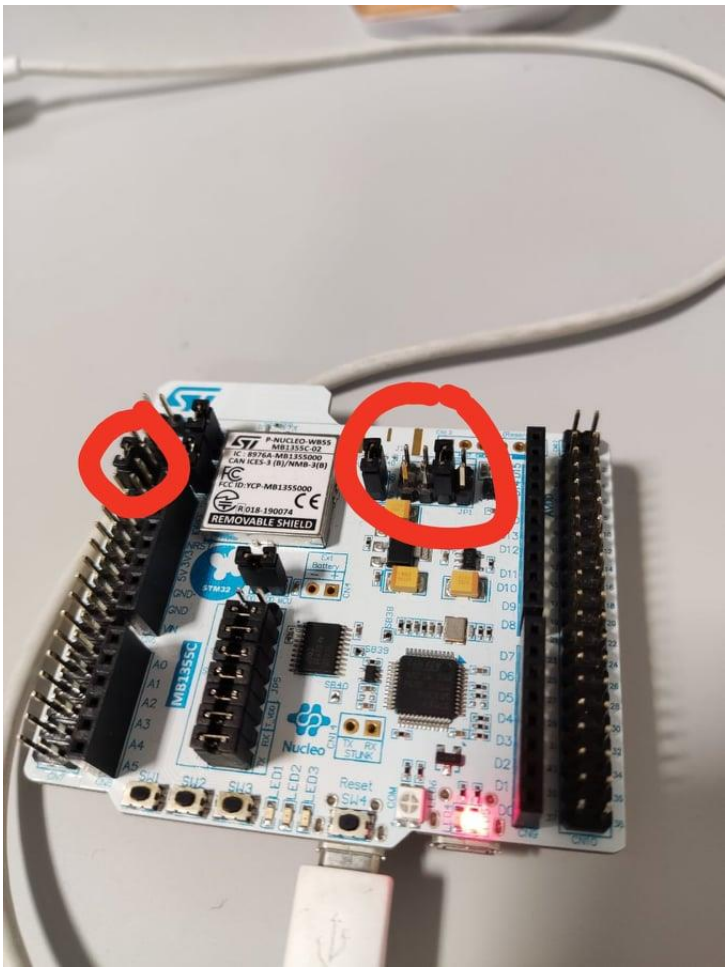
- **Les caractéristiques** sont les éléments de plus bas niveau pour les transactions GATT, elles encapsulent un unique point de mesure (une valeur pour un capteur de température ou bien un triplet (Ax, Ay, Az) pour un accéléromètre trois axes, par exemple). Les caractéristiques disposent toutes, à l'image des services, d'un UUID codé sur 16 ou 128 bits, et vous êtes libre soit d'utiliser les caractéristiques standard définies par le Bluetooth SIG, ce qui facilitera une compatibilité entre plusieurs dipositifs BLE, soit de définir vos propres caractéristiques pour un écosystème propriétaire (comme par exemple Blue-ST de STMicroelectronics).

2. PRISE EN MAIN CARTE ET EMISSION EN BLE

On a choisi d'utiliser du micropython pour émettre un signal bluetooth en BLE. Pour cela il faut configurer notre carte pour qu'elle utilise du micropython.

Si besoin, mettre la carte en DFU mode (Device Firmware Update).

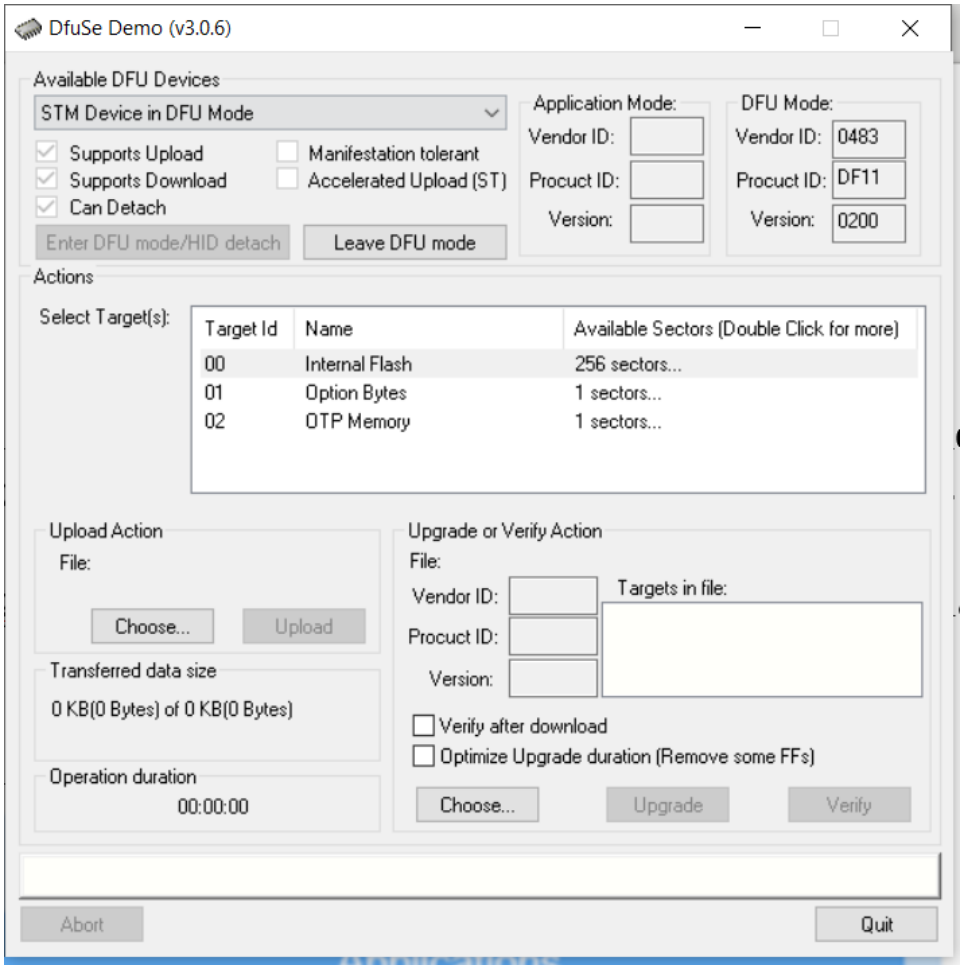
Pour la WB55 il est nécessaire me mettre les cavaliers dans une configuration spéciale pour la mettre en mode DFU. Il faut mettre le pin boot0 à 1 (entouré en rouge à gauche).



On va télécharger le dfu micropython correspondant pour la board sur le site de micropython https://micropython.org/download/.

Ensuite, en utilisant l'application DfuSe Demo, on va mettre à jour le firmware de la carte.

On a juste à sélectionner la carte si ce n'est pas déjà fait, à renseigner le dfu et à upgrade la carte.



C'est bon on peut utiliser notre carte en micropython.

3. CODE MICROPYTHON

Le code est une modification d'un exemple pris sur https://stm32python.gitlab.io/fr/docs/Micropython/.

La carte va envoyer une trame d'advertising toutes les 5 secondes et des appareils vont pouvoir s'y connecter. La trame envoyée n'a pas d'importance car on veut juste connaitre la puissance du signal reçu sur la carte MP157C-DK2. J'ai donc gardé le même que l'exemple fournit qui envoyait une température aléatoire et ai changé quelque peu le code.

main.py	15/10/2021 10:06	Python File	5 Ko
ble_advertising.py	01/10/2021 11:07	Python File	4 Ko
pybcdc.inf	31/12/2014 23:00	Informations de c...	3 Ko
README.txt	31/12/2014 23:00	Document texte	1 Ko
boot.py	24/09/2021 15:15	Python File	1 Ko

Le package est constitué de plusieurs fichiers :

- boot.py est le premier fichier chargé et traité au démarrage d'une plateforme microPython
- ble_advertising.py une bibliothèque de fonctions qui seront utilisées pour construire les trames d'avertising du protocole GAP, lancé pour et avant la connexion à un central.
- main.py est le script principal qui contiendra le code du programme utilisateur.

Dans le main.py :

On initialise notre carte en BLE:

```
# Classe pour gérer le partage de données
class BLETemperature:
    # Initialisations
    def __init__(self, ble, name="b1Dongle"):
        self._ble = ble
        self._ble.active(True)
        self._ble.irq(self._irq)
        # Prévoit une caractéristique (température : self._handle)
        ((self._handle,),) = self._ble.gatts_register_services((_ENV_SENSE_SERVICE,))
        self._connections = set()
        self._payload = advertising_payload( # on construit ici notre payload
            name=name, services=[_ENV_SENSE_UUID], appearance=_ADV_APPEARANCE_GENERIC_THERMOMETER
        )
        self._advertise()

        # Affiche l'adresse MAC de l'objet
        dummy, byte_mac = self._ble.config('mac')
        hex_mac = hexlify(byte_mac)
        print("Adresse MAC : %s" %hex_mac.decode("ascii"))
```

On gère les interruption (connexion, déconnexion, …) :

- Si une connexion est établie, on affiche “connexion établie” et on ajoute notre central dans les appareils connectés.
- Si une déconnexion est faite, on affiche “déconnexion” et on ajoute enlève notre central dans les appareils connectés. Et on relance l'advertising pour de futures connexions.
- Si on reçoit l’évènement “indicate” on renvoi un accusé de réception.

```
# Gestion des évènements BLE
def _irq(self, event, data): #gere la connexion bt
    # Lorsqu'un central se connecte...
    if event == _IRQ_CENTRAL_CONNECT:
        print("connexion établie")
        conn_handle, _, _ = data
        self._connections.add(conn_handle)

    # Lorsqu'un central se déconnecte...
    elif event == _IRQ_CENTRAL_DISCONNECT:
        print("déconnexion")
        conn_handle, _, _ = data
        self._connections.remove(conn_handle)
        # Start advertising again to allow a new connection.
        self._advertise()

    # Lorsqu'un évènement "indicate" est validé, renvoie un accusé de réception
    elif event == _IRQ_GATTS_INDICATE_DONE:
        conn_handle, value_handle, status = data
```

Il y a ensuite une méthode pour envoyer la température aux appareils connectés. Et une autre méthode pour faire l’advertise toutes les 5 secondes.

```python
# Pour envoyer la température ...
def set_temperature(self, temp_deg_c, notify=False, indicate=False):
    # Data is sint16 in degrees Celsius with a resolution of 0.01 degrees Celsius.
    # Write the local value, ready for a central to read.
    self._ble.gatts_write(self._handle, struct.pack("<h", int(temp_deg_c * 100))) #methode comprise dans lib bluetooth.BLE()
    if notify or indicate:
        for conn_handle in self._connections:
            if notify:
                # Notifie les centraux connectés du rafraichissement de la valeur de la température
                self._ble.gatts_notify(conn_handle, self._handle)
            if indicate:
                # "Indicate" les centraux connectés (comme Notify, mais requiert un accusé de réception)
                self._ble.gatts_indicate(conn_handle, self._handle)

# Envoie des trames d'advertising toutes les 5 secondes,
def _advertise(self, interval_us=500000):
    self._ble.gap_advertise(interval_us, adv_data=self._payload) #on emet la trame payload
```

Et finalement la méthode principale va créer l'instance de la classe et envoyer la température toutes les secondes.

```python
def demo():
    # Objet BLE
    ble = bluetooth.BLE() #BLE est une classe de bluetooth fournissant des methodes
    # Instance de la classe environnementale
    temp = BLETemperature(ble)

    t = 25
    i = 0

    while True:
        # Write every second, notify every 10 seconds.
        i = (i + 1) % 10

        # Envoi en BLE de la température
        # le premier parametre est la temperature
        # le second est si on veut notifier ou non l'application
        #temp.set_temperature(t, notify=i == 0, indicate=False)
        temp.set_temperature(t, notify=True, indicate=False)
        # Random walk the temperature.
        t += random.uniform(-0.5, 0.5)

        # Temporisation de cinq secondes
        time.sleep_ms(1000)


if __name__ == "__main__":
    demo()
```

4. BIBLIOGRAPHIE

https://stm32python.gitlab.io/fr/docs/Micropython/stm32wb55

https://micropython.org/download/

https://stm32python.gitlab.io/fr/docs/Micropython/stm32wb55

https://stm32python.gitlab.io/fr/docs/Micropython/BLE/index