

A blue icon consisting of three horizontal bars, resembling a menu or list symbol.

# Dossier de Conception

Projet : Clavier Numérique multifonction

Simon Martin  
Touradou Kane  
Augustin Kania



## Table des matières

Table des matières .....	2
I. PRÉSENTATION.....	3
1. Contexte et objectifs du projet .....	3
2. Problématique .....	3
3. Solution apportée.....	3
II. GESTION DU PROJET ET RÉPARTITION DES TÂCHES .....	5
1. Organisations de l'équipe .....	5
III. ANALYSE DU PROJET .....	6
1. Les besoins et les contraintes.....	6
V - Détails de conception .....	7
1 - système d'alimentation .....	7
1.1 Consommation en énergie du système .....	7
1.2 Moyens d'alimentation du système.....	8
1.3 Conception du circuit de la carte d'alimentation .....	9
2 - Microcontrôleur .....	14
2.1 Détection des touches .....	14
3- Partie amplification Audio.....	29
3.1 / Justification des choix des composants électroniques .....	29

# I. PRÉSENTATION

## 1. Contexte et objectifs du projet

Les enseignants de la formation BUT GEII ont besoin de supports interactifs pour mettre en avant la formation lors des journées portes ouvertes. Cette année, les étudiants ont été chargés de réaliser des instruments de musique en utilisant les enseignements de la formation. Notre groupe a décidé de concevoir un **clavier numérique multifonction** dans le cadre du projet SYMPHONIE.

Après validation de notre cahier des charges par les enseignants, nous avons entamé la conception du système. Lors de cette phase, nous avons identifié les différents **sous-systèmes** du clavier numérique, et chaque étudiant du groupe s'est vu attribuer la conception d'un de ces sous-systèmes.

## 2. Problématique

Comment concevoir un instrument interactif, attractif et pédagogique qui met en avant les compétences techniques du parcours Électronique et Systèmes Embarqués (ESE), tout en respectant les contraintes de **budget, de temps et d'ergonomie** ?

Ce projet doit non seulement être **fonctionnel et facile d'utilisation**, mais aussi répondre aux besoins suivants :

- Attirer l'attention des visiteurs et leur donner envie d'en apprendre d'avantage sur notre formation.
- Intégrer des **technologies embarquées communs** pour illustrer concrètement les savoirs enseignés.
- Être **transportable et ergonomique** pour faciliter son utilisation lors des démonstrations.

## 3. Solution apportée

Le projet SYMPHONIE consiste en la conception et la fabrication d'un **piano numérique électronique**, destiné à être présenté lors des **portes ouvertes de l'IUT de Neuville**.

L'objectif est de démontrer les compétences des étudiants en électronique, programmation embarquée et mécatronique, tout en proposant un instrument interactif et attractif.

Le **clavier numérique** est conçu pour intégrer **trois modes de fonctionnement** :

- **Mode manuel** : L'utilisateur joue directement sur les touches du clavier.

- **Mode semi-automatique** : Le clavier est contrôlé à distance via une application mobile.
- **Mode automatique** : Le piano joue un morceau préenregistré en suivant des commandes MIDI.

Le piano sera composé de 25 touches, on mettra une plaque de cuivre servant à détecter l'appui sur la touche. L'ensemble de la structure, y compris la base et les touches, sera imprimé en 3D pour garantir une fabrication sur mesure des composants.

Le son sera généré par un système audio embarqué comprenant deux enceintes, placées de chaque côté du clavier pour une meilleure spatialisation sonore. Le traitement du signal et la gestion des entrées/sorties seront assurés par un ESP32, qui permettra également la communication avec une application mobile via Bluetooth.

## II. GESTION DU PROJET ET RÉPARTITION DES TÂCHES

### 1. Organisations de l'équipe

Notre équipe est composée de trois étudiants en **BUT 3 GEII**, chacun étant responsable d'un aspect clé du projet :

- **Simon MARTIN** : Responsable de la partie électronique. Doit concevoir les différents circuits électroniques sur Proteus 8, soudé les composants sur des PCB. Et vérifier le bon fonctionnement des cartes électroniques.
- **Touradou KANE** : Responsable de la programmation. Il a développé le système matriciel des boutons pour minimiser l'utilisation des entrées/sorties de l'ESP32 et programmé l'application mobile permettant de contrôler le clavier. Il s'est chargé à la fois de la gestion logicielle du système de touches et de l'interface utilisateur.
- **Augustin KANIA** : Responsable de la mécanique et aussi de l'alimentation. Doit concevoir la structure physique du clavier, en prenant les mesures et en réalisant une modélisation 3D pour l'impression ultérieure. Il doit aussi concevoir la batterie du système, en étudiant sa consommation énergétique et en s'assurant que la tension de sortie était adaptée aux besoins du circuit.

## III. ANALYSE DU PROJET

### 1. Les besoins et les contraintes

Pour répondre à cet objectif, nous devons tenir compte de plusieurs **besoins et contraintes**.

#### **Besoins identifiés :**

1. **Fonctionnalité interactive** : L'instrument doit être capable d'être joué manuellement, semi-automatiquement via une application et automatiquement par n'importe quel utilisateur
2. **Qualité sonore** : Le son produit doit être clair, précis et suffisamment puissant pour être audible même dans un environnement bruyant
3. **Autonomie énergétique** : L'instrument doit fonctionner sur batterie avec une autonomie minimale d'une heure et doit être rechargeable en USB type C
4. **Facilité d'utilisation** : L'interface doit être facile d'utilisation pour que les visiteurs puissent tester l'instrument sans difficulté
5. **Démonstration des compétences GEII** : Le projet doit mettre en valeur les systèmes embarqués, l'électronique et l'informatique

#### **Contraintes à respecter :**

1. **Contraintes techniques :**
  - a. L'instrument doit être synchronisé avec un chef d'orchestre numérique via le protocole MIDI
  - b. Les touches doivent être ni trop dure/ ni trop mou grâce à un système de ressort optimisé
  - c. L'ESP32 doit gérer les entrées/sorties et la communication avec l'application mobile
2. **Contraintes budgétaires et matérielles :**
  - a. Budget limité à **200 €**
  - b. Utilisation prioritaire des **ressources déjà disponibles à l'IUT** pour limiter les achats sur RS ou autres vendeurs tiers
3. **Contraintes ergonomiques et environnementales :**
  - a. L'instrument doit être facilement transportable.
  - b. Le projet doit s'inscrire dans une démarche de développement durable (réutilisation des composants et optimisation énergétique)

## V - Détails de conception

### 1 - système d'alimentation

D'après le cahier des charges, le système doit pouvoir être alimenté par le secteur mais aussi une batterie rechargeable lui permettant d'avoir une autonomie minimum d'un heure.

#### 1.1 Consommation en énergie du système

Afin de dimensionner correctement système d'alimentation, j'ai étudié les caractéristiques électriques l'ensemble des consommateurs du système.

##### **Microcontrôleur ESP32 Wroom**

Caractéristiques Alimentation minimum : 500mA / 3,3V / 1,65W

##### **Bande de led néopixel**

Caractéristiques d'alimentation d'une led : 60mA / 5V

Les leds seront utilisées pour indiquer la touche du piano sélectionnée, on aura 2 leds par touche. En supposant qu'on ne fera aucun accord supérieur à 7 touches :

Courant max :  $7 \times 2 \times 60\text{mA} = 840\text{mA}$

On en déduit les caractéristiques d'alimentation :

5V / 0,84A / 4,2W

##### **Amplificateur et hauts parleur**

Dans l'amplificateur audio, les deux AOP alimenté en 5v utilisé pour amplifier la tension de sortie dissipent l'essentiel de l'énergie.

$$P_q = (V_+ - V_-) \times I_q = (5 - 0) \times 40 \cdot 10^{-3} = 40\text{mW}$$

$$P_{out} = 325\text{mW}$$

$$P_{out} + P_q = 325 + 40 = 365\text{mW}$$

Or le montage comprend deux AOP

$$\text{Donc } P_{ampli} = 2 \times 365 = 730\text{mW}$$

Caractéristiques d'alimentation hautes parleurs : 4W RMS

Nous utiliserons deux hauts parleurs, ainsi :  $2 \times 4 = 8\text{ W RMS}$

On en déduit les caractéristiques d'alimentation :

8,73A / 5V / 1,746A

A partir des caractéristiques j'ai pu établir un bilan de consommation

Consommateur	Tension (V)	Courant (A) (courant nominal + 25%)	Puissance moyenne (W)
ESP32 WROOM	3,3	0,7	2,31
néopixel	5	1	5
Ampli + haut-parleur	5	2,2	11
total			18,31

## 1.2 Moyens d'alimentation du système

Le système sera alimenté par la batterie mais aussi par un transformateur AC-DC.

### **Batterie**

Pour alimenter mon montage, j'ai décidé d'utiliser la batterie avec un BMS intégré.

Tension de sortie : 7,2V-8,4

Capacité : 2600mAh

Tension d'alimentation : 8,4V =- 1%

Calcul de l'autonomie de la batterie en cas d'utilisation continue de l'instrument :

$$I_{consommateurs} = \frac{\text{Puissance consommée}}{\text{tension batterie}} = \frac{18,31}{7,2} = 2\,543\text{mA}$$

$$C_{consommateurs} = I \times \text{temps} = 2543\text{mAh} < 2600\text{mAh}$$

La batterie choisit a une capacité suffisante.

### **Transformateur AC-DC**

Le transformateur est chargé de transformer le 230vac en une tension continue permettant de charger la batterie et alimenter le clavier simultanément.

$$I_{chargeur} = I_{consommateurs} + I_{batterie}$$

$$I_{chargeur} = 2543 + 900\text{mAh} = 3\text{A}$$

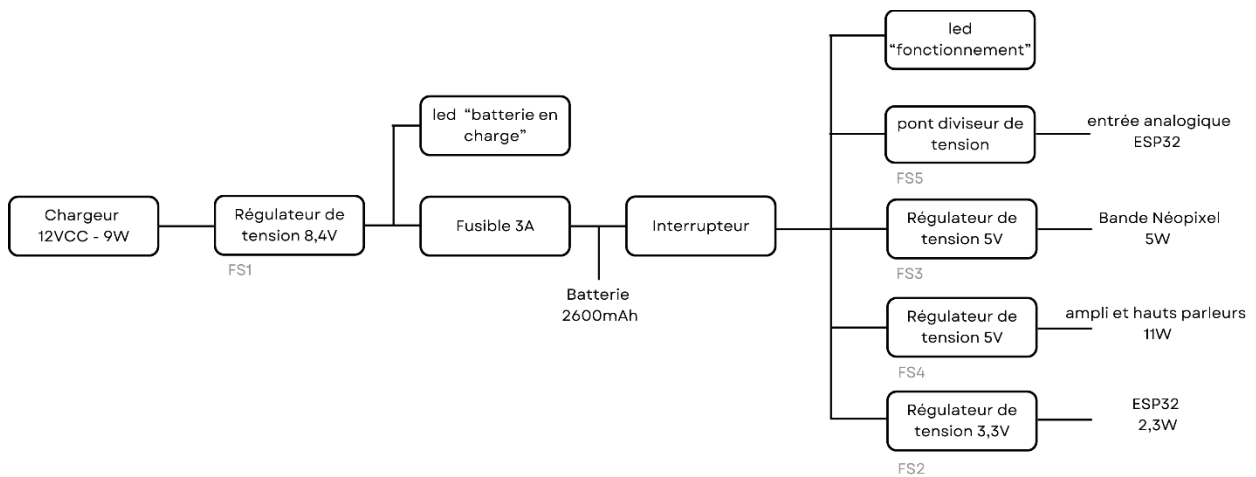
J'aurais donc besoin d'un chargeur délivrant au minimum 3A \* 8,4v = 25,2W

J'utiliserai un chargeur 12V.



### 1.3 Conception du circuit de la carte d'alimentation

Je peux maintenant réaliser un schéma du système d'alimentation. Sur ce schéma, on peut voir les 5 sous-systèmes fonctionnels qui composent ce circuit.



#### FS1 régulateur de tension 8,4V

Je me suis inspiré d'un montage de la documentation du régulateur LM317.

$$V_{OUT} = 1.25 V \left( 1 + \frac{R_2}{R_1} \right) + I_{ADJ} (R_2)$$

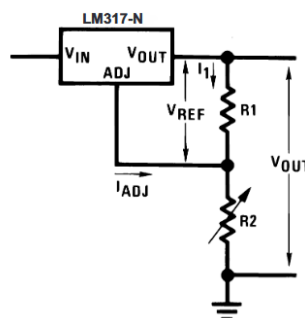


Figure 15. Setting the  $V_{OUT}$  Voltage

Figure 1 - extrait de la documentation du régulateur LM317

Les résistances  $R_1$  et  $R_2$  permettent de fixer la tension à réguler.

D'après la formule :

$$V_{out} = 1,25 \cdot \left( \frac{R_2}{R_1} + 1 \right) \Rightarrow R_2 = R_1 \times \frac{(V_{out} - 1,25)}{1,25}$$

Pour  $V_{out} = 8,4V$  et  $R_1 = 220$  on a :

$$R_2 = 220 \times \frac{(8,4-1,25)}{1,25} \Rightarrow R_2 = 1258\Omega$$

Le régulateur LM317 ne peut supporter que 1A. J'ai ajouter un transistor PNP pour augmenter le courant du régulateur. R1 est la résistance polarisation du transistor, sa valeur a été choisit de manière à faire fonctionner le transistor au plus tôt et limiter le passage de courant dans le LM317.

$$R_1 = \frac{U}{I} = \frac{0,7}{0,07} = 10$$

Pour finir j'ai ajouter des condensateurs de découplage pour éviter la transmission de parasites.

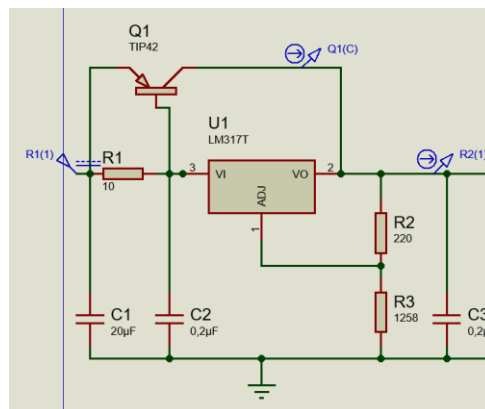


Figure 2 - extrait du schéma du circuit : FS1

## FS2 – régulateur 3,3V

Pour réaliser ce régulateur de tension, j'ai également utilisé un LM317 avec un transistor PNP. Pour fixer la tension de sortie j'ai défini les résistances avec la formule de la documentation technique :

$$V_{out} = 1,25 \cdot \left( \frac{R_6}{R_5} + 1 \right) \Rightarrow R_6 = R_5 \times \frac{(V_{out}-1,25)}{1,25}$$

Pour  $V_{out} = 3,3V$  et  $R_5 = 220\Omega$  on a :

$$R_6 = 220 \times \frac{(3,3-1,25)}{1,25} \Rightarrow R_6 = 361\Omega$$

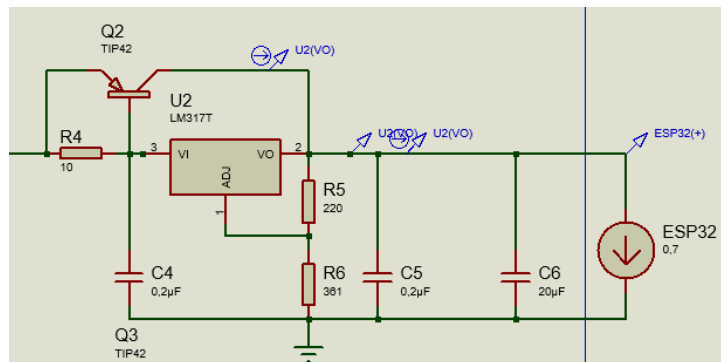
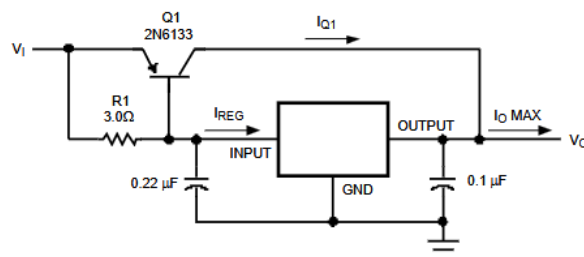


Figure 3 - extrait du schéma du circuit : FS2

### FS3/FS4 - Régulateurs 5v

Pour réguler la tension à 5V, j'ai reproduit le montage donné dans la documentation du LM7805.



$$\beta(Q1) \geq I_{O \text{ Max}} / I_{\text{REG Max}}$$

$$R1 = 0.9 / I_{\text{REG}} = \beta(Q1) V_{\text{BE}(Q1)} / I_{\text{REG Max}} (\beta + 1) - I_{O \text{ Max}}$$

Figure 24. High Current Voltage Regulator

Figure 4 - extrait de la documentation du régulateur LM7805

Ce montage utilise un transistor PNP utilisé pour augmenter le courant délivré et des condensateurs de découplages. J'ai ajouté un condensateur de découplage en sortie du montage de 2uF pour filtrer les parasites qui pourraient se former dans les fils reliant la carte d'alimentation avec les consommateurs.

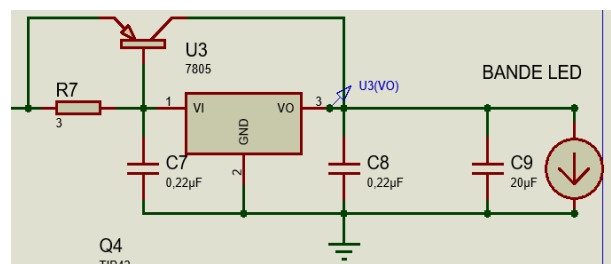


Figure 4 - extrait du schéma du circuit : FS3/FS4

### FS5 – Pont diviseur de tension

Le niveau de charge de la batterie peut être déterminé à partir la tension a ses bornes. On utilisera le microcontrôleur comme multimètre pour mesurer cette tension.

Seulement, le microcontrôleur ne tolère pas une tension de 8,4V mais seulement 3,3V.

J'ai fais un pont diviseur de tension pour traduire les 8,4v de la batterie en 3v.

$$3V = \frac{R_{10}}{R_{10} + R_9} \times 8,4V \Rightarrow R_9 = \frac{R_{10} \cdot (8,4 - 3)}{3} = R_{10} \times 1,8$$

Si je prend  $R_{10} = 1,2k$

$$R_9 = 1,2k \times 1,8 = 2,16k\Omega$$

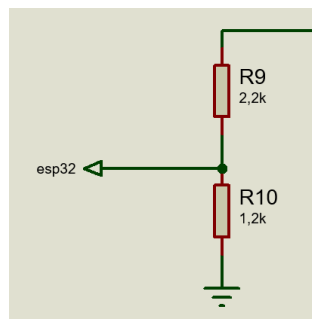


Figure 5 - extrait du schéma du circuit : FS5

### Composant supplémentaire

J'ai ajouter au montages plusieurs composants :

Des composants de sécurité : un fusible de sécurité à l'amont du circuit, une diode pour éviter les retour de courant et un interrupteur pour couper l'alimentation du système au besoin,

Des leds pour donner des information sur le fonctionnement de l'alimentation : une led indique que la batterie est en charge et une led indique que le système est alimenté.

### Circuit complet

Sur le logiciel Proteus j'ai conçu l'ensemble du circuit. Pour valider l'efficacité du circuit, j'ai utilisé l'outil de « simulation » du logiciel.

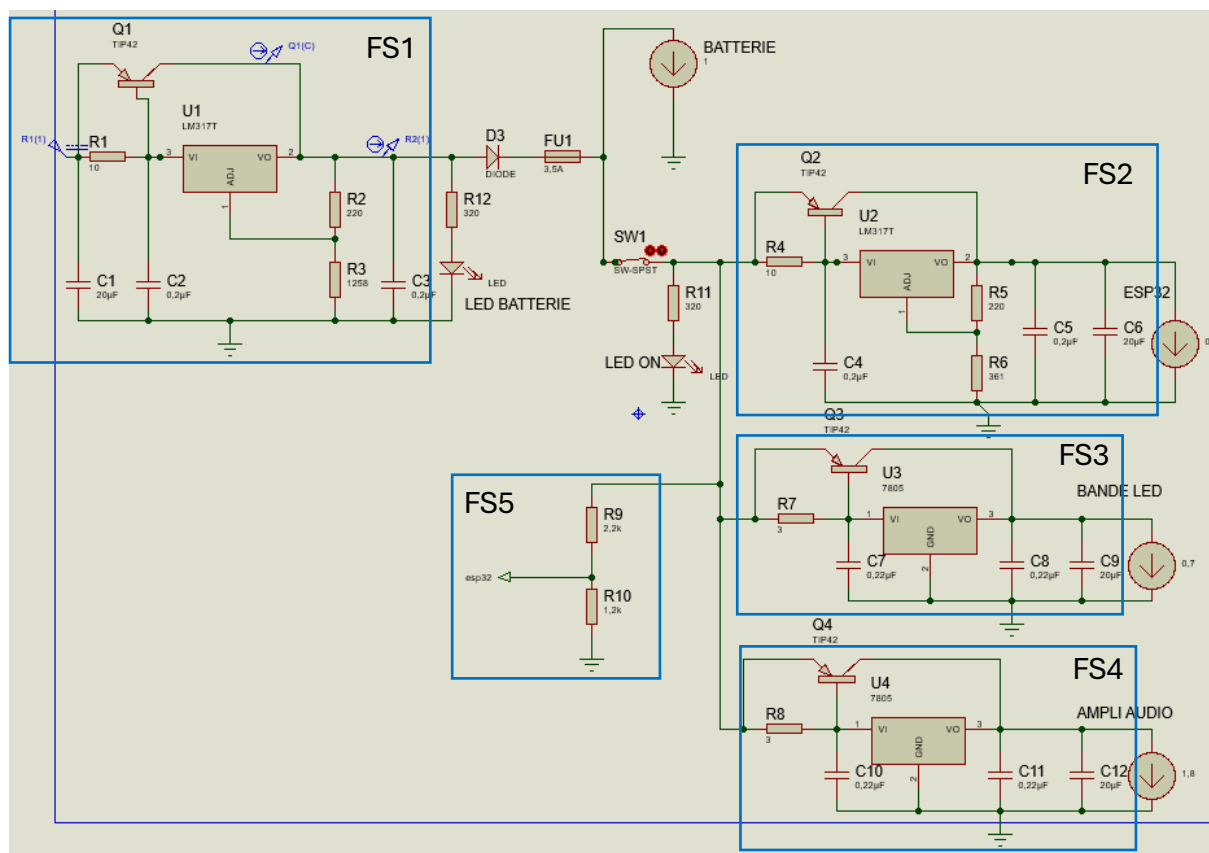
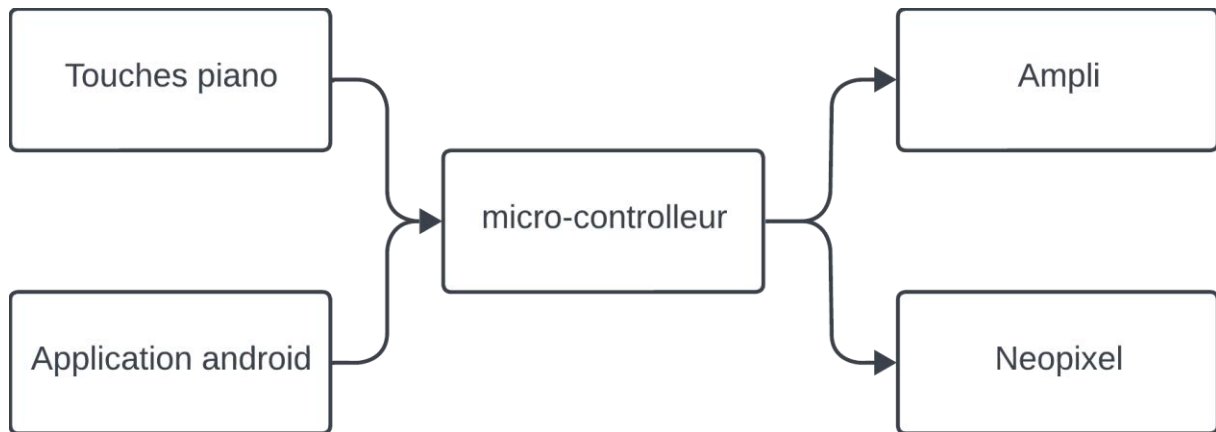


Figure 6 - schéma du circuit de la carte d'alimentation

## 2 - Microcontrôleur



### 2.1 Détection des touches

Nous avons trois modes de jeu pour notre instrument, le mode manuel, semi-automatique, et automatique. Pour le mode manuel nous avons simplement des touches à appuyer et à chaque appuie de touches nous avons un son qui est joué. Mais comme notre clavier fait 2 octaves/24 touches cela fait donc 24 GPIOs à utiliser, une touche pour un GPIO. Pour éviter d'en utiliser autant nous avons opté pour un clavier matriciel. C'est-à-dire que chaque touche, au lieu d'être connectée à un GPIO et à la masse ou 3,3V, sera connectée à deux GPIO, un correspondant à une ligne et un autre correspondant à une colonne. Ce qui fait que l'on passe de 25 à 10 GPIOs utilisées.

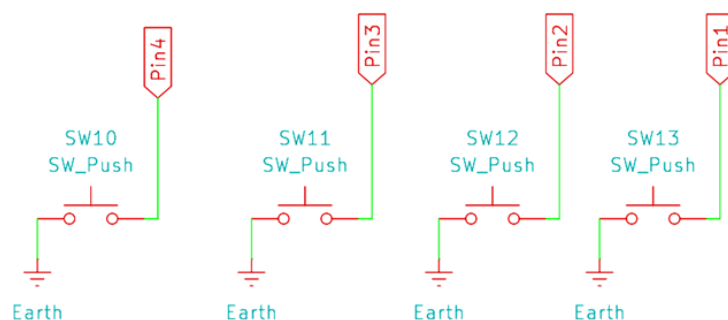


Figure 7 Boutons "Normaux"

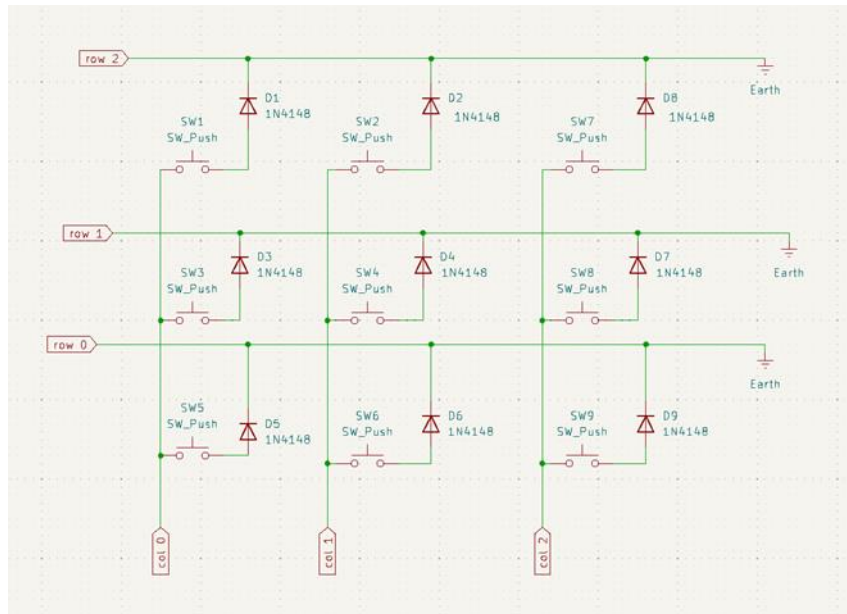


Figure 8 Boutons en matrice

Comment cela fonctionne. Nous avons les lignes en sortie, initialiser à l'état haut, et les colonnes en entrée avec des résistances de pull up. Il y aura deux boucles for. La première boucle va mettre la sortie du pin à l'état bas, la deuxième boucle elle va lire l'état de chaque colonne, et si sur une des colonnes nous avons un état bas alors un des boutons est appuyé. On peut alors récupérer la colonne et la ligne du bouton et effectuer notre traitement avec.

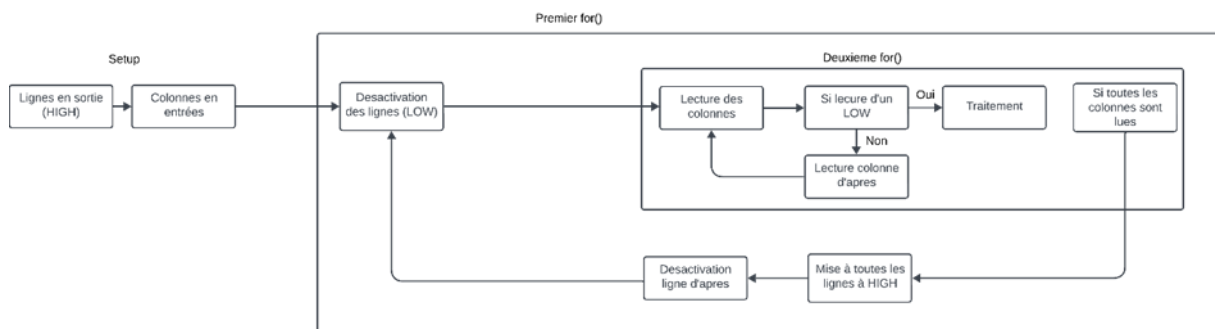


Figure 9 Algorithme de lecture de bouton en matrice

```
// Parcourir chaque ligne
for (int row = 0; row < 3; row++) {
    digitalWrite(rowPins[row], LOW);

    // Lire les colonnes
    for (int col = 0; col < 3; col++) {
        bool state = digitalRead(colPins[col]); // Lire l'état de la colonne

        // Si le bouton est pressé
        if (state == LOW) {
            Serial.printf("Bouton appuyer Colonne : %d,\t Ligne : %d\n", col, row);
        }
    }

    digitalWrite(rowPins[row], HIGH);
    delay(10); // Petit délai pour éviter les rebonds
}
```

Figure 10 Ancien programme de détection de touches

Mais pour quelque que raison que ce soit le concept n'a pas fonctionné. Nous avons donc une librairie nommée Keypad.h et en réutilisant la même configuration de boutons avec les diodes, tout fonctionne parfaitement. Et en plus de cela le créateur de la bibliothèque a ajouté à sa classe des états ce qui me facilite grandement la tâche pour envoyer des notes ON à l'appui ou des notes OFF au relâchement.

```
if (keypad.getKeys())
{
    for (int i=0; i<LIST_MAX; i++) // Scan the whole key list.
    {
        if ( keypad.key[i].stateChanged ) // Only find keys that have changed state.
        {
            switch (keypad.key[i].kstate) { // Report active key state : IDLE, PRESSED, HOLD, or RELEASED
                case PRESSED:
                    state = " PRESSED.";
                    break;
                case HOLD:
                    state = " HOLD.";
                    add2pressed_key(keypad.key[i].kcode);
                    Serial.printf("Tableau = %d ,\t %d ,\t %d \n", key_pressed[0], key_pressed[1], key_pressed[2]);
                    break;
                case RELEASED:
                    state = " RELEASED.";
                    remove_from_pressed_key(keypad.key[i].kcode);
                    Serial.printf("Tableau = %d ,\t %d ,\t %d \n", key_pressed[0], key_pressed[1], key_pressed[2]);
                    break;
                case IDLE:
                    state = " IDLE.";
                    break;
            }
            Serial.print(keypad.key[i].kchar);
            Serial.println(state);
        }
    }
}
```

Figure 11 Programme de détection de touches



## 2.2 Sortie Sonore

Pour la sortie sonore nous avons eu 3 choix. Le premier, le plus simple, utilisation de la librairie Tone. C'est une librairie qui permet de générer des signaux carrés à une fréquence voulue. Deuxième solution créer nous-même des tableaux imitant des signaux sinusoïdaux, carré triangulaire ou même quelconques et les faire sortir via un des DAC de notre ESP32. Et enfin la troisième solution, celle que nous avons choisie, l'utilisation de la librairie Mozzi. Cette librairie utilise la deuxième solution mais apporte des fonctionnalités en plus qui sont non négligeables : Fréquences d'échantillonnage configurable, fréquence d'appel à la fonction de contrôle réglable, intégration dans la librairie de multitudes de tableaux de signaux, sortie PDW, DAC ou I2S pour appareil externe.... Mais ce sont les exemples qui m'ont convaincu. Même si cette librairie est plus compliquée d'utilisation, avec le temps nous pourrions générer des sons complètement de claviers numériques utilisant de simples PWM.

Exemples d'utilisation de la librairie Mozzi : <https://sensorium.github.io/Mozzi/examples/>

Comment utiliser cette librairie ? Il faut d'abord déclarer une ou plusieurs variables qui vont stocker les valeurs de nos tableaux et ensuite on appelle la fonction startMozzi()

```
Oscil <SIN4096_NUM_CELLS, MOZZI_AUDIO_RATE> aSin1(SIN4096_DATA);
Oscil <SIN4096_NUM_CELLS, MOZZI_AUDIO_RATE> aSin2(SIN4096_DATA);
Oscil <SIN4096_NUM_CELLS, MOZZI_AUDIO_RATE> aSin3(SIN4096_DATA);

void setup() {
  Serial.begin(115200);
  ble_midi.initBLE();

  maintenant_debug = millis();

  startMozzi();

  strip.begin();
  strip.setBrightness(255);
}
```

Figure 12 Utilisation de la librairie Mozzi

Ensuite il faut déclarer les fonctions updateControl() et updateAudio(). updateControl() est la fonction qui va gérer les variables aSin1, 2 et 3 qui contiennent les tableaux pour par exemple mettre en pause le signal si un bouton est appuyé, changer la fréquence du signal ou autre. updateAudio() va retourner la prochaine valeur de notre signal qui doit sortir sur notre DAC.

```
void updateControl(){
}

AudioOutput updateAudio(){
  return MonoOutput::from8Bit(aSin1.next());
}
```

Figure 13 Génération d'un sinus simple

```

int8_t myAudioOutput = 0;
uint8_t number_of_signals = 0;

void updateControl(){
    if(key_pressed[0]!=9999)
        aSin1.setFreq(frequencies[key_pressed[0]]);
    else
        aSin1.setFreq(0);

    if(key_pressed[1]!=9999)
        aSin2.setFreq(frequencies[key_pressed[1]]);
    else
        aSin2.setFreq(0);

    if(key_pressed[2]!=9999)
        aSin3.setFreq(frequencies[key_pressed[2]]);
    else
        aSin3.setFreq(0);

    myAudioOutput = 0;
    number_of_signals = 0;
    if(key_pressed[0]!=9999){
        myAudioOutput = myAudioOutput + aSin1.next();
        number_of_signals++;
    }

    if(key_pressed[1]!=9999){
        myAudioOutput = myAudioOutput + aSin2.next();
        number_of_signals++;
    }

    if(key_pressed[2]!=9999){
        myAudioOutput = myAudioOutput + aSin3.next();
        number_of_signals++;
    }

    myAudioOutput = constrain(myAudioOutput, -128, 127);
}

AudioOutput updateAudio(){
    return MonoOutput::from8Bit(myAudioOutput + 128);
}

```

Figure 14 Changement de fréquence du sinus

On voit en figure 14 un début d'implémentation fonctionnel. Pour résumer ces deux fonctions vont changer les fréquences et additionner différents sinus selon le nombre de boutons appuyer. On remarque que la plupart du code est dans la fonction updateControl car updateAudio() est appelé à des périodes très courtes (0,3µs) et effectuer des calculs ici pourrait ralentir la fonction et changer notre signal.

## 2.3 Contrôle à distance

Pour le mode automatique et semi-automatique nous avons opter pour un contrôle à distance du clavier avec un synchronisation de bande de leds neopixels selon la touche

appuyée. Pour créer l'application j'ai utilisé Android Studio. J'ai choisi Android Studio a d'autres outils tout simplement car c'est l'outil que je connais le mieux et c'est celui qui est utilisé dans l'industrie, ce qui me fait un bon entraînement et une mention en plus dans mon CV. Pour la communication entre l'application et notre microcontrôleur j'ai opté pour le BLE.

J'aurais aussi pu choisir l'USB avec la communication série, ou même le Wifi, mais avoir un câble constamment branché à l'appareil est moins impressionnant et amusant. Et le Wifi est utilisé avec le Midi dans des cadres plus important par exemple dans un studio ou beaucoup d'appareils doivent communiquer en même temps et la consommation d'énergie n'est pas un problème. Je vais d'abord présenter l'interface de l'application puis la connexion BLE et l'intégration du Midi.

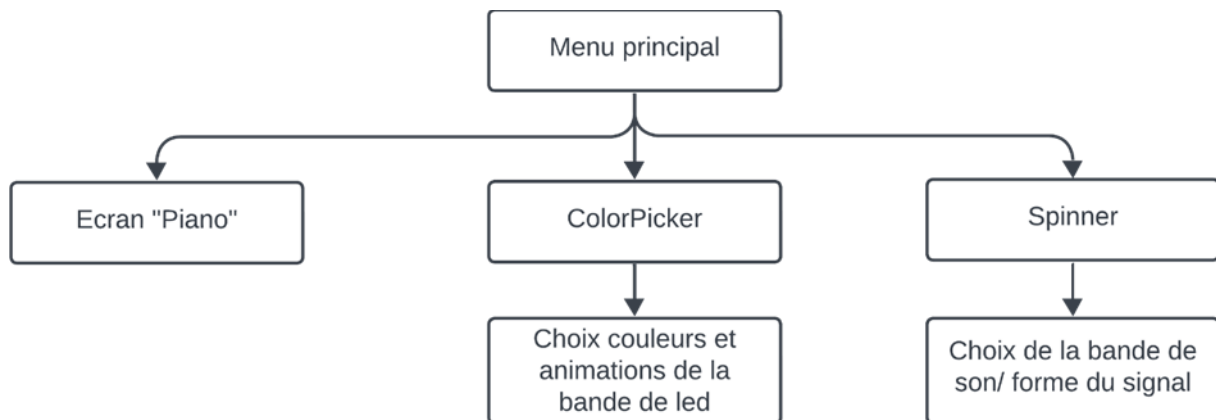


Figure 15 Architecture de l'application

L'application en elle-même est très simple. Elle a une page ColorPicker pour choisir la couleur de la bande de led, une page piano avec des boutons qui imitent les touches d'un piano et un spinner qui permet de choisir la forme du signal de sortie.

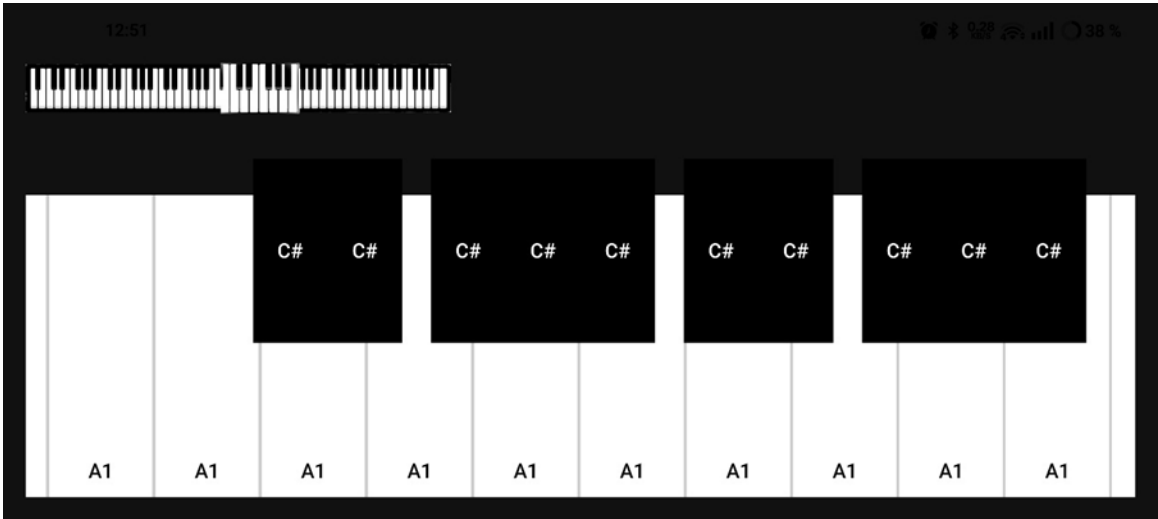


Figure 16 Ecran Piano

Je ne vais pas rentrer dans les détails de comment chaque bouton est fait car cela prendrait beaucoup trop de temps.

Je vais donc expliquer comment fonctionne le BLE. Le BLE utilise une architecture une client-serveur, le Generic Attribute Profile (GATT). Dans notre cas notre ESP32 sera le serveur et l'application le client.

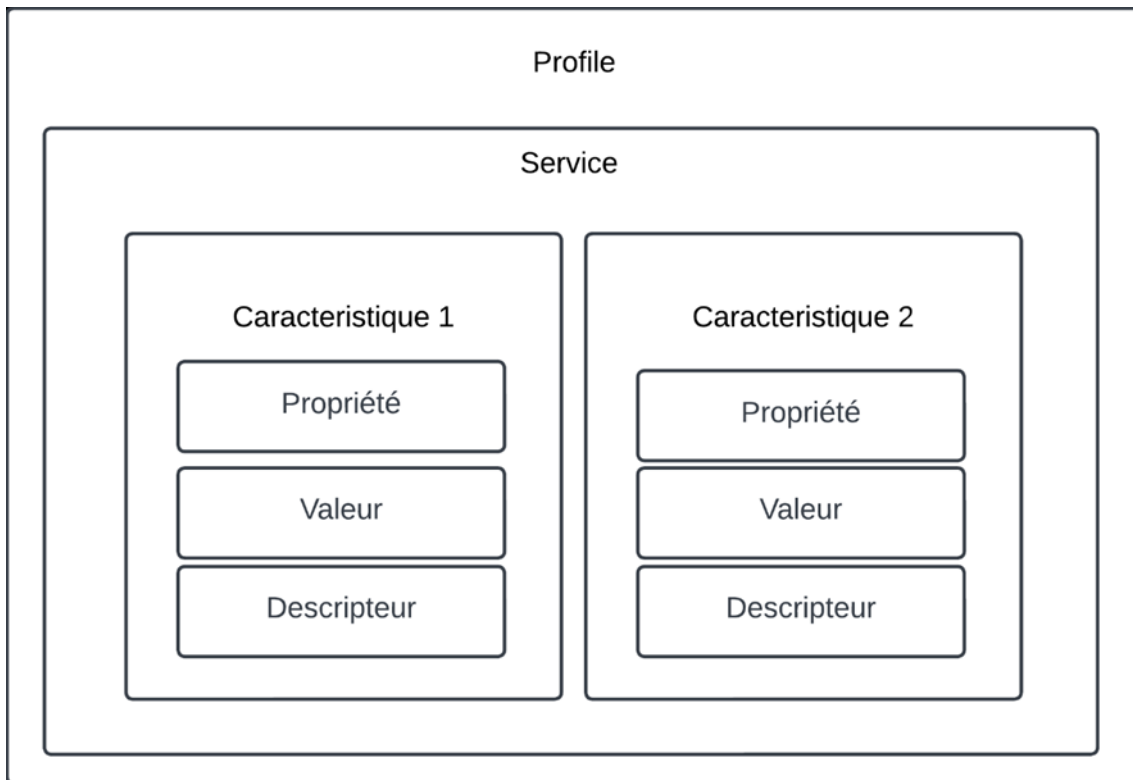


Figure 17 Serveur GATT

Voici comment sont représentés les serveurs GATT(ESP32). Chaque serveur peut avoir un ou plusieurs services et chaque service une ou plusieurs caractéristiques. Les services et caractéristiques sont définis par des UUID, des strings par exemple :

```
// Identifiants pour le service et la caractéristique
#define SERVICE_UUID "03B80E5A-EDE8-4B33-A751-6CE34EC4C700" // UUID du service
#define CHARACTERISTIC_MIDI_UUID "7772E5DB-3868-4112-A1A9-F2669D106BF3" // UUID Midi
#define CHARACTERISTIC_COLOR_UUID "12345678-1234-5678-1234-56789ABCDEF0"
#define CHARACTERISTIC_GENERIC_UUID "12345678-5678-9012-3456-56789ABCDEF0"
```

Figure 18 Déclaration des UUID

Et chaque caractéristique est définie par un descripteur, une propriété et une valeur. Le descripteur définit la métadonnée de la caractéristique, sa valeur peut être n'importe quoi : un string, un int, un char, un json et enfin sa propriété va définir comment chaque caractéristique va être utilisée.

```
// Initialiser le périphérique BLE
BLEDevice::init("ESP32 BLE Instrument");

BLEServer *pServer = BLEDevice::createServer();

// Créer un service
BLEService *pService = pServer->createService(SERVICE_UUID);

// Initialisation de la caractéristique MIDI
BLECharacteristic *midiCharacteristic;
midiCharacteristic = pService->createCharacteristic(
    CHARACTERISTIC_MIDI_UUID,
    BLECharacteristic::PROPERTY_READ |
    BLECharacteristic::PROPERTY_WRITE
    // BLECharacteristic::PROPERTY_NOTIFY
);

// Initialisation la caractéristique Couleur
BLECharacteristic *colorCharacteristic;
colorCharacteristic = pService->createCharacteristic(
    CHARACTERISTIC_COLOR_UUID,
    BLECharacteristic::PROPERTY_READ |
    BLECharacteristic::PROPERTY_WRITE
    // BLECharacteristic::PROPERTY_NOTIFY
);

// Initialisation la caractéristique Generic
BLECharacteristic *genericCharacteristic;
genericCharacteristic = pService->createCharacteristic(
    CHARACTERISTIC_GENERIC_UUID,
    BLECharacteristic::PROPERTY_READ |
    BLECharacteristic::PROPERTY_WRITE
    // BLECharacteristic::PROPERTY_NOTIFY
);
```

Figure 19 Initialisation BLE

On peut voir que j'ai utilisé les propriétés READ et WRITE pour chaque caractéristique. Grâce à ces propriétés nous pouvons définir des fonctions de callback à chaque fois que l'on lit ou écrit dans ces caractéristiques. Mais avant de parler des fonctions de callback je vais vous montrer comment fonctionne l'envoi de données avec l'application.

Tout d'abord il faut demander des autorisations et demander l'activation du Bluetooth, s'il n'est pas activé

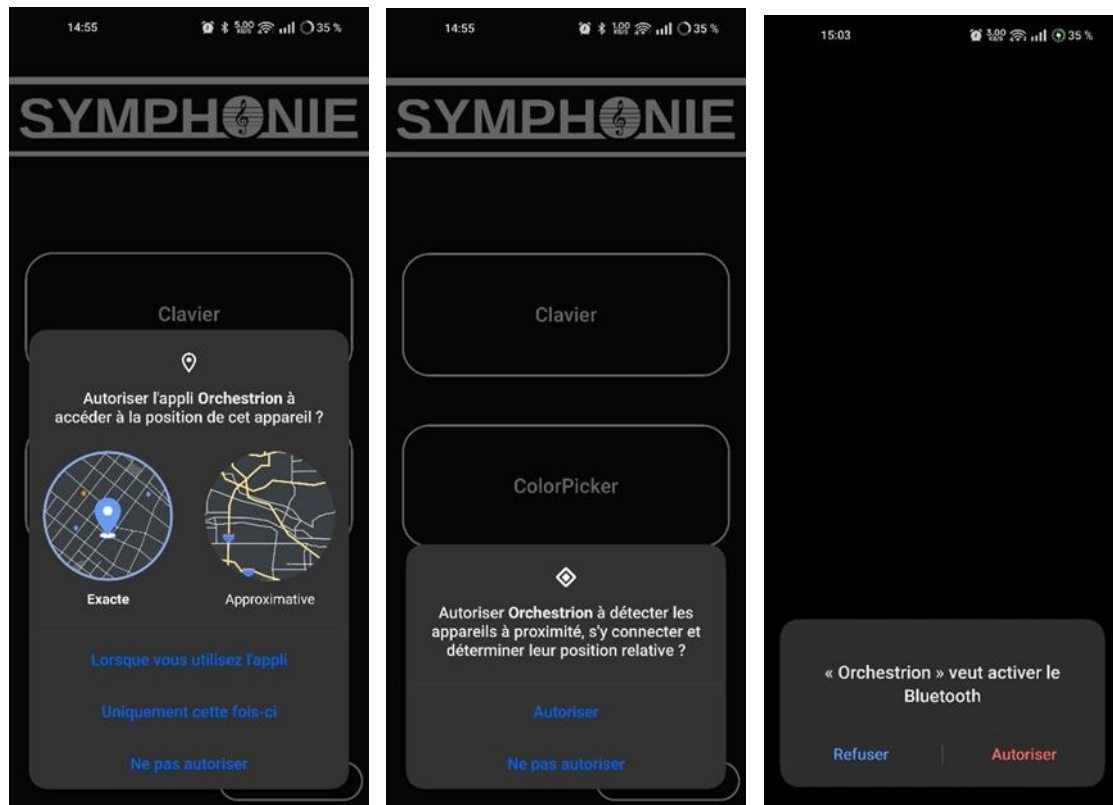


Figure 20 Demandes d'autorisations

Si les autorisations sont acceptées alors on peut scanner notre environnement pour trouver d'autre appareils avec le BLE Actif. Ensuite si on trouve un appareil avec le nom "ESP32 BLE Instrument" alors on s'y connecte et si la connexion a réussi alors, on compare les UUID pour savoir s'ils sont bons. Si les caractéristiques sont OK alors tout est OK pour l'envoi d'information.

```
@SuppressWarnings("MissingPermission")
fun sendMidiMessage(channel: Int, note: Int, velocity: Int, NoteON: Boolean = true) {
    if (channel < 1 || channel > 16) {
        Log.e(tag: "BLE", msg: "Canal MIDI invalide. Doit être entre 1 et 16.")
        return
    }
    val statusByte = (0x90 + (channel - 1)).toByte() //Channel
    val noteByte = note.toByte() //Note
    val velocityByte = velocity.toByte() //Velocity

    val midiPacket:ByteArray = if(NoteON)
        byteArrayOf(0x90.toByte(), statusByte, noteByte, velocityByte)
    else
        byteArrayOf(0x80.toByte(), statusByte, noteByte, velocityByte)
    midiWriteCharacteristic?.value = midiPacket

    bluetoothGatt?.writeCharacteristic(midiWriteCharacteristic)

    Log.d(tag: "BLE", msg: "Message envoyé: $channel, \t $note, \t $velocity")
}
```



Figure 21 Fonction d'envoi d'ordres Midi

Tout est convertis en Byte, octet puis est envoyé sur la caractéristique Midi. Il existe exactement la même fonction pour les couleurs.

```
//Callbacks
ServerCallback = MyServerCallbacks();
ColorCallBack = ColorCharacteristicCallbacks();
MidiCallBack = MidiCharacteristicCallbacks();
GenericCallBack = GenericCharacteristicCallbacks();

pServer->setCallbacks(&ServerCallback);
midiCharacteristic->setCallbacks(&MidiCallBack);
colorCharacteristic->setCallbacks(&ColorCallBack);
genericCharacteristic->setCallbacks(&GenericCallBack);
```

Figure 22 Callbacks

Pour chaque caractéristiques et services nous allons y associer une classe avec des fonctions de callback. Pour le serveur les callbacks sont onConnect() et onDisconnect(). Pour les caractéristiques tout dépend des propriétés que nous avons configuré. Si nous avons mis la propriété Read alors le callback onRead va être appelé à chaque fois que nous lisons dans une caractéristique, mais si par exemple on ne met pas la propriété Write alors la callback onWrite ne va pas être appelée lorsqu'on écrit dans la caractéristique.

```
class ColorCharacteristicCallbacks : public BLECharacteristicCallbacks {
private:
    bool update_value = false;
    uint8_t red = 0;
    uint8_t green = 0;
    uint8_t blue = 0;
    uint8_t animation = 0;

    void onWrite(BLECharacteristic* pCharacteristic) override {
        // Récupérer les données reçues
        std::string value = pCharacteristic->getValue();
        const uint8_t* data = reinterpret_cast<const uint8_t*>(value.data()); // Convertis le tableau value.data dans un uint8_t*
        size_t length = value.length();
        // reinterpret_cast change la type d'interprétation en mémoire
        // data pointe simplement l'adresse
        if (length == 5 && (data[0] == 0xFF)) {
            red = data[1];
            green = data[2];
            blue = data[3];
            animation = data[4];

            Serial.printf("Donnée Couleur reçue : RED = %d \t, GREEN = %d \t, BLUE = %d \t, ANIM = %d \n", red, green, blue, animation);

            update_value = true;
        } else {
            Serial.println("Erreur : Taille des données incorrecte");
        }
    }

    void onRead(BLECharacteristic* pCharacteristic) override { //ESP 2 Android
        Serial.println("Donnée lue !");
    }
public:
    uint8_t* getColors(){
        static uint8_t tab[4] = { red, green, blue, animation};
        return tab;
    } // static car je retourne une variable local et si pas de static alors variable supprimer apres appel a la fonction

    bool getUpdate(){
        return update_value;
    }

    void setUpdate(bool value){
        update_value = value;
    }
};
```



Figure 23 "Classe de Callback"

Les variables et les fonctions dans le public servent à récupérer nos ordres Midi depuis l'extérieur. Il existe le même type de fonction pour les couleurs.

```

NEW_MSG* BLE_Midi::loopBLE(){
    if((millis() - maintenant_loop == PERIODE) | (millis() < maintenant_loop)){//Toutes les 100ms ou si millis dépasse 47 jours

        maintenant_loop = millis();
        if(ServerCallback.getIsConnected()){
            if (MidiCallback.getUpdate()){
                MidiCallback.setUpdate(false);
                WhatsNew[0] = MIDI;
            }else
                WhatsNew[0] = {No_New_Msg};

            if (ColorCallback.getUpdate()){
                ColorCallback.setUpdate(false);
                WhatsNew[1] = Color;
            }else
                WhatsNew[1] = {No_New_Msg};

            if (GenericCallback.getUpdate()){
                GenericCallback.setUpdate(false);
                WhatsNew[2] = Generic;
            }else
                WhatsNew[2] = {No_New_Msg};
        }else
            Serial.println("Not Connected");
    }

    return WhatsNew;
}

```

Figure 24 Détection de nouveaux messages BLE

Pour mettre à jour nos variables de la Loop du main.cpp, j'ai créé la fonction loopBLE() qui retourne un tableau d'enum avec chaque enum qui représente si oui ou non nous avons reçu de nouveaux ordres. La condition if() avec le millis() est une alternative au delay() qui lui, stoppe complètement le microcontrôleur contrairement au millis().

```
memcpy(new_data, ble_midi.loopBLE(), sizeof(new_data)); // Copie des valeurs de Whats_New qui est dans loopBLE dans new_data
ble_midi.reset_tab();

for(int i=0; i < sizeof(new_data) / sizeof(new_data[0]); i++){
    // Division car sizeof retourne des octet et non le nb de variables
    switch (new_data[i])
    {
        case Color:
            Serial.println("Maj Color");
            anim.setStripColor(ble_midi.getColorOrder());
            //Changement de couleur et animation bande de led
            break;
        case MIDI:
            Serial.println("Maj Midi");
            //Traitement: Ajout ou retrait de la touche reçu du tableau de touches appuye.
            // status = 0x9x -> ajout
            // status = 0x8x -> retrait
            break;
        case Generic:
            Serial.println("Maj generic");
            //Traitement: Changement de bande de son
            break;
        default:
            break;
    }
}
```

Figure 25 Traitement donnée BLE (Extrait de la loop/main.cpp)

Et en appelant loopBLE() dans la boucle loop principale je peux récupérer toutes les données que je veux. Comme j'ai mis un timer de 100ms a la fonction loopBLE() si j'aurais récupérer le tableau WhatsNew (figure 25) comme dans la figure 26 cela n'aurait fonctionner que pour la première valeur du tableau, car à chaque ligne serait appeler loopBLE sauf que les 100ms ne serait pas écoulé. Pour cela j'ai utilisé la fonction memcpy qui me permet de copier un tableau dans un autre ce qui me fais appeler la fonction une seule fois. J'ai ajouté la fonction reset\_tab() qui permet de réinitialiser WhatsNew, car sinon au lieu de mettre à jour les traiter les donnée une seule fois, elles vont être traiter pendant 100ms à cause du timer.

```
new_data[0] = ble_midi.loopBLE()[0];
new_data[1] = ble_midi.loopBLE()[1];
new_data[2] = ble_midi.loopBLE()[2];
```

Figure 26 Moyen de récupération alternatif de nouveau ordres

```

uint8_t* BLE_Midi::getColorOrder(){
    return ColorCallBack.getColors();
}

uint8_t* BLE_Midi::getMidiOrder(){
    return MidiCallBack.getMidiOrder();
}

uint8_t BLE_Midi::getSignal(){
    return GenericCallBack.getSignal();
}

```

Figure 27 Fonctions de récupération de donnée

Ici je peux récupérer les ordres couleurs, midi et signal car les classes de callbacks sont des enfants de la classe BLE\_Midi.

```

class BLE_Midi
{
private:

    MyServerCallbacks ServerCallback;
    ColorCharacteristicCallbacks ColorCallBack;
    MidiCharacteristicCallbacks MidiCallBack;
    GenericCharacteristicCallbacks GenericCallBack;

    uint32_t maintenant_loop;
    NEW_MSG WhatsNew[3] = {No_New_Msg};

public:
    BLE_Midi(); // Déclaration du constructeur
    void initBLE();
    NEW_MSG* loopBLE();

    uint8_t* getColorOrder();
    uint8_t* getMidiOrder();
    uint8_t getSignal();
    void reset_tab();
};

```

Figure 28 Classe BLE\_Midi



Figure 29 Chemin de la variable

Grace à cela pas besoin de stocker les variables dans la classe BLE\_Midi je les récupère directement depuis leurs enfants.

Tout au long de la création de ce code j'ai eu l'intention de rendre le code le plus modulaire possible, et je pense qu'on le voit le mieux dans la figure 25. Pour la mise à jour des couleurs j'aurais pu simplement mettre dans le constructeur de la classe BLE, la classe qui contrôle la bande de led, et mettre à jour les couleurs depuis la classe BLE, mais cela aurait rendu le code beaucoup plus compliqué à réutiliser dans de futur projet. Si j'avais fait cela, dans une réutilisation future du code j'aurais dû prendre un temps pour récupérer cette classe et supprimer toutes les instances de la classe Neopix ce qui aurait été plus ou moins compliqué selon la façon dont j'aurais implémenté la classe Neopix. Et si je l'aurais fait pour la classe Neopix je l'aurais sûrement fait pour la classe gérant les boutons.

Optimiser la mémoire ou la modularité des classes n'est clairement pas nécessaire à ce projet mais ce sont de bonnes pratiques qui peuvent faire la différence dans le monde professionnel, ce qui fait de ce projet un très bon exercice.

### 3- Partie amplification Audio

#### 3.1 / Justification des choix des composants électroniques

Notre carte ampli audio reçoit un signal analogique en entrée (fréquence de la touche de piano souhaitée) et le traite avant de l'envoyer aux haut-parleurs. Pour garantir une bonne gestion du signal, nous avons adopté l'architecture suivante :

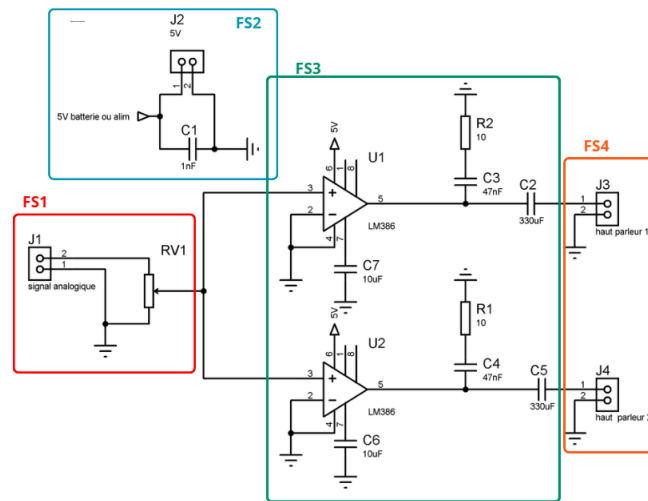


Figure 30 : Schéma de la partie amplification audio

**Entrée audio (FS1) :** Ajustement du volume avec un potentiomètre externe.

**Pré amplification (FS2) :** Entrée alimentation +5V issu de notre carte d'alimentation.

**Amplification (FS3) :** Augmentation du signal par le LM386 avant d'être envoyé aux haut-parleurs.

**Sortie audio (FS4) :** Restitution du son amplifié via nos haut-parleurs externes.

Pour la partie amplification, nous avons choisi l'amplificateur **LM386** car il est compatible avec une alimentation de **5V**, ce qui correspond à notre besoin. Son faible coût et aussi sa faible consommation en font un très bon choix pour un système alimenté par une batterie. Il permet d'avoir un gain fixe suffisant pour amplifier le signal sans distorsion ni saturation.

Des **condensateurs** de 10  $\mu$ F et 330  $\mu$ F ont été ajoutés pour stabiliser l'alimentation et améliorer la qualité du son en éliminant les parasites. Les **haut-parleurs 4 $\Omega$  - 4W** ont été sélectionnés pour être compatibles avec la sortie amplifiée du LM386.

Le schéma électronique a été conçu sous **Proteus 8**, car ce logiciel est gratuit à l'IUT et nous l'avons déjà utilisé en cours, ce qui facilite la prise en main. Il permet de simuler le circuit, détecter d'éventuelles erreurs et générer directement les fichiers Gerber pour la fabrication du PCB via notre IUT.

Tous les calculs et les justifications nécessaires sont disponibles dans le fichier "Dossier\_fabrication.pdf" dans le cy.git

### ***3.3 / Détails de conception mécanique***

La carte PCB mesure seulement 38mm x 35mm, ce qui permet une intégration compacte dans le boîtier du clavier. Les **supports DIP** facilitent le remplacement des amplificateurs en cas de panne. Le potentiomètre externe est fixé sur la façade du boîtier pour un réglage facile du volume.

Les connecteurs sont positionnés pour simplifier le câblage et l'intégration dans le système global. Les haut-parleurs seront fixés pour éviter les vibrations parasites et garantir un son clair et de meilleur basses.