SMOC - Rapport du Projet BUMPERS

Corentin Dupret, Sami Tabet

25 février 2020

Table des matières

1 Introduction **3**

2 Architecture du projet **4**

3 Éxécuter le projet **5**

4 Étapes suivies **6**
4.1 Interface web et affichage du plateau 7
4.2 Gestion de l'accéléromètre 7
4.3 Communication entre les deux cartes 8
4.4 Intégration de la carte XBee 9

5 Conclusion **10**

1 Introduction

Le jeu BUMPERS est un jeu où il y a un arbitre et des joueurs. Chaque joueur gère le mouvement d'une bille à l'aide de l'accéléromètre d'un microcontrôleur. L'arbitre place chaque bille sur le plateau dans l'ordre d'arrivée des joueurs. Puis, l'arbitre lance le jeu et gère le mouvement des billes à partir des informations envoyées par les joueurs. Ceux-ci doivent diriger leur bille vers les autres pour provoquer des chocs, comme au billard. Le but est de faire sortir les billes des autres joueurs du plateau car elles sont ainsi éliminées. Le dernier joueur dont la bille reste sur le plateau a gagné le jeu.

Le projet consiste en la création d'un client permettant de jouer au jeu BUMPERS avec d'autres clients. Le client est composé de deux microcontrôleurs STM32F4DISCOVERY, connectés entre eux, qui communiquent avec l'arbitre (c'est-à-dire le serveur) par fréquence radio.

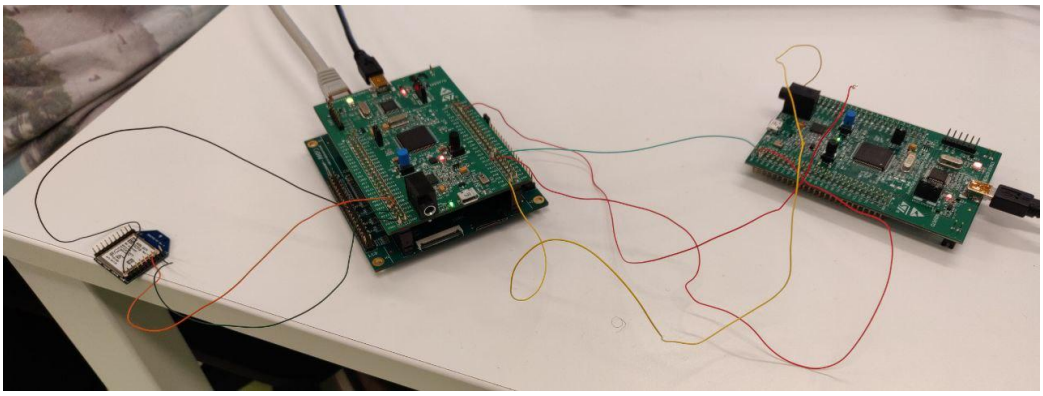

FIGURE 1 – Photo du projet

2 Architecture du projet

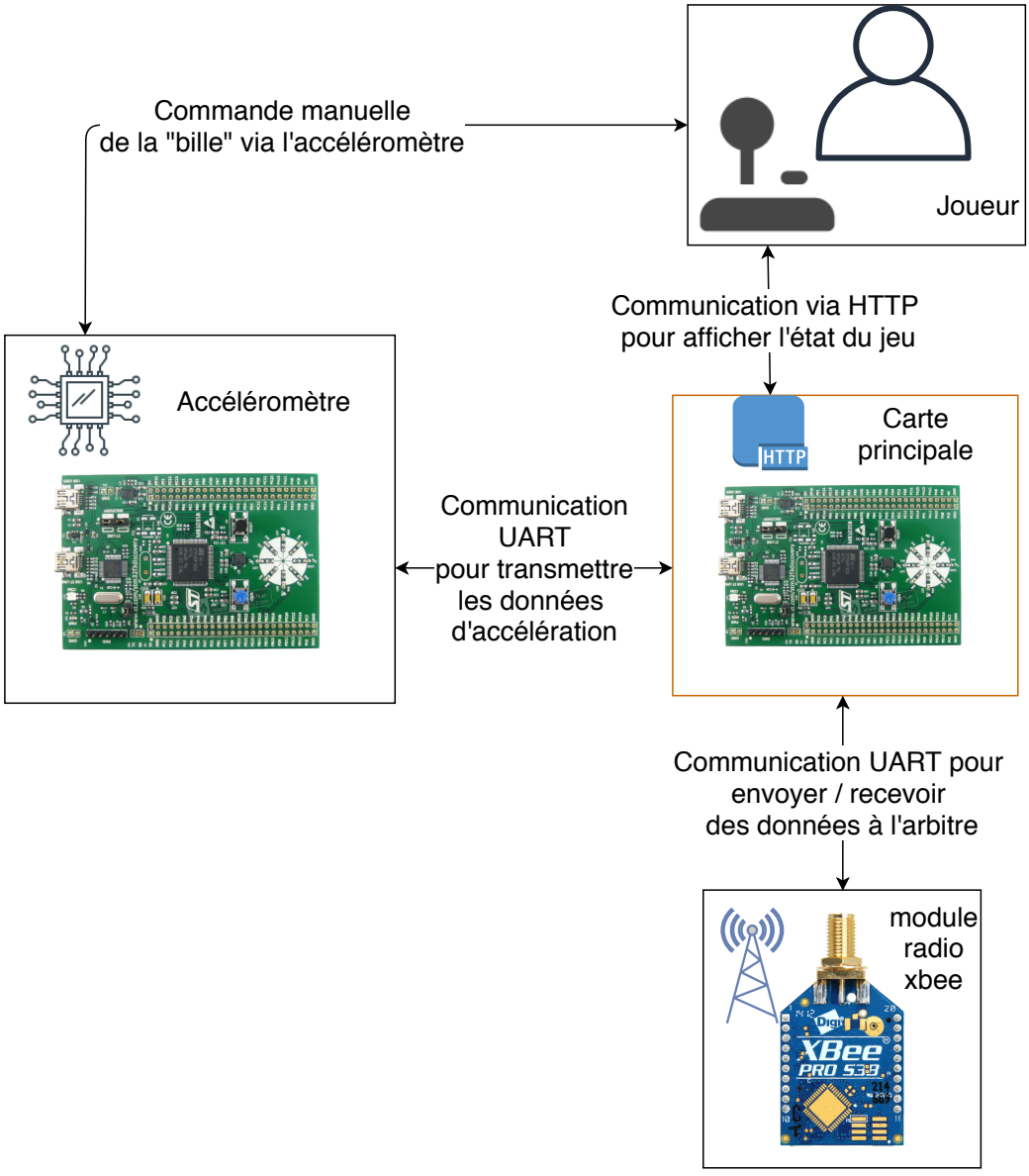

FIGURE 2 – Schéma de l'architecture du projet

Le projet se découpe en plusieurs composants intéragissant entre eux pour former le client :

— Un DevKit407 complet faisant tourner une interface web.

— Un module radio XBee permettant de communiquer avec l'arbitre par radio. Le joueur envoie les accélérations pour contrôler la bille tandis que l'arbitre envoie l'état du jeu avec, notamment, la position de la bille de chaque joueur.

— Une connexion UART entre le DevKit407 et le XBee.

— Un STM32F4DISCOVERY sans DevKit chargé de récupérer les données de l'accéléromètre MEMS intégré. Le joueur peut contrôler le mouvement de sa balle en bougeant ce microcontrôleur dans l'espace.

— Une connexion UART entre les deux microcontrôleurs pour permettre au STM32F4DISCOVERY de communiquer avec le DevKit407.

— Une connexion Ethernet entre le DevKit407 et un ordinateur pour permettre au joueur d'accéder à l'interface web afin de prendre ses décisions en connaissance de cause.

La raison pour laquelle deux microcontrôleurs sont utilisés est que l'accéléromètre ne fonctionne pas avec le DevKit407, qui est nécessaire pour disposer d'une connexion Ethernet avec le joueur. Par conséquent, deux microcontrôleurs sont utilisés et communiquent en UART pour que le microcontrôleur principal (le DevKit407) puisse disposer des données de l'accéléromètre à envoyer à l'arbitre via le module radio, tout en exposant une interface web permettant au joueur de jouer en connaissant l'état du jeu à chaque instant.

Par conséquent, notre code source contient deux dossiers distincts, permettant de construire deux binaires différents : un pour chaque microcontrôleur. `acc` contient le code source pour le STM32F4DISCOVERY utilisant l'accéléromètre, tandis que `web` contient le code source pour le DevKit407 exposant l'interface web.

Le contenu de ces dossiers sera expliqué dans les différentes parties du rapport, avec des extraits de code source.

3 Éxécuter le projet

Pour éxécuter le projet, il faut au préalable construire les fichiers binaires des deux cartes :

— La première responsable du serveur web et de la communication radio avec l'arbitre

— La seconde responsable de la partie accéléromètre

Pour cela il faut (pour chaque carte branchée en USB) :

— Aller dans le répertoire `web/acc`

— Lancer la commande `make` pour construire le fichier `.elf` qui ira sur la carte

— ouvrir 2 terminaux

— dans le premier, lancer la commande `st-util` (outil installable depuis ici : st-util)

— dans le second, lancer une session `gdb` (**attention**, ce doit être une session gdb compatible avec l'architecture de la carte, par exemple `arm-none-eabi-gdb` ou `gdb-multiarch`)

— dans cette session `gdb`, lancer les commandes :

  — `target extended-remote :4242`

  — `load build/web.elf` ou `load build/acc.elf` suivant la carte actuelle

  — Une fois le chargement terminé, lancer la commande : `continue`

4 Étapes suivies

Nous détaillons ici les différentes étapes suivies pour réaliser le projet, en expliquant le code source associé à chacune d'entre elles. Les étapes sont les suivantes :

— Réalisation de l'interface web, avec l'affichage du plateau.

— Gestion de l'accéléromètre : récupérer les accélérations calibrées et les convertir (normaliser) avant l'envoi au second microcontrôleur.

— Gestion de la communication entre les deux cartes : le STM32F4DISCOVERY envoie les accélérations normalisées au DevKit407 qui les reçoit et les décode avant de les envoyer à l'arbitre.

— Intégration de la carte XBee au DevKit407 pour communiquer avec l'arbitre.

4.1 Interface web et affichage du plateau

Le projet (la carte `web`) dispose d'une interface web qui permet notamment d'afficher le plateau de jeu (les positions de toutes les billes en jeu) mais également de :

- Lire des logs de debugging provenant du code relatif à la communication radio
- Configurer le baudrate à utiliser pour la communication radio
- S'assurer que la fonctionnalité de serveur web fonctionne correctement à l'aide de routes HTTP permettant d'allumer et d'éteindre des LEDs (permet d'aider le débugging).

Pour accéder à cette interface web il faut se connecter en ethernet sur la carte `web` (en configurant son ordinateur en réseau local) et aller sur la route : `192.168.1.29` qui a été définie au sein de CubeMX.

Les fichiers statiques sont stockés en tant qu'array de bytes dans le fichier binaire final pour éviter d'avoir à configurer un filesystem sur la carte.

Leur insertion est facilitée par des outils tels que bin2c.

Afin d'avoir un visuel en quasi "temps réel" du plateau de jeu, une requête est effectuée toutes les 50 millisecondes à la carte depuis le code javascript de la page web. Cela permet d'avoir une interface simple en HTTP sur la carte qui renvoie les dernières positions des joueurs reçues par l'arbitre sous format JSON sans avoir à gérer de protocoles temps réels tels que des WebSockets.

4.2 Gestion de l'accéléromètre

Le code correspondant à la gestion de l'accéléromètre se trouve dans le dossier `acc`.

Pour communiquer avec l'accéléromètre, nous avons choisi d'utiliser une bibliothèque écrite par Mohamed Yaqoob. Cette bibliothèque expose une API relativement simple à utiliser et elle est parfaitement adaptée à notre utilisation de l'accéléromètre. Le code correspondant se trouve dans les fichiers `acc/Src/Include/LIS3DSH.h` et `acc/Src/LIS3DSH.c`.

La bibliothèque définit une structure `LIS3DSH_DataScaled` composée de trois attributs de type `float` qui correspondent aux accélérations dans les trois dimensions spatiales. Nous définissons, dans `acc/Src/main.c`, une variable `accData` de ce type.

De plus, la bibliothèque expose une structure `LIS3DSH_InitTypeDef` qui correspond à la configuration de l'accéléromètre. Nous définissons donc une variable `accConfigDef` afin de configurer la communication avec l'accéléromètre à travers cette bibliothèque.

L'extrait de code en figure 3 montre cette configuration.

```
144       /* USER CODE BEGIN 2 */
145       accConfigDef.dataRate = LIS3DSH_DATARATE_12_5;
146       accConfigDef.fullScale = LIS3DSH_FULLSCALE_4;
147       accConfigDef.antiAliasingBW = LIS3DSH_FILTER_BW_50;
148       accConfigDef.enableAxes = LIS3DSH_XYZ_ENABLE;
149       accConfigDef.interruptEnable = false;
150       LIS3DSH_Init(&hspi1, &accConfigDef);
151
152       LIS3DSH_X_calibrate(-1030.0, 1025.0);
153       LIS3DSH_Y_calibrate(-1035.0, 1035.0);
154       LIS3DSH_Z_calibrate(-940.0, 1010.0);
```

FIGURE 3 – Code pour configurer l'accéléromètre

Les fonctions `calibrate` permettent de calibrer l'accéléromètre, c'est-à-dire faire en sorte que les valeurs minimales et maximales des accélérations soient le plus proches possible de -1000 et 1000. Les paramètres passés sont les accélérations minimales et maximales mesurées avant la calibration.

L'extrait de code en figure 4 est celui qui récupère les valeurs de l'accéléromètre toutes les secondes, normalise les valeurs pour avoir l'échelle attendue par l'arbitre et enfin transmet les valeurs normalisées (`accMove`) au DevKit407. Ce code fait appel à la fonction `convertAccData` pour normaliser les données dont le code source est présenté en figure 5.

4.3 Communication entre les deux cartes

La communication entre les deux cartes se fait via `UART`.

Les pins `UART6` de la carte `acc` sont connectées aux pins `UART5` de la carte `web`.

```
159       /* Infinite loop */
160       /* USER CODE BEGIN WHILE */
161       while (1) {
162           /* USER CODE END WHILE */
163
164           /* USER CODE BEGIN 3 */
165           if (LIS3DSH_PollDRDY(1000)) {
166               accData = LIS3DSH_GetDataScaled();
167               HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12);
168           }
169           convertAccData(&accMove, &accData);
170           HAL_UART_Transmit(&huart6, &accMove, 3, 1000);
171       }
172       /* USER CODE END 3 */
```

FIGURE 4 – Code pour récupérer les accélérations et les transmettre

Les données transitant au sein de cette connection sont les données provenant de l'accéléromètre (en soi nous avons uniquement besoin d'envoyer des données de `acc` vers `web`, ainsi connecter toutes les pins UART n'est pas nécessaire).

Ces messages ont une taille de 3 bytes et contiennent :

— L'accélération sur `x` encodée en `int8` sur 1 byte

— L'accélération sur `y` encodée en `int8` sur 1 byte

— L'accélération sur `z` encodée en `int8` sur 1 byte

4.4 Intégration de la carte XBee

La communication avec la carte `XBee` se fait également en UART, les pins `UART3` de la carte `web` sont connectées à la carte `XBee` (la carte `web` est également responsable de l'alimentation de la carte `XBee`).

Au niveau logiciel, la communication est gérée par la librairie définie dans le fichier `web/Src/bum_player.c`.

```
68  void convertAccData(int8_t accMove[3], LIS3DSH_DataScaled* accData) {
69      accMove[0] = (int8_t)(accData->x / 10);
70      if (accMove[0] < -100) {
71          accMove[0] = -100;
72      } else if (accMove[0] > 100) {
73          accMove[0] = 100;
74      }
75      accMove[1] = (int8_t)(accData->y / 10);
76      if (accMove[1] < -100) {
77          accMove[1] = -100;
78      } else if (accMove[1] > 100) {
79          accMove[1] = 100;
80      }
81      accMove[2] = (int8_t)(accData->z / 10);
82      if (accMove[2] < -100) {
83          accMove[2] = -100;
84      } else if (accMove[2] > 100) {
85          accMove[2] = 100;
86      }
87  }
```

FIGURE 5 – Code pour normaliser les accélérations

5 Conclusion

Le projet permet de jouer au jeu BUMPERS en manipulant un microcontrôleur dans l'espace. Les accélérations du mouvement du joueur dictent la trajectoire de sa balle sur le plateau de jeu. Ceci nécessitait d'implémenter différents composants (accéléromètre, interface web, XBee) puis de les connecter entre eux. Implémenter chacun des composants ne représentait pas de grande difficulté. En revanche, faire en sorte qu'ils puissent se connecter et intéragir entre eux de manière cohérente fut un défi plus important. Finalement, nous réussissons à présenter un projet quasiment terminé, mais nous avons recontré quelques difficultés, les principales étant :

— Un flux de travail inhabituel, provoquant de ce fait quelques ralentissements. Par exemple, pour une expérience de développement optimale, deux ordinateurs portables étaient nécessaires dont un avec un port

Ethernet.

— Un déverminage pas toujours évident. Bien sûr nous avons utilisé le dévermineur `gdb` mais nous avons aussi, par exemple, implémenté un script Python pour vérifier facilement que les bonnes données étaient transmises par la connexion UART.

Finalement, le projet était tout de même une bonne expérience qui nous a permis de nous familiariser une première fois avec le développement de logiciels embarqués pour des microcontrôleurs.