PFE – ScoutBOT – Exploration technique BLE – MicroPython sur STM32 WB55

Responsable du document : Nathan BRIENT

Auteur secondaire : Arnaud HINCELIN (Partie DFU)

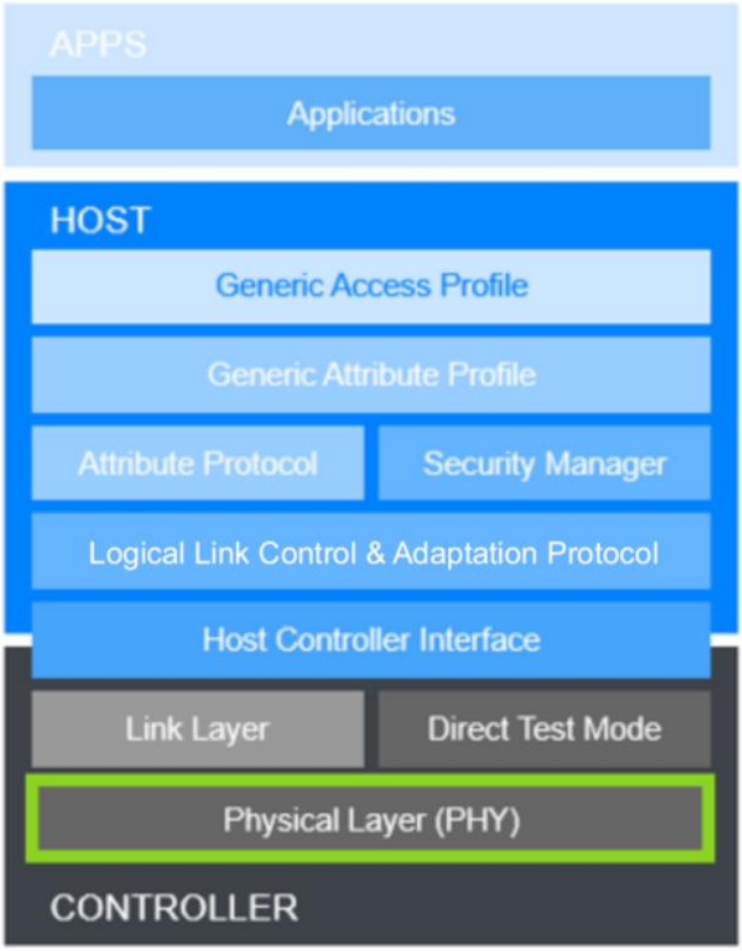
Documentation réalisée dans le cadre de notre projet de fin d'études. Cette partie concerne comment programmer des cartes STM32 de type WB55 avec du micropython pour effectuer des émission en BLE et faire de l'indor-positionning par la suite.

1. PRESENTATION DU BLE

Bluetooth : norme de 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 basée 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 certain 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.

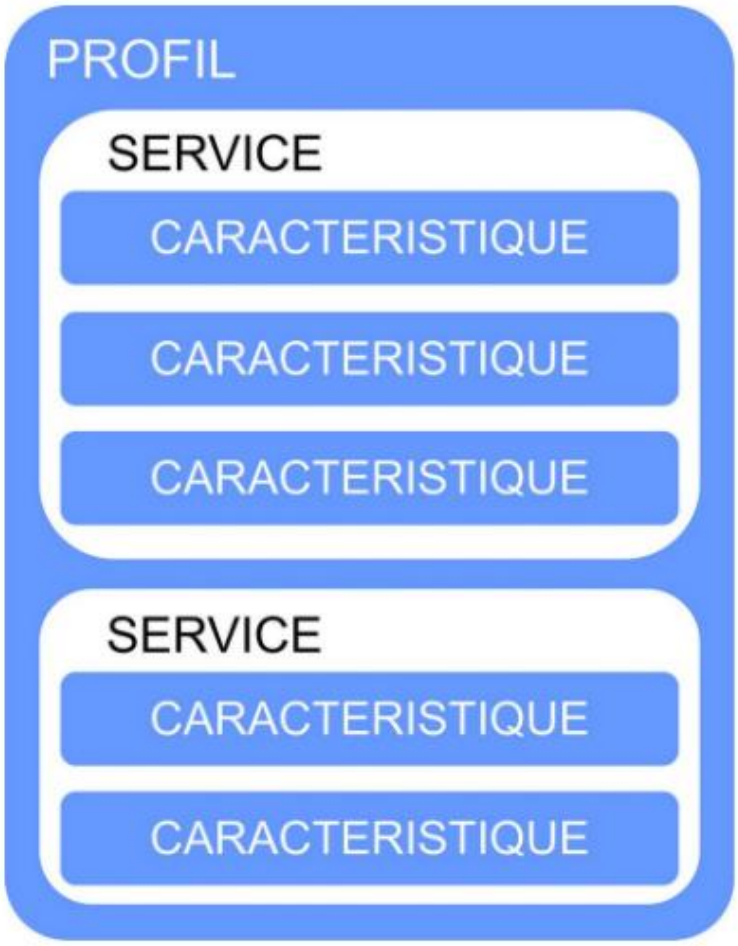
Le Generic Attributes Profile (GATT) :

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 noter 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 œuvre 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.
- **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 WB55 pour qu'elle utilise du micropython.

Ce tutoriel utilise le système d'exploitation Windows. Il n'est pas adapté un système basé sur Linux.

a. Généralités sur le mode DFU

Le mode DFU (Device Firmware Upgrade) est un mode de fonctionnement des microcontrôleur qui permet de charger un programme (avec le bootloader) via un port USB basique. Cela est utile pour mettre à jour les logiciels de la carte, ou bien si le bootloader est corrompu.

Ainsi, le mode DFU permet de flasher d'importe quel microcontrôleur de type ST (dans notre cas) sans passer par le ST-Link. Ce qui en pratique est très intéressant. Voici les différents avantages envisagés :

- Pas besoin d'outils spécifiques comme le JTAG, le ST-Link ou le USB-to-UART
- Capacité de programmer une carte STM32 vide
- Mettre facilement à jour le micro-logiciel de STM32

En mode DFU on va flasher le microcontrôleur en mettant sur la carte un fichier d'extension .dfu ; il ne faut pas confondre le fichier du firmware, qui est de type .hex ou, .s19, ou .o (binaire). En effet ces fichiers binaires ne contiennent pas la data permettant de réaliser l'opération de upgrade, seulement la data du nouveau firmware. Mais l'opération DFU nécessite aussi des informations comme le « product ID », le « vendor ID », la version du firmware, le « target ID ».

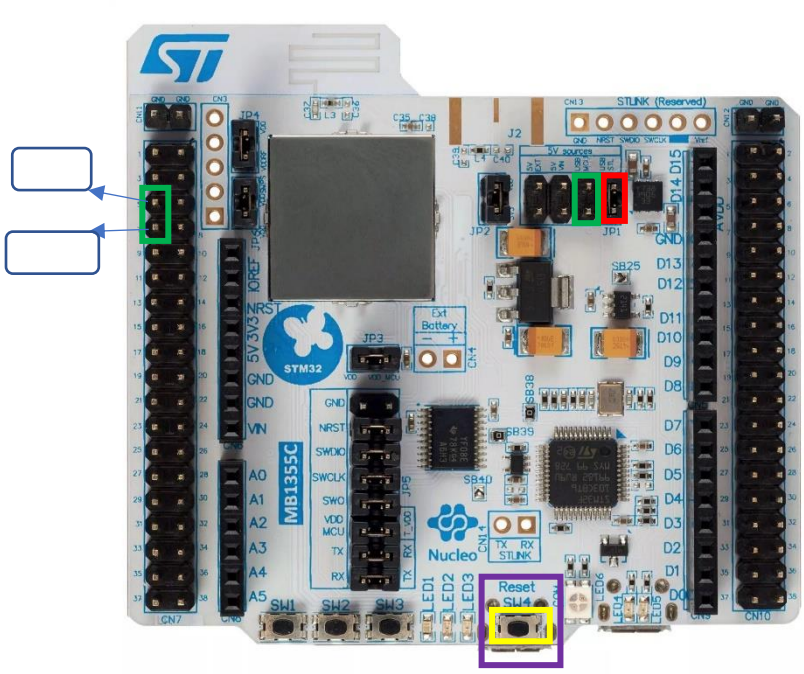
Ainsi, un nouveau type de fichier appelé fichier FDU (.dfu) contient tous ces éléments et le firmware.

b. Booter la carte en Mode DFU

<u>Etape 1 : Activer le mode DFU sur la carte</u>

La microcontrôleur sait que son mode DFU est activé lorsque la pin "boot0" est mise à l'état haut.

- Microcontrôleur nucleo WB55



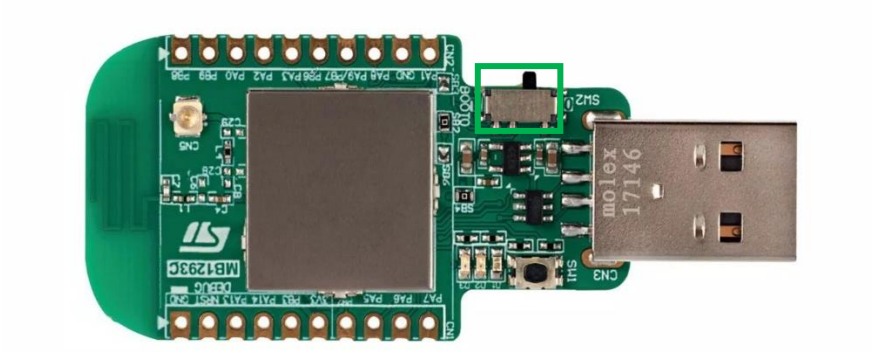
On passe le pin du BOOT0 à l'état haut, en ajoutant un connecteur entre la broche VCC et la broche BOOT0. (Connecteur en vert à gauche).

Ensuite, on enlève le connecteur (en rouge à droite) qui assure l'alimentation de la carte via le câble du ST-link, et place le connecteur afin de spécifier que l'alimentation se fera via le câble USB « USER », par lequel on transfèrera le fichier DFU (Connecteur en vert à droite).

Enfin, comme indiqué précédemment, le connecteur USB à utiliser ne sera pas le ST-link, mais le second connecteur USB annoté « USB_USER » (indiqué en violet).

Le bouton « RESET » (en jaune) doit être activé une fois ces étapes réalisées.

- Microcontrôleur WB55 USB Dongle



Pour le WB55 dongle USB, il suffit simplement de mettre le switch (en vert) sur le mode « BOOT0 », switch à gauche d'un point de vue du plan ci-dessus. Il n'y a pas de bouton « RESET » sur cette carte ; donc il faut débrancher et rebrancher le dongle USB.

Remarque 1 : Notez bien que c'est le même microcontrôleur que la carte nucleo WB55 précédemment, le dongle est plus pratique d'utilisation mais offre moins de GPIOs. Mais si on doit seulement utiliser du micropython et faire des communications BLE, le dongle USB est bien plus pratique.

Remarque 2 : Si vous utilisez la carte nucleo WB55, vérifiez bien que vous avez un unique câble USB sur le connecteur USB « USB_USER » et non sur le connecteur « ST-LINK ».

Etape 2 : Installer DfuSe

On commence par installer le logiciel DfuSe fourni par STMicroelectronics. Vous verrez pourquoi on installe maintenant ce logiciel. Il permet de flasher un microcontrôleur en mode DFU facilement.

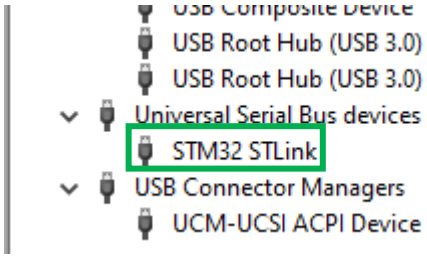
STSW-STM32080 - DfuSe USB device firmware upgrade (UM0412) (replaced by STM32CubeProgrammer) - STMicroelectronics

Etape 3 : Etablir le mode DFU sur le PC

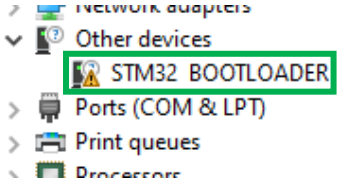
Cette étape est à réaliser si votre PC n'a jamais utilisé des microcontrôleur en mode DFU. Ces étapes ne sont documentées nulle part sur internet, seule les longues explorations techniques on permit de connaître toutes ces étapes.

On commence par ouvrir le device manager de windows.

Lorsque le microcontrôleur STM32 n'est pas en mode DFU, alors on verra le device « STM32 STLink », avec le driver connu ST-LINK, comme ci-dessous.



Mais si la carte est bien configurée en mode DFU (et que l'on a fait un « RESET » sur la carte) , on verra alors ceci la carte STM32 dans la rubrique « other devices » avec le nom de device « STM32 BOOTLOADER ». Le driver n'est normalement pas connu, ce qui explique le panneau d'avertissement.



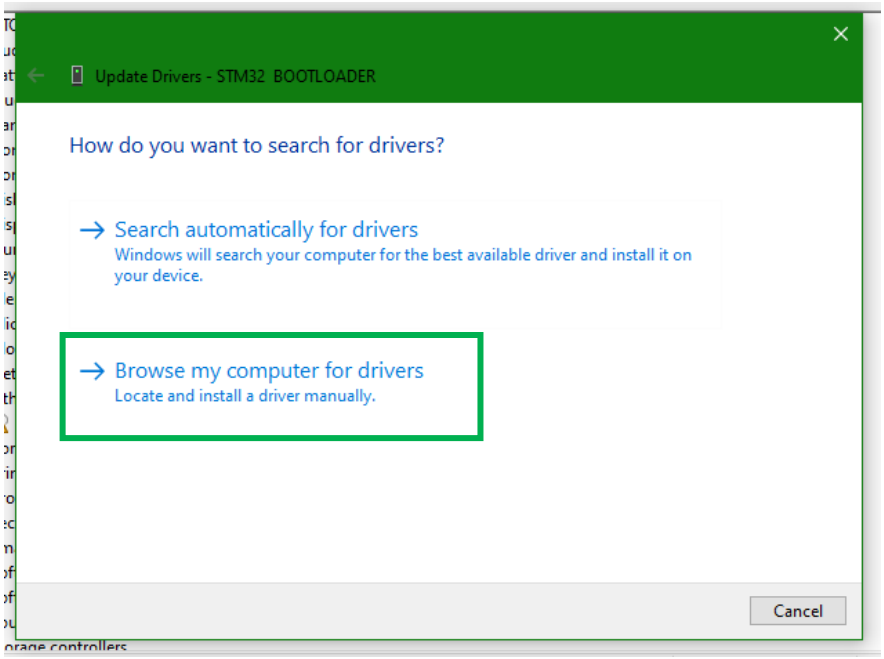
Remarque :

Dans le cas contraire, si vous ne voyez pas cela. Par exemple, si le device « STM32 BOOTLOADER » n'est pas dans la rubrique « Other devices » et que le driver paraît être connu ; alors il faut désinstaller le driver actif. Clique-droit sur le device, et désinstaller le driver. Pour information, *nous avons dû réaliser cette étape* ; cela n'est pas un problème car après lorsque que l'on aura besoin du driver ST-Link, il nous sera proposé de l'installer, mais même après réinstallation du driver ST-Link le mode DFU sera bien reconnu sur le PC.

Normalement après cette étape le device est bien dans la rubrique « Other devices » avec un nom « STM32 BOOTLOADER » et avec un driver inconnu.

Il faut maintenant mettre à jour ce driver. Clique-droit sur le device et mettre à jour.

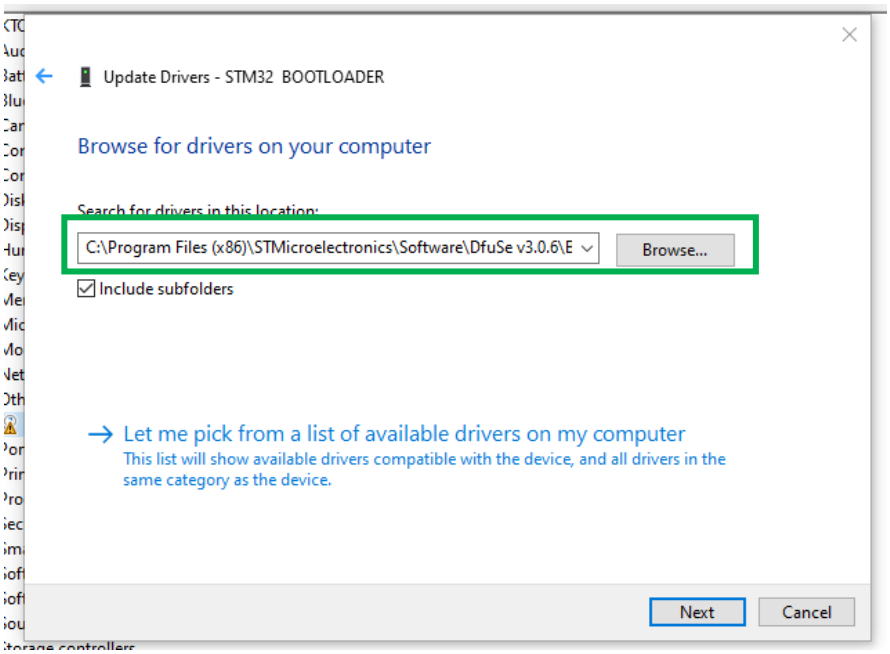
Une fois la fenêtre de mise à jour ouverte, cliquer sur rechercher des driver sur mon PC.



Le driver DFU a en réalité été téléchargé avec le logiciel DfuSe, ce qui explique pourquoi il a fallu l'installer en avance.

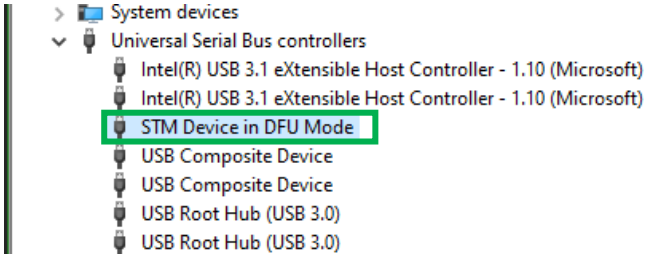
La localisation du driver DFU est donc dans les fichiers du logiciel DfuSe, un dossier appelé « Driver ».

Chemin dans notre cas : *C:\Program Files (x86)\STMicroelectronics\Software\DfuSe v3.0.6\Bin\Driver*



Puis cliquer suivre les étapes d'installation. Et au final on vous annonce le succès de l'installation du driver DFU.

Après un « RESET » de la carte, le device est cette fois détecté non plus dans « Other devices », mais dans la rubrique « USB controller » et le nom du device est « STM Device in DFU Mode ». On remarque aussi que le driver est bien connu, c'est celui que nous venons d'installer.

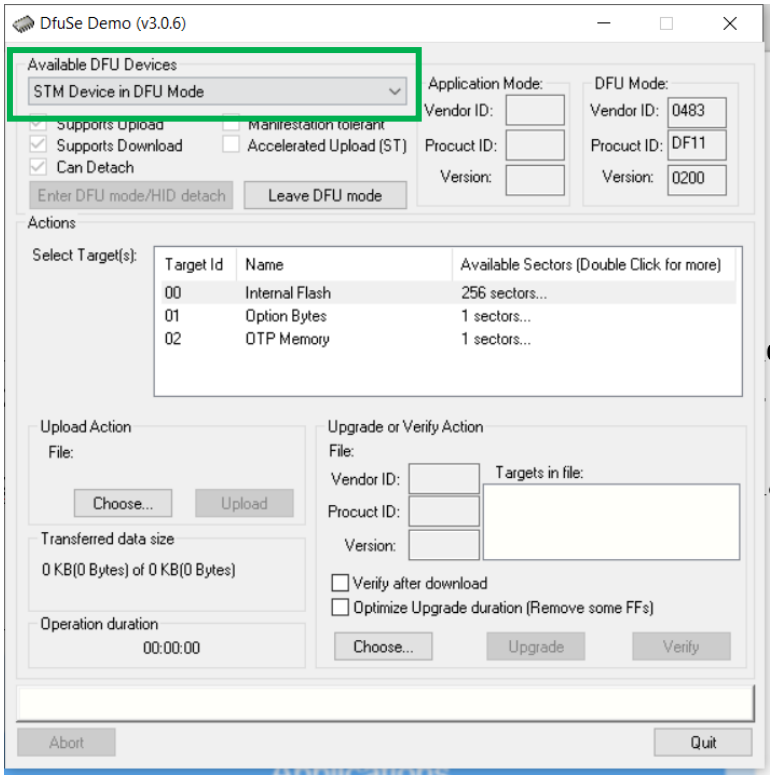


A partir de là, le PC sera capable de reconnaître tous les microcontrôleur ST en mode DFU.

c. *Flasher la carte avec du microPython*

Etape 1 : Tester le mode DFU

Afin de tester si le mode DFU a bien été configuré sur la carte et sur le PC, le moyen le plus simple est d'ouvrir le logiciel DfuSeDemo, et sans rien configurer sur le logiciel il doit normalement détecter un device en mode DFU qui correspond à la carte WB55 en mode DFU. Comme l'image ci-dessous :



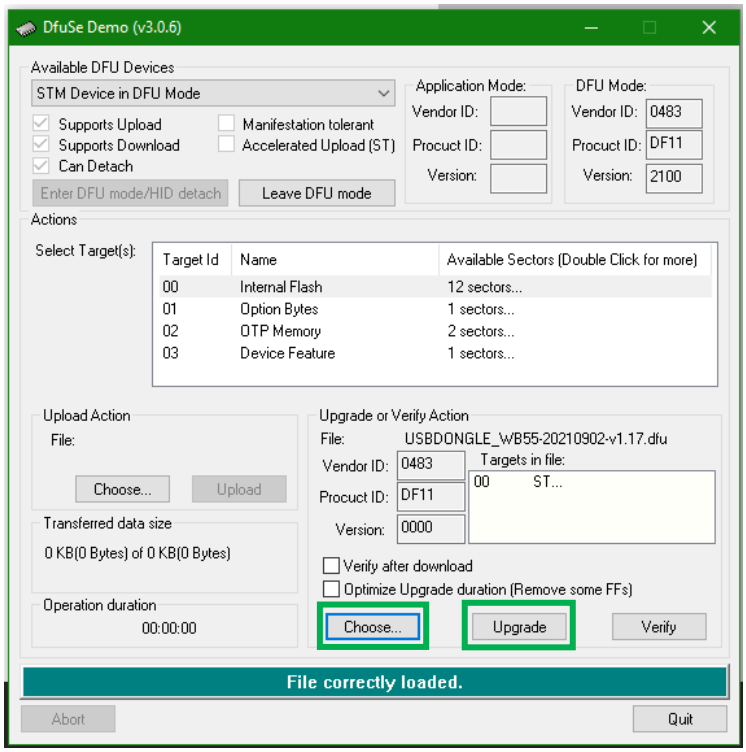
Etape 2 : Fichier DFU micropython

La carte doit maintenant être flashée avec un fichier de type .dfu ; le cas qui nous intéresse est de mettre en place du micropython sur la carte. Alors, on se rend sur le site officiel de micropython, dans « Downloads » on va dans la rubrique des cartes STM32, et il faut chercher l'endroit corresspondant à la carte WB55. On remarque qu'un certain nombre de cartes sont répertoriées et peuvent être programées en micropython.

La carte WB55 est tout en bas de la page.

MicroPython - Python for microcontrollers et on choisit la dernière version stable, dans notre cas le fichier a cet intitulé : USBDONGLE_WB55-20210902-v1.17.dfu

Puis retourner sur le logiciel DfuSe, et dans la section « Upgrade or Verify action » cliquer sur « Choose » et aller chercher le fichier DFU micropython du WB55 précédemment téléchargé.



Enfin, cliquer sur « Upgrade » et attendre que la carte soit bien flashée avec le fichier DFU de micropython.

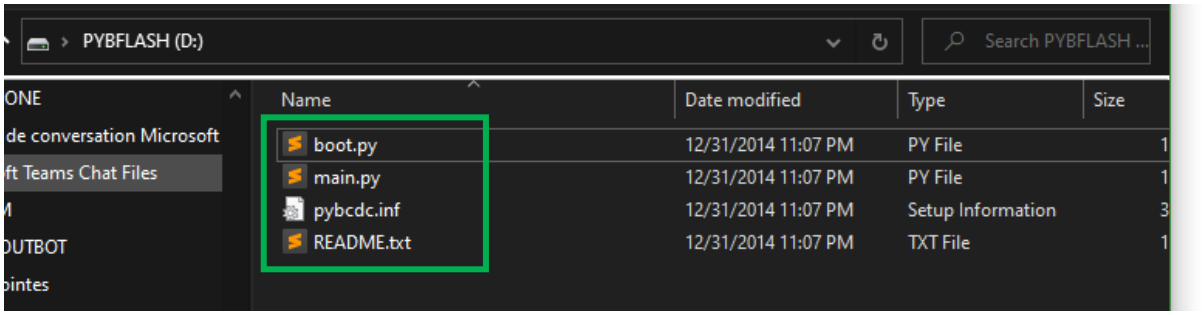
Etape 3 : Finaliser la mise en place du micropython.

Une fois le fichier DFU bien flashé sur la carte ; remettre le pin « BOOT0 » à l'état bas.

- Nucleo WB55 : Enlever le jongle du « BOOT0 »
- WB55 USB dongle : Placer le switch à l'autre état que le « BOOT0 »

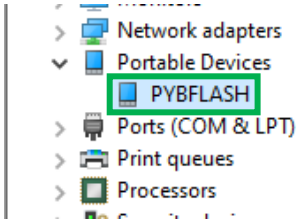
Puis faire un « RESET » de la carte avec le bouton RESET sur la nucleo, débrancher et rebrancher sur le USB dongle.

Alors immédiatement un explorateur de fichier s'ouvre avec 4 fichiers :



Un device a bien été détecté, il est visible dans l'explorateur de windows.

De plus, dans le device manager, on voit un device de type « PYBFLASH » dans la rubrique « Portable Devices » :



Notre carte est maintenant prête à être programmée en micropython.

3. CODE MICROPYTHON

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

Notre but : La carte WB55 va envoyer une trame d'advertising en BLE toutes les 5 secondes et des appareils vont pouvoir capter ce signal. La trame envoyée n'a pas d'importance car on veut juste connaitre la puissance du signal reçu sur la carte MP157C-DK2. Nous sommes repartis d'un exemple de code micropython fourni par STM32python, qui envoie une température aléatoire et avons changé ce 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 : 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 : 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.

```
# 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.

```
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