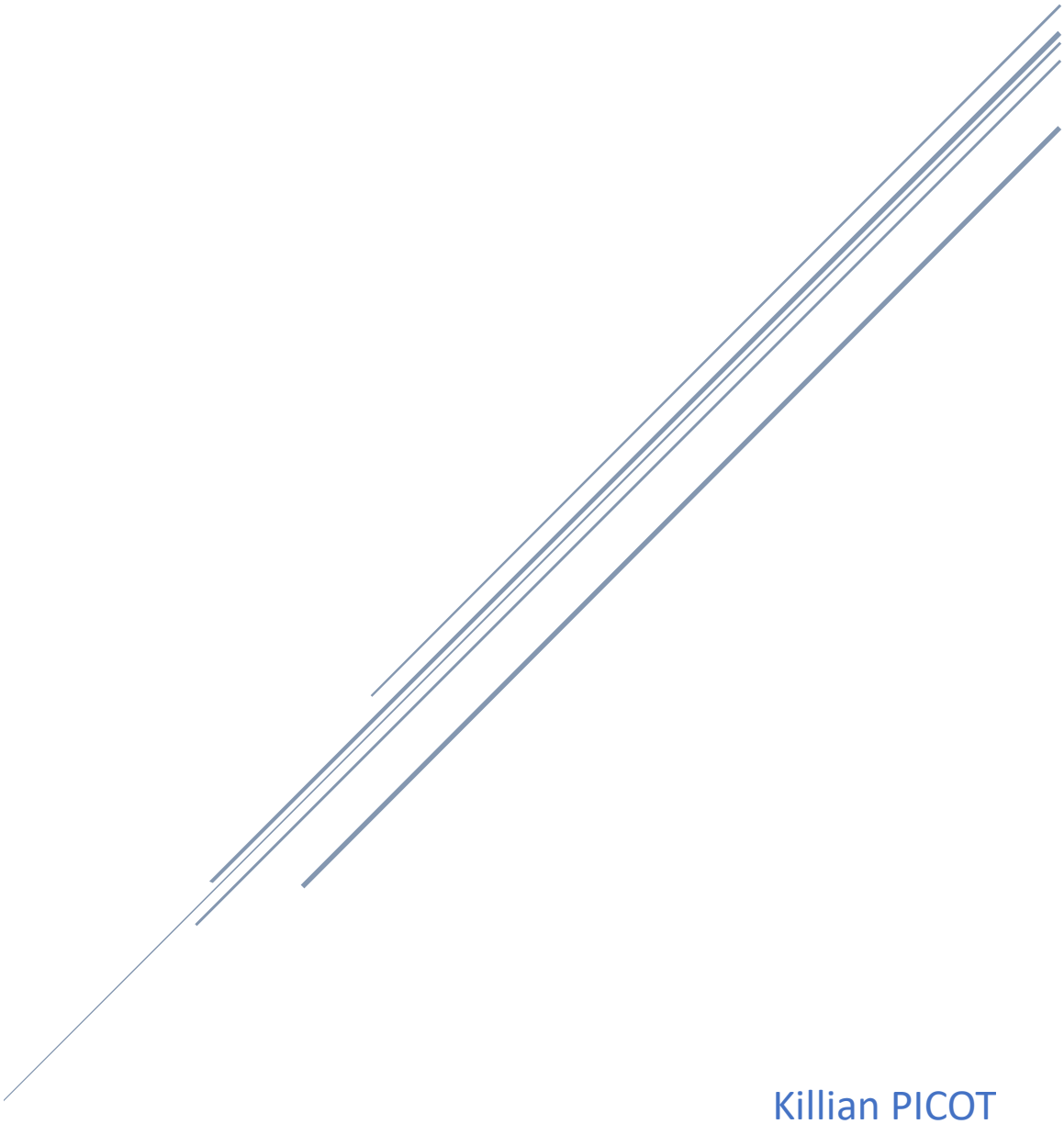
Le Mans Université

ENSIM
École d'ingénieurs
Le Mans Université

RAPPORT DE PROJET INTERMEDIAIRE

Acquisition de signaux et classification en temps réel



Killian PICOT
5A ASTRE 2024 - 2025

Table des matières

Table des figures

I. Lexique et Acronymes

- **ARM** : Architecture RISC (Reduced Instruction Set Computing) développée par Arm Holdings, utilisée dans une variété de processeurs et microcontrôleurs. Connue pour son efficacité énergétique et sa performance.
- **BDD (Base de Données)** : Ensemble de données structurées et organisées pour faciliter l'accès, la gestion et la mise à jour. Utilisée dans de nombreuses applications pour stocker des informations.
- **Buffer** : Zone de mémoire temporaire utilisée pour stocker des données en transit entre deux dispositifs ou processus. Permet de gérer les décalages entre les vitesses de transfert de données.
- **BSP (Board Support Package)** : Ensemble de logiciels et de bibliothèques nécessaires pour faire fonctionner l'ensemble des périphériques sur un matériel spécifique. Facilite le développement et l'intégration de matériel.
- **CPU (Central Processing Unit)** : Processeur central d'un ordinateur qui exécute les instructions des programmes. Il est souvent considéré comme le cerveau de l'ordinateur.
- **CODEC (Coder-Decoder)** : Dispositif ou logiciel qui compresse ou décompresse des données numériques, notamment des fichiers audios et vidéo. Utilisé pour réduire la taille des fichiers tout en conservant la qualité.
- **CSV (Comma-Separated Values)** : Format de fichier texte utilisé pour stocker des données tabulaires, où chaque valeur est séparée par une virgule. Couramment utilisé pour l'importation et l'exportation de données entre programmes.
- **DMA (Direct Memory Access)** : Technique permettant aux périphériques d'accéder directement à la mémoire sans passer par le processeur. Améliore l'efficacité du transfert de données et libère le processeur pour d'autres tâches.
- **DFSDM (Digital Filter for Sigma-Delta Modulators)** : Module utilisé pour le traitement du signal dans les microcontrôleurs, notamment pour la conversion de signaux PDM (Pulse Density Modulation) en signaux PCM (Pulse Code Modulation).

- **DSI (Display Serial Interface)** : Interface série utilisée pour transmettre des données vidéo et de contrôle à un écran LCD-TFT. Offre une connectivité haute vitesse et une faible consommation d'énergie.
- **DSE (Densité Spectrale d'Énergie)** : Mesure de la distribution de l'énergie d'un signal dans le domaine fréquentiel. Utilisée pour analyser les caractéristiques d'un signal.
- **Epoch** : Période durant laquelle un modèle d'intelligence artificielle voit l'ensemble des données de la base de données une fois durant l'entraînement. Utilisée pour mesurer les cycles d'entraînement d'un modèle.
- **FAT (File Allocation Table)** : Système de fichiers utilisé pour gérer l'espace de stockage sur des supports de mémoire, comme les disques durs et les cartes mémoire. Connu pour sa compatibilité avec différents systèmes d'exploitation.
- **FFT (Fast Fourier Transform)** : Algorithme efficace pour calculer la transformation de Fourier d'une séquence, utilisé en traitement du signal pour analyser les fréquences d'un signal. Il est largement appliqué en ingénierie et en science.
- **GPIO (General-Purpose Input/Output)** : Broches sur un microcontrôleur ou un microprocesseur, utilisées pour lire des signaux d'entrée ou envoyer des signaux de sortie. Elles sont configurables pour différentes tâches.
- **IA (Intelligence Artificielle)** : Dans le cas de ce projet, l'IA réfère à des algorithmes d'apprentissage automatique et plus particulièrement à l'apprentissage profond.
- **I2C (Inter-Integrated Circuit)** : Bus de communication série utilisé pour interconnecter des composants électroniques. Permet la communication entre un microcontrôleur et ses périphériques avec seulement deux lignes de signal.
- **I2S (Inter-IC Sound)** : Interface série utilisée pour la transmission de données audio entre des composants numériques. Utilisée couramment dans les systèmes audios numériques.
- **LCD-TFT (Liquid Crystal Display - Thin Film Transistor)** : Type de technologie d'affichage à cristaux liquides utilisant des transistors à couches minces pour

améliorer la qualité de l'image. Offre des couleurs vives et un temps de réponse rapide.

- **LED (Light-Emitting Diode)** : Diode électroluminescente utilisée comme source de lumière dans divers dispositifs électroniques. Connu pour sa faible consommation d'énergie et sa longue durée de vie.
- **LTDC (LCD-TFT Display Controller)** : Contrôleur d'affichage pour les écrans LCD-TFT, gérant l'envoi des données d'image à l'écran. Utilisé dans les applications embarquées pour afficher des graphiques et des interfaces utilisateur.
- **MCU (Microcontroller Unit)** : Microcontrôleur, un système simple permettant de faire des calculs à la manière d'un ordinateur mais avec des ressources plus limitées. Il embarque plusieurs périphériques pour des tâches spécifiques.
- **Microcontrôleur** : Système simple permettant de faire des calculs à la manière d'un ordinateur mais avec des ressources plus limitées. Il embarque plusieurs périphériques.
- **PDM (Pulse Density Modulation)** : Technique de modulation utilisée dans les systèmes de conversion analogique-numérique pour représenter le signal d'entrée sous forme de densité d'impulsions.
- **PCM (Pulse Code Modulation)** : Méthode de conversion d'un signal analogique en un signal numérique en échantillonnant l'amplitude du signal à intervalles réguliers.
- **RAM (Random Access Memory)** : Mémoire vive permettant de stocker temporairement des données pour un accès rapide par le processeur. Essentielle pour le fonctionnement des applications et du système.
- **ROM (Read-Only Memory)** : Mémoire non volatile utilisée pour stocker des données qui ne doivent pas être modifiées. Contient souvent le firmware d'un système informatique.
- **Scratch Buffer** : Mémoire tampon temporaire utilisée pour stocker des données intermédiaires ou temporaires durant le traitement des données.
- **SAI (Serial Audio Interface)** : Interface série pour la transmission de données audio numériques. Permet la communication entre des périphériques audio et des microcontrôleurs

Lexique et Acronymes généré par intelligence artificiel

II. Introduction

L'utilisation de l'IA est de plus en plus répandue au sein de nos utilisations quotidiennes. En effet, de nombreuses personnes utilisent maintenant ces logiciels de manières quotidiennes, que ce soit les IA génératives via des modèles de langage comme chat GPT de la société OpenIA ou encore Copilot de Microsoft. Ces IA nous permettent d'améliorer notre productivité, néanmoins ils nécessitent des ressources de calculs importantes afin de pouvoir les utiliser et les déployer. Ces algorithmes paraissent donc de prime-abord peu compatibles, sur des systèmes aux ressources limités tels que des microcontrôleurs. Cependant, depuis quelques années on observe l'émergence de systèmes, certes moins efficaces mais plus légers, plus simples répondant aux contraintes techniques de ces périphériques aux ressources limitées. Cet ensemble de modèles d'IA et de techniques sont regroupés au sein de ce qui a été nommé « TinyML ». Il s'agit en pratique de techniques de bon sens et algorithmiques afin de réduire l'emprunte des IA sur la mémoire et les performances des systèmes. Néanmoins, bien que l'engouement autour du TinyML soit fort, il en est réduit pour le moment à faire ces preuves dans le milieu de l'IA. En effet, peu d'acteur participe activement à son développement et il n'est encore que très peu utilisé dans la sphère professionnelle autrement que pour de la recherche.

Ce projet rentre dans ce contexte-là. Effectivement, le but est de réaliser un système permettant de faire fonctionner une IA avec des contraintes en temps réel sur un microcontrôleur. Pour ce faire, je me suis basé essentiellement sur la thèse « *Multi-purpose acoustic sensor for smart home* » (Ahmed, 2023). Cette thèse décrit spécifiquement la procédure effectuée afin de réaliser une IA permettant de classifier plusieurs bruits sonores (pluie, pas, vent, voiture) sur une carte embarquée type STM32 de la famille L4. La thèse se conclut par la faisabilité d'un tel système en temps réel sur un microcontrôleur sans pour autant parvenir à le réaliser. Le livrable final de ce projet est dans la continuité de cette thèse, car il doit être la réalisation pratique d'une IA de classification audio, fonctionnant en temps réel sur une carte embarquée. Le projet se centre donc sur la réalisation d'acquisition et de traitement de signal audio, ainsi que celui d'une IA. Les contraintes définies par le Annexe 1 : Cahier des charges (Annexe 1) sont la latence, le système sera considéré temps réel si la latence du système est inférieure à 300 ms et la performance du modèle d'IA qui doit être supérieure à 70%.

III. Gestion de projet

1) Outils de gestion de projet

Afin de gérer le temps efficacement, malgré le fait que je sois seul dans mon groupe, j'ai mis en place un système permettant de visualiser l'avancée du projet via des tâches prédéfinies à l'avance, que je mets à jour au fur et à mesure de leur complétion. Il s'agit de la plateforme https://gestion-projets.univ-lemans.fr mise en place par l'université s'appuyant sur l'outil « open source Redmine » (*figure 1)*. Cet outil permet également à mon encadrant de visualiser l'avancée du projet.

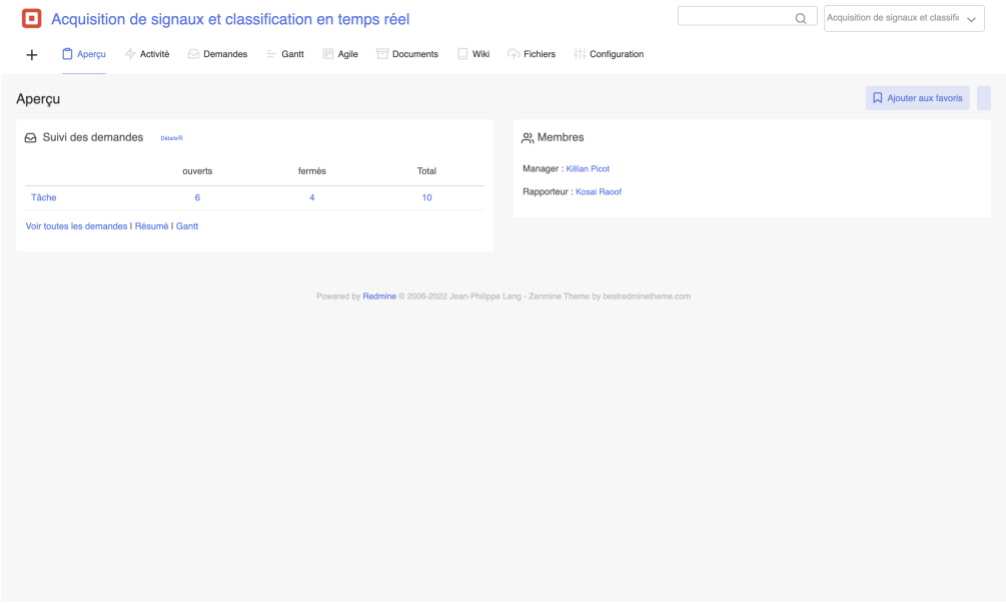


Figure 1 Page d'accueil du projet

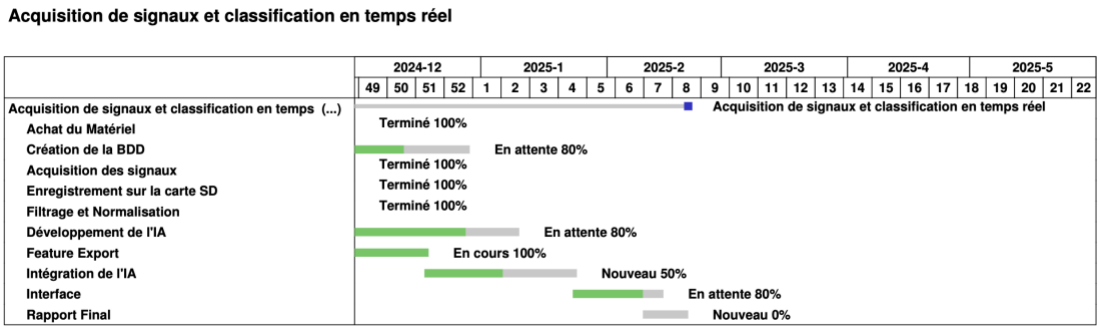


Figure 2 Gantt du 03/12/24

Jusqu'à présent, on remarque sur le Gantt (*figure 2)* que le projet avance plus rapidement que ce qui a été initialement prévu. Il n'y a donc aucun problème de retard, les délais sont pour le moment respectés.

2) Présentation d'outils utiles

GitHub

Afin de partager le code et de le versionner, j'utilise le site GitHub. Celui-ci est largement répandu dans le milieu de la programmation car il intègre le protocole git, permettant le versionnage de code ou encore la collaboration à l'écriture d'un même programme entre plusieurs personnes. Il permet également la création de plusieurs versions de code nommé « branche », chaque branche sera issue d'une autre branche ou de la version principale du programme (branche principale) et pourra alors être fusionnée par la suite avec une autre branche. Cela permet de segmenter le programme en plusieurs fonctionnalités ou chaque fonctionnalité entrainera la création d'une nouvelle branche du programme. Git permet, de plus, de passer rapidement et facilement entre toutes ces différentes versions du code. J'ai donc implémenté cette architecture dans le projet, afin de pouvoir développer le plus rapidement possible toutes les fonctionnalités. Ces-dernières sont ainsi testées indépendamment dans le but de s'assurer de leur bon fonctionnement. Quand l'ensemble des fonctionnalités majeures seront finies, elles seront alors fusionnées dans la branche principale, créant la première version fonctionnelle du programme. Il existe une version entièrement libre de GitHub nommée GitLab, dont une instance est hébergée en interne à l'université. Cependant, j'ai décidé de ne pas l'utiliser par soucis de simplicité mais également car je souhaite pouvoir récupérer ce code dans le futur et le partager au plus grand nombre.

Visual Studio Code

VS Code est un IDE open source et très versatile. Il permet d'écrire de nombreux langages de programmations différents comme le C/C++ ou encore Python, qui sont les deux langages utilisés dans ce projet. Il dispose aussi de nombreux plugins créés par la communauté et installables facilement. Ces plugins permettent d'étendre les fonctionnalités de l'IDE, comme l'ajout de débogage, de compilation de code ou encore de téléversement. Il s'agit d'un des outils de programmation les plus répandus car une seule interface permet de programmer des langages complétements différents. Ainsi, les développeurs ne sont pas perdus entre le passage d'un langage à un autre. De plus, VS code intègre une gestion native de Git permettant de créer des branches ou de mettre à jour du code via son interface graphique et non via le terminal. Pour ce projet, les extensions nécessaires sont Python (Python - Visual Studio Marketplace, s.d.), STM32 for VSCode (stm32-for-vscode - Visual Studio Marketplace, s.d.), C/C++ (C/C++ - Visual Studio Marketplace, s.d.). A l'aide de ces 3 extensions, il est possible d'écrire du code

Python et C/C++ ainsi que de débuguer/compiler/téléverser un programme créé à l'aide de STM32CubeMX. La compilation utilise GCC et Make tandis que le téléversement/débogage utilise OpenOCD.

STM32CubeMX

Il s'agit d'un logiciel développé par la société STMicroelectronics qui permet de programmer et d'utiliser plus facilement leur MCU. Après avoir sélectionné notre MCU, une interface graphique nous permet de choisir les périphériques internes/GPIO du MCU que l'on souhaite configurer et comment l'on souhaite le configurer. Par la suite, le logiciel générera automatiquement les codes associés à la configuration (*figure 3).* Il y a néanmoins certaines précautions à prendre lorsque l'on veut regénérer le code mais que ce dernier a déjà été modifié au préalable. Pour éviter que les changements utilisateurs soit écrasés, il est nécessaire de les implémenter dans les balises indiquées.

Une certaine configuration du projet CubeMX a été nécessaire au préalable. Dans l'onglet « Project Manager », il faut définir le champ « Toolchain / IDE » sur « Makefile ». Ainsi le code généré via CubeMX pourra être compilé via le plugin VSCode vue précédemment.

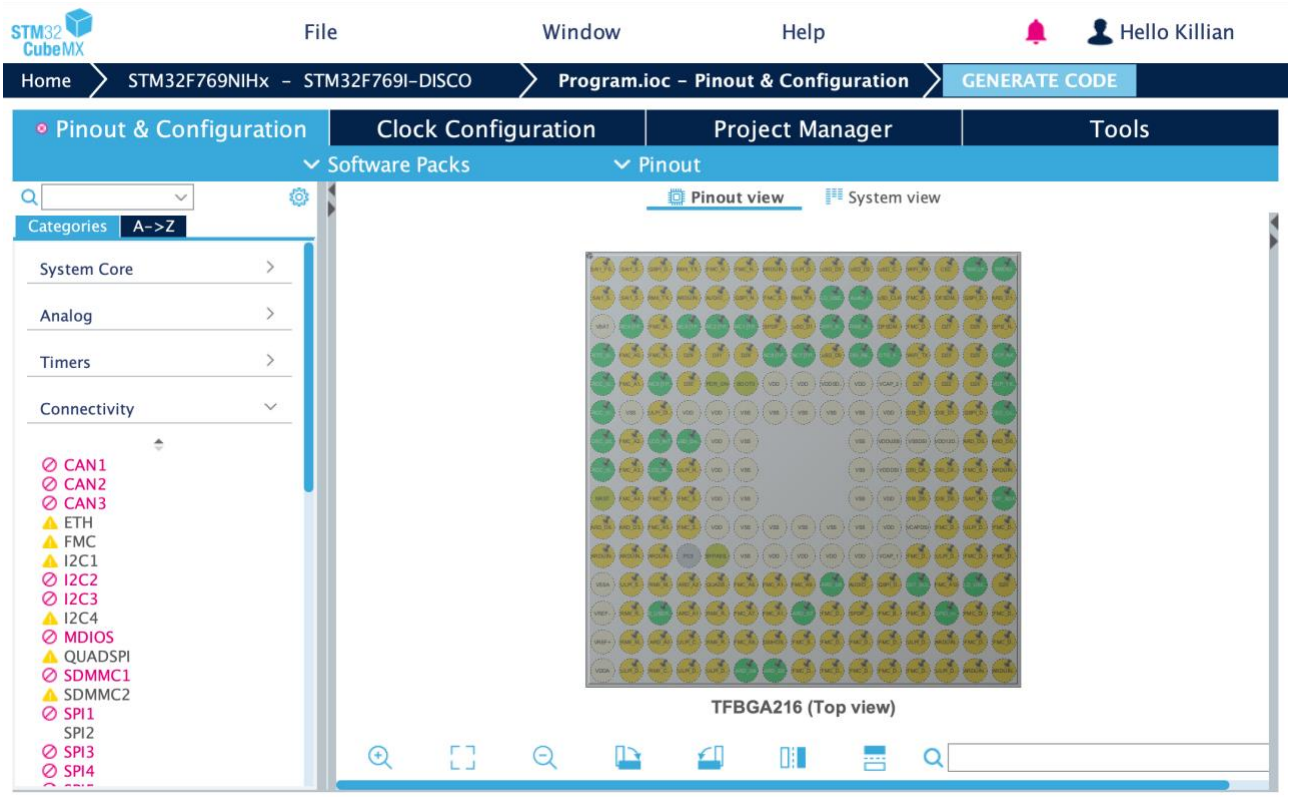


Figure 3 Logiciel STM32CubeMX

IV. Implémentations

1) Le matériel

Le choix du matériel est important bien que peu nombreux pour ce projet. Il est nécessaire de choisir une carte assez puissante pour que l'ensemble du traitement soit effectué en temps réel. De plus, sa capacitée en mémoire RAM et ROM doit être assez importante afin de permettre l'accueille d'un réseau neuronique. Le choix d'une carte embarquée de chez STMicroelectronics semble donc être l'une des meilleures solutions. Effectivement, certaines de leurs cartes disposent d'un MCU mais aussi de composants externes comme un CODEC, écran, microphones directement intégrés sur la carte. Cela permet donc de limiter fortement le risque d'erreurs au développement lié par exemple à une mauvaise connexion ou de mauvais branchement. De plus, STMicroelectronics propose un outil pour accueillir des réseaux neuroniques pré-entrainés sur certain de leur MCU ARM STM32. C'est donc tout naturellement de par mes compétences acquises sur ces systèmes et par ses avantages que je me suis tourné vers une solution STM32. Le choix a été fait sur la carte **STM32F769I-DISCO** qui dispose de l'ensemble des ressources nécessaire aux besoins du projet. Cette carte dispose : de 4 microphones digitaux ST Mems permettant l'acquisition des signaux audios, d'un écran IPS tactile pour la réalisation d'une application intuitive afin de simplifier l'utilisation du système, un port de carte µSD pour enregistrer les sons et traiter des données et enfin d'un CODEC avec une sortie audio afin d'écouter les sons captés par les microphones. D'autres périphériques externes sont présents sur cette carte mais ils ne sont pas utilisés dans ce projet, vous trouverez une liste exhaustive sur le site de STMicroelectronics (32F769IDISCOVERY - Discovery kit with STM32F769NI MCU - STMicroelectronics, s.d.) .

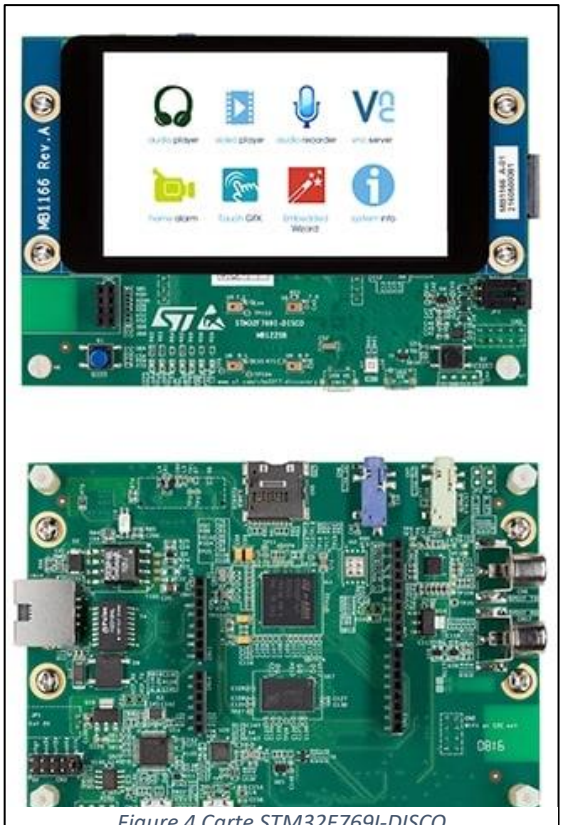


Figure 4 Carte STM32F769I-DISCO

Deux schémas de connections relatifs à la partie audio de la carte sont également disponibles en Annexe 2 : Schéma STM32F769I-DISCO. Le MCU est un STM32F7

avec un cœur ARM Cortex-M7 cadencé à 200MHz répondant au besoin de performance de l'application.

2) La programmation embarquée

La programmation a été séparée en plusieurs parties, toutes disponibles sur le GitHub du projet (Projet_5A, s.d.). Un lien vers le code spécifique à chaque fonctionnalité sera défini au début de chaque paragraphe explicatif.

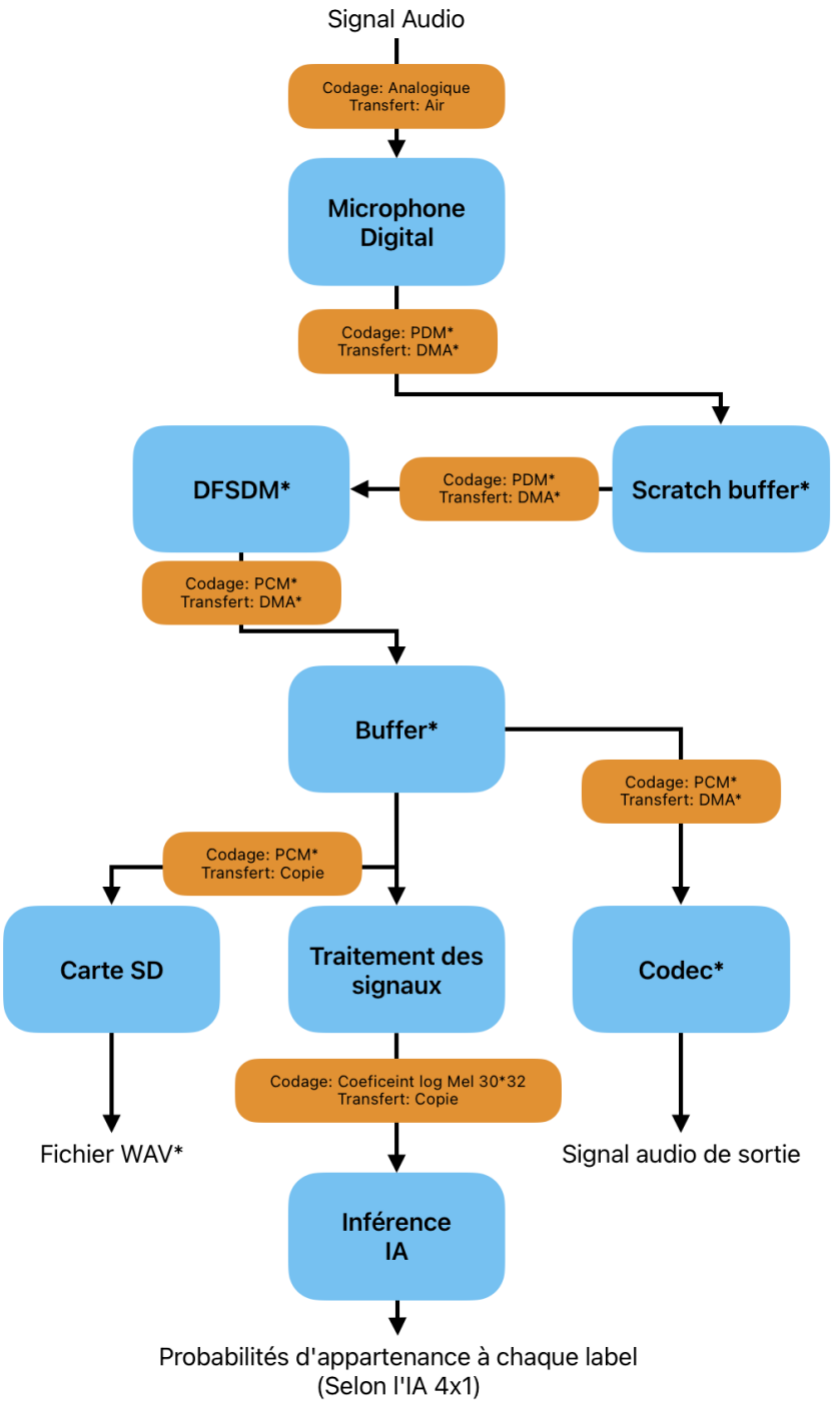


Figure 5 Schéma fonctionel de l'application

On remarque sur le schéma (*figure 5)* que l'application doit permettre de faire plusieurs actions en même temps : écrire un fichier audio sur une carte SD, envoyer les données acquises au CODEC pour sa lecture, traiter les signaux et inférer l'IA dessus. Il est donc nécessaire d'optimiser au maximum le temps CPU car l'ensemble de ses actions doivent se faire en parallèles. C'est pour cela que l'utilisation du DMA

est privilégiée, quand c'est possible. Il permet de transférer des données d'un emplacement mémoire à un autre, sans que le CPU n'intervienne dedans.

Afin de faire fonctionner les différents codes, il a été nécessaire d'importer deux bibliothèques :

- La bibliothèque BSP de la carte produit par STMicroelectronics (BSP, s.d.) qui propose un ensemble de fonctions permettant de faire fonctionner les périphériques disponibles sur la carte.
- La bibliothèque CMSIS-DSP produit par ARM (CMSIS-DSP, s.d.) qui contient des fonctions utiles pour le traitement de signal (FFT, DCT, etc…).

Écran LCD et interface

https://github.com/kiki442002/STM32_embeded_audio_classifier/tree/interface

Le code de l'écran LCD fonctionne avec les périphériques internes LTDC et DSI du MCU. Le DSI est une interface série utilisé pour une communication rapide entre écran et le MCU, il permet notamment l'envoie de données vidéo mais également des commandes de contrôles à l'écran. Le DSI n'est pas utilisé en tant que tel par le CPU dans le code. En effet, un autre composant se charge d'envoyer directement les données sur le bus DSI. Il s'agit du LTDC qui est un contrôleur d'affichage pour les écrans LCD-TFT. Il permet des interfaces accueillants des fonctionnalités avancées comme la gestion des couches ou de la transparence. Le calcul de ces fonctionnalités est fait par le contrôleur qui envoie le résultat à l'écran via le DSI. Cela permet d'optimiser le temps de calcul. L'ensemble de l'initialisation de ces périphériques se fait lors de l'appel à la fonction « *BSP_LCD_Init()* ». Par la suite, j'utilise plusieurs fonctions définies dans la BSP pour créer l'interface, les fonctions liées à l'interface sont disponibles dans le fichier « *screen.c* ».

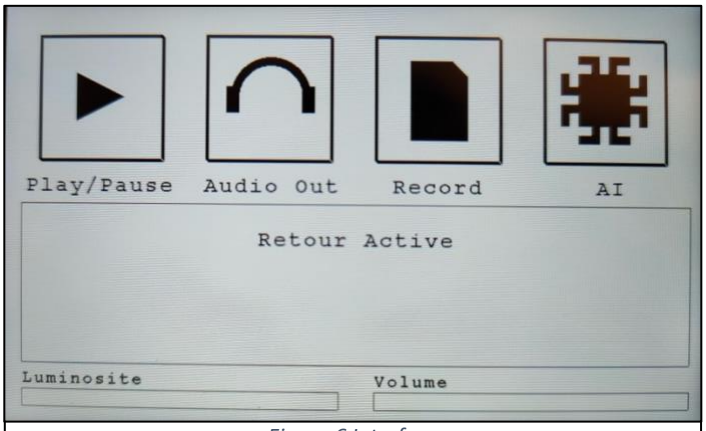


Figure 6 Interface

Il s'agit d'un écran tactile, comme on peut le voir l'interface a été réalisée pour utiliser cette capacitée (*figure 6*). La communication entre la puce tactile de l'écran et le MCU se fait via I2C initialisé dans la fonction *« BSP_TS_Init() ».* Il y a également la GPIO 13 qui est reliée entre le contrôleur tactile et le MCU. Ce pin permet au contrôleur d'avertir quand une pression a été exercée sur l'écran afin

d'engendrer une interruption cotée MCU et d'agir en conséquence. Pour configurer l'interruption une autre fonction est utilisée. Il s'agit de la fonction « *BSP_TS_ITConfig()* ». Cette fonction active l'interruption sur la broche 13 mais pour que le tout soit fonctionnel, il faut ajouter la fonction d'interruption dans le fichier « stm32f7xx_it.c ». Cette fonction doit renvoyer vers la fonction « *HAL_GPIO_EXTI_IRQHandler(TS_INT_PIN)* » avec la bonne GPIO en argument. Cela permet au système de supprimer le drapeau d'interruption pour en autoriser de nouvelles. Sans l'ajout de ces fonctions, le programme restera bloqué dans l'interruption. La fonction « *HAL_GPIO_EXTI_IRQHandler* » appelle elle-même la fonction « *HAL_GPIO_EXTI_Callback(GPIO)* ». C'est dans cette dernière fonction que le code lié à l'interruption doit être écrit. Cependant, cette fonction est appelée pour n'importe quelle interruption sur les GPIO. Il est donc nécessaire de filtrer les actions en fonction du GPIO qui a émis l'interruption. Dans notre cas, le code allume ou éteint une LED si l'on appui sur le coin supérieur gauche de l'écran. Cela permet de tester le fonctionnement du tactile de l'écran. Il sera nécessaire par la suite de rendre exploitable le tactile pour les différentes actions de l'interface.

Écriture sur carte SD

https://github.com/kiki442002/STM32_embeded_audio_classifier/tree/carte_SD

Une mémoire classique (carte SD, clé USB, etc…) dispose d'une table d'allocation des fichiers. Cette table a pour but d'indiquer dans quel espace de la mémoire les fichiers sont enregistrés, leur taille, ainsi que leur nom et leur extension. Pour cette table, tous les types de fichiers sont identiques. C'est-à-dire, que si je suis capable de créer un fichier texte, alors je pourrais produire des fichiers audio WAV, des exécutables EXE etc… Les différences entre les fichiers résident principalement dans leur entête et dans la manière dont les données sont agencées dans la mémoire. Dans ce projet, la carte SD utilise une table d'allocation de type FAT32. Dans cette table, 28 bits sont utilisés pour l'adressage de la mémoire. C'est-à-dire que la mémoire est divisée maximum en 2^{28} = 268 435 456 unités, chaque unité est appelée cluster et correspond à un nombre d'octets défini par la taille de la mémoire divisée par le nombre de cluster. La carte SD utilisée pour ce projet fait

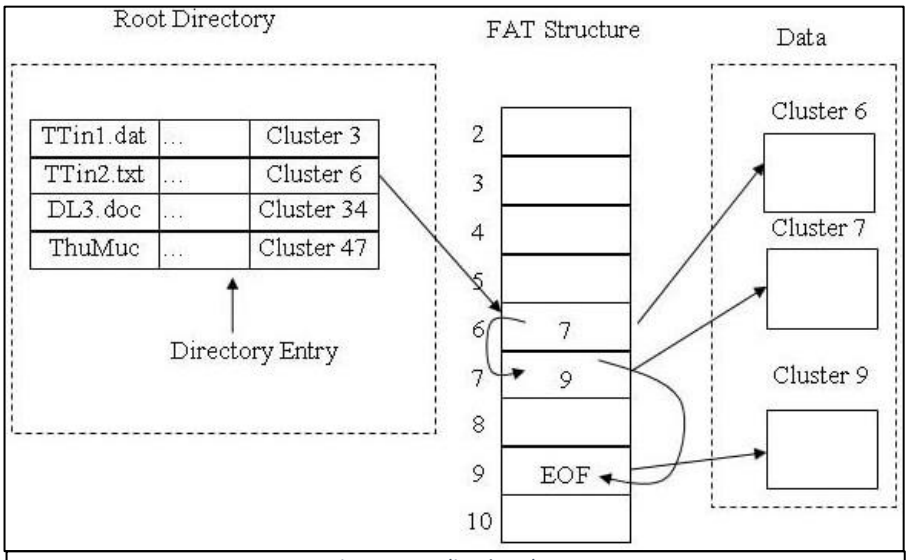


Figure 7 Explication de FAT

8Go soit : $8*1024^3$ = 8 589 934 592 octets donc la taille d'un cluster correspond à 8 589 934 592 / 268 435 456 = 32 octets = 256 bits si l'on considère le nombre maximal de clusters admissibles par le FAT32. Plus la taille de la carte SD augmentera, plus la taille d'un cluster augmentera aussi. Le cluster devient alors le plus petit élément de mémoire dans la partition. Cela veut dire que si je veux écrire un fichier de 8 octets, j'allouerais pour cela 32 octets. On remarque que ce système n'est pas optimisé pour enregistrer des petits fichiers de quelques octets. La FAT structure regroupe toutes les informations des clusters. Si un cluster est vide sa valeur sera nul et si un cluster est utilisé, il pointera vers le prochain cluster du fichier, jusqu'à ce qu'il arrive à la fin du fichier. Tous les fichiers et sous-dossiers sont regroupés dans le dossier racine. Chaque fichier/dossier aura quelques informations comme le nom du fichier, l'extension, sa taille et surtout son cluster d'entrée. Un sous-dossier sera similaire au fonctionnement du dossier racine.

Pour utiliser des tables d'allocation FAT, STMicroelectronics propose un middleware FATFS, afin de permettre l'écriture et la lecture des fichiers présents dans la structure. Cependant à ce middleware, il est nécessaire d'y ajouter un driver afin de faire fonctionner la carte SD. Le driver doit permettre de réaliser des fonctions de bases comme écrire une donnée ou la lire, initialiser la carte SD, détecter sa présence etc... Il doit être écrit dans les fichiers « *bsp_driver_sd.h* » et « *bsp_driver_sd.c* ». La bibliothèque BSP de la carte dispose déjà d'un driver avec les fonctions associées dans le fichier « *stm32f769i_discovery_sd.c* ». J'ai donc repris en grande partie ces fonctions que j'ai modifiées pour qu'elles correspondent à mon besoin. Il est également nécessaire de vérifier que l'ensemble des fonctions nécessaire à FATFS ont bien été écrites. Pour ce faire, le fichier « *sd_diskio.c* » contient toutes les fonctions drivers nécessaires. Une fois ceci effectué, il est possible d'utiliser l'outils FATFS.

Pour utiliser FATFS, il faut tout d'abord vérifier que la configuration de celui-ci dans le fichier d'entête « *ffconf.h* » est correcte et correspond à nos besoins. Ce fichier permet par exemple d'initialiser ou non certaines fonctions plus haut-niveau de FATFS ou encore d'indiquer et de paramétrer le type de table d'allocation. Après cela, il faut lier le driver créé précédemment avec FATFS. En effet, FATFS a été fait pour n'importe quel type de carte ou périphérique. L'initialisation du lien avec le driver permet de définir les fonctions qui vont être utilisés par FATFS grâce au driver créé précédemment. Par la suite, nous pouvons utiliser les fonctions « *f_mount()* » pour monter le système de fichier, « *f_open()* » pour ouvrir et/ou créer un fichier, « *f_write()* » pour écrire dans un fichier, « *f_close()* » pour fermer un fichier. Je peux donc ouvrir un fichier, lui donner un nom et une extension en .wav, écrire l'entête (*figure 8)* et les données captées par les microphones. Cela donnera un fichier écoutable via n'importe quel lecteur audio.

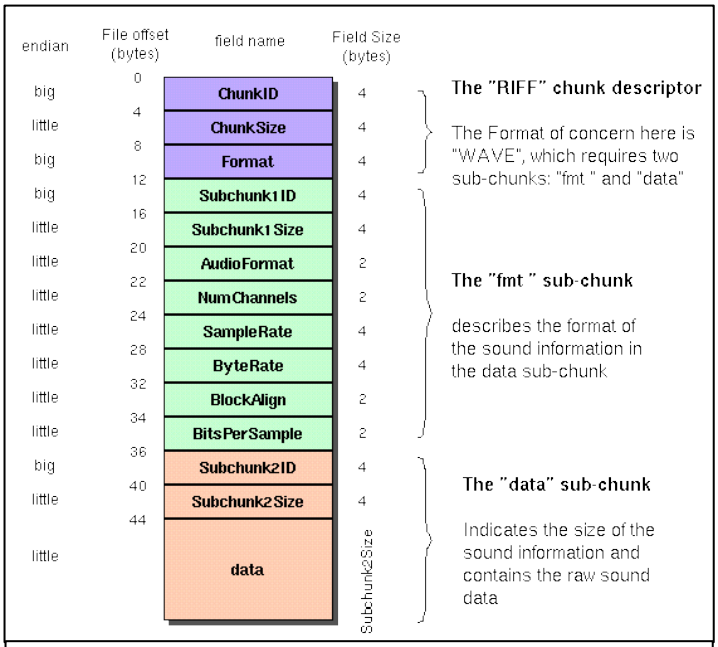


Figure 8 Description de l'entête WAV

Acquisitions des signaux sonores

https://github.com/kiki442002/STM32_embeded_audio_classifier/tree/filtrage_feature

Les microphones utilisés sont des microphones digitaux ST Mems MP34DT01 (Annexe 2). Les microphones digitaux ont la particularité d'émettre des données sous un format numérique et non analogique. Ce type de codage est appelé PDM, pour modulation en densité d'impulsion. Ce codage est seulement sur un bit et se base sur les échantillons

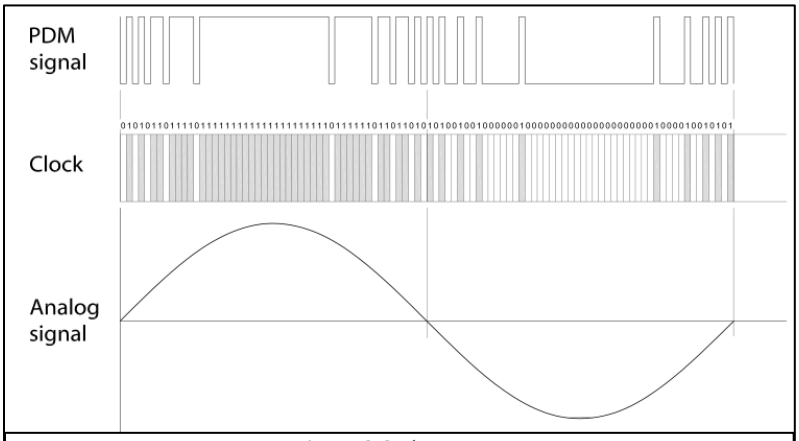


Figure 9 Codage PDM

précédents. Pour ce faire, une comparaison est réalisée avec le bit précédent tous les x temps définis par un signal d'horloge. Si le bit est à l'état haut, la quantification devra être augmentée d'une, si elle est à l'état bas, elle devra être diminuée (*figure 9)*. Cela permet de gagner en termes de bande passante par rapport à un système classique PCM codé sur plusieurs bits. Le codage PCM est simplement le fait de récupérer la valeur échantillonnée tous les x temps codé sur y bits (*figure 10)*. Bien qu'il y ait certains avantages au codage PDM comme la rapidité de transmission, cela rend plus compliqué le décodage des trames pour passer d'un codage PDM à PCM. Sans démodulateur adapté, cette démodulation prend beaucoup de temps de calcul car tous les échantillons sont liés les uns aux autres. La démodulation s'effectue via l'application d'un filtre passe bas numérique avec de nombreux coefficients d'où sa complexité. Le MCU STM32F769 dispose néanmoins d'un démodulateur DFSDM permettant de passer efficacement d'une valeur PDM à PCM. Pour utiliser ce démodulateur, les données d'entrées des microphones sont distribuées dans un buffer temporaire via un DMA, une fois le buffer plein le DMA charge les données dans le DFSDM qui retourne alors les données PDM dans le buffer final via un autre DMA (*figure 5*). Ainsi, aucun temps processeur est nécessaire à l'acquisition de ces signaux, ce qui permet d'optimiser l'application au maximum. Lorsque le DFSDM a rempli le buffer final de moitié, ou entièrement, à l'aide du DMA, une interruption est envoyée au près du CPU qui peut alors traiter les données, seulement quand elles sont disponibles. L'ensemble des fonctions de configuration et d'acquisition a été récupéré via la librairie BSP dans le fichier « *stm32f769i_discovery_audio.c* », dans laquelle une documentation d'utilisation est indiquée. Les fonctions sont sous l'appellation *BSP_AUDIO_IN.* Tous les DMA sont configurés en mode cyclique. Cela veut dire que lorsque le DMA arrive au bout du buffer, il n'attend pas la fin de l'interruption et revient directement au début. Il faut donc veiller à ce que le CPU est le temps de gérer l'ensemble des données avant que le DMA ne les écrase. Les données acquises sont codées en 16bits avec un échantillonnage à 16kHz, j'utilise les données de deux microphones afin de redistribuer un son stéréo. Néanmoins, seules les données d'un micro seront exploitées pour l'extractions des fonctionnalités pour l'IA.

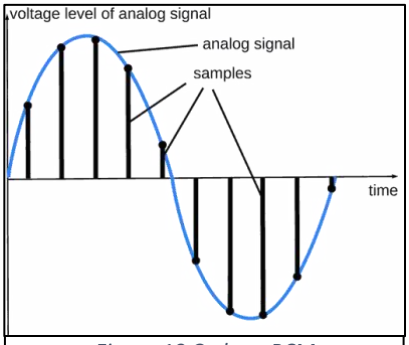


Figure 10 Codage PCM

Émissions des signaux sonores

https://github.com/kiki442002/STM32_embeded_audio_classifier/tree/filtrage_feature

Pour écouter les signaux sonores capturés via les microphones ou tout autres types de données, il est nécessaire de transmettre un signal sonore amplifier via la sortie casque (Line Output) de la carte. Pour ce faire, nous pouvons voir en annexe 2 que la sortie est reliée à un CODEC. Le CODEC est un dispositif qui permet soit de de transformer le signal audio analogique en signal numérique, soit de faire l'inverse c'est-à dire de transformer un signal numérique en signal analogique. Le CODEC va permettre également de traiter les signaux pour les amplifier, ou d'y appliquer des effets. La communication entre le CODEC et le MCU se fait via plusieurs moyens. Les commandes de configurations et de gestion du CODEC sont envoyées en I2C, alors que les données audios sont transférées via un SAI. Le SAI est un dispositif présent dans le MCU afin de l'interfacer avec des systèmes audios comme un CODEC. Il permet le transfert de données audios numériques. Il prend en charge plusieurs formats audios, plusieurs protocoles de communication et peut fonctionner en maitre ou esclave. Dans notre cas, le SAI transmet les données au CODEC via un bus série I2S. Ces données sont codées sur 16 bits avec une fréquence d'échantillonnage de 16kHz. Les données envoyées sont en stéréo. Pour ce faire, la donnée du 1er canal est envoyée puis celle du second et ainsi de suite. De la même manière que précédemment, le DMA se charge d'envoyer les données au SAI de manière cyclique. Les données des microphones sont donc écrites dans la mémoire par un DMA, et en même temps un autre DMA les récupère pour les envoyer au SAI. L'ensemble des fonctions nécessaires pour l'écoute des données audios sont présentes dans le fichier « *stm32f769i_discovery_audio.c* » sous le nom de *BSP_AUDIO_OUT.*

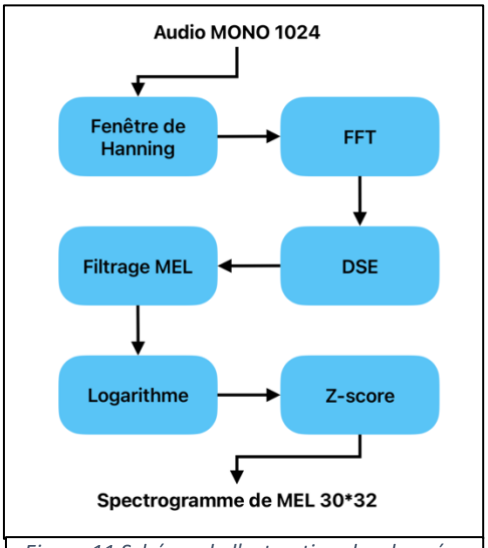


Figure 11 Schéma de l'extraction des données

Le traitement des signaux pour le réseau neuronique suit la procédure introduite dans la thèse dans la partie implémentation (Ahmed, 2023) (*figure 11*). Pour ce faire les données sont acquises selon une fenêtre de 1024 échantillons en mono avec un recouvrement de 50% soit de 512 échantillons. Il y aura donc 512 nouveaux échantillons pour chaque nouvelle fenêtre, hormis la première. Ces échantillons sont par la suite multipliés par les coefficients de Hanning :

$$h(t) = \begin{cases} \frac{1}{2} - \frac{1}{2}\cos(2\pi\frac{t}{T}) & \text{si } t \in [0, T] \\ 0 & \text{sinon.} \end{cases}$$

La fenêtre de Hanning permet d'éviter les effets de bords et de discontinuité lors du calcul de la transformée de Fourier dans le domaine fréquentiel. Une observation de l'application d'une fenêtre sur une sinusoïde a 1kHz peut être visualisée en *figure 12*.

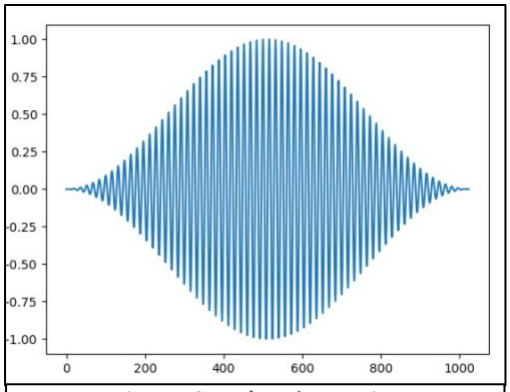


Figure 12 Fenêtre de Hanning

Une fois l'application de la fenêtre de Hanning effectuée, il faut calculer la FFT. Les signaux sont réels et non complexes, on sait donc que la FFT sera symétrique selon la fréquence nulle. J'ai donc décidé de calculer la RFTT qui est une optimisation de la FFT classique en prenant en compte les optimisations possible grâce aux signaux réels. Pour ce faire, j'ai importé la bibliothèque CMSIS-DSP de ARM. Cette bibliothèque possède une fonction permettant de calculer la FFT sur des signaux réels codés en float32 nommés « *arm_rfft_fast_f32* ». Cette fonction prend en paramètre une structure qui peut être initialisée via la fonction « *arm_rfft_fast_init_f32* » avec la taille du signal d'entrée, mais aussi les buffers d'entrée et de sortie des données. Cette fonction retourne 513 nombres complexes selon un codage particulier. En effet, les nombres sont codés à l'aide de deux flottants, le premier pour la partie réelle et le second pour la partie imaginaire. Cependant, les deux premières

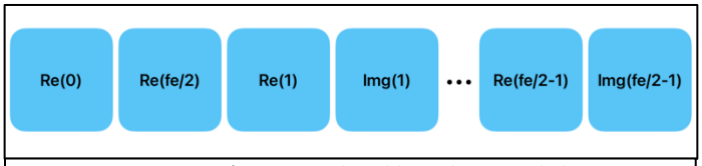


Figure 13 Représentation du tableau de sortie de la FFT

cases de ce tableau de sortie, réservées normalement pour le complexe à la fréquence nulle sont utilisées pour indiquer la valeur réelle de la fréquence nulle et de $f_e/2$ (*figure 13*). En effet, si le signal est réel alors pour ces deux fréquences la partie imaginaire sera nulle. Cela permet de trouver en sortie un tableau équivalent en taille au tableau d'entrée pour plus de commodité.

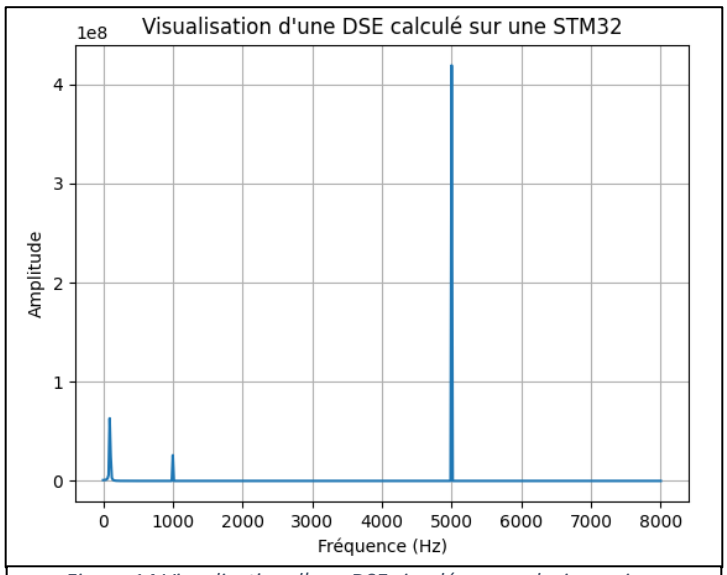


Figure 14 Visualisation d'une DSE simulée avec plusieurs sinus

Le calcul de la densité spectrale d'énergie s'effectue en calculant le module au carré des nombres complexes. Une fonction de la librairie CMSIS-DSP permet d'effectuer cela « *arm_cmplx_mag_squared_f32* ». Cependant, cette fonction n'est pas adaptée au tableau de sortie de la FFT. En effet, les données des complexes doivent être arrangées de la manière suivante partie réelle puis partie imaginaire. Il faut donc omettre les deux premières cases du tableau et diminuer la taille de nombre de complexes pour qu'elle corresponde à 511. Il est ensuite possible de calculer les modules aux carrées des complexes via cette fonction. Pour la fréquence nulle et $f_e/2$, il suffit de multiplier ces valeurs par elle-même afin de les élever aux carrés pour obtenir leur module au carré. En plaçant la donnée de la fréquence nulle au début et celle de $f_e/2$ à la fin, nous obtenons le tableau de la DSE final (*figure 14*). Normalement, il est nécessaire de calculer la densité spectrale de puissance pour y calculer les coefficients Mel par la suite. Cependant pour calculer la DSP, il est nécessaire de diviser la DSE par le temps d'échantillonnage afin de normaliser les données et obtenir une amplitude en watt. Dans notre cas, nous diviserons plus tard la DSE par le nombre d'échantillons pour obtenir une quantité d'énergie par échantillon ce qui est relativement similaire mais indépendant de la fréquence d'échantillonnage, permettant d'obtenir tout de même une DSP.

Le calcul des coefficients Mel s'effectue par filtrage du signal. Le but de ces coefficients est de simuler la représentation des signaux audios chez l'homme. Pour ce faire, les filtres Mel de forme triangulaire, ajustent leur taille, ils sont plus petits dans les basses fréquences mais plus grands dans les hautes (*figure 15*). En effet, l'oreille humaine est plus sensible aux basses qu'aux aigues. Le coefficient Mel représente l'énergie restante dans le signal après la filtration. Il faut donc sommer le produit des coefficients des filtres de Mel par ceux du signal pour obtenir les coefficients de Mel. Dans notre cas, nous utilisons 30 filtres ce qui permet d'obtenir 30 coefficients. Les filtres sont normalisés pour que leur somme soit équivalente à 1 et chaque coefficient est divisé par 1024 la taille de notre FFT afin de permettre d'effectuer le calcul des filtres de Mel sur la DSE et non la DSP. Cela permet d'économiser en temps de calcul car les filtres sont stockés dans la mémoire. Comme nous pouvons le voir sur la *figure 15*, la majorité des coefficients des filtres sont nuls. Ainsi, pour éviter des multiplications par 0, des sommes inutiles et un stockage de données supplémentaires, seuls les données non nulles sont stockées. Les filtres sont stockés à l'aide de 3 tableaux différents. Le premier correspond à l'ensemble des données non nulles des filtres mis à la suite en float32, le second aux nombres de zéros avant le début de chaque filtre en int16 et le dernier au nombre de coefficients par filtre en uint8. Cette technique permet d'économiser plus de 116ko ((1024*30*32bits - 968*32bits - 30*16bits – 30*8bits) /8 /1024) de mémoires ROM, par rapport au stockage de tous les coefficients des filtres en float32. Cela permet par ailleurs d'économiser 29 752 multiplications et sommes (1024*30-968). Il est important de prendre en compte cela pour optimiser au maximum l'application en termes de temps de calcul et de mémoire. Cela permet d'effectuer ces calculs en temps réel avant que le DMA n'écrase les données écrites précédemment. Les coefficients des filtres ont été récupérés via la bibliothèque librosa de Python et l'ensemble des tableaux sont situés dans le fichier « *mel_filters.h* » créé à partir d'un script Python présent dans le fichier

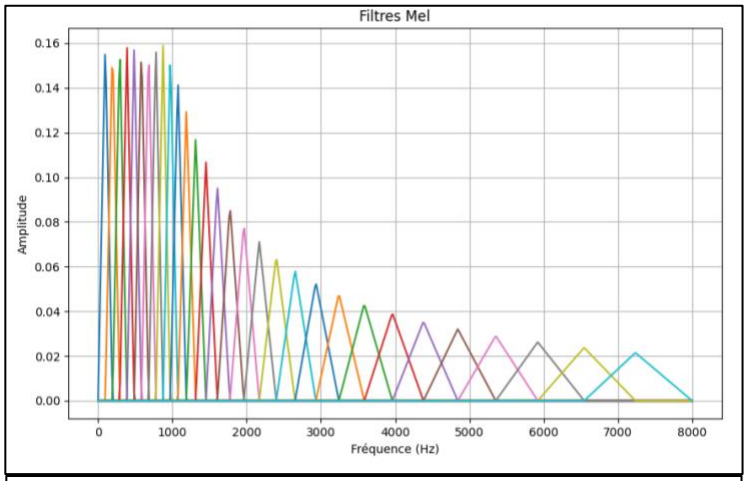


Figure 15 Banque de filtres de Mel

« */Support/mel_coef.py* » du répertoire Git du projet. Une fois les coefficients de Mel récupérés, j'applique la fonction logarithme dessus pour récupérer les log Mel.

Il ne reste plus qu'à normaliser les données pour que le modèle d'IA les comprenne mieux et améliore ses performances. La normalisation utilise le Z-Score. Cette standardisation correspond à centrer réduire l'ensemble des 30 coefficients. Pour ce faire, sur chaque jeu de coefficients d'une fenêtre (30), on calcul sa moyenne(µ) et son écart-type(Ò) et l'on applique la formule suivante $Mel\_Z\text{-}score_i = (logMel_i – µ) / Ò$.

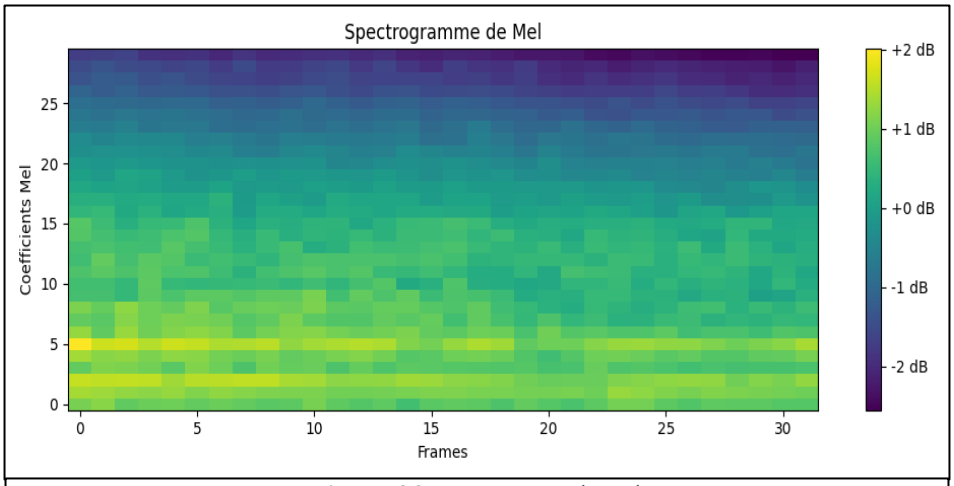


Figure 16 Spectrogramme de Mel

Au final, une fois le calcul des coefficients faits pour les 32 fenêtres, on obtient un spectrogramme de Mel que l'on peut visualiser (*figure 16*).

3) Le réseau neuronique

https://github.com/kiki442002/IA_embeded_audio_classifier/tree/main

Dans un premier temps, pour entrainer l'IA, Il est nécessaire de gérer et comprendre la base de données. Pour ce faire, j'ai pu récupérer la BDD utilisée lors de la thèse (Ahmed, 2023). Cette BDD était déjà labélisée et séparée en entrainement, test dans des fichiers CSV. J'ai néanmoins décidé de modifier cette séparation. En effet, l'ensemble des données audio de la base de données sont des fichiers de 10 secondes or le processus de l'IA prend en réalité qu'une seconde de données avant d'inférer. J'ai donc séparé les échantillons de la base de données en quatre sous-échantillons de deux secondes. J'ai omis la première et la dernière seconde de l'échantillon pour limiter les effets de bords. Le choix de deux secondes me permettrait d'augmenter la base de temps que l'IA a par inférence si nécessaire. Au final, j'obtiens une BDD de 12 000 échantillons répartie équitablement en 4 labels : bruit de pluie, de vent, de pas et de voiture. La base de données est séparée en 80% pour l'entrainement et 20% pour le test. Cependant, suite à une discussion auprès de notre professeur en IA, il serait préférable de la séparer en 3 : 80% pour

l'entrainement, 6.66% pour le développement et 13.33% pour le test. La partie développement me permettra de tester l'IA après chaque epoch pour déterminer si elle est meilleure ou non que la précédente epoch.

L'extraction des données a été fait de manière analogue à celui réaliser sur la carte embarquée, à l'exception de la FFT. Pour éviter tout risque lié à ce problème, je vais remplacer la méthode actuelle utilisant numpy par celle de la bibliothèque CMSIS-DSP pour python. Cela devrait permettre d'avoir des résultats proches de ceux qui devrait être calculés par la carte STM32 et ainsi obtenir une meilleure précision sur celle-ci.

L'IA est réalisée à l'aide de la bibliothèque pytorch pour python. Le réseau neuronique comporte 38 142 paramètres et est composé des couches suivantes :

- Conv1 :
  in = 1, out = 4, kernel = 3x3
  MaxPool2d : kernel = 2x2, stride = 2
- Conv2 :
  in = 4, out = 16, kernel = 3x3
  MaxPool2d : kernel = 2x2, stride = 2
- Linear1 :
  in = 896, out = 40
- Linear2 :
  in = 40, out = 32
- Linear3 :
  in = 32, out = 8
- Linear4 :
  in = 8, out = 4

On remarque qui s'agit d'un modèle avec un nombre limité de paramètres pour qu'il puisse facilement être accueilli au sein de la carte STM32.

Il faut maintenant finir l'élaboration du modèle, qui devra être convertit par la suite au format ONXX pour qu'il puisse être intégré sur la carte à l'aide de STM32CubeIA. Cependant, quelques métriques ont déjà pu être réalisées, sur le modèle actuel sans pour autant être confirmées. Sa performance a atteint environ 90%, et le temps d'itération sur la carte STM32 est d'environ 9ms. Ceux qui est largement acceptable puisque le modèle effectue au minimum une inférence toutes les secondes (ST Edge AI Developer Cloud, s.d.).

V. Conclusion

Pour conclure, ce projet semble être sur la bonne voie pour fournir son premier livrable. Bien que certaines difficultés aient pu être rencontrées sur l'avancement du projet. Celles-ci n'ont que très peu impacté le temps imparti à chaque tâche puisque le Gantt nous permet d'observer une avance claire. Il reste cependant, à vérifier que l'extraction des données sur la carte STM32 correspond à celle utilisée par python, avec par exemple un jeu de données de test connu à l'avance et tester sur les deux méthodes pour évaluer les différences. De plus, malgré l'ensemble des fonctions réalisées pour l'entrainement et la création de l'IA, il est nécessaire d'améliorer certain point pour s'assurer de l'optimalité des méthodes employées. Enfin, il sera nécessaire de comprendre la documentation et l'utilisation de STM32CubeIA afin de réussir à inférer le modèle d'IA sur la carte. Ce projet permet de refléter un ensemble diversifié de compétences enseignées à l'ENSIM que ce soit de l'embarqué, de l'IA ou encore du traitement de signal.

VI. Bibliographie

32F769IDISCOVERY - Discovery kit with STM32F769NI MCU - STMicroelectronics. (s.d.). Récupéré sur https://www.st.com/en/evaluation-tools/32f769idiscovery.html

Ahmed, A. (2023). *Multi-purpose acoustic sensor for smart home.* Le Mans: LE MANS UNIVERSITÉ.

BSP. (s.d.). Récupéré sur GitHub: https://github.com/STMicroelectronics/32f769idiscovery-bsp?tab=BSD-3-Clause-1-ov-file

C/C++ - Visual Studio Marketplace. (s.d.). Récupéré sur https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools

CMSIS-DSP. (s.d.). Récupéré sur GitHub: https://github.com/ARM-software/CMSIS-DSP

Projet_5A. (s.d.). Récupéré sur Github: https://github.com/kiki442002/Projet_5A

Python - Visual Studio Marketplace. (s.d.). Récupéré sur https://marketplace.visualstudio.com/items?itemName=ms-python.python

ST Edge AI Developer Cloud. (s.d.). Récupéré sur https://stm32ai-cs.st.com/home

stm32-for-vscode - Visual Studio Marketplace. (s.d.). Récupéré sur https://marketplace.visualstudio.com/items?itemName=bmd.stm32-for-vscode

VII. Annexes

1) Annexe 1 : Cahier des charges

1. Contexte

Ce projet de fin d'études vise à développer un système capable d'acquérir et de traiter des signaux audios en temps réel sur une carte embarqué. Les données seront analysées par une intelligence artificielle (IA) préalablement entraînée et implémentée sur la carte. Ce projet permettra de démontrer la faisabilité d'intégrer une IA sur un système temps réel.

2. Objectifs

- Acquisition en temps réel : Capturer des signaux audios en temps réel.
- Traitement des signaux : Filtrer et prétraiter les signaux audios capturés.
- Construire une base de données Utiliser la BDD pour entrainer l'IA, construction à l'aide de signaux enregistré par la carte SD...
- Simulation Python : entrainer et réaliser l'IA avec Python
- Classification : Utiliser l'IA pour classifier les signaux audios en différentes catégories.
- Implémentation : Assurer que tout le traitement et la classification se fassent sur le MCU.

3. Périmètre

- Parties prenantes : Étudiants en ingénierie, encadrants académiques.
- Utilisateurs finaux : Applications potentielles dans la détection de sons spécifiques.

4. Fonctionnalités

- Acquisition des signaux :
  - Utilisation de microphones pour capturer les signaux audios.
- Traitement des signaux :
  - Filtrage.
  - Normalisation des signaux.
  - Transformation et extraction des données pour l'IA
- Simulation Python :
  - Entrainer et créer son propre modèle d'IA.
- Classification :
  - Implémentation d'un modèle d'IA.
  - Classification des signaux en temps réel.
- Interface utilisateur :
  - Affichage du résultat de classification.

5. Contraintes

Techniques :

- Limitation des ressources de la carte STM32 (mémoire, puissance de calcul).
- Latence minimale pour le traitement en temps réel.

Financières :

- Budget limité pour l'achat de composants et le développement.

Temporelles :

- Délai de 6 mois pour la réalisation complète du projet.

6. Budget et Planning

- Budget estimé :
  500 € pour les composants matériels

Planning prévisionnel :

Mois 1 (7 octobre - 31 octobre 2024)

- Recherche et sélection des composants :
  - Choix de la carte STM32 et du microphone.
  - Définition des spécifications techniques.
- Début de la documentation :
  - Rédaction de l'introduction et du contexte du projet.
  - Lecture de la thèse
- Acquisition des signaux par le microphone et enregistrement sur une carte SD.

Mois 2 (1 novembre - 11 décembre 2024)

- Filtrage des signaux et normalisation
- Création de la BDD
- Documentation pour l'extraction des données.
- Début d'extraction des données et création de l'IA sur Python
- Préparation de la soutenance intermédiaire :
  - Préparation des diapositives et du discours.
  - Répétitions et ajustements.

12 décembre 2024

  - Soutenance intermédiaire :

Mois 3 (13 décembre - 31 décembre 2024)

- Traitement des signaux :
  - Tests et validation des étapes de prétraitement.

Mois 4 (1 janvier - 31 janvier 2025)

- Développement de l'IA :
  - Entraînement du modèle d'IA avec des données audio.
  - Implémentation du modèle sur la carte.
- Tests et validation :
  - Tests de classification en temps réel.
  - Ajustements et optimisations.

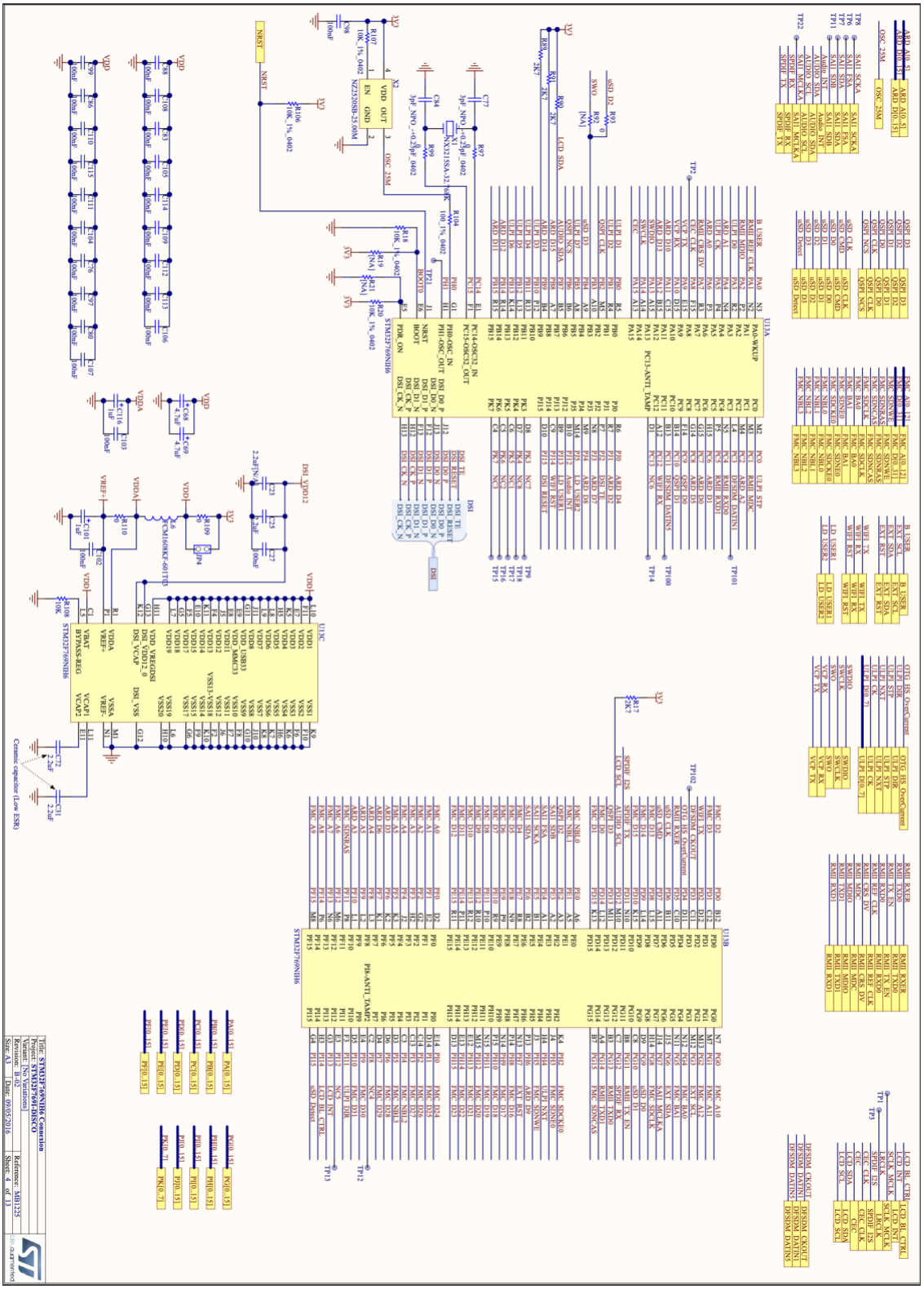
Mois 5 (1 février - 24 février 2025)

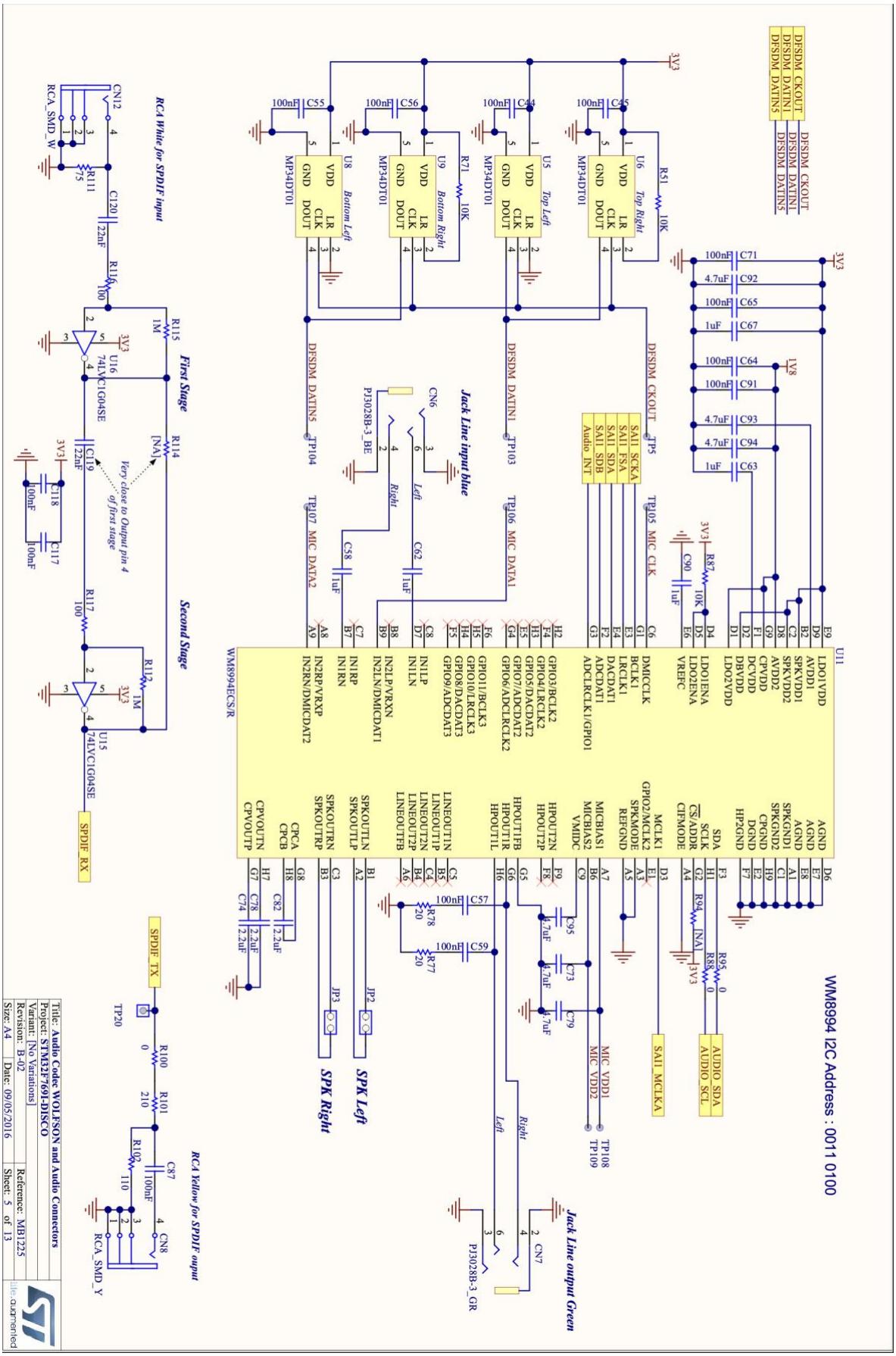
- Finalisation et intégration :
  - Intégration complète du système d'acquisition, de traitement et de classification.
  - Tests finaux et validation du système.
- Rédaction du rapport final :
  - Compilation des résultats, analyses et conclusions.
  - Révision et finalisation du rapport.
- Préparation de la soutenance finale

7. Critères de succès

- Performance : Précision de classification supérieure à 70%.
- Efficacité : Latence de traitement inférieure à 300 ms.
- Fiabilité : Système stable et robuste en conditions réelles.

2) Annexe 2 : Schéma STM32F769I-DISCO




WM8994 I2C Address : 0011 0100
WM8994ECS/R
U11
Jack Line input blue
Jack Line output Green
First Stage
Second Stage
RCA White for SPDIF input
RCA Yellow for SPDIF ouput
SPK Left
SPK Right
Very close to Output pin 4 of first stage
Left
Right
SPDIF TX
SPDIF RX
AUDIO_SDA
AUDIO_SCL
SAI1_MCLKA
MIC_VDD1
MIC_VDD2
Title: Audio Codec WOLFSON and Audio Connectors
Project: STM32F769I-DISCO
Variant: [No Variations]
Revision: B-02
Reference: MB1225
Size: A4
Date: 09/05/2016
Sheet 5 of 13