



Dossier de Conception

Projet : Clavier Numérique multifonction

Rédigé par :
Simon **MARTIN**
Augustin **KANIA**
Touradou **KANE**

Version **1.2** – 18/02/2025



Table des matières

Table des matières	2
Table des figures	3
Avant-Propos	4
I. PRÉSENTATION	5
1.1. Contexte du projet	5
1.2. Documents du projet	5
1.3. Cahier des charges	5
1.4. Dossier de conception	5
1.5. Dossiers de fabrication	5
1.6. Autres documents	5
1.7. Rappel équipe	6
1.8. Rappel des fonctions techniques	6
1. Fonction principale	6
2. Fonctions secondaires	6
3. Fonctions de contraintes	7
II – Choix des outils et logiciels utilisés	8
1 – Choix de carte	8
2 – Choix de programmation	8
Pourquoi avoir choisi le MIDI ?	9
3– Choix pour la conception des PCB	10
4– Modélisation 3D	10
III - Détails de conception	12
1 - système d'alimentation	12
Consommation en énergie du système	12
Moyens d'alimentation du système	13
Conception du circuit de la carte d'alimentation	14
2 - Microcontrôleur	19
Sortie Sonore	22
Contrôle à distance	23

3- Partie amplification Audio	35
Justification des choix des composants électroniques	35
Détails de conception mécanique	36
4- Partie à venir	37
ANNEXE	38

Table des figures

Figure 1- extrait de la documentation du régulateur LM317	14
Figure 2- extrait du schéma du circuit : FS1	15
Figure 3- extrait du schéma du circuit : FS2	16
Figure 4- extrait de la documentation du régulateur LM7805	16
Figure 5- extrait du schéma du circuit : FS3/FS4	16
Figure 6- extrait du schéma du circuit : FS5	17
Figure 7 - schéma du circuit de la carte d'alimentation	18
Figure 8 - Boutons "Normaux"	19
Figure 9 - Boutons en matrice	20
Figure 10 -Algorithme de lecture de bouton en matrice	20
Figure 11 - Ancien programme de détection de touches	21
Figure 12 - Programme de détection de touches	21
Figure 13 - Utilisation de la librairie Mozzi	22
Figure 14 - Génération d'un sinus simple	22
Figure 15 - Changement de fréquence du sinus	23
Figure 16 - Architecture de l'application	24
Figure 17 - Ecran Piano via notre application	25
Figure 18 - Serveur GATT	26
Figure 19 - Déclaration des UUID	27
Figure 20 - Initialisation BLE	28
Figure 21- Demandes d'autorisations	29
Figure 22 - Fonction d'envoi d'ordres Midi	30
Figure 23 - CODE Callbacks	30
Figure 24 - "Classe de Callback"	31
Figure 25 - Détection de nouveaux messages BLE	31
Figure 26 - Traitement donnée BLE (Extrait de la loop/main.cpp)	32
Figure 27- Moyen de récupération alternatif de nouveau ordres	32
Figure 28 - Fonctions de récupération de donnée	33
Figure 29 - Classe BLE_Midi	33
Figure 30 - Chemin de la variable	34
Figure 31- Schéma de la partie amplification audio	35
Figure 32- accès git cy de symphonie	38

Avant-Propos

Ce document constitue le **dossier de conception** du projet « Clavier numérique multifonction SYMPHONIE », un instrument électronique conçu pour jouer de la musique en intégrant trois modes de fonctionnement. Ce projet a été développé à la demande de l'IUT de Cergy-Pontoise, afin d'être présenté lors des journées portes ouvertes comme démonstration des compétences acquises en électronique et systèmes embarqués.

L'objectif de ce dossier est d'apporter une vue **complète et détaillée** sur les choix techniques, les étapes de conception, les tests de validation et les résultats obtenus tout au long du développement du système. Il s'ouvre sur un rappel du contexte du projet, bien que celui-ci soit décrit de manière plus approfondie dans le cahier des charges.

Ce document sert à la fois de référence technique pour les parties prenantes et de base documentaire pour toute amélioration ou maintenance future du système.

I. PRÉSENTATION

1.1. Contexte du projet

Le projet « Clavier numérique multifonction SYMPHONIE » consiste en la **conception d'un instrument de musique électronique**, capable de fonctionner en trois modes distincts : manuel, semi-automatique et automatique. Ce projet s'inscrit dans le cadre de la SAE (Situation d'Apprentissage et d'Évaluation) du BUT GEII (Génie Électrique et Informatique Industrielle) à l'IUT de Neuville Université.

L'objectif est de concevoir un système qui répond à des contraintes techniques, économiques et environnementales, tout en illustrant les compétences développées en électronique, programmation embarquée et mécatronique.

Ce dossier de conception documente l'ensemble du processus de développement, de la conception initiale à la réalisation finale, en passant par les choix techniques, les tests et les validations du système.

1.2. Documents du projet

Cette section référence les documents et ressources associées au projet. Ces documents sont essentiels pour la compréhension, la réalisation, et la maintenance du système. Tous les documents sont en libre accès, et peuvent être consultés sur le répertoire GitHub du projet (voir références à la fin du document).

1.3. Cahier des charges

Le CDC constitue la base du projet, avec le contexte, les objectifs, les exigences fonctionnelles et techniques, ainsi que les contraintes à respecter. Il définit en détail les attentes en matière de fonctionnement, performance, sécurité, budgétaire et respect de l'environnement.

1.4. Dossier de conception

Le dossier de conception présente tout le parcours de la réalisation du projet. Il détaille les choix pour le système, décrit les schémas électroniques et les calculs associés, la programmation de la partie numérique du système, ainsi que les étapes et les résultats de tests.

1.5. Dossiers de fabrication

Les dossiers de fabrication regroupent les plans de fabrication des cartes électroniques, les instructions d'assemblage du boîtier et des composants, ainsi que les procédures de tests. Chaque sous-système du projet comporte son propre dossier de fabrication.

1.6. Autres documents

En complément, d'autres documents sont disponibles dans le répertoire du projet, tels que les rapports de tests, les fiches techniques de composants utilisés dans le projet, la documentation logicielle, le manuel d'utilisation d'interface logicielle, et les images concernant la réalisation du projet

1.7. Rappel 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.

1.8. Rappel des fonctions techniques

Le **clavier numérique multifonction SYMPHONIE** est conçu pour offrir plusieurs modes de jeu et fonctionnalités avancées, tout en respectant diverses des contraintes techniques et environnementales.

1. Fonction principale

-> **FP** : Jouer de la musique en mode manuel

L'utilisateur doit pouvoir produire un son de piano en appuyant directement sur les touches du clavier. Chaque touche déclenche une note qui est amplifiée et diffusée par les haut-parleurs intégrés.

2. Fonctions secondaires

-> **FS1** : Lecture automatique via MIDI

L'instrument doit être capable de lire un morceau préenregistré en recevant un fichier MIDI depuis une interface externe.

-> **FS2** : Contrôle à distance via une application mobile

L'utilisateur doit pouvoir jouer du clavier à distance en commandant les touches via une application sur smartphone. Ce mode semi-automatique permet une interaction sans contact avec l'instrument.

->**FS3** : Sélection des sonorités

Le clavier doit permettre à l'utilisateur de modifier le timbre des sons en choisissant différentes banques sonores via l'application mobile. Cette option offre plus de flexibilité musicale.

->**FS4** : Autonomie sur batterie et recharge simultanée

L'instrument doit pouvoir fonctionner sur batterie pendant au moins une heure et être rechargé sans interruption lors de son utilisation.

3. Fonctions de contraintes

->**FC1** : Sonorité adaptée à un environnement bruyant

Le volume sonore doit être suffisamment puissant pour être audible même dans un environnement bruyant, comme lors des journées portes ouvertes où plusieurs démonstrations ont lieu en simultané.

->**FC2** : Ergonomie et transportabilité

L'instrument doit être facile à déplacer, compact et agréable à utiliser. Son poids, sa taille et la disposition des commandes doivent être optimisés pour assurer une expérience utilisateur fluide.

->**FC3** : Démarche écoresponsable

Le projet doit intégrer une dimension de développement durable en privilégiant l'utilisation de matériaux de récupération disponibles à l'IUT. Cette approche permet de réduire l'empreinte écologique et de limiter les coûts de fabrication

II – Choix des outils et logiciels utilisés

1 – Choix de carte

Nous avons choisi d'utiliser l'**ESP32 Wroom** car c'est un microcontrôleur assez puissant et polyvalent, parfaitement adapté à notre projet. Il offre une connectivité Wi-Fi (802.11 b/g/n) et Bluetooth (4.2), essentielle pour le contrôle à distance du clavier via une application mobile.

L'ESP32 Wroom possède **34 broches GPIO**, dont plusieurs peuvent être configurées en entrées analogiques ou numériques, permettant une gestion optimisée des capteurs, boutons et sorties audio.

Cette carte possède les périphériques suivants :

- 3 interfaces UART
- 2 interfaces I2C
- 3 interfaces SPI
- 16 sorties PWM
- 10 capteurs capacitifs
- 18 entrées analogiques (ADC)
- 2 sorties DAC

En bref, cette carte dispose assez d'entrées/sorties numériques et analogiques, permettant de gérer notre matrice de touches, l'amplification audio, ainsi que la communication MIDI. De plus, son faible coût et sa compatibilité avec de nombreuses bibliothèques sont les raisons pour lesquelles nous l'avons choisie.

2 – Choix de programmation

Dans notre projet, l'**ESP32 Wroom** joue le rôle de **cerveau du système**, assurant la gestion des entrées/sorties du clavier, la communication avec l'application mobile et l'exécution des commandes MIDI. Pour développer le firmware de l'ESP32, nous avons choisi Visual Studio Code (VS Code), qui offre de nombreux avantages par rapport à d'autres environnements de développement.



Le tableau ci-dessous qui montre pourquoi avons-nous choisi VS Code :

Avantages :	Inconvénients :
Prise en charge de nombreuses bibliothèques	Peut-être plus complexe pour les débutants
Interface moderne	Trouver les bibliothèques de son côté (pas de catalogue)
Hautement personnalisables avec des extensions	
Plusieurs langages programmables (C, C++, Python, etc....)	
Logiciel largement performant et stable	

Nous avons fait le choix de ne pas utiliser **ESP-IDF** car, bien qu'il soit le framework officiel d'Espressif, il nécessite une configuration avancée et une courbe d'apprentissage plus longue. Son utilisation en ligne de commande et sa complexité ne sont pas adaptées à notre projet, où nous privilégions un environnement plus accessible et familier par rapport à ce qu'on fait auparavant sur des projets scolaires.

De même, nous avons écarté **Arduino IDE**, car malgré sa simplicité et son interface monotone, il manque d'outils avancés, comme une bonne gestion des bibliothèques et un débogage performant. Ces limitations auraient ralenti notre développement et rendu la gestion du projet moins efficace.

Pourquoi avoir choisi le MIDI ?

Dans un premier temps, le MIDI est l'abréviation de Musical Instrument Digital Interface. C'est une langue qui permet aux ordinateurs, aux instruments de musique et à d'autres matériels de communiquer. Le protocole MIDI inclut l'interface, la langue dans laquelle les données MIDI sont transmises et les connexions nécessaires pour communiquer entre le matériel.

Tout d'abord, le MIDI est une norme largement utilisée dans le domaine de la musique électronique. Il permet de transmettre des informations de contrôle musical, telles que les notes jouées, la vélocité et les changements de paramètres, sous forme de messages numériques légers. Cela facilite la communication entre le clavier et d'autres systèmes, comme une application mobile ou un orchestrion automatisé.

Ensuite, le MIDI est optimisé pour fonctionner avec un faible volume de données, ce qui réduit la charge de calcul sur le microcontrôleur ESP32. Contrairement à un signal audio qui nécessite un traitement complexe, les messages MIDI sont de simples instructions numériques, rendant le système plus réactif.

Le tableau des notes dans un message MIDI :

Octave (-2 à 8 ou -1 à 9)	Do	Do#	Re	Re#	Mi	Fa	Fa#	Sol	Sol#	La	La#	Si
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-2 (ou -1)	0	1	2	3	4	5	6	7	8	9	10	11
-1 (ou 0)	12	13	14	15	16	17	18	19	20	21	22	23
0 (ou 1)	24	25	26	27	28	29	30	31	32	33	34	35
1 (ou 2)	36	37	38	39	40	41	42	43	44	45	46	47
2 (ou 3)	48	49	50	51	52	53	54	55	56	57	58	59
3 (ou 4)	60	61	62	63	64	65	66	67	68	69	70	71
4 (ou 5)	72	73	74	75	76	77	78	79	80	81	82	83
5 (ou 6)	84	85	86	87	88	89	90	91	92	93	94	95
6 (ou 7)	96	97	98	99	100	101	102	103	104	105	106	107
7 (ou 8)	108	109	110	111	112	113	114	115	116	117	118	119
8 (ou 9)	120	121	122	123	124	125	126	127	-	-	-	-

Figure – message sur plusieurs octaves pour le MIDI

3– Choix pour la conception des PCB

Pour la conception des circuits imprimés (PCB), nous avons retenu **Proteus 8** comme principal logiciel de développement. Ce choix s'explique par plusieurs facteurs, notamment la licence universitaire qui nous permet de l'utiliser gratuitement et sur n'importe quel PC.

De plus, ayant été formés à son utilisation tout au long de notre cursus en BUT GEII, nous sommes déjà habitués à son interface et à ses outils, ce qui optimise notre efficacité dans la conception des circuits.



Proteus 8 nous offre plusieurs **avantages** :

- Éditeur de schémas intuitif, facilitant la création et l'organisation des circuits.
- Routage assez simple + visualisation 3D.
- Génération rapide des fichiers Gerber, indispensables pour la fabrication des PCB.

Cependant, nous reconnaissons que la simulation des circuits sous Proteus est parfois limitée, avec un choix de composants restreint par rapport à d'autres outils comme **LTSpice**. Malgré cela, pour notre projet, cette limitation reste acceptable car l'accent est mis sur la conception et la fabrication du PCB plutôt que sur la simulation avancée.

On a écarté **KiCad**, bien qu'il soit puissant et open-source, car nous n'avons pas eu l'occasion de nous former dessus et il nécessiterait un temps d'apprentissage supplémentaire. Quant à **Altium Designer**, malgré ses outils avancés, il est payant et surdimensionné pour les besoins de notre projet.

4– Modélisation 3D

Pour la conception des pièces mécaniques de notre clavier numérique multifonction, nous avons utilisé **Autodesk Fusion 360**. Ce logiciel nous a permis de modéliser avec précision les différents éléments de notre piano, notamment la base du boîtier et les touches, en prenant en compte les contraintes d'assemblage et d'impression 3D.



Fusion 360 nous offre plusieurs avantages :

- **Modélisation paramétrique** : Permet d'ajuster facilement les dimensions et d'optimiser l'agencement des composants.
- **Génération des fichiers STL** : Indispensable pour l'impression 3D des pièces, garantissant une compatibilité avec la majorité des imprimantes 3D.
- **Visualisation et simulation** : Permet d'anticiper d'éventuels problèmes d'encombrement ou d'assemblage avant la fabrication.

Grâce à ce logiciel, nous avons pu concevoir un boîtier ergonomique et fonctionnel, adapté aux composants électroniques et à l'expérience utilisateur du clavier.

III - Détails de conception

1 - système d'alimentation

Conformément aux exigences du **cahier des charges**, le système doit être alimenté aussi bien par le secteur que par une batterie rechargeable, garantissant une autonomie minimale d'une heure en utilisation.

Consommation en énergie du système

Pour dimensionner correctement le système d'alimentation, il est nécessaire d'analyser les caractéristiques électriques des différents composants consommateurs d'énergie.

Microcontrôleur ESP32 Wroom

L'ESP32 Wroom, qui constitue le cœur du système, nécessite une alimentation minimale de **3,3V** avec une consommation de **500mA**, soit une puissance de **1,65W**

Bande de led néopixel

Les LEDs Neopixel sont utilisées pour indiquer la touche active du piano. Chaque LED consomme 60mA sous 5V. Étant donné que chaque touche est associée à deux LEDs et qu'un accord ne dépassera pas 7 touches simultanées, le courant maximal consommé est :

$$7 \times 2 \times 60mA = 840mA$$

On en déduit les caractéristiques d'alimentation :

$$5V \times 0,84A = 4,2W$$

Amplificateur et hauts parleurs

L'amplificateur audio utilise **deux amplificateurs opérationnels (AOP)** alimentés sous $\pm 5V$, qui dissipent une partie de l'énergie. Leur consommation est calculée comme suit :

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

La puissance de sortie est de **325mW** par amplificateur, ce qui donne une consommation totale de :

$$P_{out} = 325mW$$

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

Comme le système utilise **deux AOP**, la puissance totale dissipée est:

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

Les **haut-parleurs** utilisés ont une puissance de **4W RMS** chacun. Étant donné que le système comprend **deux haut-parleurs**, la puissance totale consommée est :

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

On en déduit les caractéristiques d'alimentation : 8,73W / 5V / 1,7

A partir des caractéristiques On a 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
néo pixel	5	1	5
Ampli + haut-parleur	5	2,2	11
Total			18,31

Tableau 1- consommation totale du système

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 :

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

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

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.

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

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

On aurait donc besoin d'un chargeur délivrant au minimum $3\text{A} \times 8,4\text{V} = 25,2\text{W}$

On utilisera un chargeur 12

Conception du circuit de la carte d'alimentation

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

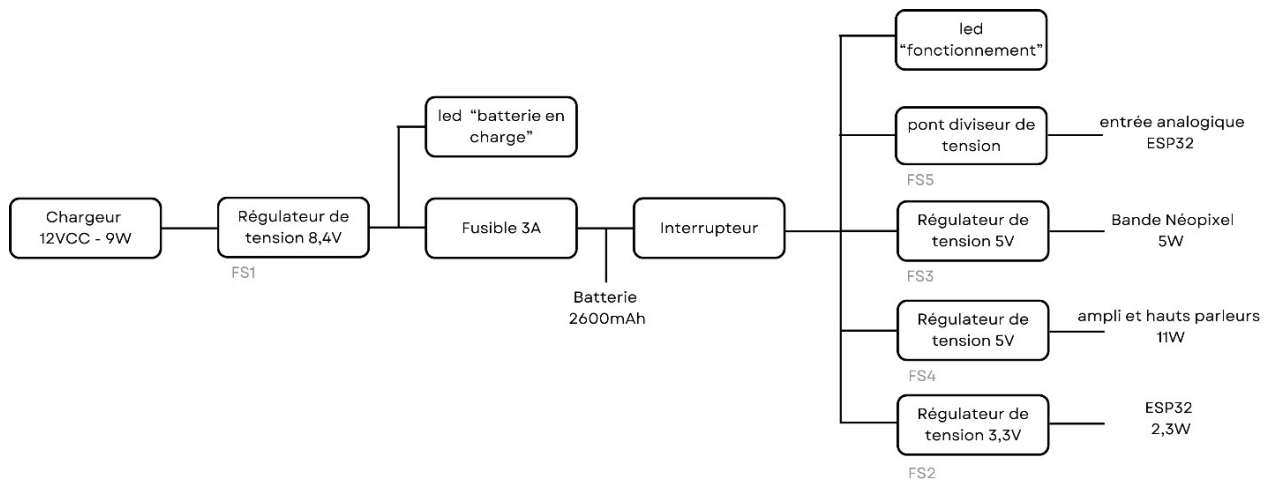


Schéma : système carte d'alimentation

FS1 régulateur de tension 8,4V

On s'est inspiré d'un montage de la documentation du régulateur LM317.

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

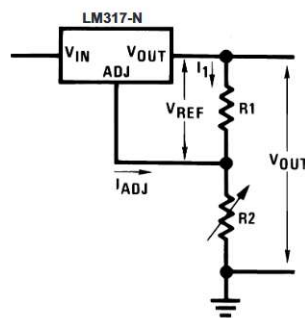


Figure 15. Setting the V_{OUT} Voltage

Figure 1- extrait de la documentation du régulateur LM317

Les résistances R1 et R2 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 qu'un courant maximal de **1A**. Afin d'augmenter la capacité de courant du régulateur, un **transistor PNP** a été ajouté. La résistance **R1**, utilisée pour la polarisation du transistor, a été dimensionnée de manière à activer le transistor dès que nécessaire tout en limitant le courant circulant dans le LM317..

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

Pour finir on a ajouté des condensateurs de découplage pour éviter la transmission de parasites.

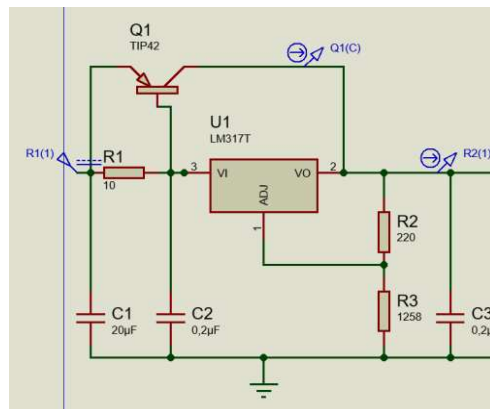


Figure 2- extrait du schéma du circuit : FS1

FS2 – régulateur 3,3V

Pour réaliser ce régulateur de tension, un LM317 a été utilisé en combinaison avec un transistor PNP. Afin de fixer la tension de sortie, les résistances ont été définies en appliquant la formule issue 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 :

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

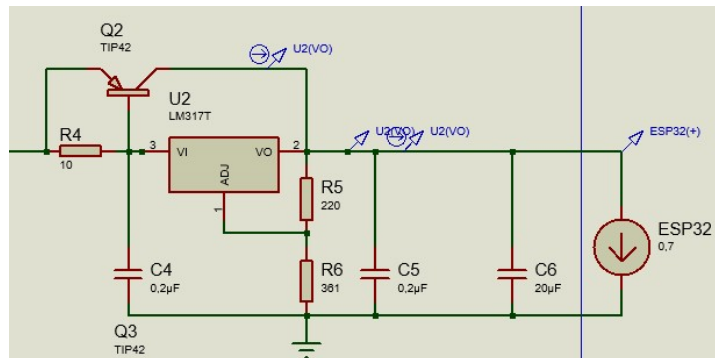
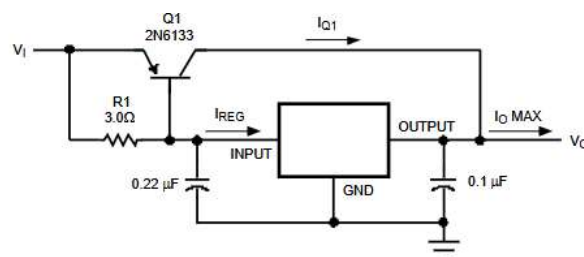


Figure 3- extrait du schéma du circuit : FS2

FS3/FS4 - Régulateurs 5v

Pour réguler la tension à 5V, On a reproduit le montage donné dans la documentation du LM7805.



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

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

Figure 24. High Current Voltage Regulator

Figure 4- extrait de la documentation du régulateur LM7805

Ce montage utilise un transistor PNP pour augmenter le courant délivré ainsi que des condensateurs de découplage. Un condensateur de 2 μF a été ajouté en sortie du circuit afin de filtrer les parasites pouvant apparaître dans les fils reliant la carte d'alimentation aux consommateurs.

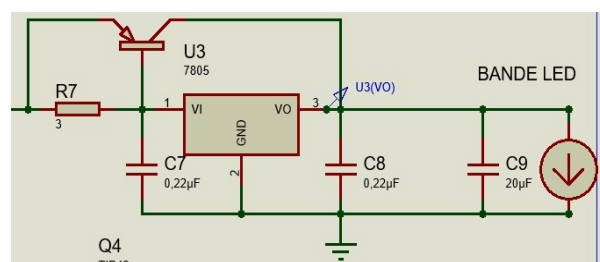


Figure 5- extrait du schéma du circuit : FS3/FS4

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.

On a fait 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 on prend $R_{10} = 1,2k$

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

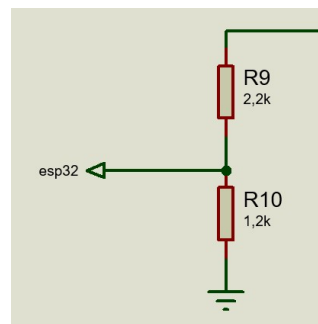


Figure 6- extrait du schéma du circuit : FS5

Composant supplémentaire

Plusieurs composants ont été ajoutés au montage afin d'améliorer la sécurité et la signalisation du système.

Tout d'abord, des composants de sécurité ont été intégrés pour protéger le circuit. Un fusible a été placé en amont afin de prévenir toute surcharge, tandis qu'une diode a été ajoutée pour éviter les retours de courant pouvant endommager les composants. De plus, un interrupteur permet de couper l'alimentation du système si nécessaire.

Ensuite, des LEDs indicatrices ont été mises en place pour fournir des informations sur l'état de l'alimentation. Une première LED signale lorsque la batterie est en cours de charge, tandis qu'une seconde LED indique que le système est alimenté et en fonctionnement.

Circuit complet

Sur le logiciel Proteus, l'ensemble du circuit a été conçu. Afin de valider son efficacité, l'outil de simulation intégré au logiciel a été utilisé pour tester son bon fonctionnement :

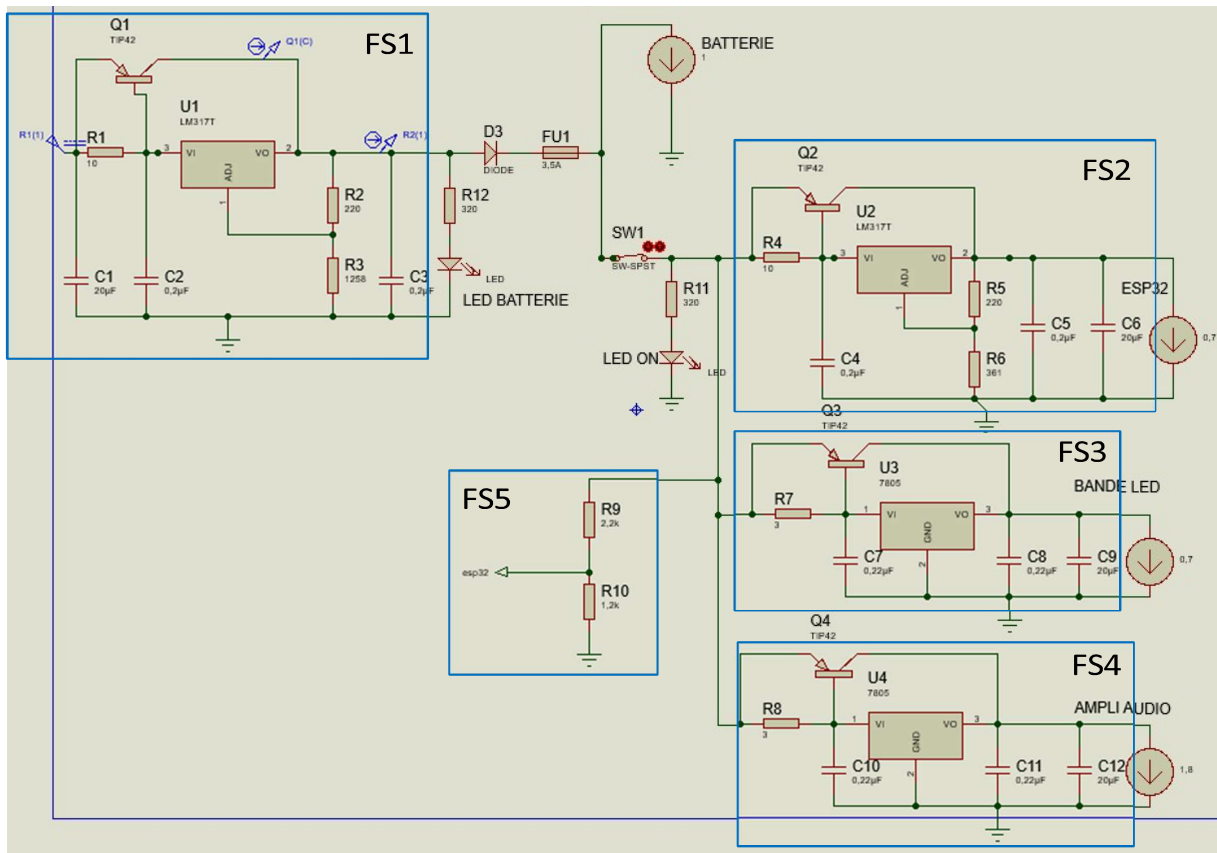


Figure 7 - schéma du circuit de la carte d'alimentation

2 - Microcontrôleur

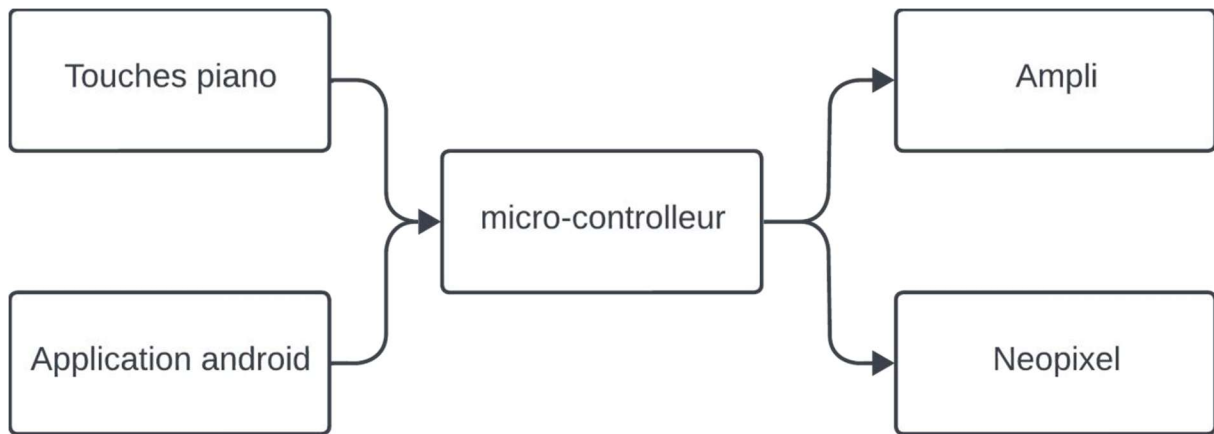


Figure : acteurs principaux du microcontrôleur

Détection des touches

Notre instrument propose trois modes de jeu : manuel, semi-automatique et automatique.

Dans le mode manuel, l'utilisateur joue en appuyant directement sur les touches du clavier. Chaque appui déclenche la production d'un son. Cependant, le clavier couvrant deux octaves avec 24 touches, une connexion individuelle de chaque touche à un GPIO distinct aurait nécessité 24 entrées/sorties, ce qui est peu optimal en termes de ressources disponibles sur le microcontrôleur.

Pour optimiser l'utilisation des GPIOs, un clavier matriciel a été adopté. Au lieu de connecter chaque touche à un seul GPIO et à la masse ou au 3,3V, chaque touche est reliée à deux GPIOs : l'un correspondant à une ligne, l'autre à une colonne. Grâce à cette configuration, le nombre de GPIOs utilisés passe de 24 à 10, réduisant ainsi la consommation des ressources tout en conservant une détection efficace des appuis.

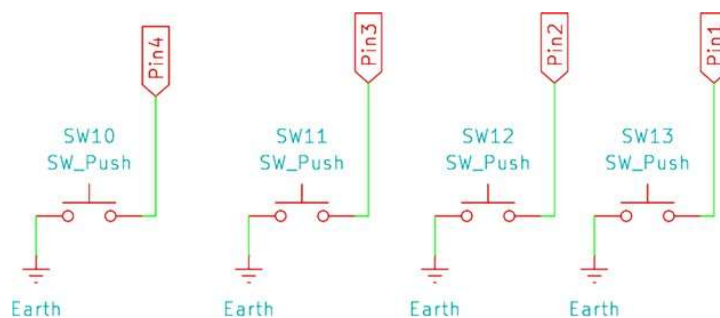


Figure 8 - Boutons "Normaux"

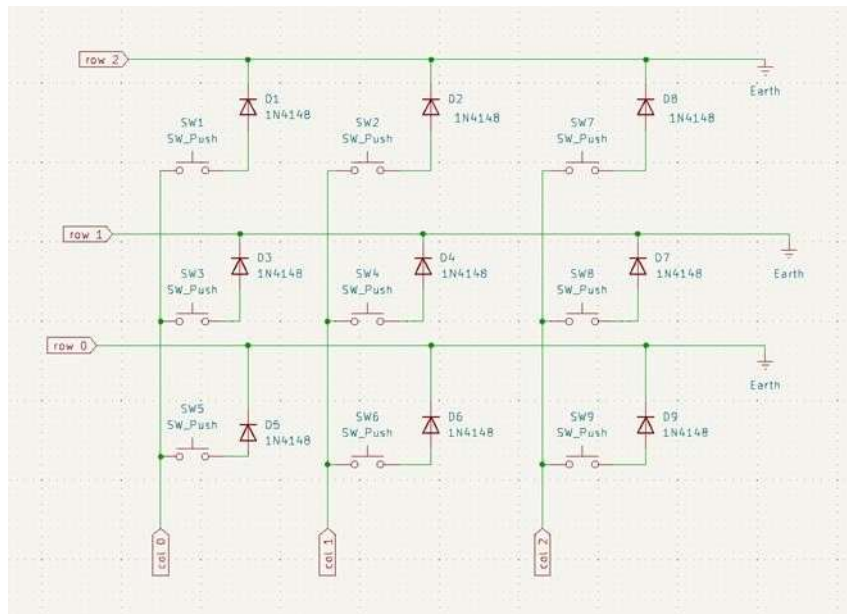


Figure 9 - Boutons en matrice

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.

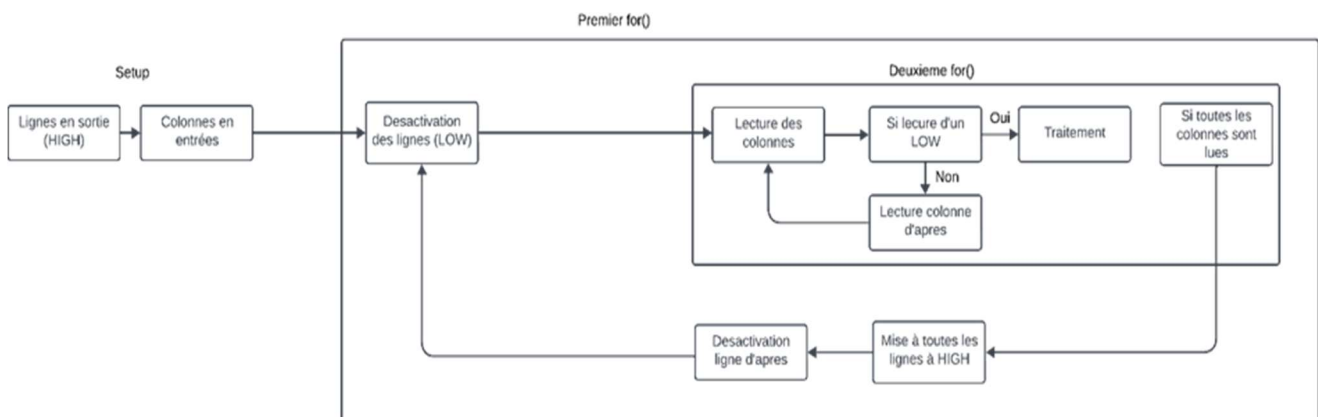


Figure 10 -Algorithme de lecture de bouton en matrice

```
// 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 11 - 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 12 - Programme de détection de touches

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 13 - 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 14 - 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 15 - Changement de fréquence du sinus

La figure 15 illustre un début d'implémentation fonctionnelle. En résumé, les deux fonctions présentées permettent de modifier les fréquences et d'additionner plusieurs sinusoïdes en fonction du nombre de boutons appuyés.

La majorité du traitement est effectuée dans la fonction `updateControl()`, car la fonction `updateAudio()` est appelée à des intervalles très courts (0,3 μ s). Réaliser des calculs complexes directement dans cette dernière risquerait de ralentir l'exécution et d'altérer le signal audio généré.

Contrôle à distance

Pour le mode automatique et semi-automatique, on a opté pour un contrôle à distance du clavier avec une synchronisation de bande de LEDs Neopixels selon la touche appuyée. Pour créer l'application, on a utilisé Android Studio. On a choisi Android Studio plutôt que d'autres outils tout simplement parce que c'est l'outil qu'on connaît le mieux et c'est celui qui est utilisé dans l'industrie, ce qui constitue un bon entraînement et une mention supplémentaire dans notre CV. Pour la communication entre l'application et notre microcontrôleur, on a opté pour le BLE.

On aurait aussi pu choisir l'USB avec la communication série, ou même le Wi-Fi, mais avoir un câble constamment branché à l'appareil est moins impressionnant et amusant. Et le Wi-Fi est utilisé avec le MIDI dans des cadres plus importants, par exemple dans un studio où beaucoup d'appareils doivent communiquer en même temps et où la consommation d'énergie

n'est pas un problème. On va d'abord présenter l'interface de l'application, puis la connexion BLE et l'intégration du MIDI..

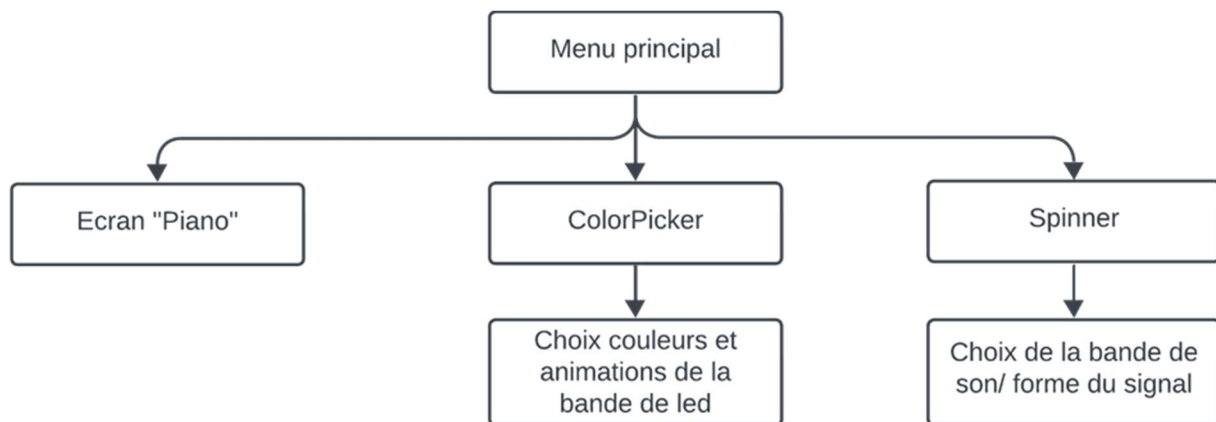


Figure 16 - Architecture de l'application

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.

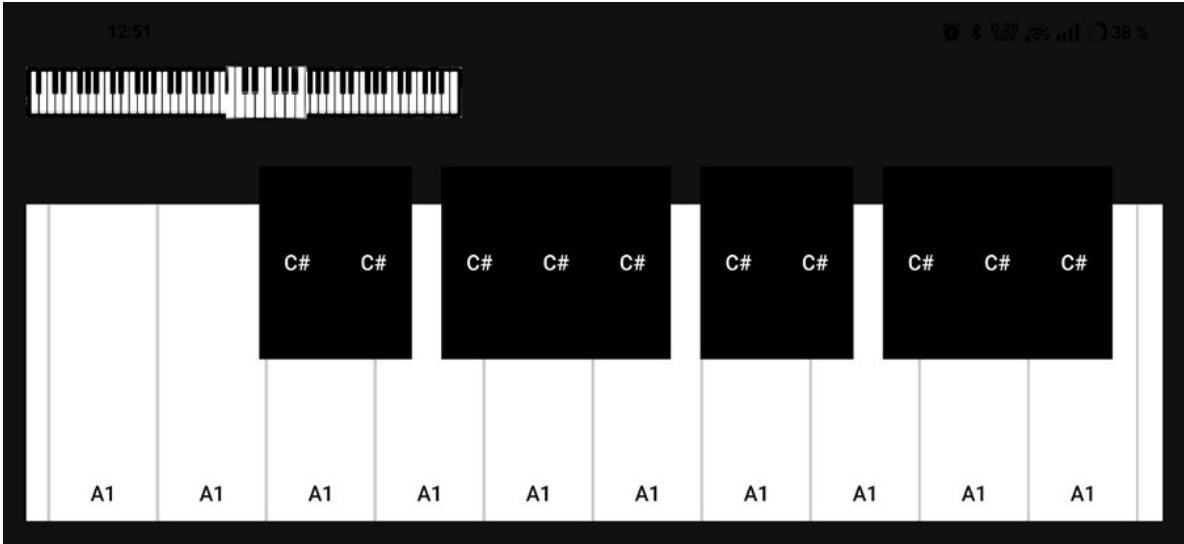
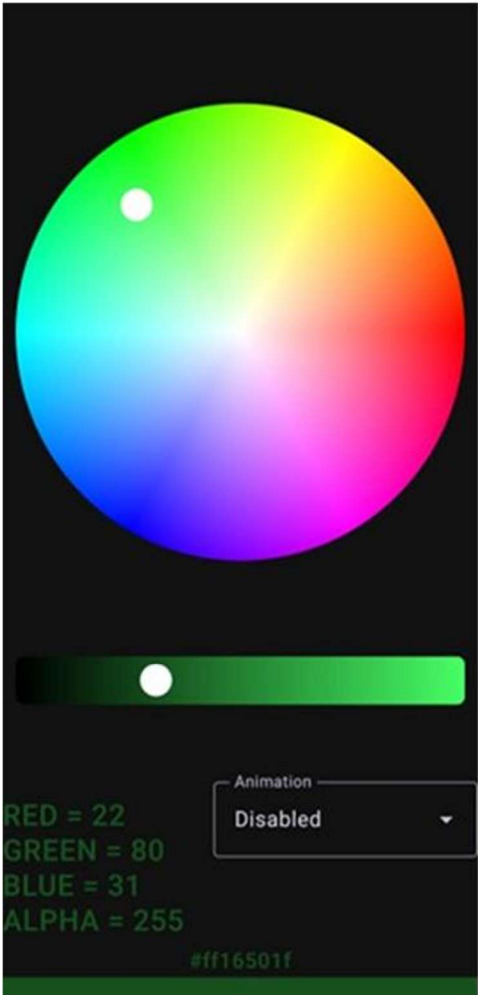


Figure 17 - Ecran Piano via notre application

Il n'est pas nécessaire de détailler la conception de chaque bouton, car cela serait trop long. L'explication se concentrera donc sur le fonctionnement du Bluetooth Low Energy (BLE).

Le BLE repose sur une architecture client-serveur basée sur le Generic Attribute Profile (GATT). Dans notre projet, l'ESP32 joue le rôle de serveur, tandis que l'application mobile agit en tant que client. Ce modèle permet au client de se connecter au serveur pour lire, écrire ou recevoir des notifications sur les données échangées, facilitant ainsi la communication entre le clavier et l'application.

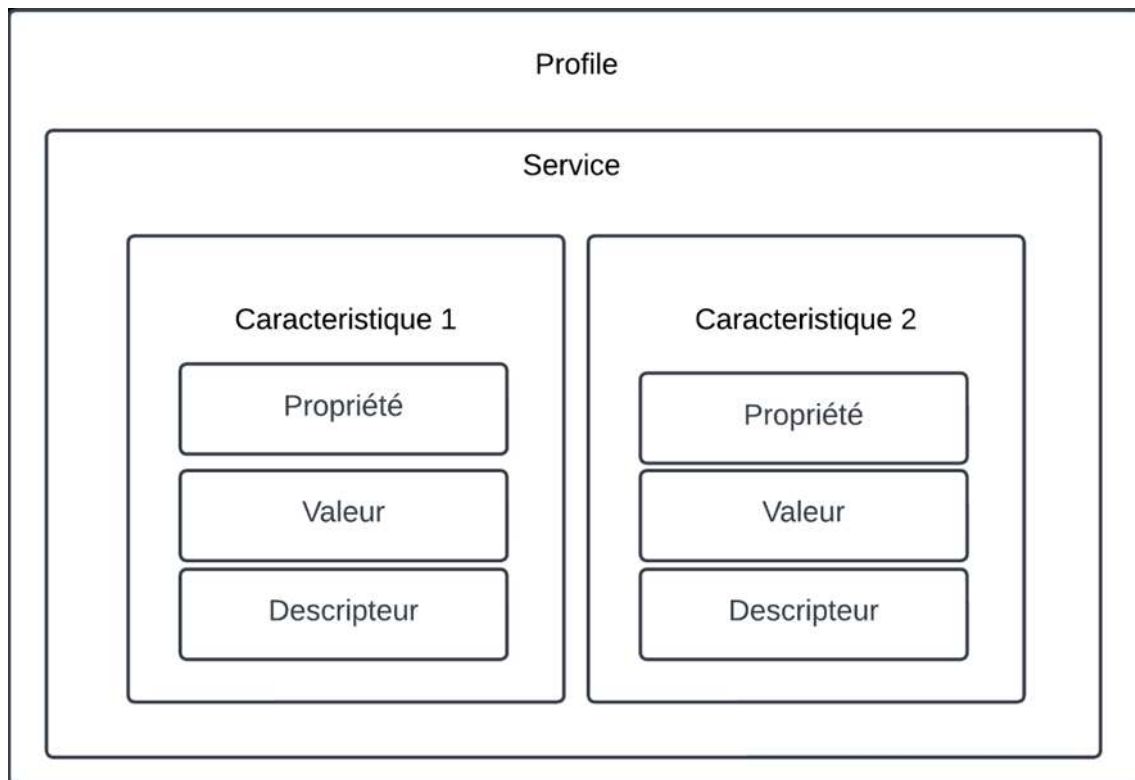


Figure 18 - Serveur GATT

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 19 - 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 20 - Initialisation BLE

On peut voir qu'on a utilisé les propriétés READ et WRITE pour chaque caractéristique. Grâce à ces propriétés, il est possible de 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, il va montrer comment fonctionne l'envoi de données avec l'application. Tout d'abord, il faut demander des autorisations et activer le Bluetooth, s'il n'est pas déjà activé.

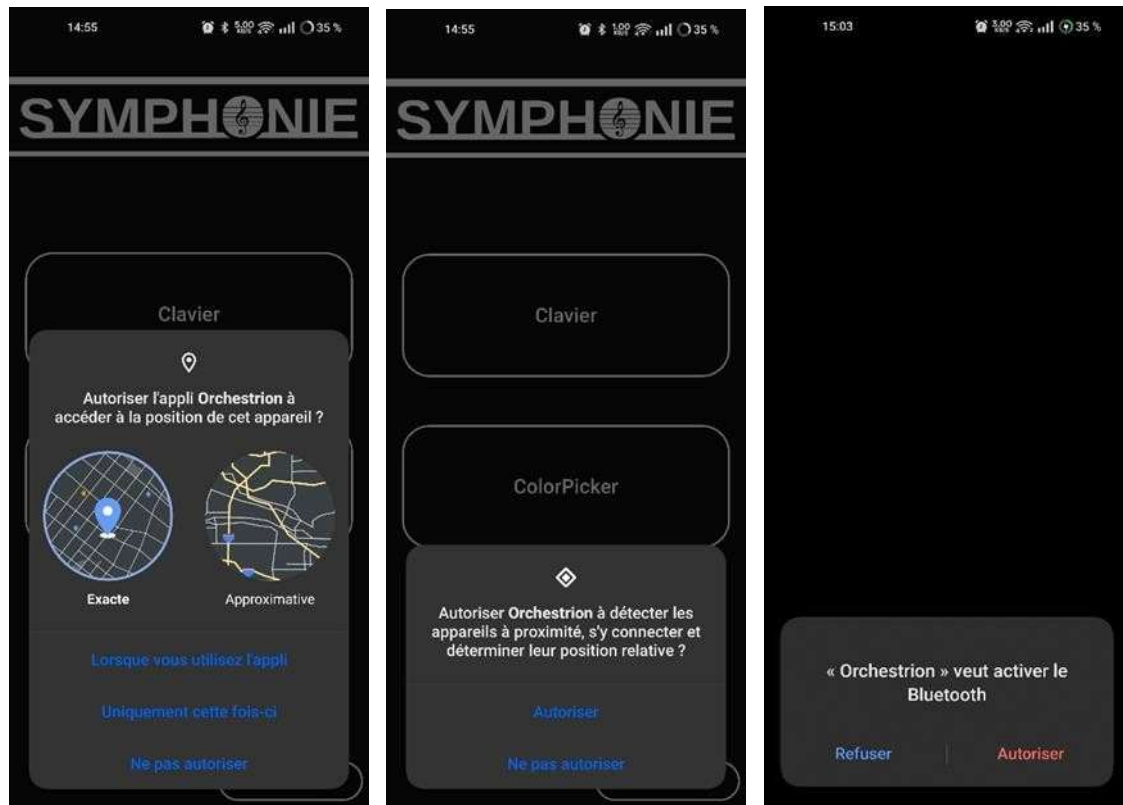


Figure 21- Demandes d'autorisations

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.

```
@SuppressLint("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 22 - Fonction d'envoi d'ordres Midi

Tout est converti 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 23 - CODE 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 en 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 24 - "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.isConnected()){
            if (MidiCallback.update()){
                MidiCallback.setUpdate(false);
                WhatsNew[0] = MIDI;
            }else
                WhatsNew[0] = {No_New_Msg};

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

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

    return WhatsNew;
}
```

Figure 25 - Détection de nouveaux messages BLE

Pour mettre à jour nos variables de la boucle du main.cpp, on a créé la fonction loopBLE() qui retourne un tableau d'enum avec chaque enum représentant 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 recu 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 26 - Traitement donnée BLE (Extrait de la loop/main.cpp)

Et en appelant loopBLE() dans la boucle principale, on peut récupérer toutes les données souhaitées. Comme on a mis un timer de 100 ms à la fonction loopBLE(), si on récupérait le tableau WhatsNew (figure 26) comme dans la figure 27, cela ne fonctionnerait que pour la première valeur du tableau, car à chaque ligne, loopBLE serait appelée, sauf que les 100 ms ne seraient pas écoulées. Pour cela, on a utilisé la fonction memcpy qui permet de copier un tableau dans un autre, ce qui fait que la fonction est appelée une seule fois. On a ajouté la fonction reset_tab() qui permet de réinitialiser WhatsNew, car sinon, au lieu de traiter les données une seule fois, elles seraient traitées pendant 100 ms 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 27- 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 28 - Fonctions de récupération de donnée

Ici on peut 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 29 - Classe BLE_Midi



Figure 30 - Chemin de la variable

Grâce à cela, il n'est pas nécessaire de stocker les variables dans la classe BLE_Midi, on les récupère directement depuis leurs enfants.

Tout au long de la création de ce code, on a eu l'intention de rendre le code le plus modulaire possible, et cela se voit le mieux dans la figure 25. Pour la mise à jour des couleurs, on aurait pu simplement inclure 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 futurs projets. Si on avait fait cela, lors d'une réutilisation future du code, il aurait fallu prendre du 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 on aurait implémenté la classe Neopix. Et si on l'avait fait pour la classe Neopix, on l'aurait sûrement fait pour la classe gérant les boutons.

Optimiser la mémoire ou la modularité des classes n'est clairement pas nécessaire pour 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

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 :

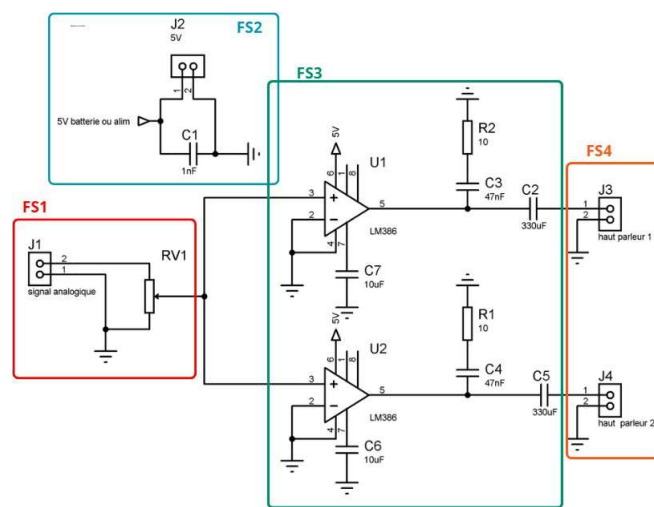


Figure 31- Schéma de la partie amplification audio

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 μF et 330 μF ont été ajoutés pour stabiliser l'alimentation et améliorer la qualité du son en éliminant les parasites. Les **haut-parleurs 4 Ω - 4W** ont été sélectionnés pour être compatibles avec la sortie amplifiée du LM386.

Le schéma électronique a été conçu sous **Proteus 8**.

Aux bornes du bornier J1, c'est-à-dire à la sortie de l'ESP32 WROOM, on doit avoir une tension AC maximum de 880mV crête à crête, ajustable entre [-400mV ; +400 mV] grâce au potentiomètre.

Tous les calculs et les justifications nécessaires sont disponibles dans le fichier "**Dossier_fabrication_SimonMARTIN_V1_3.pdf**" dans le cy.git

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.

4- Partie à venir

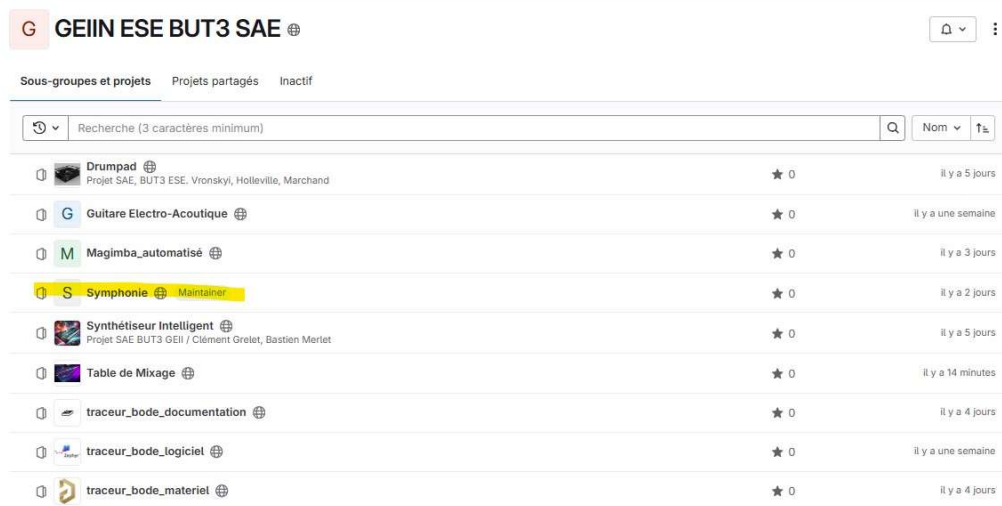
Le développement du projet est **toujours en cours**, et ce dossier de conception reste **évolutif**. Actuellement, l'ensemble des pièces 3D du boîtier et des touches du clavier ont été imprimées. La prochaine étape consiste à vérifier leur assemblage avant de les intégrer définitivement au dossier, car il serait prématuré d'inclure cette partie dans ce dossier tant que la validation mécanique n'a pas été effectuée.

En parallèle, la partie hardware est toujours en développement. Le système d'alimentation a récemment été finalisé et sa simulation a été validée. Il reste à réaliser le routage du circuit imprimé (PCB), puis à effectuer les tests de validation avant son intégration dans le prototype alpha.

Enfin, d'autres sections de ce dossier seront amenées à évoluer afin d'apporter plus de précisions et de s'assurer qu'elles répondent pleinement aux exigences du client.

ANNEXE

Pour accéder aux documents nécessaires (cdc, df,...) par rapport à ce projet, rendez-vous dans le git.cyu.fr, du GEII ESE BUT3 SAE -> choisir *Symphonie* :



The screenshot shows the GitHub-like interface of git.cyu.fr. At the top, the repository name 'GEIIN ESE BUT3 SAE' is displayed. Below it, there are tabs for 'Sous-groupes et projets', 'Projets partagés', and 'Inactif'. A search bar is present with the placeholder text 'Recherche (3 caractères minimum)'. A list of projects is shown, each with a repository icon, a name, a description, a star count, and a last update time. The project 'Symphonie' is highlighted with a yellow background. It is maintained by 'Maintainer' and has 0 stars. Other projects include 'Drumpad', 'Guitare Electro-Acoustique', 'Magimba_automatisé', 'Synthétiseur Intelligent', 'Table de Mixage', 'traceur_bode_documentation', 'traceur_bode_logiciel', and 'traceur_bode_materiel'.

Repository	Description	Stars	Last Update
Drumpad	Projet SAE, BUT3 ESE, Vronskyi, Hotteville, Marchand	0	il y a 5 jours
Guitare Electro-Acoustique		0	il y a une semaine
Magimba_automatisé		0	il y a 3 jours
Symphonie		0	il y a 2 jours
Synthétiseur Intelligent	Projet SAE BUT3 GEII / Clément Grelet, Bastien Merlet	0	il y a 5 jours
Table de Mixage		0	il y a 14 minutes
traceur_bode_documentation		0	il y a 4 jours
traceur_bode_logiciel		0	il y a une semaine
traceur_bode_materiel		0	il y a 4 jours

Figure 32- accès git cy de symphonie