Berney Alec & Forestier Quentin & Herzig Melvyn

SYM – 29.01.2022

Capteurs et Bluetooth Low Energy

Labo 4



Table des matières

[Introduction 2](#_Toc94360396)

[Démarrage 2](#_Toc94360397)

[Manipulations 2](#_Toc94360398)

[Capteurs 3](#_Toc94360399)

[Utilisation 3](#_Toc94360400)

[Mise en place 3](#_Toc94360401)

[Question 4](#_Toc94360402)

[Bluetooth Low Energy 5](#_Toc94360403)

[Utilisation 5](#_Toc94360404)

[Mise en place 7](#_Toc94360405)

[Question 10](#_Toc94360406)

# Introduction

Ce laboratoire est une introduction à l’utilisation de capteurs et de communications Bluetooth Low Energy. Pour la partie Bluetooth, un écran Espruino Pixl.js a été fourni.

# Démarrage

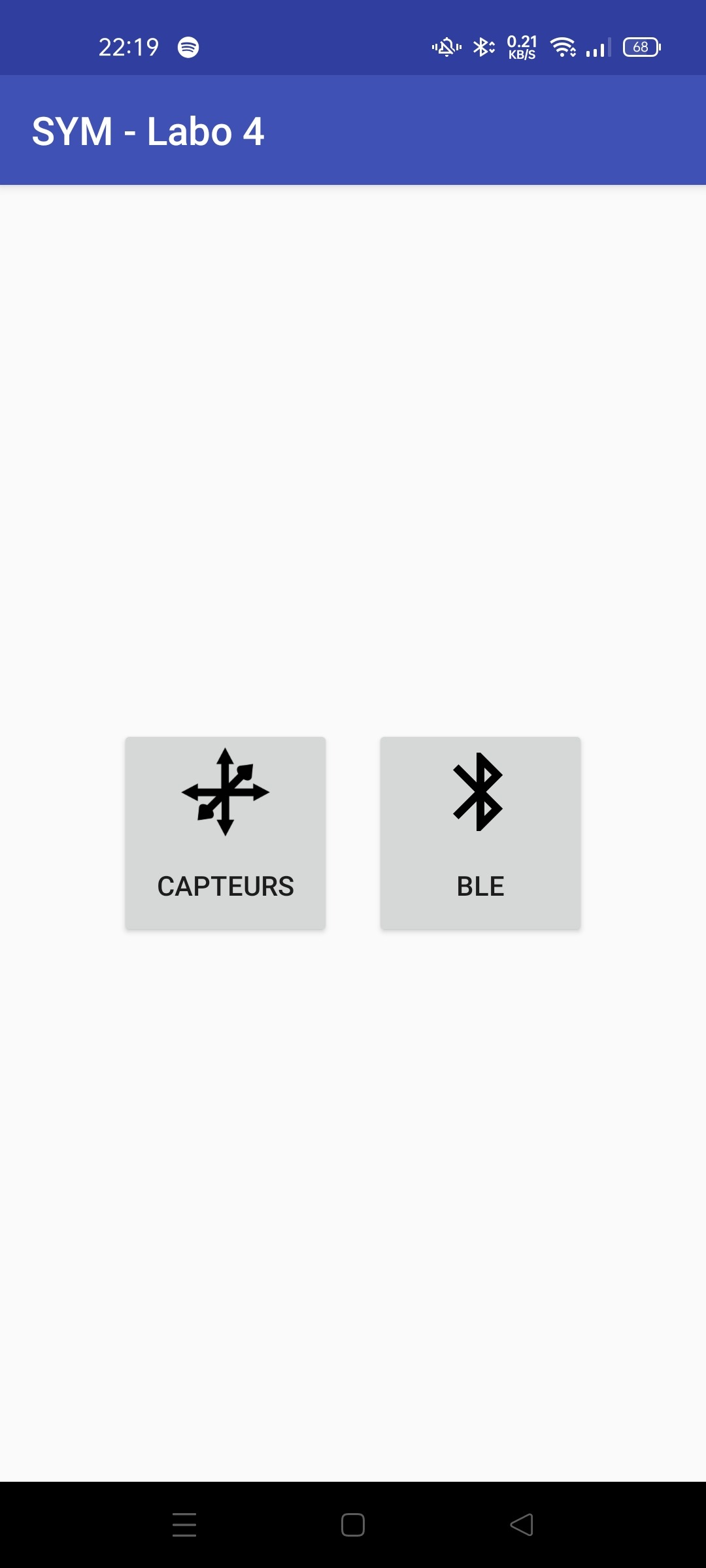


Figure Écran de démarrage

# Manipulations

Le laboratoire consiste en 2 manipulations majeures. Une activité a été réalisé pour chaque manipulation. Au cours des points suivants, nous allons décrire de manière incrémentale ce que nous avons mis en place pour arriver à l’était final de ce laboratoire.

## Capteurs

### Utilisation

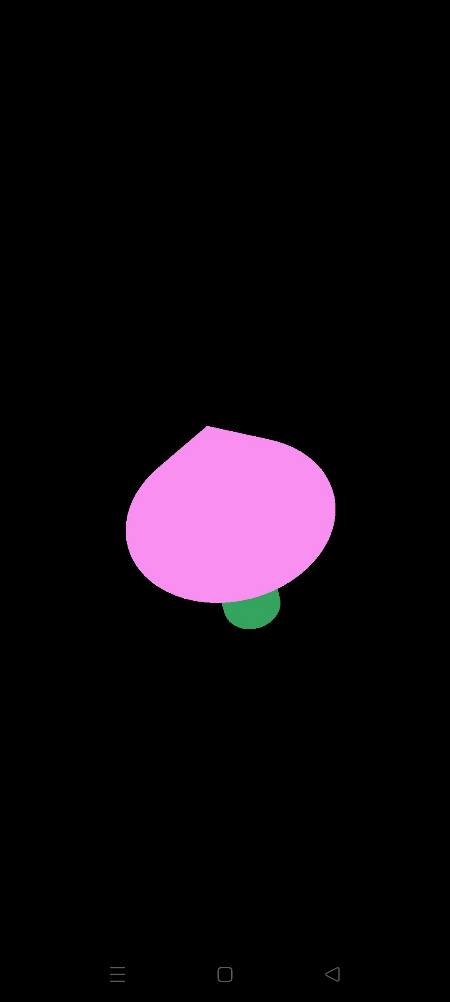
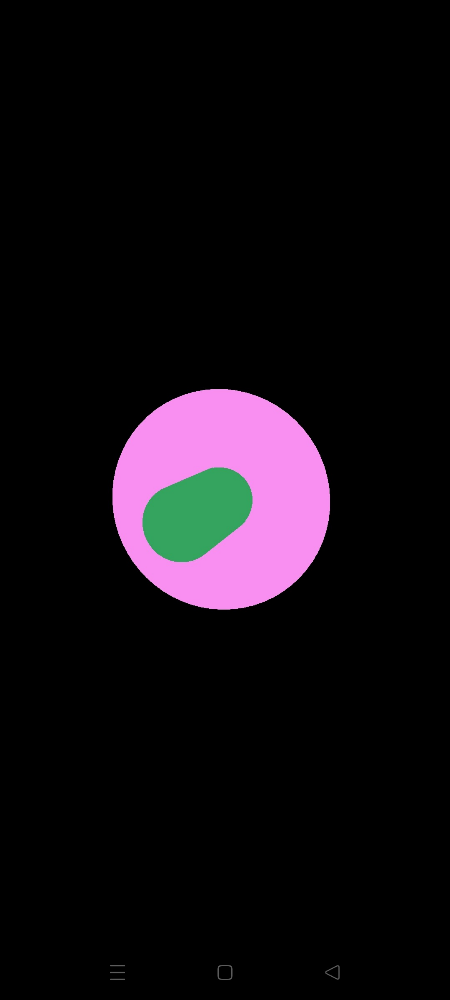
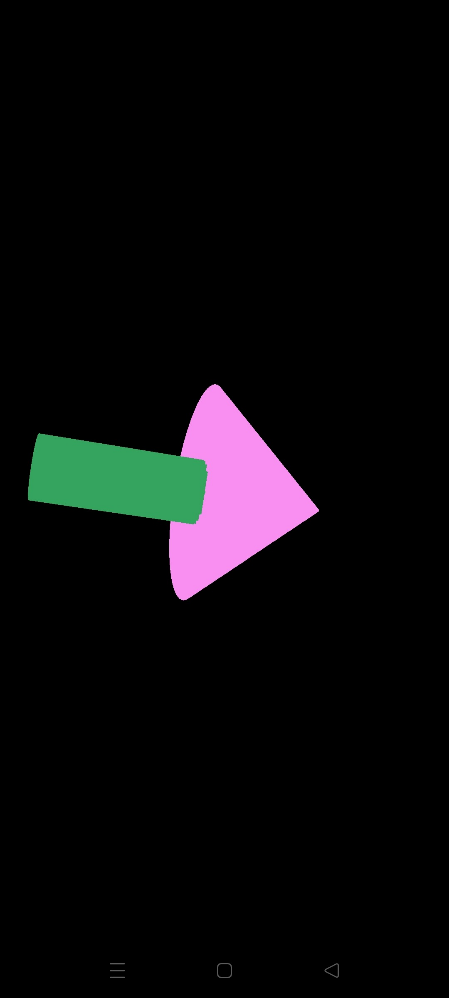
L’interface est très simple.

Figure Nord à droite

Figure Nord derrière

Figure Nord en haut

En effectuant des rotations du téléphone, l’aiguille va se tourner pour continuer d’indiquer le nord.

### Mise en place

Nous allons omettre toute la partie graphique de cette activité car elle nous a été fournie. Il est cependant important de noter qu’openGL est utilisé pour afficher la flèche.

Le point le plus compliqué de cette activité est le SensorEventListener :

*/\*\*  
 \* Sensor listener to handle sensors event  
 \*/*val mSensorListener : SensorEventListener = (object : SensorEventListener {  
  
 // Sensors data  
 private var mAccelerometerData = FloatArray(3)  
 private var mMagneticData = FloatArray(3)  
 private var mRotationMatrix = FloatArray(16)  
  
 */\*\*  
 \* React to sensors  
 \*/* override fun onSensorChanged(event: SensorEvent?) {  
 // Set sensors value  
 when (event?.sensor?.*type*) {  
 Sensor.*TYPE\_ACCELEROMETER* -> mAccelerometerData = event.values  
 Sensor.*TYPE\_MAGNETIC\_FIELD* -> mMagneticData = event.values  
 }  
  
 // Get rotation matrix  
 SensorManager.getRotationMatrix(mRotationMatrix, null, mAccelerometerData, mMagneticData)  
  
 // set new rotation  
 mRotationMatrix = opglr.swapRotMatrix(mRotationMatrix)  
 }  
  
 override fun onAccuracyChanged(p0: Sensor?, p1: Int) {  
 // Not needed  
 }  
  
})

Une fois inscrit pour les 2 types de capteurs, la fonction onSensorChanged sera appelé lors d’un changement. On stocke la valeur ayant changée, puis on recalcule la matrice de rotation.

### Question

**Une fois la manipulation effectuée, vous constaterez que les animations de la flèche ne sont pas fluides, il va y avoir un tremblement plus ou moins important même si le téléphone ne bouge pas. Veuillez expliquer quelle est la cause la plus probable de ce tremblement et donner une manière (sans forcément l’implémenter) d’y remédier**

Les valeurs renvoyées par les capteurs changent, et cela même si le téléphone ne bouge pas. Cela peut être dû, par exemple, à :

* La très légère imprécision du capteur dû au bruit
* Des vibrations, par exemple à l’endroit où est posé le téléphone, que nous ne pouvons pas percevoir

Il est cependant évident que les changements de valeurs sont très faibles, et une solution serait de recalculer la matrice de rotation seulement lorsque la nouvelle valeur est « significativement » différente. Ceci permettrait d’éviter les micro-changements de valeurs, mais ferait perdre un peu en précision.

## Une image contenant texte Description générée automatiquementBluetooth Low Energy

Figure Activité Bluetooth

### Utilisation



Figure Scan des appareils Bluetooth

Après avoir cliqué sur le bouton de scan, placé en haut à droite, la liste des appareils possédant le Service Custom SYM. Il est alors possible de se connecter à l’appareil en cliquant sur son nom dans la liste.

Une image contenant texte

Description générée automatiquementUne fois l’appareil sélectionné, l’écran suivant s’affiche :

Figure Gestion de l’appareil Bluetooth

Sur cet écran vous pouvez :

* Lire la température donnée par le périphérique en cliquant sur le bouton « Lire ».
* Observer le nombre de cliques réalisés sur le périphérique.
* Mettre à jour la date et l’heure sur le périphérique en cliquant sur le bouton « Mettre à jour ».
* Envoyer une valeur à ajouter au graphique du périphérique en entrant une valeur dans le champ texte sous « Valeur à envoyer », puis en appuyant sur le bouton « Envoyer ».

### Mise en place

Tout d’abord, il a fallu filtrer les appareils ayant le Service Custom SYM. Nous scannons donc les appareils possédant l’UUID souhaité.

//we scan for any BLE device  
//we filter them based on advertised service:  
//"SYM" (UUID: "3c0a1000-281d-4b48-b2a7-f15579a1c38f")  
val uuidFilter = ScanFilter.Builder()  
 .setServiceUuid(ParcelUuid.fromString(("3c0a1000-281d-4b48-b2a7-f15579a1c38f")))  
 .build()  
  
//reset display  
scanResultsAdapter.clear()  
bluetoothScanner.startScan(  
 *listOf*(uuidFilter),  
 builderScanSettings.build(),  
 leScanCallback  
)

Nous vérifions que tous les services et caractéristiques soient opérationnels et présents lors de la connexion grâce aux fonction suivante :

*/\*\*  
 \* Init the supported services using their UUID  
 \** ***@param*** *services List of services available  
 \*/*private fun initServices(services: List<BluetoothGattService>) {  
 for (service in services) {  
  
 when (service.*uuid*) {  
 TIME\_SERVICE\_UUID -> {  
 timeService = service  
 initTimeCharacteristics(timeService!!.*characteristics*)  
 }  
 SYM\_SERVICE\_UUID -> {  
 symService = service  
 initSYMCharacteristics(symService!!.*characteristics*)  
 }  
 }  
  
 }  
}  
  
*/\*\*  
 \* Init the supported characteristics of Time service using their UUID  
 \** ***@param*** *characteristics List of characteristics of Time service  
 \*/*private fun initTimeCharacteristics(characteristics: List<BluetoothGattCharacteristic>) {  
 for (char in characteristics) {  
 when (char.*uuid*) {  
 CURRENT\_TIME\_UUID -> currentTimeChar = char  
 }  
 }  
}  
  
*/\*\*  
 \* Init the supported characteristics of SYM service using their UUID  
 \** ***@param*** *characteristics List of characteristics of SYM service  
 \*/*private fun initSYMCharacteristics(characteristics: List<BluetoothGattCharacteristic>) {  
 for (char in characteristics) {  
 when (char.*uuid*) {  
 INTEGER\_UUID -> integerChar = char  
 BUTTON\_CLICK\_UUID -> buttonClickChar = char  
 TEMPERATURE\_UUID -> temperatureChar = char  
 }  
 }  
}

Si tout a bien été initialisé, la méthode « isRequiredServiceSupported » retourne alors true.

Suite à cela, on s’enregistre pour recevoir les notifications souhaitées.

override fun initialize() {  
 /\*  
 Ici nous somme sûr que le périphérique possède bien tous les services et caractéristiques  
 attendus et que nous y sommes connectés. Nous pouvous effectuer les premiers échanges BLE:  
 Dans notre cas il s'agit de s'enregistrer pour recevoir les notifications proposées par certaines  
 caractéristiques, on en profitera aussi pour mettre en place les callbacks correspondants.  
 \*/  
  
 setNotificationCallback(buttonClickChar).with **{** \_: BluetoothDevice, data: Data **->** buttonClickValue.*value* = data.getIntValue(Data.*FORMAT\_UINT8*, 0)  
 **}** setNotificationCallback(currentTimeChar).with **{** \_: BluetoothDevice, data: Data **->** currenttimeValue.*value* = String.*format*(  
 "%02d/%02d/%02d %02d:%02d:%02d",  
 data.getIntValue(Data.*FORMAT\_UINT8*, 3), // Day  
 data.getIntValue(Data.*FORMAT\_UINT8*, 2), // Month  
 data.getIntValue(Data.*FORMAT\_UINT16*, 0), // Year  
 data.getIntValue(Data.*FORMAT\_UINT8*, 4), // Hour  
 data.getIntValue(Data.*FORMAT\_UINT8*, 5), // Second  
 data.getIntValue(Data.*FORMAT\_UINT8*, 6) // Minute  
 )  
 **}** enableNotifications(buttonClickChar).enqueue()  
 enableNotifications(currentTimeChar).enqueue()  
  
 readTemperature()  
}

Et on met à disposition des méthodes pour les autres services. Ces méthodes sont encapsulées par des méthodes du ViewModel, permettant ainsi de s’assurer qu’un appareil est bien connecté. Les voici :

*/\*\*  
 \* Read the temperature on the ble device connected  
 \** ***@return*** *true if everything went well  
 \*/*fun readTemperature(): Boolean {  
 /\*  
 on peut effectuer ici la lecture de la caractéristique température  
 la valeur récupérée sera envoyée à l'activité en utilisant le mécanisme  
 des MutableLiveData  
 On placera des méthodes similaires pour les autres opérations  
 \*/  
 if (temperatureChar != null) {  
 readCharacteristic(temperatureChar).with **{** \_: BluetoothDevice, data: Data **->** temperatureValue.postValue(  
 (data.getIntValue(Data.*FORMAT\_UINT16*, 0)?.div(10)).*toString*()  
 )  
 **}**.enqueue()  
 return true  
 } else {  
 return false  
 }  
}

*/\*\*  
 \* Send an integer to the ble device connected  
 \** ***@param*** *integer integer to send  
 \** ***@return*** *true if everything went well  
 \*/*fun sendInteger(integer: Int): Boolean {  
 return if (integerChar != null) {  
 integerChar!!.setValue(integer, Data.*FORMAT\_UINT32*, 0)  
 writeCharacteristic(integerChar, integerChar!!.*value*, *WRITE\_TYPE\_DEFAULT*).enqueue()  
 true  
 } else {  
 false  
 }  
}

*/\*\*  
 \* Set the current time in the ble device connected  
 \** ***@param*** *calendar Date to set  
 \** ***@return*** *true if everything went well  
 \*/*fun setDate(calendar: Calendar): Boolean {  
 return if (currentTimeChar != null) {  
  
 currentTimeChar!!.*value* = calendarToByteArray(calendar)  
  
 writeCharacteristic(  
 currentTimeChar,  
 currentTimeChar!!.*value*,  
 *WRITE\_TYPE\_DEFAULT* ).enqueue()  
 true  
 } else {  
  
 false  
 }  
}

On peut noter que toutes les récupérations de valeurs sont stockées dans des MutableLiveData. Ceci permettant d’être notifié lors d’un changement et ainsi pouvoir mettre à jour la vue en conséquence.

//ble events  
bleViewModel.isConnected.observe(this, **{** updateGui() **}**)  
  
bleViewModel.temperatureValue.observe(this, **{**updateValue()**}**)  
bleViewModel.buttonClickValue.observe(this, **{**updateValue()**}**)  
bleViewModel.currenttimeValue.observe(this, **{**updateValue()**}**)

### Question

**La caractéristique permettant de lire la température retourne la valeur en degrés Celsius, multipliée par 10, sous la forme d’un entier non-signé de 16 bits. Quel est l’intérêt de procéder de la sorte ? Pourquoi ne pas échanger un nombre à virgule flottante de type float par exemple ?**

Pour commencer, il est important d’évaluer les valeurs que peuvent prendre une température et les valeurs que l’on souhaite connaître / garder. Voici les observations réalisées par notre groupe :

* La température est quasiment toujours située entre -50°C et +100 °C.
* Les températures en dessous de 0°C sont quasiment impossible car non supportée par le « device », dans le sens où l’on suppose que le « device » ne supporte pas physiquement cette température.
* Seul un chiffre après la virgule est utilisé pour représenter la température.

Pour donner suite à ces observations, on remarque qu’il faut comme valeur maximale 1'000, car on peut obtenir 100 \* 10 au maximum, le \* 10 permettant d’avoir une précision d’un dixième. On se donc compte que l’on a besoin d’uniquement 16 bits non-signés pour représenter les valeurs maximales et minimales possibles.

Le fait de pouvoir représenter la valeur en non-signé 16bits, en utilisant des nombres entiers plutôt que des flottants, cela nous permet de significativement alléger la taille des données transmises (minimum 2x plus léger) entre le périphérique et le téléphone. Cela est intéressant lorsque l’on utilise un périphérique utilisant du BLE et ayant un débit maximum de 100 kbit/s. Ce gain n’est donc pas à négliger.

**Le niveau de charge de la pile est à présent indiqué uniquement sur l’écran du périphérique, mais nous souhaiterions que celui-ci puisse informer le smartphone sur son niveau de charge restante. Veuillez spécifier la(les) caractéristique(s) qui composerai(en)t un tel service, mis à disposition par le périphérique et permettant de communiquer le niveau de batterie restant via Bluetooth Low Energy. Pour chaque caractéristique, vous indiquerez les opérations supportées (lecture, écriture, notification, indication, etc.) ainsi que les données échangées et leur format.**

Le service énoncé ci-dessus existe déjà et s’appelle « Battery\_Service ». Ce service possède l’UUID 180F et utilise la caractéristiques « Battery\_Level ».

Voici le tableau représentant les opérations supportées par le « Battery\_Service » :

|  |  |
| --- | --- |
| Caractéristiques |  |
| Lecture | Obligatoire |
| Écriture | Exclu |
| Écriture sans réponse | Exclu |
| Écriture signée | Exclu |
| Écriture fiable | Exclu |
| Notifications | Optionnel |
| Indications | Exclu |
| Auxiliaires écrivables | Exclu |
| Broadcast | Exclu |

Les données échangées par ce service sont un simple pourcentage de batterie allant de 0 à 100%. 100% représentant un périphérique entièrement chargé et0% l’inverse. Le format est un entier non-signé 8bits (UInt8).

Voici les sources utilisées :

* Battery\_Service : <https://www.bluetooth.com/wp-content/uploads/Sitecore-Media-Library/Gatt/Xml/Services/org.bluetooth.service.battery_service.xml>
* Battery\_Level : <https://www.bluetooth.com/wp-content/uploads/Sitecore-Media-Library/Gatt/Xml/Characteristics/org.bluetooth.characteristic.battery_level.xml>