GHILENE RAYANE
LASSO JARAMILLO NICOLAS

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

The goal of this lab: *This 16-hour mini-project aims to study various aspects of the real-time audio signal acquisition, processing, and playback chain on the STM32F746-Discovery board. The ultimate goal is to implement a real-time audio effect (reverberation, dynamic compressor, etc).*

1. Quelques pistes pour démarrer (première séance)

1.1 Quelques questions a se poser... pour savoir estimer rapidement les performances d'un système audio temps-réel

- *Que dit le theorème de shannon?*

The sampling frequency (Fs) must be bigger than the double of the maximum frequency of the signal (Fmax)

$$F_s > 2 * F_{max}$$

- *Pour du traitement de la parole, quelle fréquence d'échantillonnage préconisez-vous ?*

The maximum frequency of speech sounds is generally below 8 kHz. A sampling rate of 16 kHz can well represent human vocal characteristics and is commonly used for speech and telephony.

- *Idem pour du signal de musique ?*

For the music signal, the sampling frequency for standard CD quality is 44.1 kHz (it may vary depending on the quality of the recording)

- *Qu'est-ce que le "double-buffering" ?*

Double-buffering is a technique used to ensure that data is correctly managed between the two buffers and that processing can continue uninterrupted (in this tp we will use double buffering by splitting our buffer in half, we wait until the first half is full to process it and then move to the second half).

- *Qu'est-ce que le DMA transfer ?*

DMA transfer (Direct Memory Access transfer) is a hardware technique that allows a device (peripheral) to directly access memory without constant intervention from the central processing unit (CPU). This frees up the CPU for other tasks while the device transfers data.

- *A 48kHz de fréquence d'échantillonnage, de combien de temps dispose-t-on pour traiter un échantillon audio ?*

For audio sampling at 48 kHz, the sampling period is the inverse of the sampling frequency, i.e.:

$$T_e = \frac{1}{F_e} = \frac{1}{48,000Hz} \approx 20,83ms$$

This means that for a sampling frequency of 48kHz, an audio sample is captured every 20.38ms.

- *A titre de comparaison, quelle est la durée moyenne d'une instruction (par exemple une addition de deux registres) sur un processeur STM32F7 cadence à 200MHz ?*

Durée moyenne d'une instruction = (Nombre de cycles d'horloge par instruction) / Fréquence d'horloge

La **latence audio** est definie comme le temps séparant l'arrivée d'un échantillon sur l'entree 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) :

- Quelle est la dynamique (a estimer en dB) offerte par une quantification sur 16 bits (int16_t) ?

The dynamic (dB) is defined as the difference between the minimum audible signal and the loudest one. in the case of a quantification of 16 bits the dynamic is calculated using the following equation:

$$\begin{aligned}&\text{Dynamic(dB)} = 20 \cdot \log_{10}(2^n)\\&\text{In the case of 16 bits}\\&\text{Dynamic(dB)} = 20 \cdot \log_{10}(2^{16})\\&\text{Dynamic(dB)} \approx 96.32\,\text{dB}\end{aligned}$$

This means that the dynamic for quantification over 16 bits is around 96.32 dB

- Quelle dynamique offre a contrario un encodage sous forme de float (32 bits au format IEEE) ?

We use the same formula we used previously

$$\text{Dynamic(dB)} = 20 \cdot \log_{10}(2^n)$$
$$\text{In the case of 32 bits}$$
$$\text{Dynamic(dB)} = 20 \cdot \log_{10}(2^{32})$$
$$\text{Dynamic(dB)} \approx 180.6\,\text{dB}$$

2. Tester l'influence de quelques paramètres audio

- *En modifiant le paramètre AUDIO_BUF_SIZE (cf. préambule du fichier audio.c), estimer l'influence de la taille du buffer DMA sur la latence perçue?*

```
// whole sample count in an audio frame: (beware: as they are interleaved stereo samples, true audio frame duration is given
#define AUDIO_BUF_SIZE   ((uint32_t)512)
/* size of a full DMA buffer made up of two half-buffers (aka double-buffering) */
#define AUDIO_DMA_BUF_SIZE   (2 * AUDIO_BUF_SIZE)
```

The AUDIO_BUFF_SIZE is initialized at 512 in the audio.c file
using a smaller buff size results in a smaller latency

$$\frac{\text{Buffer Size}}{\text{Sampling Frequency}} = Latency$$

$$\frac{512}{160000} = 0.032s$$

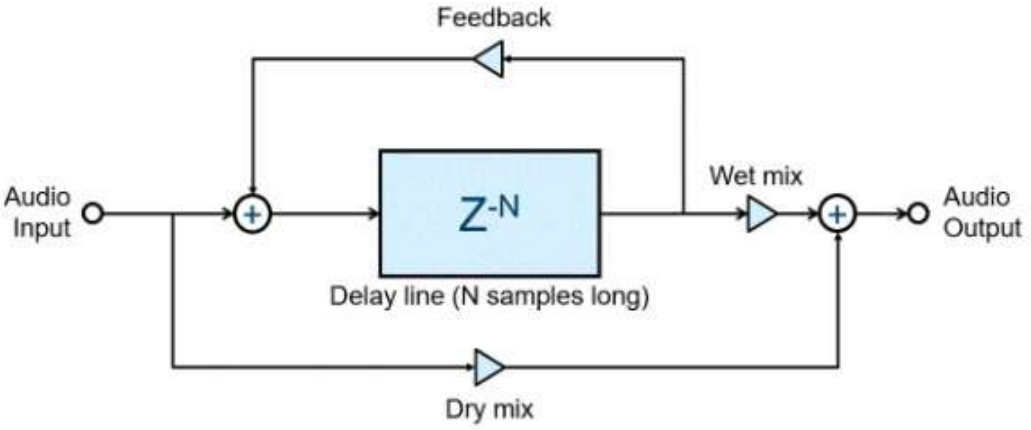
- *Quel est l'influence de la fréquence d'échantillonnage sur la qualité perçue ? (on se reportera au fichier source main.c, ligne 861, où l'on peut modifier le paramètre hsai BlockA 2.Init.Audio Frequency*

In our case, the audio frequency is set to 16kHz

When testing smaller sampling frequencies the quality of the audio deteriorates and the delay is aggravated. Using a higher frequency increases the quality of the audio.

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

The first algorithm we implemented was a simple delay:



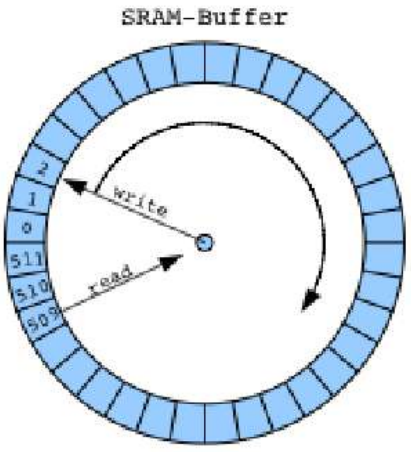
This effect consisted of two steps:

1. We read the audio using the method readFromAudioScratch(pos)
2. We calculate the output audio and write it using the method writeToAudioScratch(pos)

these methods allow us to, read and write in an external SDRAM which is useful for applying effects on our audio signals

To do so we will use a circular buffer where we will store the samples (the buffer must at least be bigger than the delay factor)

Here we used a ring buffer so that we don't exceed the buffer size when



Ring buffer size 512

2. Use of RTOS

An RTOS, a Real-Time Operating system, is a multitask OS useful for the real-time applications of effects (modifications) to our signals by scheduling them.
In our case, it is useful for plotting the spectrogram of our audio in real-time without altering the audio reconstruction and testing our effects.

We defined two main methods: **defaultTask** used for real-time audio processing and **uiTask** Which is used for the graphic manipulations of our microcontroller.
To inform our microcontroller about the priority order of these methods we used the two methods, osSignalWait(signal, millisec) and osSignalSet(task, signal).

2.4 Fourier Transform plot

To Visualize the spectrogram of our audio on the board, we calculate the Fourier transform of each audio sample and then plot them.
We used "Double buffering" by splitting our buffer in half. We wait until the first half of the buffer is full, and then we process it, (calculate the Fourier transform/ apply audio effects) while we wait for the second half to be filled.

Fourier Transform
From the library CMSIS-DSP, we imported the two methods arm_fft_fast_f32(), which calculates the Fourier transform, and arm_cmplx_mag_f32() which returns the Fourier spectrum magnitude, here's how we structured our code:

```
0
1  void calculateFFT(int16_t *in){
2      for (int i = 0; i < FFT_Length; i++){
3          aFFT_Input_f32[i] = in[i];
4      }
5      arm_rfft_fast_f32(&FFT_struct, aFFT_Input_f32, aFFT_Output_f32, 0);
6      arm_cmplx_mag_f32(aFFT_Output_f32, aFFT_Input_f32, FFT_Length/2);
7      osSignalSet(uiTaskHandle, 0x0003);
8  }
```

Method implemented for calculating the Fourier spectrum

when calling this method, calculateFFT, we first need to initialize using the arm_rfft_fast_instance_f32 and arm_rfft_fast_init_f32()

Fourier spectrum plot

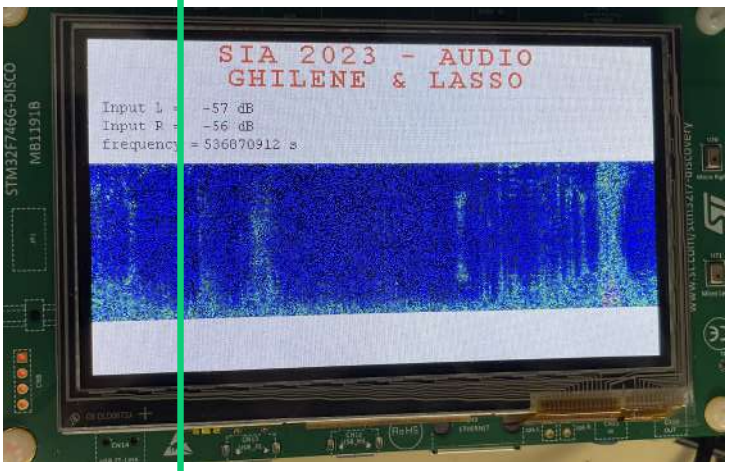
Using the methods already implemented in the disco_lcd.c file we plotted the Fourier spectrum of our audio on the card, here's how we did it:

```
/* Infinite loop */
for(;;)
{
    z++;
    osSignalWait(0x0003, osWaitForever);
    uiDisplayInputLevel(inputLevelL_cp, inputLevelR_cp);
    for(int y = FFT_Length/2 + y1; y > y1; y--){
        LCD_DrawPixel_Color(x + time, y, aFFT_Input_f32[(FFT_Length/2 + y1) - y]);

    }
```

Fourier Spectrum plot on card

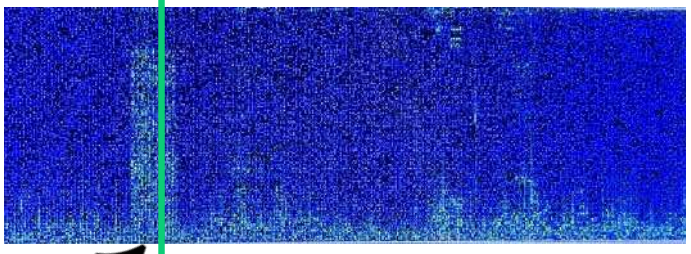
(This code was implemented in the uiTask)
Using the LCD_DrawPixel_Color, we were able to plot the Fourier transform for each sample of our signal, here's how it looked like:



Fourier Spectrum plot

When plotting the Fourier Spectrum on the card, we were able to prove that it was auto-updating each time it reached the end of the screen it restarts from the left, except when plotting the values we found out that we had issues when calculating our Fourier transform.

When testing distinct audio frequencies we were able to visualize continuous forms of the spectrum



As we can see, when testing a continuous audio frequency we get a homogeneous spectrogram plot

Implementation of effects:

1. Echo

We started by implementing a simple Echo effect

```
8
9⊖static void echo(int16_t *out, int16_t *in) {
0      float K;
1      float DRY = 0.4;
2      float WET = 0.6;
3      float fb = 0.4;
4      int delay = 50 * AUDIO_BUF_SIZE;
5      for (int n = 0; n < AUDIO_BUF_SIZE; n++){
6          K = in[n] + fb * readFromAudioScratch(pos);
7          out[n] = DRY * in[n] + WET * K;
8          writeToAudioScratch(out[n],pos);
9          if (pos < delay) {pos = pos + 1;}
0          else{pos = 0;}
1      }
2  }
```

Echo Implementation

2. Noise Gate

Using a predefined threshold, the noise gate method blocks all the audio samples that are below this threshold.

```
3
4⊖static void noise_gate(int16_t *out, int16_t *in) {
5      float threshold = 1000;
6      for (int n = 0; n < AUDIO_BUF_SIZE; n++){
7          if (abs(in[n])<threshold){
8              out[n] = 0;
9          }
0          else {out[n] = in[n];}
1      }
2  }
```

very naïve version

Noise Gate Implementation

When testing different thresholds, we visualize a difference in the spectrogram.

- High threshold: we don't hear ambient noise, but we can hear normal conversations
- Low threshold: we can hear some noise
-

Conclusion

This project allowed us to discover different audio processing techniques using an STM32 microcontroller.

Due to time restrictions, we couldn't go all the way through in the Tp, but at least we got the opportunity to learn how to use an RTOS to ensure that all of our tasks were correctly executed, learn how to use a ring buffer, and how to implement audio effects.