

Real-time digital audio signal processing with a STM32F746 Discovery board

3.1 Quelques questions à se poser pour savoir estimer rapidement les performances d'un système audio temps-réel

- Que dit le théorème de Shannon ?

La fréquence d'échantillonnage doit être strictement 2 fois supérieure à la fréquence du signal.

- Pour du traitement de la parole, quelle fréquence d'échantillonnage préconisée seriez vous ? Idem pour le signal de musique ?

Pour la parole on préconise du 16 kHz et 48 kHz pour la musique (en fonction du théorème de Shannon).

- Qu'est-ce que le "double-buffering" ? Qu'est-ce que le DMA transfert ? Comment est implémenté le double-buffering dans le code fourni (rôle des IRQ Handler notamment, cf. `audio.c`) ?

Le double-buffering et le DMA transfert sont des concepts couramment utilisés pour garantir un flux de données fluide et sans interruption. Le double-buffering est une technique utilisée pour éviter les interruptions dans le flux de données lors de l'envoi ou de la réception de données. Dans le contexte de l'audio, cela signifie que deux tampons (buffers) sont utilisés pour stocker les échantillons audio. Pendant que l'un des tampons est en cours de lecture ou d'écriture, l'autre peut être rempli ou vidé, de sorte que le flux audio puisse se poursuivre de manière continue. Le DMA est une technique de transfert de données qui permet à un périphérique matériel de transférer des données directement entre la mémoire et le périphérique sans l'intervention constante du processeur. Cela libère le processeur pour effectuer d'autres tâches, améliorant ainsi l'efficacité du système.

- A 48kHz de fréquence d'échantillonnage, de combien de temps dispose-t-on pour traiter un échantillon audio ? A titre de comparaison, quelle est la durée moyenne d'une instruction (par exemple une addition de deux registres) sur un processeur STM32F7 cadencé à 200MHz ?

Lorsque la fréquence d'échantillonnage est de 48kHz alors on dispose de $\frac{1}{48000}$ secondes, soit 20 microsecondes. Sur un processeur cadencé à 200MHz cela fait une durée de 5 nanosecondes.

- **La Latence audio est définie comme le temps séparant l'arrivée d'un échantillon sur l'entrée audio et sa restitution sur la sortie casque (pour fixer les idées, imaginons un effet audio dont la fonction de transfert est simplement $T(z) = 1$) : donnez-en une estimation en fonction de AUDIO_BUF_SIZE et de la fréquence d'échantillonnage.**

La latence audio serait de $\text{AUDIO_BUF_SIZE}/\text{fs}$ avec fs la fréquence d'échantillonnage.

- **Quelle est la dynamique (à estimer en dB) offerte par une quantification sur 16 bits (int16_t) ? quelle dynamique offre à contrario un encodage sous forme de float (32 bits au format IEEE) ?**

(rq : provient de $20 \log_{10} 2^{N\text{bits}}$)

Une quantification sur 16 bits offre une dynamique d'environ 96,32 dB. Un encodage en virgule flottante sur 32 bits (IEEE 754 à simple précision) offre une dynamique d'environ 193 dB.

- **Que représentent les formats q15_t et q31_t ? Ils sont notamment utilisés dans la bibliothèque DSP-CMSIS (FFT, filtres, etc)**

Le "q" signifie "quantized", et 15 bits sont utilisés pour représenter la partie fractionnaire du nombre. Le bit restant est utilisé pour le signe (positif ou négatif). Pareil pour le second format mais cette fois 31 bits sont utilisés pour le nombre et un pour le signe pour un total de 32 bits.

3.2 Tester l'influence de quelques paramètres audio

Lorsque l'on augmente le paramètre AUDIO_BUF_SIZE, la latence augmente également (pour une taille de 2048 la latence s'entends beaucoup plus que pour 512 qui nous semble instantanée) et inversement quand on réduit le paramètre.

Pour la qualité lorsque l'on réduit la fréquence d'échantillonnage la qualité réduit drastiquement également. Cependant nous observons un autre phénomène, la fréquence d'échantillonnage a l'air d'impacter le gain car il est en moyenne plus bas d'une petite vingtaine de dB.

Finalement nous modifions la quantification tout d'abord en passant à 0 les 4 derniers bits, cela entraîne une réduction de l'information. Puis nous décalons de 4 bits l'information (ainsi nous sommes sur 12 bits) et cela réduit fortement l'amplitude (et le gain) du signal.

3.3 Un premier algorithme d'effet audio : "the simple delay"

$x[n]$ l'échantillon d'entrée et $y[n]$ l'échantillon de sortie à l'instant n , alors le retard introduit est déterminé par la période d : $y[n] = x[n] + fb * y[n-d]$.

Cependant ici nous travaillons avec 2 buffers, SDRAM qui est très grand ainsi que Audio (celui sur lequel nous travaillons jusqu'à la principalement). Le but est de tourner

selon i sur SDRAM et n pour Audio et ainsi utiliser la grande possibilité de stockage de SDRAM. On remplit donc SDRAM grâce aux informations in[n] et on revoit en out[n] un mixe entre in[n] et un signal avec un délai d'une durée choisie (une seconde de délai correspond à 16000 de décalage par exemple).

```

int i =0;
static void processAudio(int16_t *out, int16_t *in) {
    int Delay = 0.5 * 16000;

    LED_On(); // for oscilloscope measurements...
    for (int n = 0; n < AUDIO_BUF_SIZE; n++){
        writeInt16ToSDRAM(in[n], (i+Delay)%100000);
        out[n] = in[n] + 0.5* readInt16FromSDRAM(i);
        i += 1;
        i = i % 100000;
    }
    LED_Off();
}

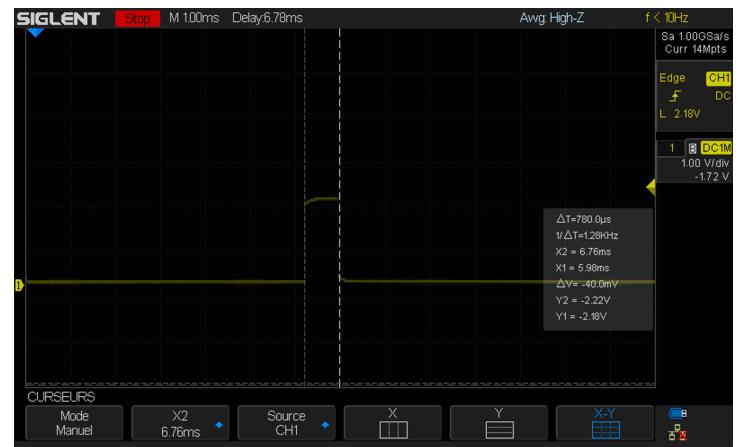
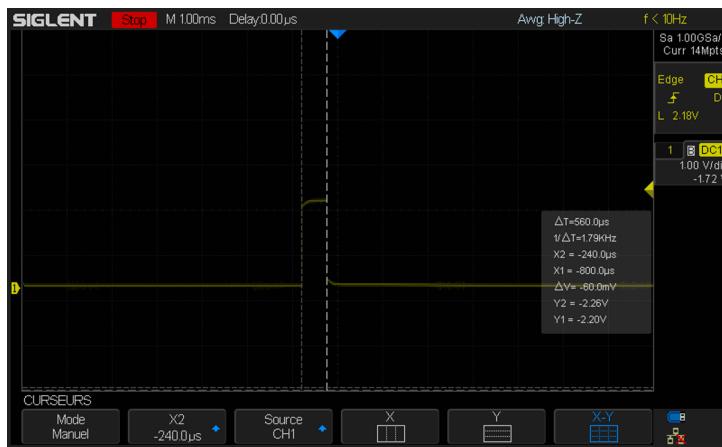
```

3.4 Analyse expérimentale des performances

On ajoute la led qui s'allume au début de notre fonction de delay et on l'enlève après afin de pouvoir observer le temps consommé par notre bloc de code.

On se met en q15_t :

Lorsque l'on regarde notre code tel quel il met 780 microsecondes, mais lorsque l'on utilise l'option “Optimized for speed” cela se réduit à 560 microsecondes.



On se met en float :

Lorsque l'on regarde notre code tel quel il met 670 microsecondes, mais lorsque l'on utilise l'option “Optimized for speed” cela se réduit à 480 microsecondes.

A la suite de ces mesures, nous décidons donc de travailler en float en mode de compilation “Optimized for speed” car nous voulons faire du temps réel.

4.1 Utilisation des fonctionnalités de CMSIS-RTOS

Nous allons avoir 2 tâches principales :

- Default Task
- UI Task

La Default Task sera celle avec la plus haute priorité, son objectif est d'enregistrer le signal audio et de le traiter. La seconde, la UI task, sera moins prioritaire, elle permettra d'afficher le résultat de notre traitement du signal audio.

Dans notre code, la Default Task (correspondant au traitement audio) est grâce aux fonctions HAL_SAI_RxCpltCallback et HAL_SAI_RxHalfCpltCallback. Ceci correspond à un appel lorsque le transfert DMA est complété.

La UI Task (en charge de la partie graphique) est appelée dans la fonction AudioLoop2 dès que le traitement audio est fini.

4.2 Calcul en temps réel de la TF d'une trame audio et affichage

On souhaite afficher en temps réel la TF de l'audio entrant dans le micro de la STM32. Nous allons utiliser les fonctionnalités de CMSIS-RTOS vues précédemment. Nous ferons continuellement en priorité la default task et une fois le traitement audio terminé, on appelle la UI Task afin d'afficher la TF. A chaque appel de la UI Task :

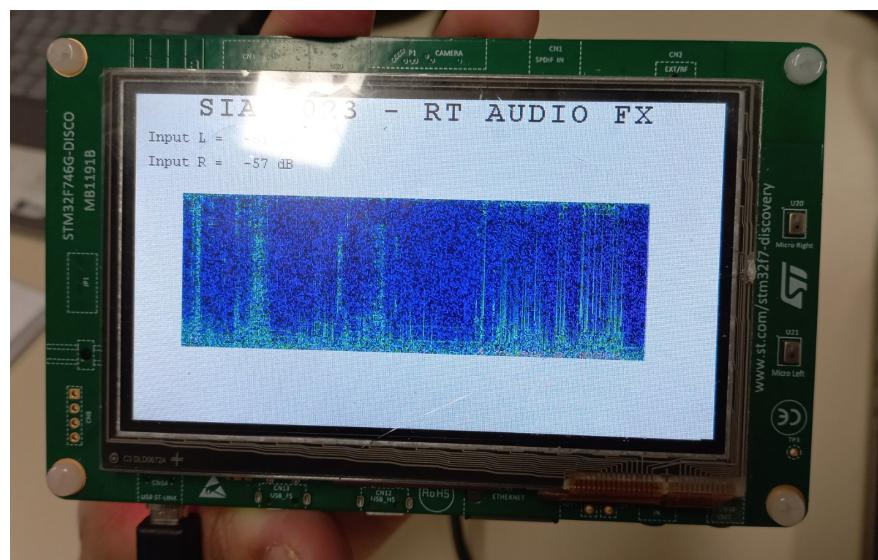
- On affiche l'amplitude (on pourrait le faire moins souvent cependant afin de libérer le processeur mais cela reste intéressant de l'avoir continuellement en temps réel).
- On affiche une "ligne" de pixel colorés associée au calcul de la TF.

Pendant ce temps la default task :

- Traite un 1/2 buffer en calculant la TF

On pourrait représenter les tâches comme cela :

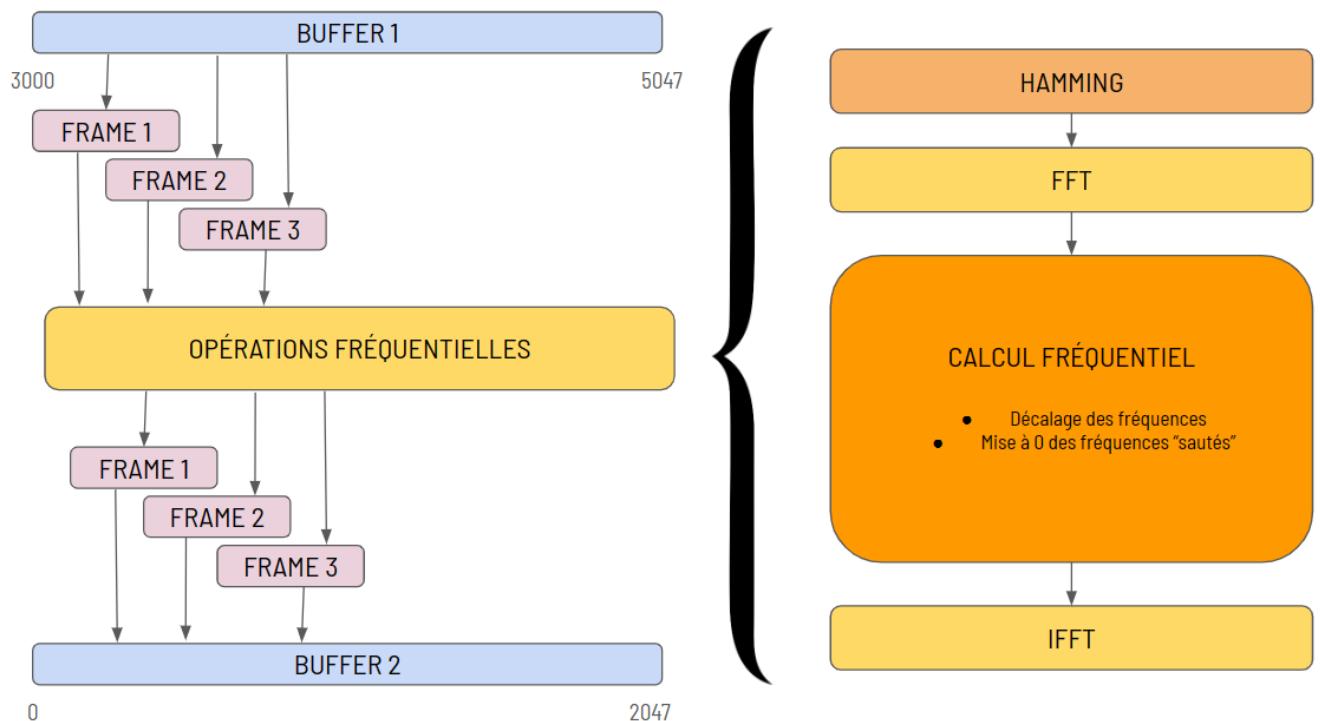
Remplissage du 1/2 buffer (DMA)						
Default task						
UI task						



L'écran fait au total 480 x 272 pixels (de 4,3 pouces), on a choisi de centrer notre affichage, pour ça on le commence à 40 pixels et on le finit à 440 pixels. Cela permet un affichage sur 400 pixels centrés.

5.3 Vocoder de phase

Le vocodeur de phase est un effet complexe utilisé en traitement du signal audio pour effectuer deux opérations non triviales sur un signal audio : **la modification de la hauteur tonale (pitch) sans altérer la durée**. En effet, cela est loin d'être trivial car la hauteur tonale et la durée d'un signal audio sont techniquement liées. Si on augmente la fréquence d'échantillonnage, le signal semblera plus aigu, mais il sera également lu plus rapidement, ce qui entraînera une réduction de la durée du signal. À l'inverse, si on diminue la fréquence d'échantillonnage, le signal semblera plus grave, mais il sera également lu plus lentement, ce qui rallongera la durée du signal.



Nous avons suivi les étapes suivantes pour créer notre vocoder sur STM32 :

1. Nous écrivons dans notre **BUFFER 1**.
2. Nous prenons des trames de taille **FRAME_SIZE**, qui se chevauchent les unes avec les autres.
3. Nous appliquons à nos trames une fenêtre de Hamming.
4. Nous effectuons la **FFT** (Transformée de Fourier rapide).
5. Nous calculons les nouvelles fréquences avec un décalage que nous avons décidé ($a=2$ correspond à un ajout d'octave et $a=0,5$ correspond à une diminution d'octave) et mettons à zéro les fréquences 'sautées' en raison de la multiplication.
6. Nous effectuons la **FFT inverse**.
7. Nous récupérons nos trames et additionnons celles qui se chevauchent dans le **BUFFER 2**.
8. Il nous suffit de lire nos valeurs de sortie dans le **BUFFER 2**.

Nous avons réalisé une première version avec des buffers linéaires en entrée et sortie mais ceci entraîne des coupures et donc rend la voix “robotique”.

Dans un deuxième temps, nous avons essayé d'utiliser des buffers circulaire mais malgré nos efforts pour implémenter cette méthode, nous n'avons pas réussi à avoir des résultats concluants.

Il existe encore plusieurs autres méthodes d'implémentation de vocoder que nous n'avons pas pu aborder.