Projet IoT : BykeEmbedded

Franck Labracherie, Victor Costa, Arno Dupère

2022-2023

Sommaire

1 Introduction

Au cours du premier semestre de l'année scolaire 2022-2023, le projet Byke fut l'un des projets assignés aux élèves de 5A. Une des tâches de ce projet est le développement d'une application sur un micro-controlleur, qui sera connecté au reste du projet.

Nous vous présenterons premièrement le projet de manière succincte, avant de nous concentrer sur le micro-contrôleur et le déroulement de l'exécution du projet.

2 Byke

Byke est un projet visant à développer un service pour cyclistes. Ce dernier est composé de plusieurs applications:

- un micro-service développé en Java avec le framework Spring Boot,
- une application web développée avec VueJs et TypeScript,
- une application mobile développée en Kotlin,
- une interface sur micro-contrôleur développée en Rust.

Le micro-contrôleur est connecté à l'application mobile, qui est elle-même connectée au micro-service. L'application web est également connectée au micro-service et sert à gérer les données. Nous allons nous concentrer sur le micro-contrôleur.

3 Déroulement du projet

3.1 Lancement

3.1.1 Resources

Au lancement du projet, un M5Go IoT Starter Kit nous a été fourni. Il s'agit d'un kit vendu par M5Stack contenant un micro-contrôleur de type ESP32, et un ensemble de 6 composants:

- un potentiomètre,
- un hub à trois ports,
- une télécommande infrarouge,
- un capteur de proximité,
- un capteur d'humidité,

- trois LED (rouge, verte et bleue).

Le micro-contrôleur contient lui-même plusieurs composants:

- une bande de 10 LEDs RGB,
- un microphone,
- un écran LCD,
- un haut-parleur,
- trois boutons,
- un port UART,
- un port I/O,
- un port I2C.

Un GPS nous a également été fourni, car une des contraintes du projet est d'utiliser ce dernier. Nous en avons reçu un de type Adafruit Ultimate GPS FeatherWing.

Après quelques semaines, deux autres contrôleurs nous ont été fournis: un M5CoreInk et un m5StickC Plus.

3.1.2 Conception

Nous avons commencé par nous demander ce que nous allions développer. Après réflexion, nous avons décidé plusieurs choses:

1. le micro-contrôleur serait connecté à l'application mobile par Bluetooth Low Energy, et non pas par Bluetooth, pour consommer moins;
2. ce dernier servirait comme compagnon de route du cycliste (cf Annexes Figure 1):
    - il donnerait des informations sur le trajet,
    - il permettrait de définir une nouvelle étape sur un trajet;
3. Nous allions utiliser les boutons, les LEDs, le haut-parleur et l'écran.

3.2 Langage

Bien que le langage utilisé en cours pour les programmes destinés aux systèmes embarqués est le C++, nous avons décidé d'utiliser le Rust. En effet, c'est un langage bas niveau et compilé, mais l'avantage par rapport au C++ est, en plus de meilleurs temps de compilation et d'une plus grande rapidité (cf Annexe 2), le fait que ce langage est plus sûr pour la mémoire. À cela on peut ajouter une syntaxe que nous trouvons plus intéressante. Le problème majeur

de ce langage est l'apprentissage, car Rust est très complexe et intègre des concepts peu répandus dans les autres langages, comme le principe de propriété (*ownership*). Cependant un membre du groupe avait déjà de l'expérience avec ce langage, ainsi le problème ne s'est pas vraiment posé.

Cela a été assez compliqué au début de mettre en place tout l'environnement de travail, mais il existes plusieurs tutoriels en ligne, ainsi cela n'a pas non plus pris trop de temps.

Un des avantages que nous avons pu avoir plus tard est l'existence de plusieurs bibliothèques liées à la programmation sur ESP32 et sur les différents composants disponibles sur le M5Go (par exemple les LEDs ou le Blueooth Low Energy). Cela a représenté un gain de temps considérable: par exemple sur l'écran, la bibliothèque codait déjà plusieurs méthodes de base comme l'envoi des commandes nécessaires à la configuration de l'écran.

Cependant le fait qu'il y ait des bibliothèques peut engendrer des problèmes de dépendances et de versions. Cela a été le cas pour le Bluetooth Low Energy, et dans ce cas nous avons dû *fork* le dépôt Git, et nous l'avons modifié nous-même pour pouvoir l'utiliser.

Pour le développement à proprement parler nous avons utilisé VSCode, qui propose des extensions pour le Rust permettant un développement optimal.

3.3 La bibliothèque

3.3.1 Les boutons

Après avoir mis en place l'environnement de développement, nous nous sommes dit que la meilleure manière de commencer ce projet serait de créer une bibliothèque qui nous permettrait à plus long terme de développer l'interface plus facilement. La tâche était de créer une bibliothèque permettant à n'importe qui de développer des applications le plus facilement possible sur un M5Go.

Au départ, nous n'avons rencontré aucun problème: nous arrivions à faire fonctionner les boutons. En effet ce n'était pas compliqué, une fois que nous avions compris que l'état *Low* correspondait au bouton appuyé, et l'état *High* au bouton relaché.

Plus tard dans le projet, nous avons essayé de faire fonctionner les *interrupts*. Tout étant implémenté dans la bibliothèque de base pour ESP32, cela n'a pas été compliqué, mais la recherche a pris du temps. La seule complication que nous avons rencontrée est que les méthodes appelées sur interrupt ne peuvent pas utiliser la bibliothèque **std** (bibliothèque standard qui s'occupe elle-même de l'allocation), pour des problèmes de mémoire. Ainsi il a fallu que toutes les

méthodes appelées respectent cette spécification. Il a également fallu trouver un moyen de garder des données de manière statique pour pouvoir y accéder dans les *interrupts*, donc également des sections critiques pour éviter les accès simultanés à une même ressource.

3.3.2 Les lumières:

Premièrement, nous avons simplement essayé d'allumer le Gpio15 (broche sur laquelle la bande est branchée). Cela a fonctionné, mais ce n'était pas ce que nous voulions: il fallait pouvoir écrire sur des LEDs en série. Notre premier réflexe a été de chercher un moyen d'importer la bibliothèque Adafruit Néopixel, qui est la bibliothèque C++ utilisée en cours pour ce type de bande. En effet, il est possible d'intégrer à un projet Rust du code C++ qui pourra être exécuté[1]. Nous avons alors créé une petite bibliothèque contenant deux méthodes: une à importer et l'autre non. Nous avons ensuite intégré une référence au fichier *header* dans le fichier de *build* de la bibliothèque. Nous avons pu remarquer que le code était effectivement compilé et qu'un fichier de lien en Rust avait effectivement été créé, mais nous n'arrivions pas à appeler le code. Après plus d'une dizaine d'heures de recherches et d'échec nous avons décidé d'abandonner, et d'essayer de tout faire en Rust (le problème le plus récurrent était un problème de *linker* pas trouvé).

Nous avons finalement trouvé une bibliothèque qui permettait d'écrire sur la bande et qui intégrait même des couleurs. Nous avons donc pu prendre en main cette dernière et l'intégrer à notre bibliothèque.

3.3.3 L'écran

Grâce à notre expérience précédente nous savions qu'il faudrait en premier lieu chercher si une bibliothèque existait pour notre type d'écran, et ce fut le cas. Nous avons alors importé cette bibliothèque et grâce aux exemples qu'elle fournissait et à la bibliothèque C++ de M5Stack pour la gestion de l'écran, nous avons pu comprendre le code, et utiliser ce dernier afin de coder notre propre surcouche de cette dernière.

Nous avons donc intégré à la bibliothèque quelques fonctions basiques:

- allumer / éteindre l'écran (avec la broche *blk*)
- remplir l'écran d'une couleur
- dessiner du texte

À cela nous avons rajouté ultérieurement une méthode pour dessiner une image.

[1] D'ailleurs, étant donné que ce code ne provient pas du Rust, il n'y a aucun moyen de savoir s'il est *memory-safe*, ainsi dans le Rust, ce code est marqué et exécute comme *unsafe*.

3.3.4 Le haut parleur

Le haut-parleur est contrôlé avec un contrôle LEDC. Comme son nom l'indique, ce dernier est à la base utilisé pour gérer les LEDs et leur fréquence de clignotement, mais a été généralisé à tout type de signal de type PMW (*Pulse Width Modulation*). Ainsi en modulant la fréquence et l'intensité des signaux (pour cette dernière grâce aux *duty*), on peut générer tout type de signal audio par exemple. C'est donc ce que nous avons utilisé pour la gestion du speaker.

Au début nous ne savions pas vraiment comment il fonctionnait, alors nous avons mis sa fréquence à 1s pour pouvoir l'analyser. Nous avons pu remarquer que l'impulsion envoyée se traduit par une sorte de vibration que l'on pourrait représenter grossièrement par l'impact d'une bille sur un mur. En augmentant la fréquence nous pouvions donc générer des tonalités intéressantes. Nous avons donc d'implémenter les différentes notes (ensuite il suffit de diviser la fréquence par $2^{8-octave}$ pour un plus grand spectre).

Dans le projet nous avons eu l'idée de l'utiliser pour faire un son lorsque l'on appuie sur un bouton. Cependant à cause des spécificités de l'implémentation du contrôle LEDC dans la bibliothèque utilisée et du manque de temps, nous n'avons pas pu l'utiliser dans notre projet. En effet le problème est qu'il est impossible pour l'instant de simplement modifier la fréquence: on est obligé de créer un nouveau *driver*. Ainsi cela pose des problèmes de section critique.

Nous aurions voulu ajouter à la bibliothèque un moyen de mieux moduler le son afin changer le timbre par exemple mais nous n'avons pas eu le temps nécessaire pour.

3.3.5 Le GPS: gestion du port UART

À ce moment là dans le projet nous avons commencé à essayer de faire fonctionner le GPS. Le GPS envoyant de l'information en continu, il faut le brancher au port UART (port C) (*Universal Asynchronous Receiver Transmitter*) de notre système. Cela nous permet de lire l'information en continu.

Premièrement, nous avons essayé de lire ce que le GPS envoie pour comprendre comment récupérer les informations. Nous avons remarqué qu'il envoyait du texte contennant du texte, des virgules, des retours à la lignes ('\n'), et des sortes de commandes de type $GP[trigramme] (GP pour *Global Positioning*). Avec quelques recherches nous avons pu trouver que ce sont des trames NMEA (*National Marine & Electronics Association*). Nous avons trouvé une bibliothèque implémentant un *parser* NMEA, ainsi nous n'avons pas eu à le faire nous même.

Concernant la lecture des trames nous avons réimplémenté la méthode trouvée sur l'exemple GPS disponible sur l'IDE Arduino Uno: lire les caractères un à un, en attendant les symboles de début ('$') et fin ('\n') de trame.

Étant donné que c'est un cas spécial, nous n'avons pas rajouté d'implémentation de la gestion du GPS à notre bibliothèque, mais nous avons rajouté le port C.

3.3.6 Le potentiomètre: gestion du port I/O

Pour apprendre à utiliser le port I/O, nous avons décidé d'utiliser le potentiomètre fourni avec le kit M5Go (composant Angle). En lisant ce que nous recevions, nous avons remarqué que l'intensité envoyée avait un maximum et un minimum, alors nous avons simplement implémenté un exemple[2] changeant l'intensité des lumières en fonction de l'angle du potentiomètre.

3.3.7 Le Bluetooth Low Energy

Comme son nom l'indique, le BLE diffère du Bluetooth par sa faible consommation énergétique, et le fait qu'il est fait pour envoyer peu d'informations.

Nous avons trouvé une bibliothèque permettant de faire tourner le BLE sur un système ESP32. Cependant, comme expliqué plus tôt, elle posait un problème de dépendance lorsqu'on l'intégrait, alors nous l'avons modifiée conformément à nos dépendances.

Nous aurions pu implémenter la connexion BLE sécurisée, mais nous n'avions pas le temps, car cela voudrait dire implémenter un autre type d'appairage côté application mobile. Nous nous sommes donc contentés de la connexion basique. Nous avons testé cette dernière avec un exemple, et tout fonctionnait. Nous avons quand même modifié la bibliothèque, pour lui rajouter la possibilité de gérer les déconnexions.

3.3.8 Le capteur environnemental: gestion du port I2C

La connexion I2C est une connexion maître-esclave. L'esclave décide de son adresse, et peut *broadcast* des messages avec une certaine taille maximale. Le maître écrit et/ou lit sur une seule adresse.

Pour illustrer cette fonctionnalité nous avons implémenté un exemple qui lit des informations de température et humidité de l'air. Ce qui est intéressant avec ce composant est que l'on peut lui demander d'envoyer les données de manière unitaire, mais aussi d'envoyer en boucle les données à une certaine fréquence, le tout en lui envoyant certaines commandes.

[2]Les bibliothèques Rust peuvent intégrer des fichiers "exemples" permettant aux utilisateurs de tester ces dernières et leurs différentes fonctionnalités.

3.4 L'application

Après avoir terminé avec la bibliothèque, nous avons commencé à développer l'application même sur le système.

3.4.1 Accueil

La première page que nous avons développée est une page d'accueil, qui correspond à une liste d'options. Cette page au final propose 3 options:

- une page de connexion BLE,
- une page d'affichage d'informations sur l'excursion,
- une page d'options.

3.4.2 Bluetooth

Le problème que nous avons eu a été que lorsqu'on a activé les *interrupts* pour les boutons, nous avons remarqué que l'*interrupt* du bouton A s'activait de manière répétée. Après quelques recherches nous avons remarqué qu'en effet le lancement du Bluetooth sur un ESP32 envoie des signaux à la broche GPIO39. Nous avions envisagé de récupérer un bouton en plus, mais au final il nous a semblé plus intéressant de connecter un autre système. Nous avons donc décidé d'utiliser le système M5StickC Plus cité plus haut, sur lequel nous avons réimplémenté le BLE. Ce dernier est connecté au M5Go en I2C par un hub sur lequel est aussi branché le capteur environnemental. C'est le choix idéal car ça permet au système principal de communiquer avec le système secondaire et inversement.

La connexion au BLE de l'application mobile se fait par scan d'un QR code contenant l'adresse mac du composant qui se connecte (ici le M5StickC Plus).

Sur l'écran d'informations sur l'excursion l'utilisateur peut créer de nouvelles étapes au fur et à mesure qu'il avance, et ainsi crér son propre chemin.

3.4.3 Communication

Il nous fallait donc un moyen de faire communiquer les objets entre eux, donc nous avons décidé d'implémenter un système de trames commandes sur 256 bytes:

1. Byte 1: code identifiant de la commande;
2. Byte 2: longueur du message contenu;
3. Bytes 3-256: message au format JSON.

Cela permet l'interopérabilité entre tous les appareils, dont le mobile, car les trames sont définies et leur logique est commune, ce qui facilite l'ajout de nouvelles trames. De plus le JSON permet pour tous les appareils une désérialisation en objets connus.

Le problème que nous avons eu avec ce système est son implémentation avec le BLE. En effet, 256 bytes c'est possible en I2C, mais pas en BLE, qui peut transmettre une vingtaines de bytes. Nous avons donc dû implémenter un système de demandes répétitives en cas de trames incomplètes, côté IoT et Android.

3.4.4 Affichage

Pour la gestion de l'affichage nous avons décidé d'implémenter le système suivant:

- Nous avons un objet *GraphicBox* qui peut contenir du texte, et peut être dessinée;
- Un *Screen* contient une liste de *GraphicBox*, et le dessiner revient à dessiner l'ensemble de ces dernières;
- Un objet *App*, qui correspond à l'application, contient une liste de *Screen*, et on ne dessine que l'écran actif.

On en arrive à la gestion d'un state général.

3.4.5 Le state

Un objet State est passé à l'objet *App*, et ce dernier l'envoie à ses enfants. Le state est utilisé par les screens pendant les événements déclenchés par les interrupts, et pendant un update qui se fait à chaque itération du *loop* principal. Cela permet à notre application d'être plus ergonomique, et d'être facile à la prise en main.

4 Pour conclure...

Ce projet nous a appris beaucoup sur les systèmes embarqués en général et une multitude de notions tels que les différents types de connexion. Cela nous a également forcé à trouver des solutions et chercher plus loin qu'une simple demande sur StackOverflow.com, tout en pensant aux différentes problématiques que chaque solution va apporter.

De plus nous avons pu en apprendre plus sur comment faire fonctionner plusieurs projets inter-dépendants. Ça a été une expérience plus qu'enrichissante et nous avons pu gagner beaucoup d'expérience sur plusieurs types de compétences, autant techniques que théoriques.

5 Annexes

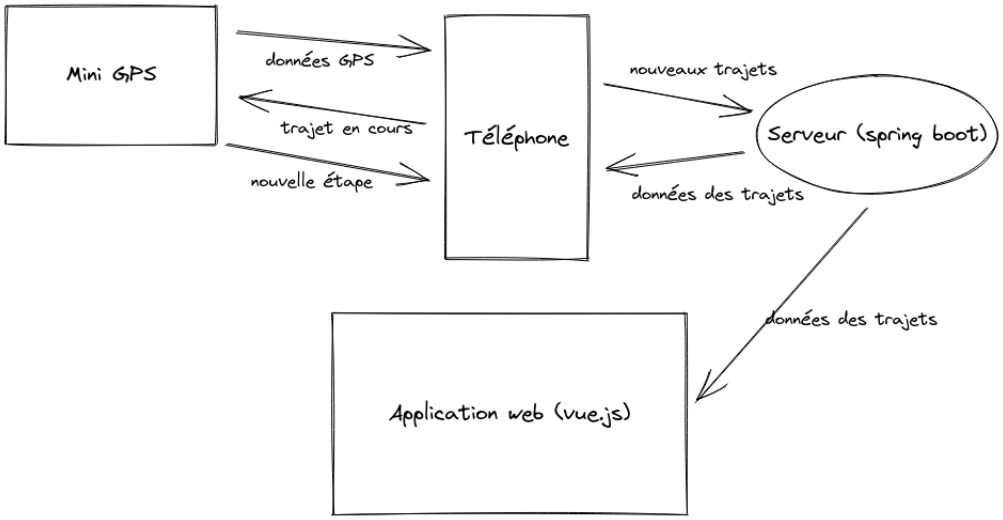

Figure 1: Maquette du fonctionnement de Byke

Table 4. Normalized global results for Energy, Time, and Memory

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Figure 2: Classement des langages de programmation par performance (étude de Green Software Lab)