

# Projet A3 TC AUDISEN

## Description détaillée

## Table des matières

1	Introduction.....	3
1.1	Contexte.....	3
1.2	AUDISEN.....	4
1.2.1	Partition cible.....	4
1.2.2	Encodage .....	5
1.2.3	Accentuation.....	5
1.2.4	Fichier AMS.....	5
2	Description des blocs.....	6
2.1	Architecture du programme demandé.....	6
2.2	Lecture d'une playlist.....	8
2.2.1	Principe .....	8
2.2.2	Traitement de la playlist en C .....	9
2.3	Lecture d'une partition électronique.....	9
2.4	Création d'une trame.....	9
2.4.1	Trame et somme de contrôle.....	9
2.4.2	Usage dans le projet : .....	10
2.5	Communication avec la carte électronique .....	11
2.5.1	Mise en place du matériel .....	11
2.5.2	Développement de l'envoi de trame par câble USB.....	13
2.5.3	Code à fournir.....	14
2.6	Création d'une partition électronique .....	14
3	Côté Pratique.....	15
3.1	Environnement de travail .....	15
3.2	Simulateur.....	16
3.2.1	Simulateur Python .....	16
3.2.2	Simulateur Matlab .....	17
3.3	Utilisation de la carte électronique.....	18
4	Evaluation .....	18
4.1	Rendu .....	18
4.2	Recette .....	18
4.3	Audit des étudiants .....	19
4.4	QCM .....	19
5	Annexes .....	19
5.1	Définitions .....	19

5.2	Signature des fonctions.....	19
5.2.1	Lecture d'une playlist .....	19
5.2.2	Lecture d'une partition électronique .....	19
5.2.3	Création de trames .....	20
5.3	Communication avec USB .....	20
5.3.1	Création de partition électronique.....	20
5.4	Tests automatiques.....	20
5.5	Fréquences des notes .....	21

## 1 Introduction

### 1.1 Contexte

De nos jours, l'écoute de musique se fait principalement via des applications (Spotify, Deezer, Youtube,...), pour lesquelles les morceaux de musique sont stockées sur des serveurs. On estime que la musique en « streaming » générait en 2016 entre 200 et 300 millions de kilogrammes de gaz à effet de serre, là où les disques compacts (CD) généraient en 2000 157 millions de kilogrammes de gaz à effet de serre. L'industrie du « streaming » est donc 2 fois plus polluantes que celle du CD il y a 20 ans. Le téléchargement définitif de morceau de musique localement est également moins polluant (époque des baladeurs MP3). Notamment car il n'implique le transfert du morceau qu'une unique fois.

L'objectif de projet A3 tronc-commun 2023-2024, nommé AUDISEN, est de réaliser un système de lecture de morceaux de musique capable de lire en entrée des partitions musicales numérisées, de synthétiser les sons associés aux partitions puis de les restituer de façon analogique. En bref, de créer un baladeur audio complet lisant des morceau de musique stocké localement.

La Figure 1 décrit le découpage du projet :

- Partie informatique :
  - Lecture d'une playlist
  - Lecture d'une partition numérique
  - Création de trames de communications avec la carte microcontrôleur
  - Ecriture sur port USB
- Partie microcontrôleur
  - Réception des trames en provenance du PC via la liaison USB/UART
  - Crédit des signaux de sortie :
    - Emphasis : signal impulsionnel de période variable de rapport cyclique fixe
    - Audio : signal audio analogique
- Partie analogique : transformation du signal audio en y ajoutant :
  - Effet d'attaque (dynamisme, accentuation)
  - Réglage tonalité
  - Distorsion « douce » tube / « froide » transistor
  - Effet trémolo (modulation d'amplitude BF)

## Audisen Music Player

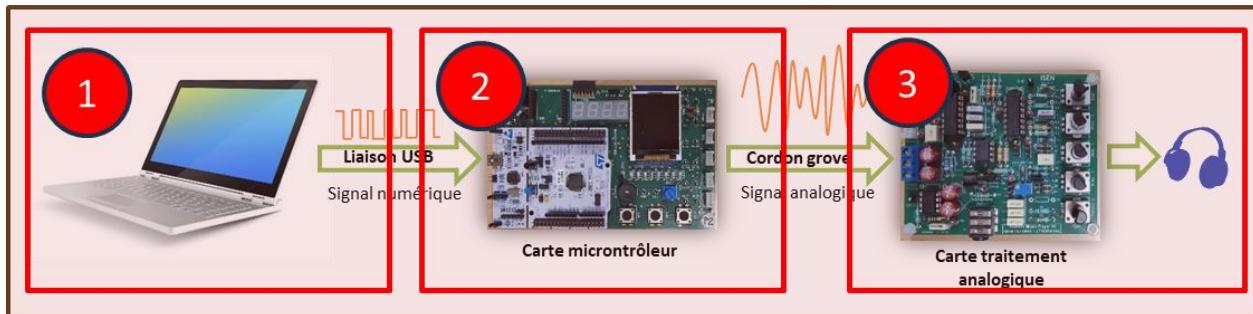


Figure 1: Découpage du projet AUDISEN

### 1.2 AUDISEN

#### 1.2.1 Partition cible

On suppose une partition restreinte aux symboles suivants :

Symbol musical	Nom	Durée
○	Ronde	4 temps
♩	Blanche	2 temps
♪	Noire	1 temps
♪	Croche	0,5 temps
♩	Demi-soupir	0,5 temps

Figure 2 : Symboles musicaux

Un exemple de partition est donné en Figure 3.



Figure 3 : Exemple de partition

La plus petite durée symbolisée est de 0.5 temps, nous appellerons cette durée un « tick ». Ainsi une croche correspond à un tick, une noire 2 ticks, une blanche 4 ticks et ainsi de suite ...

Le tempo fait le lien entre la durée d'une note (temps) et le temps réel d'exécution (seconde). Un tempo de 72 noires/min comme dans la Figure 3, devient pour AUDISEN : 144 ticks/min (tpm)

## 1.2.2 Encodage

La partition est découpée en « tick », on associe à chaque note sa notation internationale en supposant 5 octaves maximum (C1 à B5) et pas plus de 4 notes jouées simultanément.

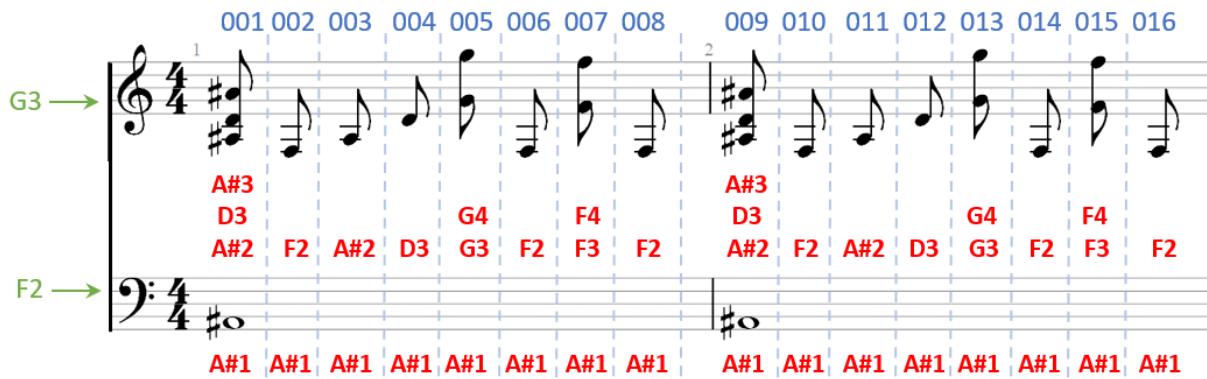


Figure 4 : découpage par ticks

En suivant le découpage donné en Figure 4, on obtient la suite de commandes suivantes :

```
Ligne 1 : A#1 , A#2 , D3 , A#3
Ligne 2 : A#1 , F2
Ligne 3 : A#1 , A#2
...
Ligne 16 : A#1 , F2
```

## 1.2.3 Accentuation

On dit qu'une note est accentuée, lorsqu'elle jouée pour la première fois. Nous noterons « ^ » quand la note est accentuée et « x » lorsqu'elle n'est pas accentuée. Ainsi pour la partition donnée en Figure 3, on obtient :

```
Ligne 1 : A#1 ^ , A#2 ^ , D3 ^ , A#3 ^
Ligne 2 : A#1 x , F2 ^
Ligne 3 : A#1 x , A#2 ^
...
Ligne 16 : A#1 x ; F2 ^
```

## 1.2.4 Fichier AMS

La dernière étape consiste à écrire les informations précédentes dans un fichier de type \*.ams dans le format décris en Figure 5.

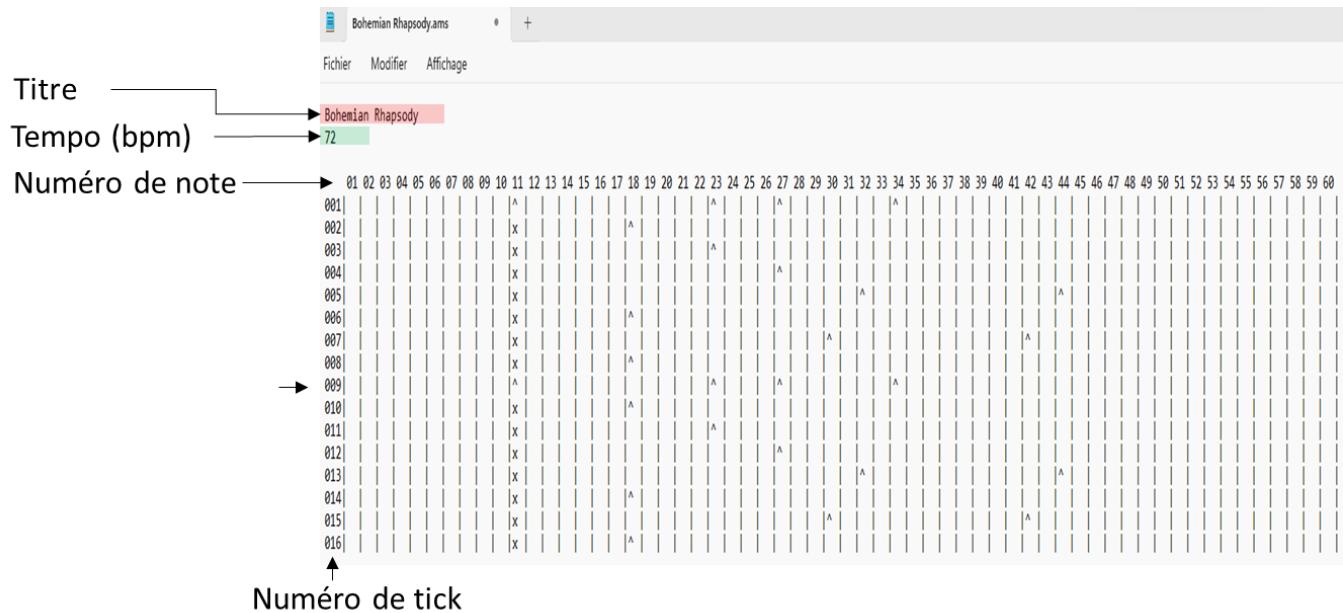


Figure 5 : Exemple de fichier AMS

On écrit successivement

- Le titre du morceau
- Le tempo en bpm

Ensuite on écrit sous forme de matrice les notes à jouer avec en colonne le numéro de la note et en ligne le numéro du tick, un « ^ » signifie que la note est accentuée et « x » que la note est jouée normalement.

## 2 Description des blocs

### 2.1 Architecture du programme demandé

Le programme demandé est décrit en Figure 6, il prend en entrée une playlist décrise dans un fichier \*.amp, des partitions numériques décrise dans des fichiers \*.ams (ou bien de façon simplifiée dans des fichiers \*.txt) et envoie par un port USB des commandes afin qu'une carte STM32 joue la playlist choisie en entrée. Afin de tester le programme principal, Il est demandé d'écrire les trames/commandes qui devraient être envoyées par USB dans un fichier \*.frm. Ce fichier .frm qui remplace le transfert USB et sera lu par un simulateur mis à disposition (Python et/ou Matlab).

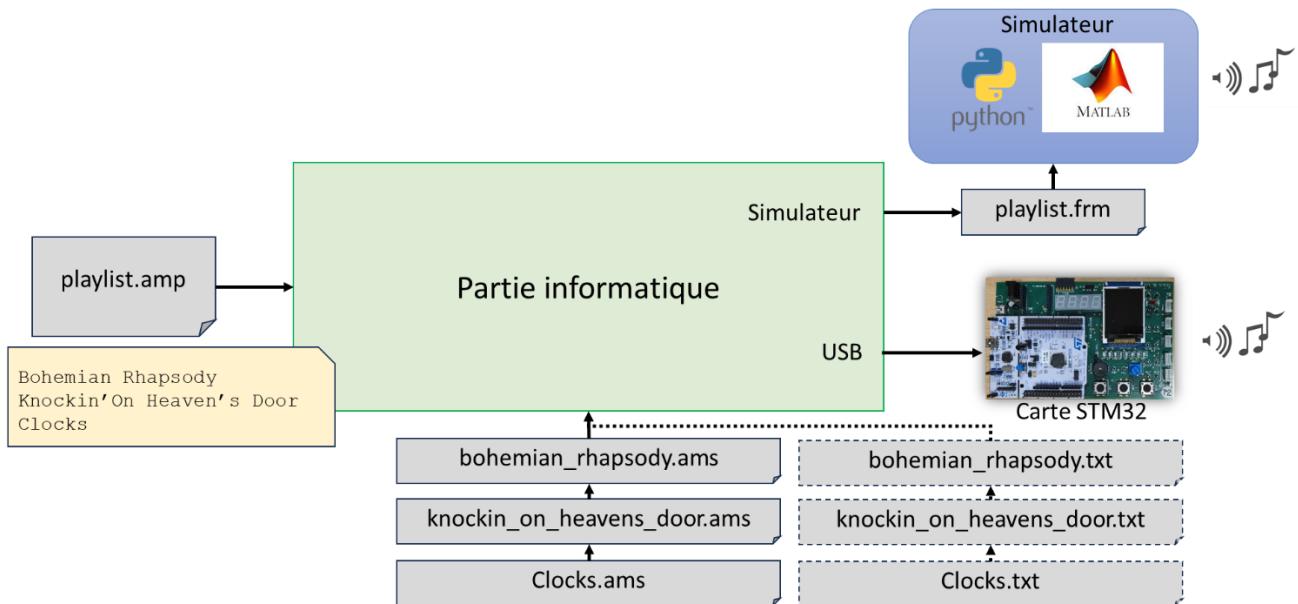


Figure 6 : description de la partie informatique

Le synoptique du programme montrant les différentes interactions entre les blocs demandés est décrit en Figure 7.

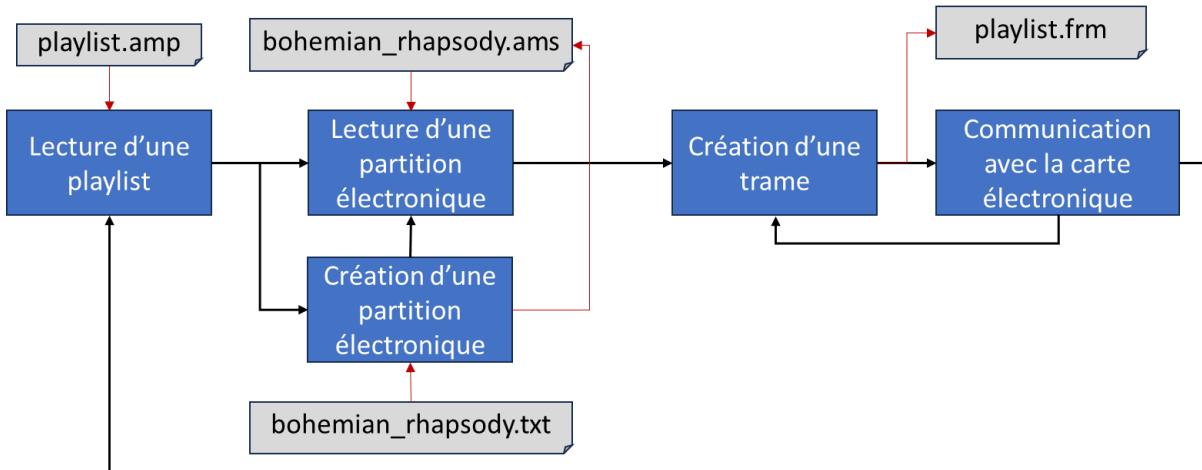


Figure 7 : Synoptique de la partie informatique

Le programme demandé doit être codé en Langage C, il est demandé d'utiliser les structures définies en annexe section 5.1, les prototypes de fonction décrits en section 5.2 ainsi que les noms de fichiers donnés dans le Tableau 1. On vous demande de fournir un makefile afin d'automatiser la compilation du projet.

Nom fonctions	Fichiers	Param	Retour	Description
initAMP	amp.c et amp.h	char* fileName	FILE* pf	Ouverture fichier .amp
readAMP	amp.c et amp.h	FILE* pf, char* songFileName		Renvoie le titre d'une chanson
closeAMP	amp.c et amp.h	FILE* pf		Ferme le fichier .amp
readAMS	ams.c et ams.h	char* fileName	s_song mySong	Lit le fichier .ams d'une chanson
createAMS	ams.c et ams.h	char* txtFileName char* amsFileName		Créé un fichier .ams d'une chanson à partir d'une partition simplifiée décrite dans un fichier .txt
createInitFrame	frame.c et frame.h	s_song mySong, char* frame		Créé la trame d'initialisation
createTickFrame	frame.c et frame.h	s_tick myTick, char* frame		Créé une trame pour un tick
initUSB	usb.c et usb.h		FT_HANDLE myHandle	Ouvre l'USB
writeUSB	usb.c et usb.h	char* frame FT_HANDLE myHandle		Écrit une trame sur l'USB
closeUSB	usb.c et usb.h	FT_HANDLE myHandle		Ferme l'USB

Tableau 1 : Fonctions demandées

## 2.2 Lecture d'une playlist

### 2.2.1 Principe

Le fichier playlist fourni est un fichier texte dont l'extension se termine en `.amp`. Un fichier playlist peut donc être édité via un bloc-notes ou tout autre application capable de modifier un fichier texte.

Le fichier playlist `Playlist.amp` contient exactement autant de lignes que de morceaux à jouer. Il n'y a pas de ligne vide à la fin du fichier. La dernière ligne contient le nom du dernier morceau à jouer. Chaque ligne contient un texte qui représente le titre du morceau seul : `Bohemian Rhapsody` par exemple. Le passage à la ligne dans le fichier signifie que le titre est terminé. Il peut y avoir des espaces, des apostrophes, des caractères spéciaux qu'il faudra traiter. Même si dans le code, un titre ne doit pas dépasser 40 caractères, dans la playlist il peut dépasser 40 caractères.

Des fichiers `ams` seront présents sur votre disque. Ils représentent les morceaux à jouer. L'objectif de la lecture de la playlist est de renvoyer **un par un** les titres du fichier de la playlist dans une chaîne de caractère qui correspond à un nom de fichier « `.ams` ». Les noms de fichiers `ams` ne contiennent que des minuscules, les espaces et les caractères spéciaux sont remplacés par le caractère « `_` ». Il ne doit pas rester de caractère « `_` » plusieurs fois à la suite.

**Bohemian Rhapsody**

Donnera en sortie

**bohemian\_rhapsody**

Puis

**Knockin' On Heaven's Door**

Donnera en sortie

**knockin\_on\_heaven\_s\_door**

On supprimera éventuellement tous les caractères superflus en début de titre (pas de `__titre`) , ainsi que des `__` à répétition.

Les fichiers `ams` en question seront fournis pour le projet. Pour ce qui touche aux fichiers `ams`, lire la section 2.3.

## 2.2.2 Traitement de la playlist en C

Trois fonctions sont à implémenter en respectant impérativement la signature donnée :

```
FILE* initAMP(char* filename);
void readAMP(FILE* pf, char* song_name);
void closeAMP(FILE* pf);
```

**initAMP** doit tenter d'ouvrir le fichier dont le nom sur le disque est dans la variable **filename** : "**playlist.amp**". Si le fichier s'ouvre en lecture, on renvoie un pointeur vers ce fichier. Sinon on renvoie NULL.

**closeAMP** doit vérifier que le fichier pointé par **pf** existe bien et le fermer le cas échéant.

**readAMP** consiste à lire **une ligne** du fichier pointé par **pf** et renvoyer une chaîne de caractère. Cette chaîne de caractère doit absolument respecter les critères exigés plus haut. Si la lecture du fichier se termine, la fonction doit remplacer **song\_name** par NULL. A l'usage, NULL indique à celui qui utilise **readAMP** qu'il n'y a plus de titre à lire.

Conseil : Nous conseillons de faire les traitements sur la chaîne de caractère dans des fonctions courtes et séparées au fur et à mesure du projet. Chaque aspect (majuscule, apostrophe etc.) sera évalué séparément, il est donc possible de ne pas traiter un de ces aspects. Il est évidemment possible (même recommandé) de créer des fonctions « en plus » pour alléger le code des fonctions principales.

L'appel de ces trois fonctions **initAMP**, **readAMP**, **closeAMP**, doit être organisé dans une autre fonction permettant d'ouvrir la playlist, de lire les titres au fur et à mesure puis de fermer la playlist à la fin.

ATTENTION : ne pas modifier la signature des fonctions obligatoires **initAMP**, **readAMP**, **closeAMP**.

## 2.3 Lecture d'une partition électronique

La partition électronique est un fichier **\*.ams** tel que décrit en section 1.2.4. On vous demande de créer une fonction avec la signature suivante :

```
s_song readAMS (char* fileName)
```

La fonction **readAMS ()** lit le fichier nommé **fileName** qui est de type **.ams** et renvoie une structure de type **s\_song** contenant tous les informations sur le morceau. Si le fichier ne peut être ouvert la fonction doit renvoyer une structure contenant un titre vide (« ») et que des '0' sur tous les champs.

## 2.4 Crédit d'une trame

### 2.4.1 Trame et somme de contrôle.

Lorsque l'on communique avec l'extérieur, que ce soit via USB, via internet, ou via des ondes radio par exemple, il est nécessaire d'envoyer une information simple et courte permettant d'être reconnue par le récepteur. En numérique on envoie une suite de 0 et de 1 qui auront une signification particulière, on parle d'une trame, et il est d'usage d'y inclure également un moyen de vérifier si l'information a été ou non dégradée durant son voyage (auquel cas on jettera la trame et on pourra

demander à l'envoyeur une nouvelle trame par exemple, jusqu'à recevoir une information non dégradée).

Ce moyen de vérification est usuellement une « somme de contrôle » (checksum en anglais) : l'idée est d'avoir un chiffre calculable à partir des données de la trame, ainsi si un bit change au hasard dans la trame, ce chiffre calculé et celui qui nous a été envoyé ne correspondront plus.

Exemple :

0 22 -10 5 33 44 2 **96**

→ 96 est la somme simple des éléments précédent, si l'un des éléments change au hasard :

0 22 -10 5 33 44 -1 **96**

Alors il devient facile de refaire la somme des éléments et de vérifier que ce n'est pas égale à 96. C'est une méthode pratique qui est inclue dans presque tous les systèmes de communications et même dans le stockage ou encore dans les RAM modernes (DDR ECC). (note : l'exemple est illustratif, il est en base décimal, en général on résonne plutôt en base binaire car ce qu'on envoi en pratique en informatique est très majoritairement une suite de 0 et de 1.

Évidemment, si par hasard le chiffre de la somme de contrôle change également pour s'accorder pile à l'erreur des données, il n'y aucune manière de vérifier la chose. Mais cela arrive rarement d'une part, et l'idée est de choisir une fonction qui n'est pas forcément une somme simple comme dans l'exemple ci-dessus, mais une opération plus complexe afin qu'il faille modifier complètement la checksum pour une petite modification des données, ce qui rends le cas encore plus rare.

#### 2.4.2 Usage dans le projet :

Deux types de trames sont utilisées dans le projet : les trames d'initialisation et les trames de tick. Les trames sont des suites d'octets de la forme suivante :

#<content>\*<checksum><CR><LF>

1. Caractère '#'
2. Contenu de taille variable
3. Caractère '\*'
4. Checksum qui correspond au XOR bit à bit **des octets du contenu**, le tout écrit en hexadécimal (obligatoirement sur 2 caractères, par exemple '0d', en utilisant des minuscules : pas '0D')
5. Séquence de caractères <CR><LF>, codes ascii 13, puis 10, "\r\n" en C.

Comme vous pouvez le constater, on choisit ici une somme de contrôle (checksum) qui est un XOR de chaque octet avec le précédent résultat du XOR. C'est une fonction pratique car le résultat est différent en cas de changement d'un bit tout en s'assurant d'une taille fixe à la fin (ici on fait des XOR octet par octet donc la checksum finale fait un octet). N'oublions pas qu'un octet s'affiche en deux lettres hexadécimales, ce pourquoi il faut deux lettres pour le représenter. On code ces deux lettres en deux chars qui sont accolée au résultat. Cette conversion, au lieu de mettre le résultat en octet, nous assure qu'un petit changement de l'information envoyée aboutit à un changement notable de la somme de contrôle.

Le site <https://www.scadacore.com/tools/programming-calculators/online-checksum-calculator/> peut vous être utile pour vérifier le calcul de vos checksums.

Une trame d'initialisation doit être émise au début de la lecture d'une chanson. Il s'agit d'une suite d'octets défini de la façon suivante (exemple) :

```
#Bohemian Rhapsody,144,16*<checksum><CR><LF>
```

Ici le nom de chanson à afficher sur le lecteur est “Bohemian Rhapsody”, la chanson a un tempo de 144 (écriture décimale) et comportera 16 trames de ticks (écriture décimale).

Le checksum est donc calculé comme le XOR bit à bit des codes ASCII des caractères compris strictement entre # et \*, y compris les virgules.

Les trames de tick sont émises pendant la lecture d'une chanson. Il s'agit d'une suite d'octets définie de la façon suivante :

```
#<mode>,<accent>,<note1>,<note2>,<note3>,<note4>*<checksum>*<CR><LF>
```

- Mode : '0' (playlist) ou '1' (snaper)
- Accent : '0' (pas d'accent) ou '1' (accent). Accent si au moins une des 4 notes est accentuée
- Note1 : '00' (demi-soupir) ou '01'...'60' (note)
- Note2, 3, 4 : '00' (pas de note) ou '01'...'60' (note)

Les trames *doivent* être initialisées par les fonctions ci-dessous, fournies avec leurs signatures :

```
void createInitFrame(s_song mySong, char* frame);
void createTickFrame(s_tick myTick, char* frame);
```

Ces fonctions *doivent* être définies dans une paire de fichiers **frame.h** et **frame.c**.

Les paramètres d'entrée sont une structure du type adapté et un tableau de char d'une taille suffisante pour recevoir l'intégralité de la trame. Attention !, nous vous demandons d'ajouter un '\0' en fin de tableau afin de procéder à des tests automatiques. La détection de fin de trame peut se faire sur la séquence de deux caractères <CR><LF>.

## 2.5 Communication avec la carte électronique

### 2.5.1 Mise en place du matériel

Plusieurs carte STM32 sont mises à disposition des élèves. Elles sont préprogrammées. Elles permettent de tester l'envoi de donnée via un câble USB. La carte doit être reliée à une sortie analogique avec adaptateur jack pour un casque audio. Il n'y a qu'un seul adaptateur par groupe de TP. Il faut donc planifier ses tests avec le groupe de TP.

Voici une vue matérielle du dispositif entier :

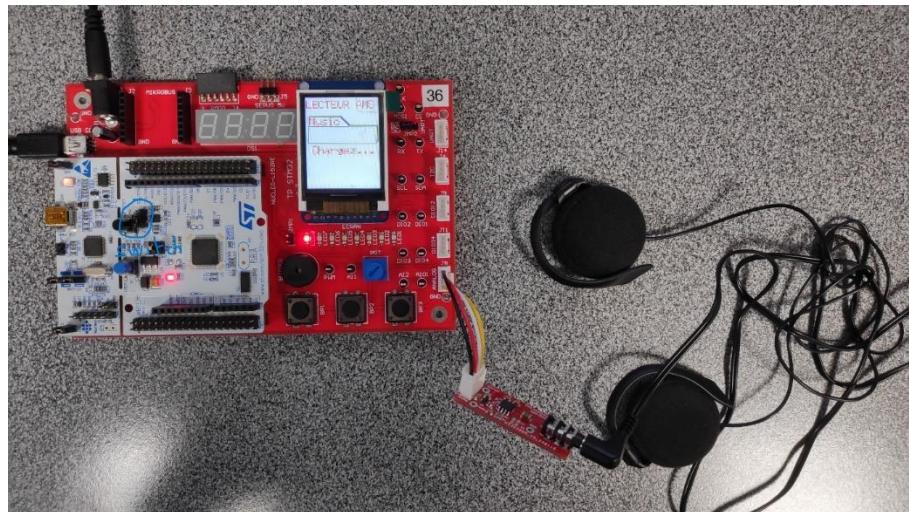


Figure 8 : Dispositif complet d'écoute

**Important:**

Pour déployer le dispositif complet il faut :

- Brancher la sortie USB sur le port **USB COM** (et pas sur power USB de la STM32)
- Déplacer le switch E5V-U5V de la STM32 sur **E5V** pour permettre une alimentation sur le connecteur J1
- Brancher l'alimentation **5V** sur le connecteur **J1**
- Optionnellement, on peut brancher la sortie analogique sur le port ANALOG. Sans lui, il n'y a juste pas de son (brancher le casque également).

Une fois le matériel mis en place et configuré, on peut le débrancher et le prêter à un autre binôme. Il n'est pas utile pour le développement, seulement pour les tests.

Sans casque, vous pouvez tout de même vérifier que l'envoi des trame titre fonctionne, modifier le tempo. Lorsqu'une mélodie est chargée on voit son titre sur l'interface de la carte. Le tempo peut être modifié après appui sur le bouton **Play** avec le bouton + ou -.

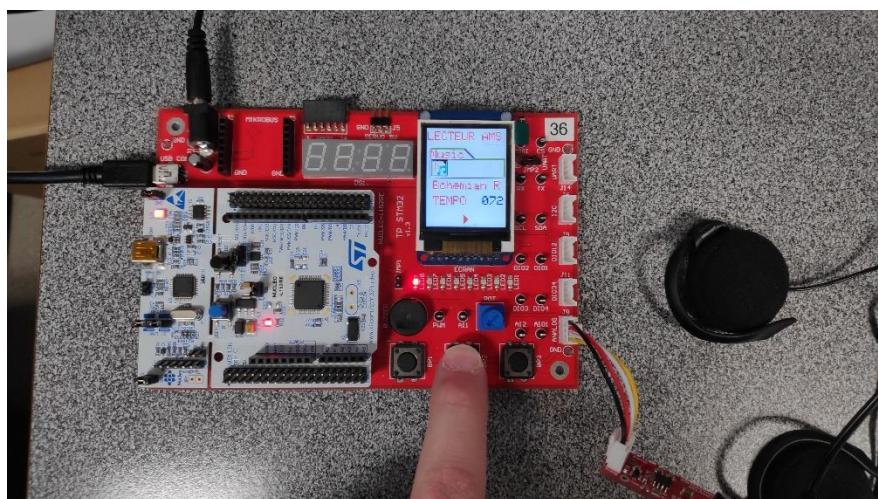


Figure 9 : Appui sur play

## 2.5.2 Développement de l'envoi de trame par câble USB

La partie suivante décrit l'envoi de trames USB sous un environnement Windows. La description sous mac et linux n'est pas faite. Se référer à la doc [ftd2xx](#) pour une programmation USB avec une Mac ou en Linux.

*Installation de l'environnement de programmation Windows.*

*Idée : avant de changer d'environnement de DEV, faites une sauvegarde complète de votre projet fonctionnel, ceci pour éviter d'avoir un projet qui ne fonctionne plus du tout sous Windows...*

Si vous utilisez un IDE de programmation (VS CODE, CLION) vérifiez au préalable si votre installation comprend un compilateur compatible Windows de type **MinGW**. Si c'est le cas, dans les settings de votre projet, changez votre compilateur **gcc** de WSL pour celui compatible Windows.

Sinon, installez le compilateur **MinGW**. Un document explicatif d'installation de **MinGW** est disponible sur Moodle. Sinon référez-vous aux explications fournies sur le web.

Avant de passer à l'étape suivante, vérifiez que votre projet fonctionne bien sous Windows avec **MinGW**.

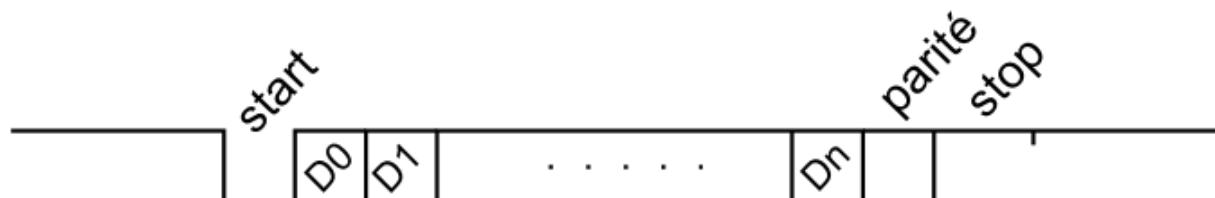
*Développement de la communication avec la lib ftd2xx.*

On trouve de la documentation pour écrire le code ici

<https://ftdichip.com/software-examples/code-examples/c-builder/>

L'envoi de chaque trame se fait en UART, c'est un protocole simple dit « série » qui était autrefois le protocole qu'on utilisait pour envoyer des informations sur un unique fil électrique (le protocole est hérité du télégraphe). Aujourd'hui les ordinateurs envoient plusieurs informations en parallèle grâce à des prises et câbles possédant de multiples fils (par exemple en USB4 il y a trois canaux de données en parallèle, chacun dédoublés car étant [une paire différentielle](#)), mais la maîtrise fine de cela dépasse le cadre de notre cours. Heureusement, pour ceux désirant simplement transmettre des informations en série simple (comme en USB1, en USB2 ou sur un connecteur série [RS-232](#)) il suffit de télécharger ce pilote ftd2xx qui nous permet d'utiliser l'UART sur n'importe quelle connexion USB.

Une trame UART ressemble à ceci :



Les bits D0 à D1 correspondent à des bouts de la trame que vous avez construite précédemment, le nombre de bit est à définir quand on construit la liaison. Mais en plus de votre trame, ils sont eux même entourés d'identifications supplémentaires, notamment un bit de start qui définit le début du signal et un bit de stop qui en définit la fin.

Il y a également un bit de parité optionnel, qui est comme une mini somme de contrôle. C'est la somme des bits précédent, sur un unique bit, qui définit donc uniquement si la somme est paire ou impaire. C'est un système de contrôle assez rudimentaire.

Il est aussi important de noter que le bit de stop peut durer un certain temps (1 temps de bit normal, ou 1.5 ou même 2.).

Notez également que du fait de l'héritage des transmissions télégraphique, le niveau de base de la ligne est 1 et non 0. Le bit de start est donc un 0 qui vient couper la ligne, et non un 1.

Enfin, tout ceci est transmis à un certain rythme, un nombre de bit par seconde. On parle ici plus généralement de « symbole par seconde » et l'unité de cela est le baud.

Tout ceci définit notre façon de communiquer et il faut évidemment que ce soit en accord avec ce que la carte STM32 attend, sinon ça ne fonctionnera pas ou mal.

Notre carte STM32 fonctionne à 9600 Baud, 8 bits de données, un bit de stop d'une longueur de 1 et pas de contrôle de la parité (vu qu'on a déjà notre somme de contrôle). On doit également préciser qu'elle n'utilise pas de « flow control », un système de contrôle permettant d'éviter à un récepteur de recevoir trop de donnée (ce qui n'arrivera pas ici).

Vous allez devoir vous référer à la documentation ftd2xx pour comprendre comment paramétrier la connexion afin de respecter ce que la carte STM32 attend.

### 2.5.3 Code à fournir

Dans deux fichiers **usb.h** **usb.c**, implémenter les trois fonctions suivantes :

Les signatures à respecter pour l'envoi de données en USB sont les suivantes

```
#include "ftd2xx.h"

FT_HANDLE initUSB();
void closeUSB(FT_HANDLE ftHandle);
void writeUSB(char* frame, FT_HANDLE ftHandle);
```

Finalement, une fois ces fonctions implémentées, il faudra les tester dans un programme principal.

Le programme principal doit permettre soit de générer un fichier .frm (mode fichier) soit une écriture sur USB (mode USB)

## 2.6 Crédit à une partition électronique

L'objet de ce bloc est de créer une partition électronique de type \*.ams à partir d'une partition simplifiée décrivant les notes et les symboles du morceau. Ce fichier d'entrée est de type .txt et suit le format suivant :

```
Bohemian Rhapsody
72

A1# R, A2# C, D3 C, A3# C
F2 C
A2# C
D3 C
...
```

La première ligne du fichier donne le titre du morceau ici « Bohemian Rhapsody » tandis la seconde ligne donne le tempo du morceau en bpm. Ensuite chaque ligne correspond à un « tick » où chaque

note est caractérisée par sa valeur en notation internationale (de C1 à B5) et sa durée (B,R, N ou C) décrite dans le Tableau 2. Chaque note est séparée par une virgule (« , »).

Notation	Signification	Temps
R	Ronde	8 ticks
B	Blanche	4 ticks
N	Noire	2 ticks
C	Croche	1 tick
D	Demi-soupir	1 tick

Tableau 2 : Notation des symboles de durée de note

La fonction aura la signature suivante :

```
void createAMS(char* txtFileName, char* amsFileName);
```

Les paramètres sont les suivants :

- `txtFileName` : nom du fichier d'entrée (.txt)
- `amsFileName` : partition électronique (.ams)

### 3 Côté Pratique

#### 3.1 Environnement de travail

Pour la partie simulation, l'environnement de travail n'est pas imposé, veuillez-vous reporter aux consignes données dans le module « Algorithme et Langage C ».

Pour la partie connexion à la carte électronique, les drivers USB sont prévus pour fonctionner sous Windows uniquement. **Un développement sous windows devient ainsi nécessaire.** Nous vous conseillons d'installer MSYS2 ainsi que le compilateur MinGW64 (<https://www.msys2.org/>). A partir de MSYS2 les installations de gcc et make se font de la façon suivante :

```
pacman -S mingw-w64-ucrt-x86_64-gcc
pacman -S make
```

(plus de détail sur le tuto d'installation sur Moodle)

Notez qu'il est possible d'utiliser la version de MinGW inclue dans Clion par exemple, mais cela vous empêchera d'installer simplement de nouveau package comme le permet MSYS2.

Afin de faciliter la recette on vous demande d'organiser les fichiers de votre projet de façon à avoir une architecture identique à la Figure 10. Attention toute architecture différente pourra entraîner un malus lors de la recette

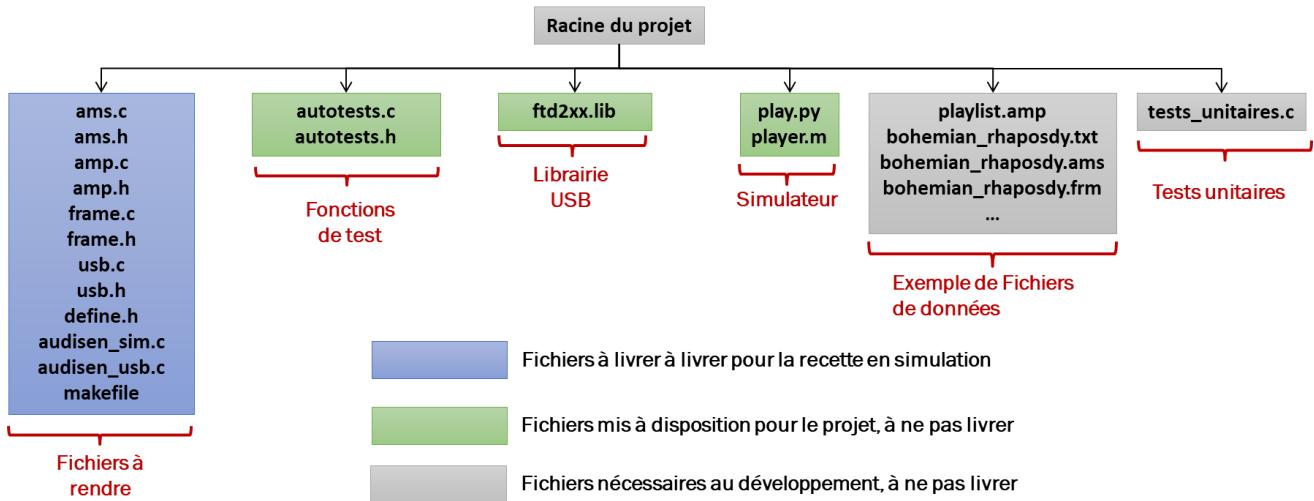


Figure 10 : architecture de fichiers demandée

### 3.2 Simulateur

Afin de lire les fichiers .frm produits par votre code principal C, nous vous fournissons deux simulateurs capables de lire ces fichiers et de jouer de la musique pour remplacer la carte STM32 et l'envoi par USB. Vous devez choisir un des deux simulateurs, selon vos facilités en Python ou en Matlab. Le professeur présent dans votre salle pourra aussi vous imposer l'un des deux, Python ou Matlab.

Pour tester l'écriture de vos trames. Le principe est le suivant :

En C : Exporter toutes les trames (titres et ticks) à la suite dans un fichier texte d'extension .frm. Pas de ligne vide. Chaque mélodie commence par son titre suivi de tous les ticks. Une trame par ligne. Pas de ligne vide à la fin.

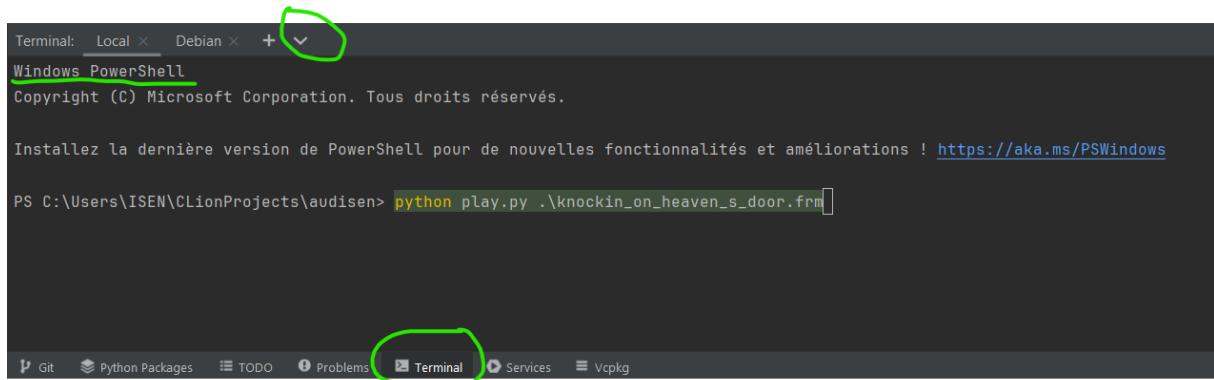
#### 3.2.1 Simulateur Python

Le simulateur python consiste en un seul fichier play.py. Il peut être placé dans le même répertoire que votre projet, au même niveau que les fichiers .frm.

Un fichier play.py est disponible sur moodle. L'installation du module python `simpleaudio` est obligatoire. Le fichier play.py contient les instructions d'installation. Il est possible d'utiliser `pycharm` ou autre IDE pour python, mais python en ligne de commande vous fera gagner du temps !

Lancer le script python en ligne de commande ou via votre IDE habituel. Mettre un casque !

Il est possible d'utiliser le terminal de votre IDE (ici CLION) pour lancer le script python directement. Si python est installé sur votre environnement Windows (cela devrait être le cas normalement) il faut bien mettre la console en mode PowerShell (ou Windows PowerShell ou Command Prompt) avec l'icône  de la figure suivante. Ne pas utiliser de console WSL, Debian... Pour les mac, il n'y a qu'une console, pas de choix à faire.



```
Terminal: Local × Debian × + ▾
Windows PowerShell
Copyright (C) Microsoft Corporation. Tous droits réservés.

Installez la dernière version de PowerShell pour de nouvelles fonctionnalités et améliorations ! https://aka.ms/PSWindows

PS C:\Users\ISEN\CLionProjects\audisen> python play.py .\knockin_on_heaven_s_door.frm
```

Vous pouvez aussi utiliser le **powershell** de votre PC (menu Windows, power... puis lancer powershell), l'interface est exactement la même. Seul souci : il faudra retrouver le chemin d'installation de votre projet pour lancer votre script avec des commandes **cd**. **cd** sous **powershell** est identique à **cd** sous linux.

### 3.2.2 Simulateur Matlab

#### 3.2.2.1 Installation

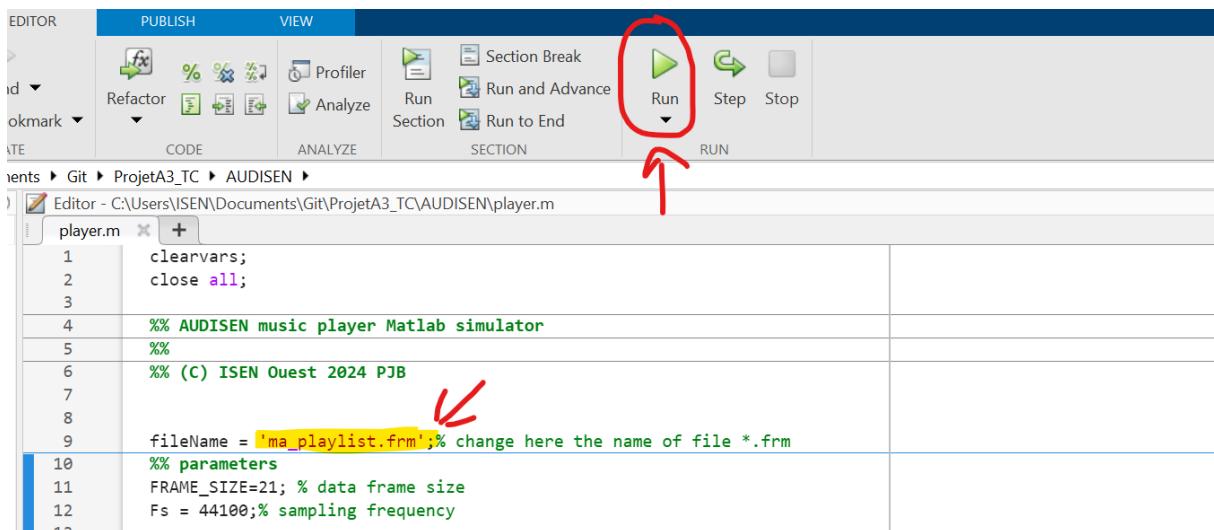
Le logiciel Matlab est un logiciel scientifique qui va sera en utile notamment en M1 dans les cours liés au traitement du signal. Vous pouvez l'installer gratuitement en suivant le lien suivant :

<https://www.mathworks.com/academia/tah-portal/isen-yncrea-ouest-31160968.html>

Il faut pour cela s'enregistrer en utilisant (uniquement) votre adresse @isen-ouest.yncrea.fr.

#### 3.2.2.2 Utilisation de simulateur

Ouvrez le fichier player.m avec le logiciel matlab. En ligne 9, écrivez le nom du fichier \*.frm souhaité. Il ne reste plus qu'à cliquer sur « Run » comme le montre la Figure 11. Le logiciel va alors jouer la playlist représentée par le fichier \*.frm.



```
EDITOR PUBLISH VIEW
Refactor Profiler Analyze Run and Advance Step Stop
CODE ANALYZE SECTION RUN
Run Section Run to End
RUN

Editor - C:\Users\ISEN\Documents\Git\ProjetA3_TC\AUDISEN\player.m
player.m
1 clearvars;
2 close all;
3
4 %% AUDISEN music player Matlab simulator
5 %%
6 %% (C) ISEN Ouest 2024 PJB
7 %
8 %
9 fileName = 'ma_playlist.frm'; % change here the name of file *.frm
10 %% parameters
11 FRAME_SIZE=21; % data frame size
12 Fs = 44100; % sampling frequency
```

Figure 11 : Simulateur Matlab

### 3.3 Utilisation de la carte électronique

L'utilisation de la carte électronique est réservée au binôme qui vont transférer les données par USB. Il n'y a que quelques cartes par groupe. Demandez à votre professeur référent de vous fournir le matériel. La description de l'utilisation des cartes est faite en 2.5.1.

## 4 Evaluation

### 4.1 Rendu

. La recette de votre programme sera effectuée le dernier jour entre 15:00 et 18:00. Vous devrez rendre avant 15 :00 le jour de la recette une archive contenant vos sources que vous nommerez sources\_xx\_yy.zip (où xx et yy désignent les initiales respectives du binôme). Les fichiers à inclure sont détaillés en Figure 12.

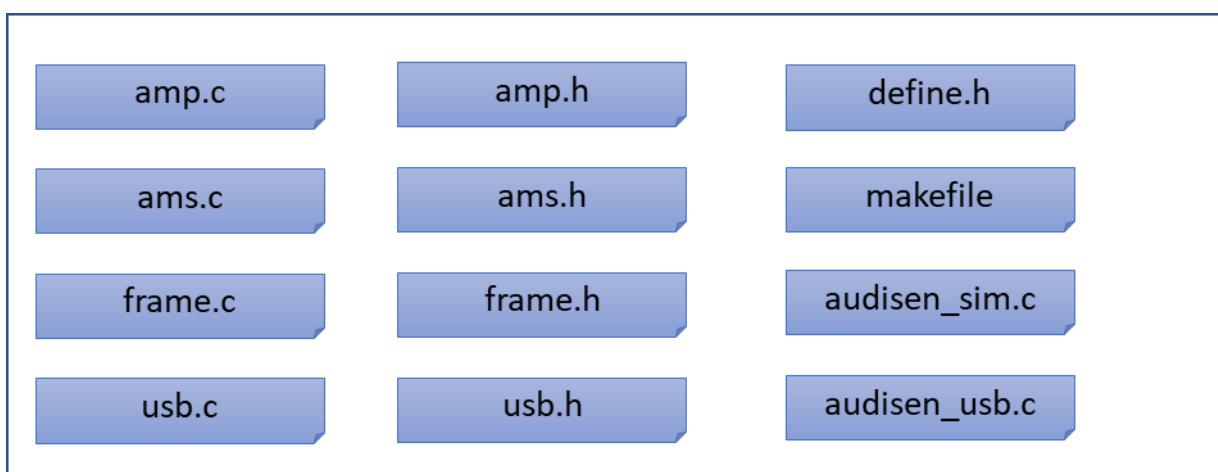


Figure 12 : Fichiers à rendre

### 4.2 Recette

Lors de la recette, l'évaluateur extrait sur son ordinateur l'archive postée par chaque groupe d'étudiants et procède aux tests détaillés dans le Tableau 3 avec le barème associé. Une partie des tests est effectuée au moyen de tests automatiques décrits en section

#	Nom du test	Type de test	Fonction de test	A3 hors CIR3	CIR3
1	Lecture d'un fichier AMP	Automatique	testReadAMP()	4	2
2	Lecture d'un fichier AMS	Automatique	testReadAMS()	4	3
3	Création de trames	Automatique	testFrame()	4	2
4	Création d'un fichier AMS	Automatique	testCreateAMS()	4	5
6	Programme global en simulation	Manuel		4	4
7	Programme global en USB	Manuel		4	4
<b>Total</b>				<b>24</b>	<b>20</b>

Tableau 3 : Barème de la recette

### 4.3 Audit des étudiants

Lors de la recette, les étudiants seront audités sur le code soumis. La note générée sera proportionnelle au nombre de fonctionnalités validées lors de la recette (une fonction non validée ne sera pas auditée).

### 4.4 QCM

Lors du jour de la recette, une épreuve théorique de validation des connaissances acquises sera effectuée sous la forme d'un questionnaire à choix multiple (QCM) d'une durée de 30 min. Ce QCM sera effectué en monôme.

## 5 Annexes

### 5.1 Définitions

```
#define MAX_SIZE_TITLE 40
#define MAX_SIZE_LINE 190
#define MAX_NUMBER_TICKS 999
typedef struct tick{
    int accent;//accentuation du tick (0=Non, 1:Oui)
    int note[4];// Tableau de 4 notes (0 à 60)
}s_tick;

typedef struct song{
    int tpm;// tick par minutes
    int nTicks; // Nombre de ticks dans le morceau
    char title[MAX_SIZE_TITLE];// Titre du morceau
    struct tick tickTab[MAX_NUMBER_TICKS]; // Tableau de ticks
}s_song;
```

### 5.2 Signature des fonctions

#### 5.2.1 Lecture d'une playlist

```
FILE* initAMP(char* filename) {
    FILE * pf = NULL;

    return pf;
}

void readAMP(FILE* pf, char* songFileName) {

}

void closeAMP(FILE* pf) {

}
```

#### 5.2.2 Lecture d'une partition électronique

```
s_song readAMS(char* fileName) {
```

```
s_song mySong;

return mySong;
}
```

### 5.2.3 Création de trames

```
void createInitFrame(s_song mySong, char* frame) {

}

void createTickFrame(s_tick myTick, char* frame) {

}
```

## 5.3 Communication avec USB

```
FT_HANDLE initUSB() {
    FT_HANDLE ftHandle;

    return ftHandle;
}

void closeUSB(FT_HANDLE ftHandle) {

}

void writeUSB(char* frame, FT_HANDLE ftHandle) {

}
```

### 5.3.1 Création de partition électronique

```
void createAMS(char* txtFileName, char* amsFileName) {

}
```

## 5.4 Tests automatiques

Nous vous mettons à disposition un fichier de fonctions (autotests.c/autotests.h) permettant de tester unitairement les différents blocs :

- testReadAMP()
- testReadAMS()
- testFrame()
- test CreateAMS()

Chaque fonction affiche une valeur entre 1 (bloc fonctionnel) et 0 (bloc non fonctionnel). Un programme permettant de tester unitairement les différents blocs est le suivant :

```
---- Autotest results of block ReadAMP ----
--> test 0 : 1/1
--> test 1 : 1/1
--> test 2 : 1/1
--> test 3 : 1/1
--> test 4 : 1/1
--> test 5 : 1/1
Finish autotest of block ReadAMP => total score : 100.0 %

---- Autotest results of block ReadAMS ----
--> test 0 : 1/1
--> test 1 : 1/1
Finish autotest of block ReadAMS => total score : 100.0 %

---- Autotest results of block Frame ----
--> test 0 : 1/1
--> test 1 : 1/1
--> test 2 : 1/1
--> test 3 : 1/1
Finish autotest of block Frame => total score : 100.0 %

---- Autotest results of block CreateAMS ----
--> test 0 : 1/1
--> test 1 : 9/9
Finish autotest of block CreateAMS => total score : 100.0 %
```

## 5.5 Fréquences des notes

Notation		Numéro note	Fréquence note (Hz)
Française	Internationale		
Si5	B5	60	1975,534
La#5	A#5	59	1864,656
La5	A5	58	1760
Sol#5	G#5	57	1661,219
Sol5	G5	56	1567,982
Fa#5	F#5	55	1479,978
Fa5	F5	54	1396,913
Mi5	E5	53	1318,511
Ré#5	D#5	52	1244,508
Ré5	D5	51	1174,66
Do#5	C#5	50	1108,731
Do5	C5	49	1046,503
Si4	B4	48	987,767
La#4	A#4	47	932,328
La4	A4	46	880
Sol#4	G#4	45	830,61
Sol4	G4	44	783,991
Fa#4	F#4	43	739,989
Fa4	F4	42	698,457
Mi4	E4	41	659,256
Ré#4	D#4	40	622,254
Ré4	D4	39	587,33

Do#4	C#4	38	554,366
Do4	C4	37	523,252
Si3	B3	36	493,884
La#3	A#3	35	466,164
La3	A3	34	440
Sol#3	G#3	33	415,305
Sol3	G3	32	391,996
Fa#3	F#3	31	369,995
Fa3	F3	30	349,229
Mi3	E3	29	329,628
Ré#3	D#3	28	311,127
Ré3	D3	27	293,665
Do#3	C#3	26	277,183
Do3	C3	25	261,626
Si2	B2	24	246,942
La#2	A#2	23	233,082
La2	A2	22	220
Sol#2	G#2	21	207,653
Sol2	G2	20	195,998
Fa#2	F#2	19	184,998
Fa2	F2	18	174,615
Mi2	E2	17	164,814
Ré#2	D#2	16	155,564
Ré2	D2	15	146,833
Do#2	C#2	14	138,592
Do2	C2	13	130,813
Si1	B1	12	123,471
La#1	A#1	11	116,541
La1	A1	10	110
Sol#1	G#1	9	103,827
Sol1	G1	8	97,999
Fa#1	F#1	7	92,499
Fa1	F1	6	87,308
Mi1	E1	5	82,407
Ré#1	D#1	4	77,782
Ré1	D1	3	73,417
Do#1	C#1	2	69,296
Do1	C1	1	65,407

Tableau 4 : correspondance entre note, numéro et fréquence