Développement d'une montre connectée

Nathan Olff

Automne 2015

Table des matières

1. Introduction

Depuis quelques années maintenant, un nouveau marché de l'électronique grand public se développe : celui de la smartwatch (montre connectée). Véritable compagnon du smartphone, la montre connectée est conçue pour permettre à son utilisateur d'être informé ou de consulter des informations présentes sur son téléphone, en un clin d'oeil, sans avoir à le sortir de son sac / sa poche.

Intéressé par le sujet, j'ai commencé à me renseigner sur les différents modèles de smartwatch existants, leurs modes de communication et leurs composants. Une grosse majorité d'entre elles fonctionne avec des processeurs de même type/architecture que ceux de nos smartphones. En faisant ce constat, je me suis demandé s'il n'était pas possible de développer une smartwatch fonctionnant avec de simples micro-contrôleurs à la puissance réduite entrainant de ce fait une diminution de la consommation énergétique.

2. Objectif du projet

L'objectif de ce projet est de développer une smartwatch basée sur des microcontrôleurs au lieu des processeurs d'applications que l'on trouve dans des Moto 360 ou Apple Watch par exemple.

L'appareil devra posséder au minimum deux capacités essentielles : pouvoir communiquer avec un smartphone afin de récupérer des informations et les transmettre à l'utilisateur.

Il sera également nécessaire de développer une application mobile permettant au smartphone de se connecter à la montre et de communiquer avec elle.

Tel que défini ci-dessus, l'appareil jouera uniquement le rôle d'afficheur. La montre ne serait en fait qu'un moyen pour le smartphone de faire parvenir des informations à l'utilisateur. Si on regarde le marché des smartwatch actuel, on se rend vite compte qu'un autre type d'interaction est essentiel à son fonctionnement : l'**interaction avec l'utilisateur**.

Il faut que l'utilisateur soit en mesure d'agir sur le fonctionnement de la montre : modifier le type d'informations qu'elle lui présente voire même transmettre de l'information vers le smartphone par son intermédiaire. Pour cela, il faut prévoir à la fois une solution technique permettant à l'utilisateur de transmettre des commandes à la montre, mais également un système de navigation par menus sur la montre elle-même.

Un des points forts des smartwatch commerciales est la présence de nombreux capteurs : luminosité, fréquence cardiaque, accéléromètre, gyroscope, etc... Il serait intéressant de pouvoir implémenter certains de ces composants dans ce projet. Malheureusement ces améliorations seront considérées comme optionnelles dans le cadre de ce projet dans un souci d'efficacité.

3. Choix du matériel

3.1 Communication avec le smartphone

Lorsque l'on parle de smartphone et de connexion sans fil, deux technologies viennent tout de suite à l'esprit : le Wi-fi et le Bluetooth.

Le Wi-fi présente plusieurs inconvénients : le premier étant la nécessité de la présence d'un point d'accès. Une fois le téléphone et la montre connectés au même réseau Wi-fi, encore faut-il arriver à les relier ensemble en évitant toute interférence avec d'autres smartphones potentiellement sur le même réseau. Un deuxième inconvénient du Wi-fi (en tout cas tel qu'il est disponible aujourd'hui), est sa consommation énergétique : il ne faudrait pas avoir à transporter une batterie de la même taille, si ce n'est plus grosse, que la smartwatch en elle-même.

Le Bluetooth est un standard de communication utilisé depuis bien longtemps déjà pour la communication entre téléphone et périphériques externes, tel qu'un autre téléphone ou une oreillette sans fil. Depuis quelques années, une norme appelée Bluetooth Low Energy (BLE) se développe. Elle est spécifiquement conçue comme un moyen de communication moderne pour l'IoT (*Internet of Things* – Internet des Objets). Elle se focalise sur des débits de transfert très faibles qui sont de nature économes en énergie.

Mon choix s'est donc porté sur le BLE.

3.2 Interaction avec l'utilisateur

A l'exception des Pebble, toutes les smartwatch disponibles dans le commerce contiennent un écran tactile. L'efficacité de l'écran tactile comme interface homme machine n'est plus à prouver. Cependant, dans un souci de complexité et de consommation d'énergie, j'ai préféré me tourner vers un écran non tactile à technologie OLED (*Organic Light-Emitting Diode* – Diode électroluminescente organique).

Concernant la partie commande de l'utilisateur, j'ai découvert en faisant quelques recherches l'existence de capteurs de mouvement (*gesture sensor*). Ces capteurs sont en réalité un combiné de capteurs de luminosité ambiante, de couleur (RGB) et de proximité. Ils peuvent être utilisés pour détecter les mouvements d'une main à une distance de 10 à 20 cm du capteur. En plus des mouvements de translations horizontaux dans les quatre directions, ces capteurs permettent également de détecter l'approche ou l'éloignement d'une main (ou tout autre objet). Il serait donc intéressant d'implémenter un de ces capteurs, le APDS-9960 [1].

1. SparkFun RGB and Gesture Sensor

3.3 Les micro-contrôleurs

N'ayant pour l'instant que peu de connaissances dans le développement de logiciels embarqués purs, j'ai préféré opter pour des cartes de développement compatibles avec des API (*Application Programming Inferface* – Interface de programmation) apportant un certain niveau d'abstraction.

3.3.1 Sparkfun Microview

Afin de limiter à la fois la complexité générale du système et sa taille, j'ai concentré mes recherches sur des cartes de développement incluant déjà un petit écran. J'ai choisi sur le Microview.

Le Microview est composé d'un micro-contrôleur ATMEL ATmega 328P et d'un écran OLED d'une définition de 64x48 pixels. Produit par SparkFun, le Microview se programme via Arduino, une API bien connue dans le monde des hobbyistes électroniques. Il ne mesure que 26.5 x 26.5 x 10.5 mm.

Après une campagne de financement sur la plateforme Kickstarter [2] ayant récolté plus de 500 000 $ (sur un objectif de 25 000 $), le Microview est désormais disponible en vente libre sur de nombreuses plateformes de vente de produits électroniques.

Les parties matériel du Microview (plans, schémas techniques, modèles 3D de la coque...) ont été publiées par SparkFun sous licence libre Creative Commons [3]. Une librairie a également été mise à disposition sous licence libre GPLv3 [4] permettant une utilisation relativement simple de l'écran, avec des fonctions de dessin et d'affichage de caractères.

Un programmeur USB permet de relier le MicroView à un ordinateur et de charger le programme à l'aide de l'*Integrated Development Environment* – Environnement de développement intégré (IDE) Arduino.

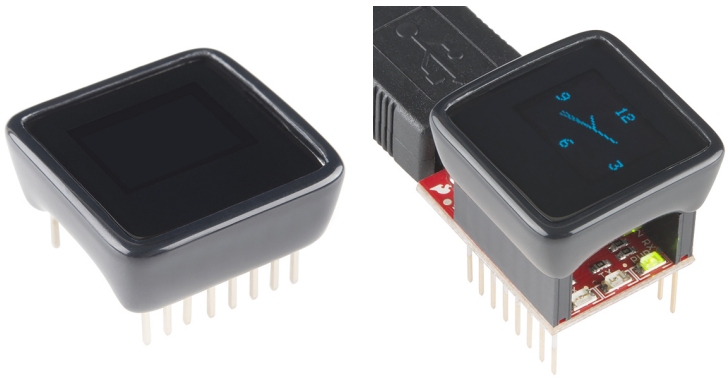

FIGURE 3.1 – Sparkfun Microview, accompagné de son programmeur sur l'image de droite

3.3.2 RedBear BLE Nano

Une fois le Microview et son écran sélectionnés, il restait encore la question de la connectivité Bluetooth. J'ai d'abord cherché des modules BLE que j'aurais pu relier au Microview. Ces derniers

2. Campagne Kickstarter du Microview : `https://www.kickstarter.com/projects/1516846343/microview-chip-sized-arduino-with-built-in-oled-di/description`
3. GitHub du hardware : `https://github.com/sparkfun/MicroView`
4. GitHub de la librairie Arduino : `SparkFun_MicroView_Arduino_Library`

étaient cependant soit d'une dimension soit d'un prix relativement élevé.

J'ai donc ensuite cherché du côté des cartes de développement avec BLE intégré, de la même façon que l'écran est intégré au Microview. C'est à ce moment-là que j'ai découvert l'existence de micro-contrôleurs avec le BLE intégré dans la puce elle-même. Le Nordic nRF51822 [5] est un micro-contrôleur construit autour d'un coeur ARM Cortex-M0 [6] et possède une connectivité Bluetooth 4.0 Low Energy.

Le RedBear BLE Nano [7] est une petite carte de développement intégrant un nRF51822. Il est compatible avec Arduino, mbed [8], et est fourni avec un *Software Development Kit* – Kit de développement (SDK) pour du développement embarqué pur.

Tout comme pour le Microview, le BLE Nano nécessite l'utilisation d'un programmeur USB.

En termes de dimensions, la carte du BLE Nano ne mesure que 18.5 x 21.0 mm.

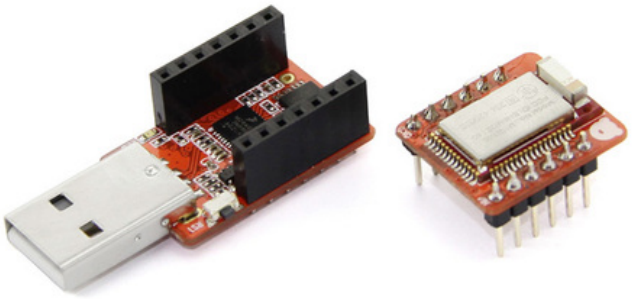

FIGURE 3.2 – RedBear BLE nano à droite et son programmer USB à gauche

3.3.3 Communication interne

La smartwatch est donc composée de deux éléments principaux : le BLE Nano et le Microview. La communication entre ces deux éléments est effectuée via un bus Inter-Integrated Circuit (I^2C). Le BLE Nano agit comme maître de la connexion, et le Microview comme esclave.

Etant donné que le BLE Nano fonctionne en 3.3V et le Microview en 5V, j'ai du mettre en place un système de conversion de niveaux à l'aide de transistors et de résistances pour permettre aux deux microcontrôleurs de communiquer (voir figure 3.3).

Lorsque l'application Android détecte une notification, elle récupère son contenu et détermine si elle doit être ou non transférée à la smartwatch. Si oui, elle transfère l'information au Nano via BLE, qui s'occupe ensuite de formater l'information et de la transmettre via I^2C au Microview.

5. Présentation du nRF51822sur le site internet de Nordic : `https://www.nordicsemi.com/eng/Products/Bluetooth-Smart-Bluetooth-low-energy/nRF51822`
6. Le Cortex-M0 est le micro-contrôleur très basse consommation développé par ARM : `http://www.arm.com/products/processors/cortex-m/cortex-m0.php`
7. BLE Nano sur le site de RedBear : `http://redbearlab.com/blenano`
8. mbed est une API développé par ARM – plus d'informations dans la section **??**

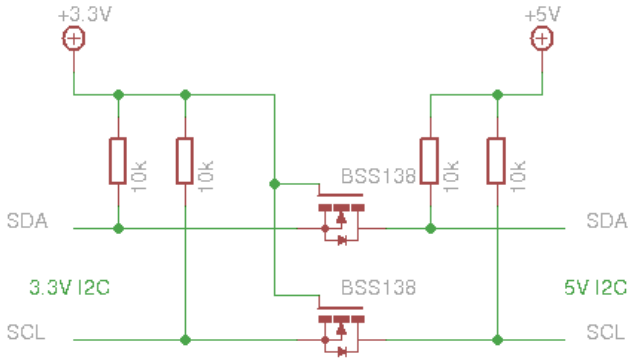

FIGURE 3.3 – Convertisseur de niveau pour bus I^2C

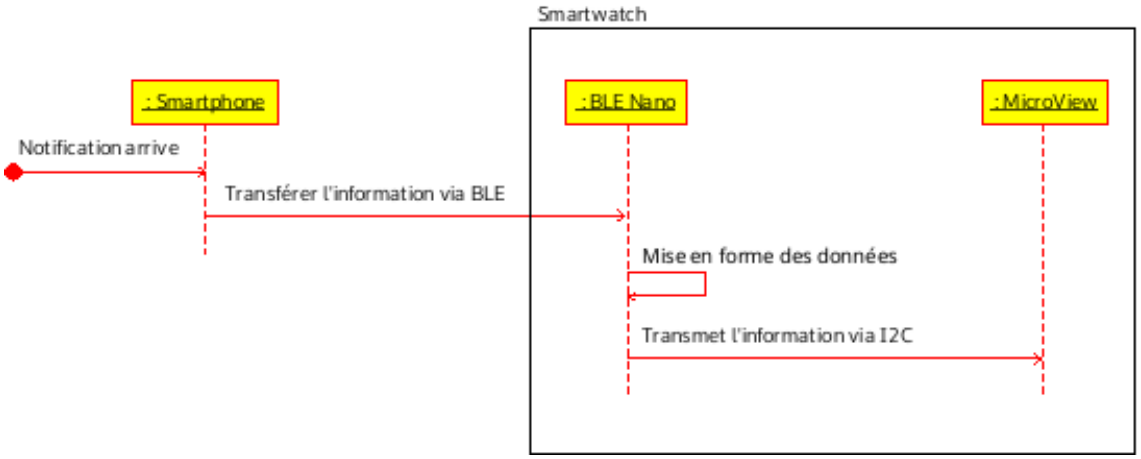

FIGURE 3.4 – Diagramme de séquence présentant la transmission d'information

4. Trois modèles de programmation

4.1 Arduino

Arduino est une plateforme libre (open source) de prototypage. A la fois matériel et logiciel, elle permet au plus grand nombre de créer des projets en électronique et informatique.

Créé en 2005 en Italie, Arduino est aujourd'hui utilisé à travers le monde. De l'éducation, aux projets scientifiques, en passant par la robotique ou encore la domotique, le nombre de projets développés utilisant Arduino est incalculable.

En termes de matériel, il existe une dizaine de modèles de cartes mère Arduino officielles, développées par la fondation Arduino. De par la nature open source du projet, de très nombreuses personnes / entreprises, se sont mises à développer des cartes mère "compatibles" Arduino. Basées en général sur les mêmes micro-contrôleurs que les cartes officielles, elles disposent souvent de caractéristiques propres avec notamment des fonctionnalités additionnelles intégrées à la carte (port(s) de connexion, régulateur, afficheur...) ou encore des formes et tailles particulières.

Historiquement, les cartes Arduino contenaient des micro-contrôleurs ATMEL 8-bits de type AVR. Ce n'est que récemment que la fondation Arduino, ainsi que d'autres par la suite, ont commencé à proposer des cartes Arduino à base de micro-contrôleurs ARM 32-bits. Ces derniers permettent un important gain de terme de puissance et ouvrent donc la porte à de nombreux projets gourmands en ressources.

FIGURE 4.1 – Logo Arduino

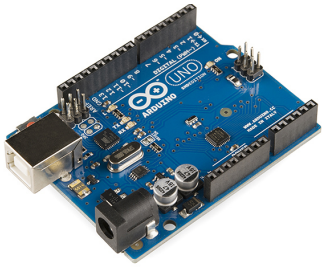

FIGURE 4.2 – Arduino Uno

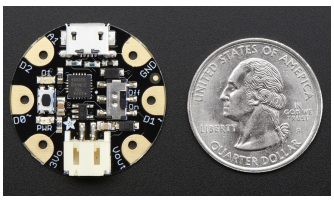

FIGURE 4.3 – Adafruit Gemma (compatible Arduino)

D'un point de vue logiciel, les cartes Arduino se programment à l'aide d'un IDE homonyme, également open source.

Les programmes Arduino, également appelés sketch Arduino, sont développés en C. Les entrées et sorties des cartes sont manipulées à travers une API spécifique au projet Arduino. Un sketch doit être composé au minimum de deux fonctions : une fonction `setup()`, exécutée au début du programme et pouvant contenir l'initialisation des pins, et une fonction `loop()`, exécutée en boucle et qui contient le programme principal.

La communauté est à l'origine de nombreuses bibliothèques pouvant être intégrées aux sketch

Arduino, notamment pour l'utilisation de certains capteurs, de protocoles de communication (SPI, I^2C...) ou de certains *shield* (cartes filles venant se monter sur les cartes mère Arduino).

L'API Arduino combinée à la présence de nombreuses librairies, permet de cacher la complexité liée à la configuration des GPIO (ports d'entrées / sorties) du micro-contrôleur au programmeur lambda. Le but d'Arduino est de faciliter le développement de projets d'électronique au plus grand nombre, et principalement aux personnes étrangères à la programmation. Il n'est en effet pas nécessaire d'être un ingénieur en systèmes embarqués pour mener à terme un projet à base d'Arduino.

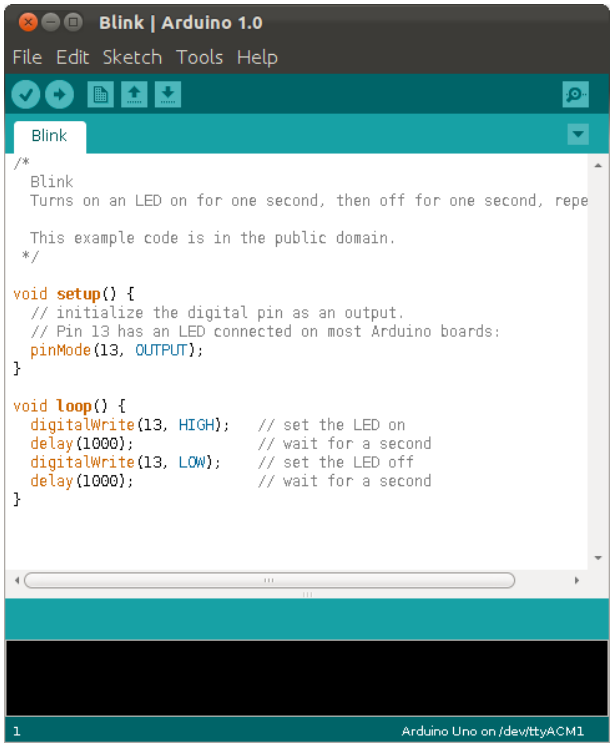

FIGURE 4.4 – Arduino IDE – Blinky

Pour des personnes déjà familiarisées à la programmation et au paradigme des systèmes embarqués, l'utilisation d'Arduino peut parfois être frustrante. J'ai moi-même été surpris en découvrant l'IDE et son manque effarant de fonctionnalités par rapport à ce que j'étais habitué (Eclipse par exemple). En effet, l'IDE ne propose pas de gestion de projets multi-fichiers, ni de débuggeur ou encore d'auto-complétion : des fonctions qui paraissent pourtant indispensables à tout IDE digne de ce nom.

Bien qu'une bonne partie de la complexité du programme soit cachée, on peut tout de même accéder aux registres internes du micro-contrôleur et avoir ainsi un meilleur contrôle sur son fonctionnement.

4.2 ARM mbed

mbed est une plateforme de développement open source dont l'objectif est de proposer une interface de programmation facile à prendre en main pour le développement de projets liés à l'IoT.

ARM est une société britannique spécialisée dans le développement de micro-processeurs. C'est en 2009 qu'elle présente la plateforme mbed. Elle est utilisable avec un nombre grandissant de

cartes mère et n'est compatible qu'avec les micro-contrôleurs ARM de la série Cortex-M [1]. De grands noms de l'informatique embarquée comme Freescale, Nordic ou encore STMicroelectronics se sont joints à ARM et proposent aujourd'hui des cartes mère mbed.

FIGURE 4.5 – Logo ARM mbed

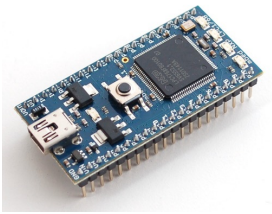

FIGURE 4.6 – NXP LPC1768

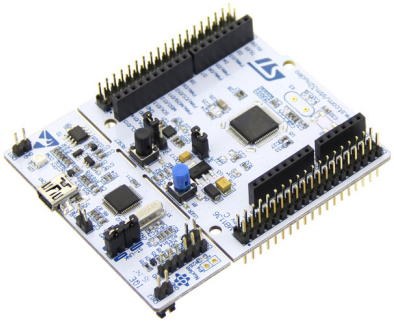

FIGURE 4.7 – ST NUCLEO F-401RE

mbed est disponible soit sous la forme d'un SDK pouvant être intégré à n'importe quel IDE / chaine de compilation, soit via l'IDE et compilateur en ligne, accessibles gratuitement sur le site pour développeur mbed (`https://developer.mbed.org/compiler/`).

Contrairement à l'IDE Arduino, celui de mbed propose toutes les fonctionnalités de base d'un IDE (projets multi-fichiers, auto-complétion, suivi d'appels de fonctions) ainsi que d'autres très intéressantes comme la gestion de code source (via des dépôts internes au site pouvant être partagé avec le public) ou l'exportation de programme sous forme de projet GCC, DS-5 ou encore Keil µVision. L'IDE en ligne utilise en interne une version du compilateur commercial d'ARM.

Le développement d'un programme mbed est plus proche du C++ que du C. Chaque port et chaque périphérique est représenté par un objet. Un changement d'état sur un pin de sortie s'apparente à une assignation de variable (voir figure 6.1).

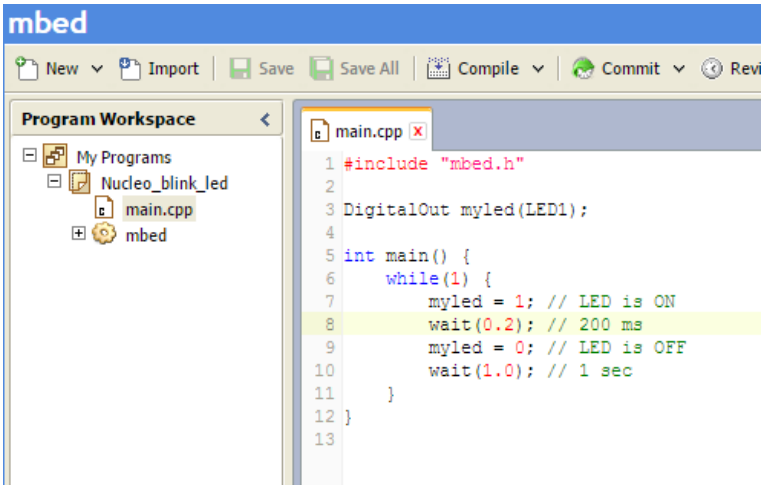

FIGURE 4.8 – IDE mbed – Blinky

1. Les deux autres séries de micro-processeurs d'ARM, les Cortex-A et Cortex-R sont plus des processeurs d'applications (utilisés notamment dans nos smartphones) que des micro-contrôleurs

4.3 Android Studio

Android est un système d'exploitation mobile lancé en 2007 par Google, suite au rachat deux ans plus tôt d'une startup. Détenant plus de 80% de parts de marché dans le domaine des smartphones, il s'exporte désormais sur TV (Android TV), montres connectées (Android Wear) ou encore voitures (Android Auto). Basé sur un noyau Linux, il est publié sous licence libre.

Si le système d'exploitation est essentiel au fonctionnement d'un smartphone, ce sont les applications et non le système qui le rend de plus en plus indispensable dans nos vies de tous les jours. Il existe aujourd'hui plus d'un million d'applications sur le magasin d'applications officiel de Google, le Google Play Store. Ces applications sont développées en Java grâce au Android SDK. Ce SDK peut être soit intégré à un IDE classique (par exemple Eclipse) soit utilisé au travers d'Android Studio.

Android Studio est un IDE développé par Google et basé sur IntelliJ IDEA. Il est exclusivement pensé pour le développement d'applications Android. Le SDK est intégré, mais également une série d'outils comme ADB (nécessaire au debug d'applications), un afficheur de log, un éditeur d'interface graphique ou encore un émulateur Android.

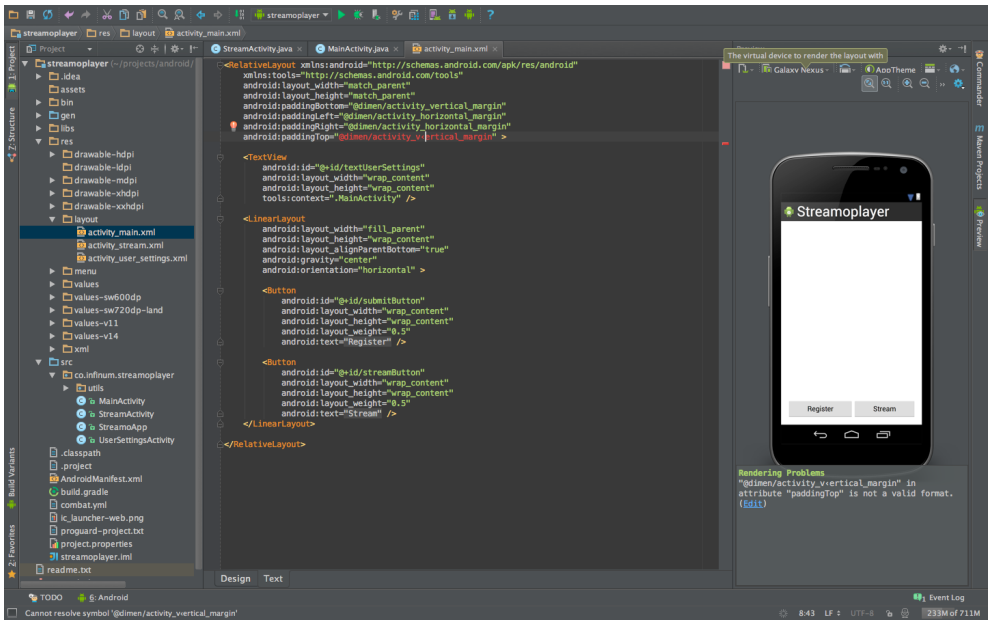

FIGURE 4.9 – Android Studio

5. Bluetooth Low Energy

Le Bluetooth Low Energy (BLE) est une technologie de communication sans fil utilisée notamment par beaucoup d'objets connectés tels que bracelets sport, vêtements ou même bouteilles d'eau.

Il ne s'agit pas simplement d'une nouvelle version du Bluetooth classique, mais bel et bien d'une nouvelle façon de voir les choses. Pour comprendre le BLE, il y deux notions à connaitre : GAP et GATT.

5.1 GAP : Generic Access Profile

C'est la partie qui contrôle la connexion entre deux appareils, ainsi que la visibilité d'un appareil. Deux types d'appareils existent :

— GAP Peripheral : des petits appareils à basse consommation, contenant souvent un ou plusieurs capteurs (exemple : un moniteur de rythme cardiaque)
— GAP Central : généralement un smartphone ou une tablette

Un GAP Central peut être connecté à plusieurs GAP Peripheral.

Afin d'initier une connexion entre un central et un périphérique, il faut que le périphérique soit visible pour le central. Une procédure, appelée *advertising*, permet de faire cela. En plus de partager le nom de l'appareil et son adresse réseau, le périphérique envoie également la liste des services et caractéristiques dont il dispose (voir section 5.2). Il peut d'ailleurs utiliser cet *advertising* pour transmettre des données à des appareils qui ne seraient pas connectés à lui.

5.2 GATT : Generic Attribute Profile

GATT est un protocole définissant la façon dont deux appareils communiquent au travers du BLE.

Un service permet de séparer les informations à transmettre en plusieurs entités logiques appelées caractéristiques. Il peut avoir une ou plusieurs caractéristiques. Un service se distingue d'autres services grâce à la présence d'un UUID (Universal Unique Identifier).

Le Bluetooth SIG (Special Interest Group) est un organisme chargé de la définition des normes et standards du Bluetooth. Ils ont défini un certain nombre de services standards. On retrouve notamment un service de mesure du rythme cardiaque, d'état de la batterie, ou encore de positionnement

intérieur. Chacun de ces services possède un UUID sur 16-bits fixé par le SIG [1]. Cela permet d'avoir un certain niveau d'interopérabilité entre central et périphériques. En effet, le service ainsi que sa structure étant connus, un développeur peut par exemple écrire une application de monitoring du rythme cardiaque de l'utilisateur en utilisant l'UUID `0x180D` et être ainsi compatible avec tous les capteurs de rythme cardiaque implémentant ce standard.

Il est également possible de définir ses propres services personnalisés, dont l'UUID est alors encodé sur 128-bits au lieu de 16.

Une caractéristique est le niveau le plus bas dans la pile BLE. Elle contient un seul point de données (ou un tableau de données liées dans le cas par exemple d'un accéléromètre à trois axes). Ici aussi, on peut utiliser des caractéristiques standards sur 16-bits [2] ou des caractéristiques personnalisées sur 128-bits.

Une caractéristique peut être définie comme accessible en lecture seule, écriture seule ou lecture et écriture. Ce niveau d'autorisation correspond aux actions mises à disposition du central connecté au périphérique.

1. Liste des services standards : `https://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx`
2. Liste des caractéristiques standards : `https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicsHome.aspx`

6. Présentation du code

6.1 Arduino pour le MicroView

Le Microview est une carte compatible Arduino. J'ai donc développé son code en utilisant l'IDE homonyme ainsi que plusieurs bibliothèques.

6.1.1 Bibliothèques

MicroView.h

Sparkfun, constructeur du Microview, a mis à disposition sur son site internet une bibliothèque Arduino nommée Microview, permettant à tout un chacun d'utiliser relativement facilement l'écran présent sur l'appareil sans avoir à gérer la communication interne entre le micro-contrôleur et l'écran via SPI.

On retrouve des fonctions d'affichage de caractères (avec plusieurs polices pré-enregistrées) ou encore de dessins (ligne, pixel, cercle, rectangle...).

Documentation : `http://learn.microview.io/doco/html/index.html`

Wire.h

Wire est une bibliothèque développée par la communauté Arduino. Elle permet de communiquer avec d'autres micro-contrôleurs via un bus I^2C. Dans le cadre de ce projet, elle a donc servi à la réception de messages depuis le BLE Nano.

Documentation : `https://www.arduino.cc/en/Reference/Wire`

QueueList.h

QueueList est une implémentation d'une file FIFO (premier arrivé, premier dehors) pour Arduino. Elle est utilisée comme mémoire tampon entre la réception d'un message et l'affichage à l'écran. La plupart du temps, cette file ne contiendra aucun ou un élément. Elle se remplira petit à petit dans le cas où l'utilisateur reçoit une grande quantité de notifications sur un laps de temps très court.

Documentation : `http://playground.arduino.cc/Code/QueueList`

Time.h

La bibliothèque Time permet d'accéder et de modifier la valeur de l'horloge interne. L'appareil devant être en capacité de donner l'heure à son utilisateur, elle est donc nécessaire à son bon fonctionnement.

Documentation : `http://www.pjrc.com/teensy/td_libs_Time.html`

6.1.2 Initialisation

GPIO

Les pins 3, 5 et 6 sont définis comme des sorties à l'aide de la fonction Arduino `pinMode()`. Ils sont reliés à trois LEDs. Lorsqu'un message arrive via le port I^2C, l'état d'une des LED est modifié. Ces trois LEDs ont essentiellement été utilisées pour du débug mais pourraient tout à fait être intégrées dans un produit fini, afin d'attirer l'attention de l'utilisateur.

Protocoles de communication

Le port série est relié à un ordinateur via le port USB du programmeur du Microview. C'est le seul moyen fiable permettant d'indiquer des messages de log ou de debug lors de la phase de développement.

On se connecte également à un bus de communication I^2C via la bibliothèque Wire et avec une adresse de `0x44`.

Synchronisation du temps

Une première fonction de réception des messages I^2C est définie à l'aide de Wire. Cette fonction, appelée `syncEvent`, permet de décoder un message de synchronisation envoyé par le BLE Nano sur le bus I^2C et d'utiliser la valeur transmise (un timestamp [1]) pour modifier sa propre valeur d'horloge interne (avec l'addition manuelle d'une heure pour le fuseau horaire).

Tant qu'un premier message de synchronisation valide n'a pas été reçu, la montre reste en attente et n'affiche rien d'autre.

Une fois ce premier message reçu, une autre fonction de réception est alors mise en place : `receiveEvent()`. En plus de permettre une resynchronisation de l'horloge, la fonction ajoute également les autres messages reçus à la file FIFO des messages à afficher. [2]

Timer

L'initialisation du Timer1 est faite en désactivant d'abord les interruptions, puis en éditant les valeurs des registres directement, avant de réactiver les interruptions.

Le Timer1 est configuré en mode CTC (Clear Timer on Compare Match), ce qui signifie que lorsque la valeur du compteur (stockée dans `TCNT1`) atteint le seuil défini dans le registre `OCR1A`,

1. Un timestamp est une façon de comptabiliser le temps sur certains systèmes informatiques. C'est un nombre entier correspondant au nombre de secondes écoulées depuis le 1er janvier 1970.
2. Un octet d'en-tête est ajouté par le BLE Nano avant transmission sur le bus I^2C pour définir si le message est un message de synchronisation d'horloge ou un texte à afficher.

le compteur est remis à zéro. Ce mode de fonctionnement est défini par les bits `WGM` des registres `TCCR1A` et `TCCR1B`.

L'horloge utilisée par le timer est l'horloge interne du micro-contrôleur, sur laquelle on applique un pre-scaler de 1024 (défini par les bits `CS` de `TCCR1B`). On obtient ainsi un signal d'une fréquence d'environ 16 KHz. En prenant en compte la valeur de 200 définie dans `OCR1A`, on voit que le flag du Timer1 est levé toutes les 12,8 ms.

Enfin, le bit `OCIE1A` du registre `TIMSK1` permet d'activer l'interruption et l'appel à la fonction correspondante contenue dans le vecteur d'interruption.

La routine d'interruption du Timer1 est définie dans le programme à l'aide de
`ISR(TIMER1_COMPA_vect) { ... }`

Cette routine a pour but d'afficher et de faire défiler le message courant sur l'écran du Microview. Pour cela, un calcul de coordonnées est effectué pour définir le premier pixel du premier caractère, puis le premier pixel à afficher sur l'écran une fois ce premier caractère disparu de l'écran. Cette fonction prend en compte la taille de la police utilisée pour l'affichage ainsi que la largeur de l'écran OLED.

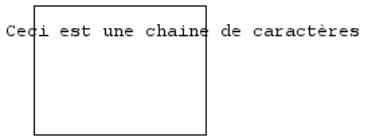

FIGURE 6.1 – Exemple d'affichage d'une chaine de caractère effectuée par la routine du Timer1

6.1.3 Boucle principale

Algorithm 1 Boucle d'exécution principale du Microview

```
fileNestPasVide ← Non file.estVide()
Si Non defillementEnCours Alors
  afficherLheure()
Fin Si
Si fileNestPasVide Alors
  Si defillementEnCours Alors
    dernierTour ← Vrai
    Tant que defillementEnCours Faire
      attendre()
    Fin Tant que
  Fin Si
  stopInterruptions()
  message ← file.pop()
  Si dernierCaractere(message) = '\r' Alors
    dernierTour ← Faux
    dernierCaractere(message) ← '\0'
  Fin Si
  activerTimer1()
  redemarrerInterruptions()
Fin Si
```

La boucle principale est une fonction relativement simple. Si la file de message n'est pas vide, alors elle en retire un élément et active le défilement de ce même message. Si aucun message n'est

en cours de défilement, alors elle affiche l'heure.

6.2 mbed pour le BLE Nano

Le BLE Nano a pour but de transmettre les messages du smartphone vers le Microview. Avant de transférer un message, il effectue une étape de mise en forme avec ajout d'un header permettant au Microview de savoir quel type d'informations il reçoit.

6.2.1 I^2C

Pour se connecter à un bus I^2C en tant que maître dans un programme mbed, il suffit d'instancier un objet (appelé ici `i2c_port`) de la classe `I2C` en donnant en paramètres les pins utilisés pour `sda` et `sdl` (respectivement `P0_10` et `P0_8`).

On définit une constante globale (`addr`) afin d'enregistrer l'adresse de l'esclave Microview (`0x44`).

Afin d'envoyer une commande à l'esclave via le bus I^2C, il suffit d'appeler une fonction : `i2c_port.write(addr, str, length(str))`

6.2.2 Bluetooth Low Energy

Afin de mettre en place la partie BLE du programme, je me suis inspiré des exemples présents sur le site développeur mbed : `https://developer.mbed.org/teams/Bluetooth-Low-Energy/code/BLE_GATT_Example/`

L'interaction avec la puce BLE est effectuée au travers d'une instance de la classe `BLEDevice`.

Définition des services et caractéristiques

La première étape consiste à définir, en variables globales, les UUID des services et des caractéristiques, ainsi que les tableaux ayant pour but d'accueillir le contenu des caractéristiques. Une fois cela fait, il faut définir les caractéristiques en elles-mêmes de la façon suivante :

```
WriteOnlyArrayGattCharacteristic<uint8_t, sizeof(writeValue)> writeChar(writeCharUUID,
    writeValue)}
```

Une fois toute les caractéristiques définies, il faut créer le service qui va les contenir :

```
GattCharacteristic *characteristics[] = {&readChar, &writeChar, &timeSyncChar};
GattService customService(customServiceUUID, characteristics,
    sizeof(characteristics)/sizeof(GattCharacteristic *));
```

Callbacks

Deux fonctions d'appels (callbacks) sont définies dans ce projet : `disconnectionCallback` et `writeCharCallback`.

La première se charge de redémarrer l'annonce (advertising) du périphérique BLE sur le réseau.

La seconde est un peu plus complexe. Elle possède deux comportements, en fonction de la caractéristique qui a été écrite.

S'il s'agit de `writeChar`, on ajoute une en-tête 'S' au début du message, met à jour le contenu de la caractéristique `readChar` et envoie enfin le message complet (avec en-tête) au Microview au travers du bus I^2C.

S'il s'agit de la caractéristique `timeSyncChar`, on ajoute une en-tête 'T' et on envoie le tout au Microview. L'octet d'en-tête permet au Microview de définir s'il doit mettre à jour son horloge ou simplement afficher le contenu du message sur son écran.

Initialisation

Une première fonction d'initialisation du `BLEDevice ble` est effectuée par un appel à `ble.init()`. Ensuite, on enregistre les deux fonctions de callbacks définies dans la section 6.2.2.

L'annonce est mis en place grâce au code ci-dessous :

```
ble.accumulateAdvertisingPayload(GapAdvertisingData::BREDR_NOT_SUPPORTED |
    GapAdvertisingData::LE_GENERAL_DISCOVERABLE);
ble.setAdvertisingType(GapAdvertisingParams::ADV_CONNECTABLE_UNDIRECTED);
ble.accumulateAdvertisingPayload(GapAdvertisingData::SHORTENED_LOCAL_NAME,
                                 (const uint8_t *)"BLE NOTIF", sizeof("BLE NOTIF") - 1);
ble.accumulateAdvertisingPayload(GapAdvertisingData::COMPLETE_LIST_16BIT_SERVICE_IDS,
    (uint8_t *)uuid16_list, sizeof(uuid16_list));
ble.setAdvertisingInterval(1600);
```

Une petite explication des constantes utilisées ci-dessus est nécessaire :

- BREDR_NOT_SUPPORTED : Le périphérique est compatible uniquement avec le Bluetooth Low Energy, et non le Bluetooth classique
- LE_GENERAL_DISCOVERABLE : Le périphérique est visible et peut être découvert à tout moment
- ADV_CONNECTABLE_UNDIRECTED : N'importe quel central peut se connecter à ce périphérique
- SHORTENED_LOCAL_NAME : permet de définir le nom du périphérique tel que visible par les utilisateurs en train de scanner la zone.

Pour finir, on ajoute le service avec `ble.addService(customService)` et on démarre la phase d'annonce avec `ble.startAdvertising()`

Boucle principale

La boucle principale du programme contient uniquement une ligne `ble.waitForEvent()`. Elle met en sommeil le micro-contrôleur en attendant de recevoir un message via Bluetooth.

6.3 Application Android

Le support du Bluetooth Low Energy est une fonctionnalité présente dans Android depuis la version KitKat 4.4. De nombreux changements dans l'API concernant le BLE ont été apportés dans la version suivante, Android Lollipop 5.0. J'ai donc décidé de partir sur une application ayant comme version minimum la version 5.0 (correspondant au Android SDK 21).

Une application Android peut contenir plusieurs types d'éléments :

— Activité : Il s'agit d'un écran de l'application. Une activité est généralement liée à une fonction de l'application.
— Service : Un service fonctionne en arrière plan. Il permet d'effectuer des actions longues ainsi que des actions qui doivent être exécutées lorsque l'application n'est pas au premier plan.
— BroadcastReceiver : Il s'agit d'une sorte de service qui est appelé lorsqu'un évènement précis intervient dans le système.
— AndroidManifest : Un fichier de configuration contenant notamment les différentes activités, services ainsi que les permissions requises pour l'exécuter.

6.3.1 AndroidManifest

Le fichier AndroidManifest de cet application précise le numéro minimum de SDK nécessaire à l'exécution de l'application. Dans notre cas, il s'agit de l'API 21 (voir section 5.

Afin de permettre à notre application d'utiliser le BLE, on doit définir la ligne suivante dans le fichier AndroidManifest. L'attribute `android:required` permet d'indiquer qu'un appareil ne supportant pas le BLE ne pourra pas installer l'application :

```
<uses-feature android:name="android.hardware.bluetooth_le" android:required="true"/>
```

Quatres permissions sont requises au bon fonctionnement de l'application :

— `BLUETOOTH` : nécessaire à toute communication Bluetooth
— `BLUETOOTH_ADMIN` : permet à l'application d'initier une connexion avec un périphérique Bluetooth
— `RECEIVE_SMS` : permet à l'application d'être notifiée à l'arrivée d'un SMS (voir section 6.3.6)
— `READ_SMS` : autorise l'application à lire le contenu des SMS reçus

Les activités et services de l'application sont également définis dans le Manifest. Une mention spéciale pour le `BluetoothLeService` est ajoutée : `android:enabled="true"`. Elle précise que le système doit démarrer le service au démarrage de l'application.

6.3.2 BluetoothLeService

Le service BluetoothLeService se lance au démarrage de l'application. Il fonctionne tant que l'application tourne, même si elle n'est pas au premier plan.

Son rôle est de gérer la transmission et réception de messages à travers le BLE. Chaque activité ou service souhaitant communiquer avec la montre doit se lier d'abord au service et appeler les méthodes du service.

6.3.3 MainActivity : première écran de l'application

Au démarrage, l'application affiche un simple bouton permettant de démarrer une seconde activité `DiscoverActivity`.

6.3.4 DiscoverActivity

L'activité récupère le BluetoothAdapter du système et démarre le scan de périphériques. Elle contient une liste des périphériques trouvés ainsi qu'un menu permettant de démarrer ou d'arrêter

le scan.

Elle vérifie que le Bluetooth est bien activé. Si non, elle demande à l'utilisateur de l'activer.

Lorsque l'utilisateur sélectionne un périphérique dans la liste, une troisième activité est démarrée.

6.3.5 DeviceControlActivity

Au démarrage, elle récupère le nom et l'adresse du périphérique depuis les données extras envoyées par l'activité précédente, et les affiche à l'utilisateur.

Elle se lie également au service `BluetoothLeService`. La communication Bluetooth n'étant faite que par le service, il est nécessaire de se lier à lui pour effectuer des actions.

Un BroadcastReceiver appelé `mGattUpdateReceiver`, est créé. Il écoute tout changement apporté à une caractéristique GATT (connexion, déconnexion, changement de valeur, découverte...).

Lorsque des services sont découverts par ce BroadcastReceiver, une fonction nommée `displayGattServices` est appelée. Elle passe en revue chaque service et caractéristiques du périphérique. Pour la caractéristique de synchronisation temporelle, elle écrit la valeur du timestamp de l'appareil Android. Pour la caractéristique de texte à afficher, elle définit l'attribut `watchCharacteristic` avec la caractéristique et enregistre le `SMSReceiver` pour recevoir les SMS entrants.

6.3.6 SMSReceiver

`SMSReceiver` est une classe héritant de `BroadcastReceiver`. Un `BroadcastReceiver` est un objet destiné à recevoir des Intents.

Lorsqu'un SMS est reçu par le smartphone, la fonction `onReceive()` de `SMSReceiver` est appelée. La fonction crée un tableau de `SmsMessage` à partir des informations contenues dans l'Intent reçu.

Au départ, j'ai essayé de transmettre directement le contenu du SMS à `BluetoothLeService`. Malheureusement, il n'est pas possible pour un `BroadcastReceiver` de se lier à un service existant. Afin de contourner le problème, j'ai donc créé un service `ForwardService` (voir section 6.3.7) qui est démarré par `onReceive()` avec le contenu du message en extra.

6.3.7 ForwardService

`ForwardService` est donc un service créé exclusivement pour créer transmettre le contenu d'un SMS du `SMSReceiver` vers le `BluetoothLeService`.

À son démarrage, le service se lie au `BluetoothLeService`. Il récupère le contenu du SMS à partir des données extras communiqué par `SMSReceiver` et le renvoie au `BluetoothLeService` avec la fonction `sendToWatch()`. Une fois cela fait, le service s'arrête de lui-même.

7. Caractéristiques et coûts

Type	**Modèle**	**Prix**
Microcontrôleur	Sparkfun Microview [1]	35.50 €
Programmeur	SparkFun Microview USB Programmer [2]	13.29 €
Microcontrôleur	RedBear BLE Nano Kit (+ Programmer) [3]	30.35 €
Total		79.14 €

1. https://www.sparkfun.com/products/12923
2. https://www.sparkfun.com/products/12924
3. http://redbearlab.com/blenano/

8. Conclusion

Le développement de ce projet m'a permis de découvrir trois plateformes de développement : Arduino, mbed et Android. J'avais dors et déjà effectué de petits programmes exemples de type Hello World sur chacune des trois plateformes mais jamais de projet de cette envergure.

Ce fût également pour moi l'occasion de découvrir le fonctionnement du Bluetooth Low Energy, une technologie sans fil pleine d'avenir dans un monde où l'Internet des Objets devient de plus en plus présent.

Une fois le projet terminé, je me suis rendu compte d'un problème lors de l'affichage de certains SMS. Seul les SMS courts sont transmis et affichés sur l'écran de la smartwatch. Il se trouve que le BLE ne permet d'écrire qu'une série de 20 octets dans une caractéristique à la fois. Tout SMS de plus de 20 caractères n'est donc pas transmis (ou en tout cas pas reçus par le BLE Nano).

Une solution possible serait de découper le message pour le transmettre en plusieurs fois avec un processus de handshake permettant au BLE Nano de demander au smartphone l'envoie du paquet suivant.

Comme expliqué au début du rapport, il aurait été intéressant d'implémenter quelques capteurs ainsi que des moyens d'interaction de l'utilisateur avec la montre (boutons et menus). Malheureusement, cela n'a pas été possible par manque de temps.