TP5 : Les Interruptions

Par Curtis Martelet

Licence EEA ; HAE404

Table des matières

23/05/2022

I. Introduction aux interruptions

Avant tout, ces TP ont été réalisé sur une carte microcontrôleur STM 32L152C-Discovery.

Tous les bouts de codes & fonctions utilisées sont disponibles dans l'annexe.

A. Interruption Externes

Une interruption est une fonction électronique du microcontrôleur.
A la différence d'une fonction que l'on programme, une interruption est conçue pour que l'on n'ait pas à attendre.

Pour débuter, nous souhaitons piloter les DELs PB**6** et **PB7** de la carte à l'aide du bouton **PB0**. On utilisera une interruption pour réaliser cette fonction.

Le changement d'état des DELs se fera sur le front montant et descendant de **PB0**.

A l'aide du registre du microcontrôleur (stm32l1xx.h), on trouve que c'est *EXTI_Trigger* qui définit l'activation de l'interruption en fonction du front que l'on veut :

- Front Descendant : *EXTI_Trigger_Falling*.
- Front Montant : *EXTI_Trigger_Rising*.
- Front Montant/Descendant : *EXTI_Trigger_Rising_Falling*.

On utilise la fonction *IRQ_EXTI0_Config* pour configurer l'interruption :

```
void IRQ_EXTI0_Config() { // Configure GPIO pour l'Interrupteur
    /* Activer GPIOA */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA,ENABLE);
    GPIO_InitTypeDef gpio_a;
    GPIO_StructInit(&gpio_a);
    gpio_a.GPIO_Mode = GPIO_Mode_IN;
    gpio_a.GPIO_Pin = GPIO_Pin_0; // Pin 0
    GPIO_Init(GPIOA,&gpio_a);

    /* Activer SYSCFG sur APB2 pour permettre l'utilisation des interruptions
externes */
    RCC_APB2PeriphClockCmd (RCC_APB2Periph_SYSCFG,ENABLE);
    /* Declarer PA0 comme source d'interruption */
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA,EXTI_PinSource0);
    EXTI_InitTypeDef EXTI0_params;
    EXTI_StructInit(&EXTI0_params);
    EXTI0_params.EXTI_Line = EXTI_Line0; // Pin 0
    EXTI0_params.EXTI_LineCmd = ENABLE;
    EXTI0_params.EXTI_Trigger = EXTI_Trigger_Rising_Falling; // Montant-Descendant
    EXTI_Init(&EXTI0_params);

    /* Activer l'interruption dans le NVIC */
    NVIC_InitTypeDef nvic;
    NVIC_Init(&nvic);
    nvic.NVIC_IRQChannel = EXTI0_IRQn;
    nvic.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&nvic);
}
```

Code 1. Configuration de l'interruption

Une fois l'interruption configurée, il s'agit maintenant de définir ce qu'elle réalisera.

La fonction *EXTI0_IRQHandler* est faite pour cela : elle s'exécute quand l'interruption a lieu.

On souhaite donc inverser l'état d'une DEL que l'on aura choisie. *GPIO_ToggleBits* est la fonction du GPIO qui permet d'inverser l'état d'un pin. Il est configuré en fonction du coté (A ou B) et du numéro du pin.

```
void EXTI0_IRQHandler(void) // Le code a exécuter quand il y a interruption.
{
    if(EXTI_GetITStatus(EXTI_Line0) != RESET) // Permet de faire une seule fois
l'interruption au lieu de la répéter.
    {
        /* Clear the EXTI line 0 pending bit (enlève le flag) */
        EXTI_ClearITPendingBit(EXTI_Line0);
        GPIO_ToggleBits(GPIOB,GPIO_Pin_7); // Inverse état du pin 7
        GPIO_ToggleBits(GPIOB,GPIO_Pin_6); // Inverse état du pin 6
    }
}
```

Code 2. Interruption Externe

Le *if* évite que l'interruption se réalise plusieurs fois d'affilé, par exemple dans le cas du rebond d'un bouton.

Avec cette interruption, on inversera les états des DELs **PB6** et **PB7** sur la pression et la relève du bouton.

Pour que les états des DELs soient inversés l'un par rapport à l'autre, il suffit de définir l'état de base d'une des DELs comme étant allumé :

On ajoute, avant la boucle du *main*, la fonction *GPIO_SetBits* qui se paramètre comme *GPIO_ToggleBits*. Cette fonction "set" à 1 le pin configuré, ce qui revient à alimenter la DEL.

La fonction main de cet exercice devrait donc ressembler à cela :

```
int main(void) {
    // # LED sur PB7 et PB6
    /* Activer GPIOB sur AHB */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB,ENABLE);
    GPIO_InitTypeDef gpio_b;
    GPIO_StructInit(&gpio_b);
    gpio_b.GPIO_Mode = GPIO_Mode_OUT; // Sortie tout-ou-rien
    gpio_b.GPIO_Pin = GPIO_Pin_7|GPIO_Pin_6; // Pin 7 et 6
    GPIO_Init(GPIOB,&gpio_b);
    GPIO_SetBits(GPIOB,GPIO_Pin_7); // Set à 1 le pin 7 du coté B. La DEL est
allumée.
    IRQ_EXTI0_Config(); // configuration de l'interruption

    while(1) {// Rien car on attend l'interruption}
}
```

Code 3. Configuration DELs

B. Interruptions Périodique

Si le premier exercice ordonnait une interruption sur l'appui d'un bouton (donc une commande extérieure), ce second exercice se basera sur l'horloge du microcontrôleur pour commander les interruptions.

On souhaite faire clignoter la DEL toutes les 500ms.
On va utiliser le **Timer 2** de la carte pour mesurer ce temps.

```
void TIM2_IRQ_Config()
{
    /*Activer TIM2 sur APB1 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);
    /* Configurer TIM2 a 500 ms */
    TIM_TimeBaseInitTypeDef timer_2;
    TIM_TimeBaseStructInit(&timer_2);
    timer_2.TIM_Prescaler = 16000-1; // Prescalaire et Période ont au final le
même résultat
    timer_2.TIM_Period = 500; // Cependant, on utilise Prescalaire pour compter le
temps et Période pour mesurer le temps.
// Fhorloge = 44 000 Hz ; CPU = 16*10^6
// Thorloge = Modificateur/16x10^6 <=> 1/44000 = TIM_Period/16x10^6 <=> TIM_Period
= 16x10^6/44000 = 363.6 (on arrondira au supérieur)
// On retrouve 2kHz, la moitié de la fréquence prévue, à cause du fonctionnement
de l'horloge
    TIM_TimeBaseInit(TIM2,&timer_2);
    TIM_SetCounter(TIM2,0);
    TIM_Cmd(TIM2, ENABLE);

    /* Associer une interruption a TIM2 */
    TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);

    NVIC_InitTypeDef nvic;
    /* Configuration de l'interruption */
    nvic.NVIC_IRQChannel = TIM2_IRQn;
    nvic.NVIC_IRQChannelPreemptionPriority = 0;
    nvic.NVIC_IRQChannelSubPriority = 1;
    nvic.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&nvic);
}
```

Code 4. Configuration du Timer

La fréquence du microcontrôleur étant de 16Mhz, on choisit un prescaler de 16000 : *TIM_Period* s'incrémentera toutes les millisecondes.
Voulant une interruption toutes les 500ms, il suffira de définir la valeur de *TIM_Period* à 500.

Le code de l'interruption est identique à celle des interruptions extérieures, au détail près qu'elle s'appelle *TIM2_IRQHandler*.

```
void TIM2_IRQHandler() {
    if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
    {
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
        GPIO_ToggleBits(GPIOB, GPIO_Pin_7); // Inversion du pin 7
    }
}
```

Code 5. Interruption périodique

Changement de Fréquence

On nous demande cette fois-ci une fréquence de 44kHz.
On décide cette fois-ci de mettre le prescaler à 0 (*TIM_Prescaler* = 0).
Fréquence voulue = 44 kHz ; Fréquence du CPU = 16 MHz.

$$T_{horloge} = \frac{(TIM\_Period + 1) \times (\text{TIM\_Prescaler} + 1)}{16 \times 10^6}$$

$\frac{1}{44\times10^3} = \frac{TIM\_Period+1}{16\times10^6}$ $\qquad$ $TIM_{Period} = \frac{16\times10^6}{44\times10^3} - 1 = 362.6$ (on arrondie au supérieur)

Une fois reconfiguré, on observe sur l'oscilloscope que la fréquence observée n'est pas 44kHz, mais 22kHz : ramené en période, 44kHz = 22.7us et 22kHz = 45.5us.

Une période de la DEL est quand elle a été allumée et éteinte une fois. Or, le code actuel fait que tous les 22.7us (44kHz), on inverse l'état de la DEL.
Ainsi, 22.7us + 22.7us = 45.5us, soit 22kHz.

Pour remédier à ce problème, on peut par exemple réduire de moitié la valeur de *timer_2.TIM_Period*e, soit 182 (on arrondie à l'inférieur) au lieu de 363.

II. Générer une tension sinusoïdale

Maintenant que l'on sait comment réaliser des interruptions périodiquement, on veut générer une tension sinusoïdale sur le pin ***PA4*** de la carte.

Cependant, le microcontrôleur n'est pas capable de lui-même de faire ce signal, c'est pourquoi nous allons l'aider en calculant en amont les valeurs qu'il générera.

A. Le Sinus

Avant toute chose, nous avons besoin de la bibliothèque ***math*** (*math.h*) pour utiliser le sinus.

Nous stockerons les valeurs du sinus dans un tableau nommé *Tension_* que l'on définira à l'aide de la fonction `malloc`.
Malloc permet d'allouer à une variable un espace qui sera calculé en fonction du type de variable présente dedans :

```
Tension = malloc(100*sizeof(float));
```

Code 6. Malloc

Tension fait maintenant la taille de 100 valeurs du type float.
Ce tableau devra être déclaré globalement pour que l'interruption puisse s'en servir.

Les préparations maintenant faites, il faut maintenant calculer le sinus.

L'équation du sinus est :

$$Tension = 511 \times \sin\left(2 \times \pi \times \frac{k}{100}\right) + 2047$$

Où k est un incrément qui ira de 0 à 99 (100 valeurs).

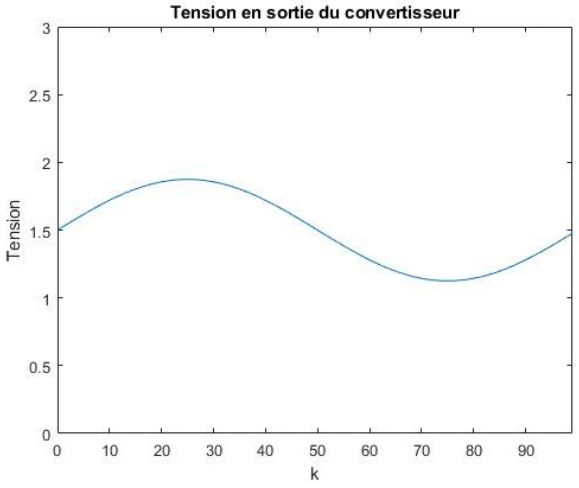

Figure 1. Fonction Tension(k)

Le début de la fonction ressemble donc à cela :

```
#include "stm32l1xx.h"
#include <math.h>

float* Tension;
int n=0;

int main(void)
{
    Tension = malloc(100*sizeof(float));
    for(int k=0;k<100;k++) {
        Tension[k] = 511 * sin(2*3.14159*k/100) + 2047;
    }
}
```

Code 7. Sinus

Le microcontrôleur ne peut générer des tensions qu'entre 0 et 3V.
On choisit donc de décaler la valeur moyenne du sinus de +1.5V.

Le convertisseur analogique à numérique de la carte fonctionne sur 12 bits : 2^12 = **4096** valeurs.

Donc pour 3V, le convertisseur est égal à 4095 ; pour 1.5V (la valeur moyenne), 2047 ; et pour 0V, 0.

C'est pour cela que l'on ajoute 2047 à l'équation, pour centrer le sinus sur 1.5V.

- La valeur minimale du sinus est donc :
  2047 - 511 = **1536**, ce qui donne une tension en sortie du convertisseur de **1.25V**.
- La valeur minimale du sinus est donc :
  2047 + 511 = **2558**, ce qui donne une tension en sortie du convertisseur de **1.87V**.

J'ai utilisé un produit en croix pour convertir les valeurs du convertisseur en tension.

Maintenant que le sinus est calculé et stocké dans la variable *Tension*, il est temps de générer le signal en sortie de la carte.
Pour ce faire, on réemploie une fonction du code du précédent exercice, **TIM2**.
On devra également utiliser le convertisseur Digital à Analogique (**DAC**).

Configuration du TIMER

Nous n'avons pas besoin de toucher à la configuration précédente de **TIM2**. Le prescaler (`_timer_2.TIM_Prescaler`) et la période (`timer_2.TIM_Period`) restent les mêmes puisque l'on garde la fréquence de 44kHz.

La seule partie qui change est l'interruption en elle-même :

```
void TIM2_IRQHandler() {
    if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
    {
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
        GPIO_ToggleBits(GPIOB, GPIO_Pin_7); // Debugger
        DAC1_Set(T[n%100]); // Converti la valeur en tension
        // On ne dépasse pas 99 valeurs : à n = 100, n%100 = 0
        n++; // Incrémentation
    }
}
```

Code 8. Interruption

On déclarer en variable globale *n*, l'incrémenter de *Tension*.

Configuration du DAC

Le ***DAC*** est un convertisseur Digital à Analogique.
Dans ce TD, il est nécessaire pour convertir les valeurs du sinus en une tension.

```
void DAC1_Config()
{
    /*Activer GPIOA sur AHB */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
    /* Configurer PA4 en mode analogique*/
    GPIO_InitTypeDef gpio_a;
    GPIO_StructInit(&gpio_a);
    gpio_a.GPIO_Mode  = GPIO_Mode_AN; // Mode Analogique
    gpio_a.GPIO_Pin = GPIO_Pin_4; // Sortie sur PIN 4
    GPIO_Init(GPIOA, &gpio_a);

    /*Activer DAC sur APB1 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);
    /* Configurer DAC1 avec parametres par defaut */
    DAC_InitTypeDef dac_1;
    DAC_StructInit(&dac_1);
    DAC_Init(DAC_Channel_1, &dac_1);
    /* Activer DAC1 */
    DAC_Cmd(DAC_Channel_1, ENABLE);
}
```

Code 9. Configuration du DAC

Maintenant que le DAC est configuré, il ne reste plus qu'à convertir les valeurs et à les émettre.

Le DAC utilise deux fonctions pour convertir puis générer une tension :

- *DAC_SetChannel1Data* et initialise la tension.
- *DAC_SoftwareTriggerCmd* émet la valeur.

La fonction *DAC1_Set* utilisé par l'interruption ressemblera donc à cela :

```
void DAC1_Set(uint16_t value)
{
    DAC_SetChannel1Data( DAC_Align_12b_R, value ); // Init convertisseur 12 bits
    DAC_SoftwareTriggerCmd( DAC_Channel_1, ENABLE ); // Emet la tension

}
```

B. Ecoute au Casque

Il n'est pas possible de brancher directement un casque à notre microcontrôleur. Le casque fonctionne avec des tensions comprises entre -1.5V et 1.5V, or, la carte n'est pas capable de générer ces valeurs négatives.

En réalisant un pont diviseur de tension et en y ajoutant un condensateur, on peut abaisser la tension moyenne de 1.5V à 0V. Ainsi, on peut câbler le casque au microcontrôleur.

Une fois que l'on branche un casque sur le circuit, le son entendu n'est pas un DO, mais une note plus aiguë.

Cela s'explique par le casque qui a besoin qu'on l'attaque avec une tension supérieure à celle fournie en sortie du condensateur.
Ce manque de tension provoque une perturbation du signal généré par la carte, et donc crée des fréquences parasites qui perturbent l'écoute :
Le son entendu est plus aiguë que ce que l'on devrait avoir.

Image 1. Signal du casque sans suiveur

En mettant un montage suiveur entre le condensateur et le casque, l'AOP joue le rôle d'une alimentation qui va fournir un signal de meilleure qualité au casque, permettant d'entendre le DO.

Image 2. Signal casque avec suiveur

II. Le Mini-Synthétiseur

A l'aide des fonctions précédemment utilisées, cet exercice demande que l'on émette du son à partir du microcontrôleur.

Dans un premier temps, on dit pouvoir modifier la fréquence du DAC sur l'appui du bouton ***PA0***. Chacune de ces fréquences correspondent à une note de musique : à chaque pression, la note produite changera.

Note	Do	Re	Mi	Fa	Sol	La	Si	Do
Fréquence	262	294	330	350	392	440	494	524
Valeur DAC	611	544	484	458	408	363	323	306

Ces valeurs sont calculées de la même façon que dans l'exercice sur les interruptions périodiques :

$$F_{note} = \frac{1}{\frac{V_{DAC} \times 100}{F_{horloge}}} \qquad V_{DAC} = \frac{F_{horloge}}{F_{note} \times 100}$$

Avec : $F_{horloge} = 16 \times 10^6 Hz$; F_{note} la fréquence de la note ; V_{DAC} la valeur DAC (valeur pour TIM_Period)

Si on multiplie V_{DAC} par 100, c'est que le sinus est constitué de 100 points.

Ces valeurs sont initialisées dans une variable nommée *note_periode*.

```
unsigned int note_periode[8] = {611,544,484,458,408,363,323,306};
```

Code 10. Tableau des notes

On récupère du premier exercice la configuration du GPIO pour faire fonctionner le bouton ***PA0***.

Cette fois-ci, on n'utilisera pas une interruption sur l'appui du bouton car cela causerait des problèmes lors de l'exécution.
On place alors dans le while :

```
int switch_status = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0); // 1 si appuie
if (switch_status == Bit_SET && prev_switch_status == 0) // Empêche la
réactivation tant que le bouton est appuyé
{
interrupteur++; // Nb d'appuie sur l'interrupteur
TIM_Cmd(TIM2, DISABLE); // Désactive le Compteur
TIM_SetCounter(TIM2, 0); // Réinitialise le Compteur
TIM_TimeBaseInitTypeDef timer_2;
TIM_TimeBaseStructInit(&timer_2);
timer_2.TIM_Prescaler = 0;
timer_2.TIM_Period = note_periode[interrupteur%8]; // Change valeur de la période
du timer
// A 8, on redescend à 0
TIM_TimeBaseInit(TIM2, &timer_2);
TIM_Cmd(TIM2, ENABLE);
}
prev_switch_status = switch_status; // Stock l'état n dans une variable n-1.
```

Code 11. Action sur pression

Le reste du code est identique à l'exercice précédent :

- Utiliser le Timer 2 pour enclencher l'interruption, TIM2_IRQHandler.
- Utiliser le DAC (et `DAC_Set`) pour générer le signal
- Stocker les valeurs du signal dans une variable globale (*T*), ainsi que son compteur (*n*).

Signal Sinus

```
void buildSinus() {
    for (int k = 0; k < 100; k++) {
        T[k] = 511 * sin(2 * 3.14159 * k / 100) + 2047;
    }
}
```

Code 12. Générer un Sinus

Image 3. Signal Sinusoïdal

Signal Triangle

```
void buildTriangle() {
    int k;
        for(int k=0; k<100/2;k++) {
            T[k] = 2047-511 + (k*1022*2/100);
        }
        for(; k<100;k++) {
            T[k] = 2047 + 511 -((k-100/2)*1022*2/100);
        }
}
```

Code 13. Générer des Triangles

Signal en Dent de Scie

```
void buildSawTooth() {
    for(int k=0; k<100;k++) {
        T[k] = 2047-511 + (k*1022/100);
    }
}
```

Code 14. Générer des Dents de Scie

Image 4. Signal en Dent de Scie

De ces 3 signaux, celui qui me parait le plus agressif à l'écoute est le signal en triangle.

En série de Fourier, ce signal se décompose en une principale harmonique (ici, 440Hz), puis des harmoniques impair (h=2k+1).
Comparé au signal en dent de scie, les harmoniques du signal triangulaire vont plus loin dans les fréquences hautes (pour un k égal, h triangulaire > h dent).[1]

[1] https://arsonor.com/comment-bien-aborder-les-eq-2-la-perception-du-timbre/

III. Mini Projet : Filtrage Numérique d'un Signal

Cette ultime partie du TP transformera le microcontrôleur en une pseudo carte de mixage audio.

Les fonctions primordiales de ce programme, *Get_Adc_quickly* et *DAC1_Set_Quickly*, sont écrites en ***BareMetal*** : s'ils étaient écrits en langage bibliothèque (avec *stm32l1xx.h*), les conversions analogique -> numérique (et inversement) ne seraient pas assez rapide pour traiter le signal.

Get_Adc_quickly

Cette fonction est utilisée pour convertir le signal arrivant du PC en des valeurs que la carte peut traiter.

```
uint16_t Get_Adc_Quickly() {
    ADC1->CR2 |= (uint32_t)ADC_CR2_SWSTART;
    while((ADC1->SR & ADC_FLAG_EOC) == 0) ;
    return (uint16_t) ADC1->DR;
}
```

Code 15. Get_Adc_Quickly

ADC1 = ADC_TypeDef (ADC1_BASE = 0x2400 + (APB2PERIPH_BASE = 0x10000 + (PERIPH_BASE = 0x40000000)))

ADC1->CR2 => *(*ADC1).CR2* : on écrit donc dans CR2, le registre de contrôle, ADC_CR2_SWSTART qui est égale à 0x40000000. Selon la bibliothèque, *ADC_CR2_SWSTART* démarre la conversion : on en conclut donc

SR est le registre d'état de ADC. Son offset est de 0. *ADC_FLAG_EOC* est égal à 0x0002 (0b0000 0000 0000 0010).
Ainsi, tant que le second bit de ADC est 0, le programme attend.

Enfin, quand celui-ci est à 1, le programme retourne DR, qui a un offset de 0x58.

DAC1_Set_Quickly

Cette fonction est l'inverse Get_Adc_quickly : elle converti les valeurs traitées en une tension pour le casque.

Faire varier *adc* revient à faire varier le volume en sortie.
Ainsi, diviser par 10 divisera le son en sortie de 10 ; multiplier par 10 aura l'effet inverse ce qui multipliera par 10 le son.

```
void DAC1_Set_Quickly(uint16_t value)
{
    static __IO uint32_t tmp = (uint32_t)DAC_BASE + (uint32_t)0x00000008 +
DAC_Align_12b_R;
    *(__IO uint32_t *) tmp = value;
}
```

Code 16. DAC1_Set_Quickly

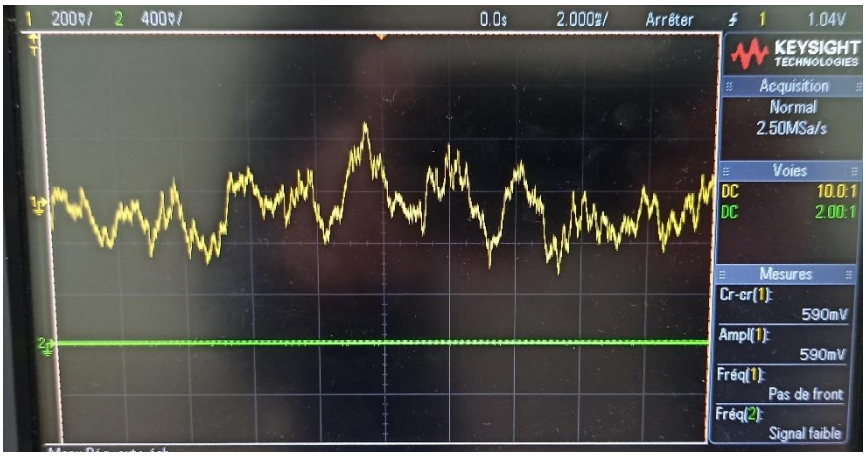

Image 5. Signal en entrée de circuit

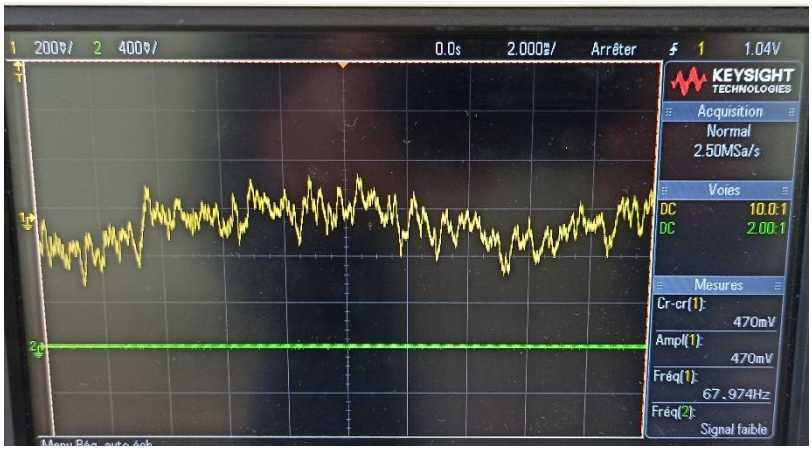

Image 6. Signal en sortie de circuit

B. Filtre Passe-Bas

Le filtre passe-bas est un filtre qui ne laisse passer que les fréquences inférieures à sa fréquence de coupure (*fc*) ; les fréquences au-dessus de celle-ci sont coupées.

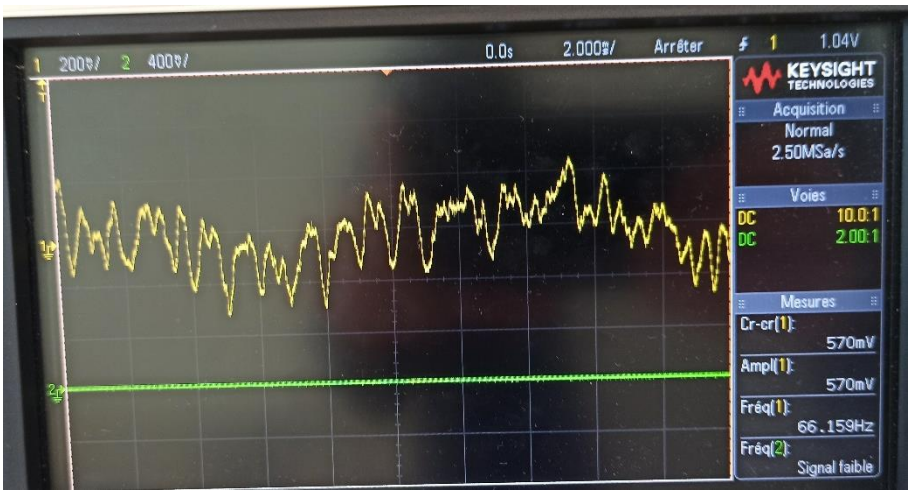

Image 7. Signal normal

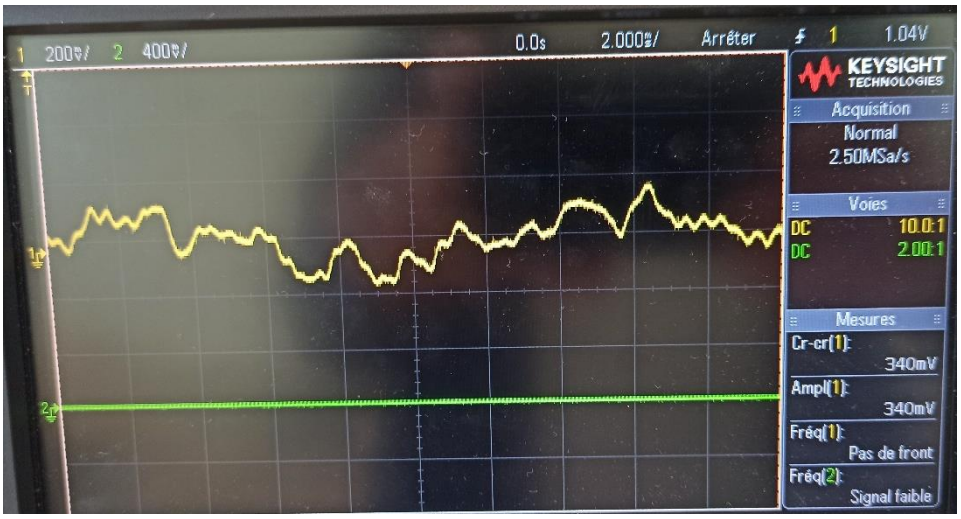

Image 8. Signal après Passe-Bas

On peut voir sur les photos de l'oscilloscope que le filtre Passe Bas supprime un très grand nombre de pic sur l'oscilloscope.

À l'oreille, le son parait plus grave (plus la fréquence est élevée, plus le son est aigu).

	Filtre Analogique	Filtre Numérique (microcontrôleur)
Avantage	- Le filtre analogique converti bien plus rapidement les signaux. - Moins cher.	- Le filtre numérique est bien plus simple à configurer. - Il propose une large variété de configuration (facteur compris entre 10 et 100 pour le passe-bas). - Très fiable.
Inconvénient	- Usure plus rapide. - Configuration rigide. - Manque de fiabilité.	- Prix. - Lent.

Après de plusieurs recherches sur des sites spécialisés[2], les seules raisons que j'ai trouvé de choisir un filtre analogique sont son prix et sa vitesse de traitement. Bien supérieur au numérique, un filtre analogique peut-être nécessaire pour le traitement de signaux trop rapide.
Cependant, les filtres numériques semblent bien meilleurs sur presque tous les autres aspects.

[2] https://www.advsolned.com/traitement-analogique-du-signal-asp-ou-le-traitement-numerique-du-signal-dsp-lequel-choisir/
https://www.petoindominique.fr/pdf/cours8.pdf

C. Compresseur Dynamique Audio

Le compresseur audio est un amplificateur dont le gain varie en fonction de la tension à son entrée.

Le but d'un compresseur audio est d'égaliser les sons : les sons les plus forts sont atténués tandis que les plus faibles sont amplifiés.

Avec ce graphique, on peut observer qu'en fonction de la tension et du coefficient de compression (*m*), la valeur du gain est radicalement différente.

Plus le coefficient est élevé, plus le gain maximal observé est important.
Également, c'est quand la tension est faible que l'on a gain qui y est important, tandis que plus la tension est grande, plus le gain tend vers 0.

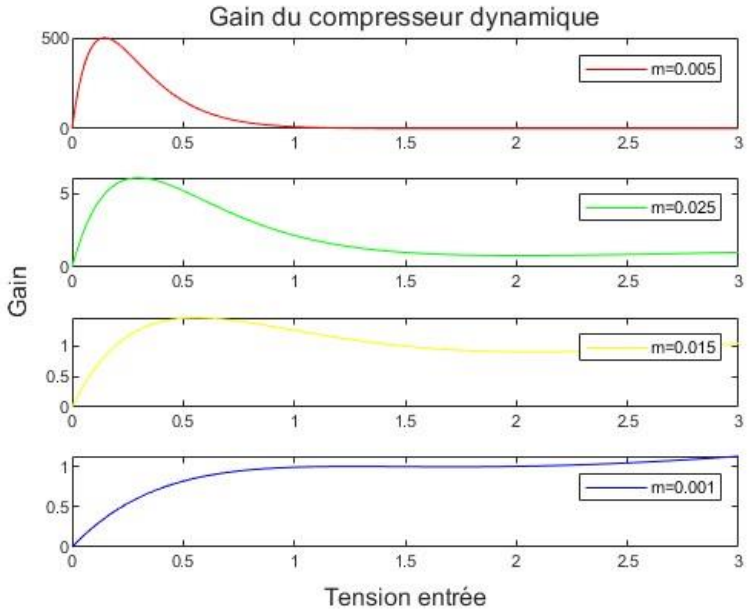

Figure 2. Gain du compresseur

L'exception se trouvant quand le coefficient est le plus faible, le compresseur ne fait que transmettre les signaux, sans vraiment les changer.

Le programme pour générer ces signaux se trouvera en annexe.

Pendant l'écoute, on a l'impression tous les sons ont le même volume.

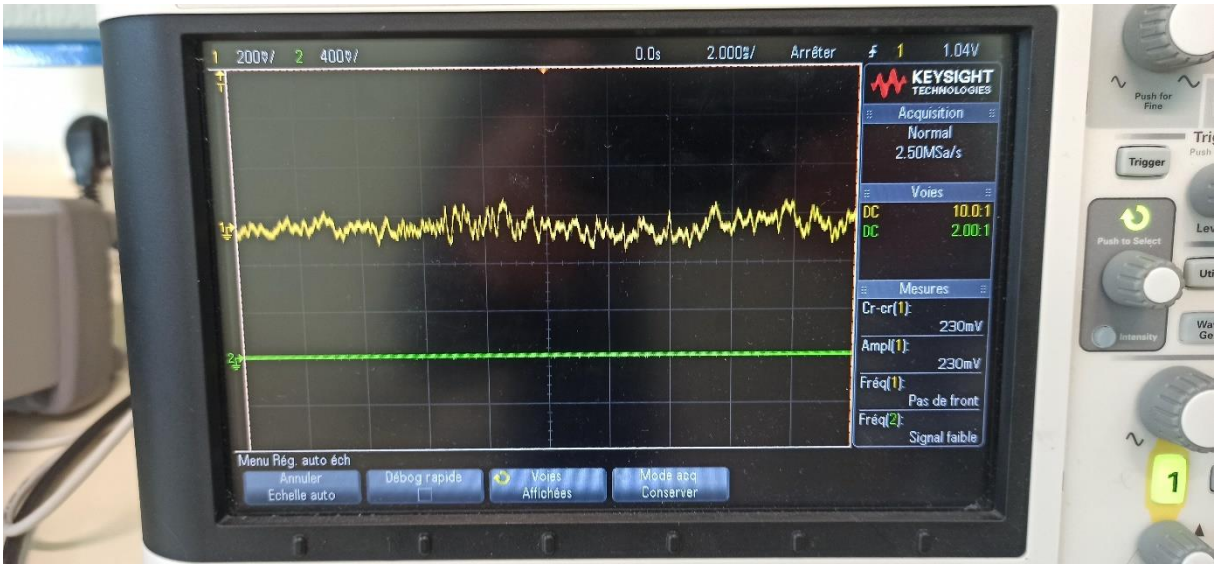

Image 9. Atténuation par Compresseur Dynamique

Un filtre linéaire est, en traitement du signal, un dispositif qui applique un opérateur linéaire à un signal d'entrée. [3]

En électronique, un filtre numérique est un élément qui effectue un filtrage avec une succession d'opérations mathématiques sur un signal discret. C'est-à-dire qu'il modifie le contenu spectral du signal d'entrée en atténuant ou éliminant certaines composantes spectrales indésirées.[4]

Le gain du compresseur varie en fonction de la tension en entrée.

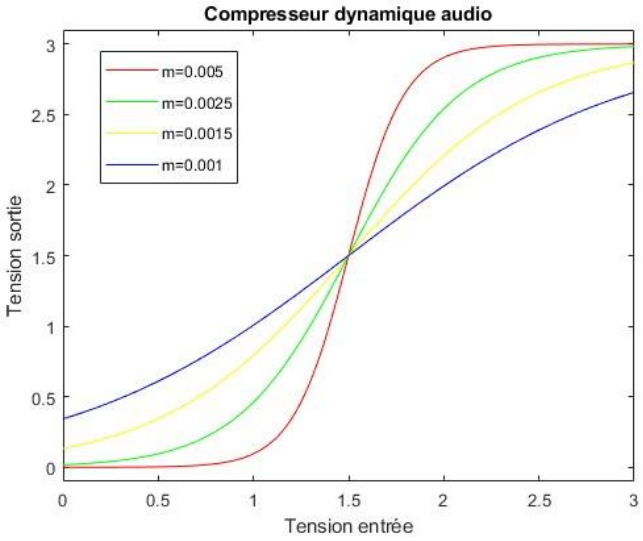

Figure 3. Compresseur dynamique

Comme donné dans la définition ci-dessus, un gain linéaire est de la forme $S = \alpha A + B$.
Par exemple, le gain du filtre passe-bas est :

$$H(jw) = \frac{K}{1 + j\frac{\omega}{\omega_c}}$$

Or, le compresseur dynamique est un filtre numérique.
Comme le montre la figure 3, ce dernier n'applique pas le même gain α à toutes les valeurs du signal.
De plus, à la différence de

[3] http://www.traitement-signal.com/filtre_lineaire.php
[4] http://www.traitement-signal.com/filtre_numerique.php

IV. Annexe

```
%--- Sinus ---%
k = linspace(0,99,100000);
y = 0.374267578*sin(pi*k*2/100) + 1.5
% Affichage
plot(k,y)
xlabel('k')
ylabel('Tension')
title("Tension en sortie du convertisseur")
axis([0 99 0 3])

%--- Compresseur Audio ---%
% Coefficient
m1 = 0.005
m2 = 0.0025
m3 = 0.0015
m4 = 0.001
% Calcul Sortie
E = linspace(0,3,1000);
Eb = 4096*E./3;
S1 = (4095)./(1+exp(-m1*(Eb-2047)));
S2 = (4095)./(1+exp(-m2*(Eb-2047)));
S3 = (4095)./(1+exp(-m3*(Eb-2047)));
S4 = (4095)./(1+exp(-m4*(Eb-2047)));
% Affichage des courbes
plot(E,S1*3/4096,'r',E,S2*3/4096,'g',E,S3*3/4096,'y',E,S4*3/4096,'b')
legend('m=0.005','m=0.0025','m=0.0015','m=0.001')
xlabel('Tension entrée')
ylabel('Tension sortie')
title('Compresseur dynamique audio')
axis([0 3 -0.1 3.1])
% Calcul des gains
G1 = E./S1b;
G2 = E./S2b;
G3 = E./S3b;
G4 = E./S4b;
% Affichage des gains
figure
t = tiledlayout(4,1);
nexttile, plot(E,G1,'r')
legend('m=0.005')
nexttile, plot(E,G2,'g')
legend('m=0.025')
nexttile, plot(E,G3,'y')
legend('m=0.015')
nexttile, plot(E,G4,'b')
legend('m=0.001')
% Configuration de la vue
title(t,'Gain du compresseur dynamique')
xlabel(t,'Tension entrée')
ylabel(t,'Gain')
```

```c
#include "stm32l1xx.h"

void IRQ_EXTI0_Config();

int main(void)
{
    // # LED sur PB7
    /* Activer GPIOB sur AHB */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB,ENABLE);
    /* Configurer PB7 comme sortie tout-ou-rien */
    GPIO_InitTypeDef gpio_b;
    GPIO_StructInit(&gpio_b);
    gpio_b.GPIO_Mode = GPIO_Mode_OUT;
    gpio_b.GPIO_Pin = GPIO_Pin_7|GPIO_Pin_6;
    GPIO_Init(GPIOB,&gpio_b);
    /* allumer la LED*/
    GPIO_SetBits(GPIOB,GPIO_Pin_7);
    // configuration de l'interruption
    IRQ_EXTI0_Config();

    while(1) {

    }
}

// callback pour l'interruption externe EXTI0_IRQ
void EXTI0_IRQHandler(void) // Le code a executer quand il y a interruption.
{
    if(EXTI_GetITStatus(EXTI_Line0) != RESET) // Permet de faire une seule fois
  l'interruption au lieu de la répéter.
    {
        /* Clear the EXTI line 0 pending bit (enlève le flag) */
        EXTI_ClearITPendingBit(EXTI_Line0);
        GPIO_ToggleBits(GPIOB,GPIO_Pin_7); // Inverse état du pin 7
        GPIO_ToggleBits(GPIOB,GPIO_Pin_6); // Inverse état du pin 6
    }
}

// ### EXTI0 sur PA0
// Configuration
void IRQ_EXTI0_Config()
{
    // # Interrupteur
    /* Activer GPIOA sur AHB */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA,ENABLE);
    /* Configurer PB7 comme entree tout-ou-rien */
    GPIO_InitTypeDef gpio_a;
    GPIO_StructInit(&gpio_a);
    gpio_a.GPIO_Mode = GPIO_Mode_IN;
    gpio_a.GPIO_Pin = GPIO_Pin_0;
    GPIO_Init(GPIOA,&gpio_a);

    /* Activer SYSCFG sur APB2
     * pour permettre l'utilisation des interruptions externes */
    RCC_APB2PeriphClockCmd (RCC_APB2Periph_SYSCFG,ENABLE);
    /* Declarer PA0 comme source d'interruption */
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA,EXTI_PinSource0);
    /* Param. des signaux qui declencheront l'appel de EXTI0_IRQHandler()
     * autrement dit on parametre les signaux associes a la "ligne 0"
     * ici : sur front montant ("Trigger_Rising")
     */
    EXTI_InitTypeDef EXTI0_params;
```

```
62      EXTI_StructInit(&EXTI0_params);
63      EXTI0_params.EXTI_Line = EXTI_Line0;
64      EXTI0_params.EXTI_LineCmd = ENABLE;
65      EXTI0_params.EXTI_Trigger = EXTI_Trigger_Rising_Falling;
66  // Front Descendant : EXTI_Trigger_Falling
67  // Front Montant : EXTI_Trigger_Rising
68  // Front : EXTI_Trigger_Rising_Falling
69
70      EXTI_Init(&EXTI0_params);
71
72      /* Activer l'interruption dans le NVIC */
73      NVIC_InitTypeDef nvic;
74      NVIC_Init(&nvic);
75      nvic.NVIC_IRQChannel = EXTI0_IRQn;
76      nvic.NVIC_IRQChannelCmd = ENABLE;
77      NVIC_Init(&nvic);
78  }
79
```

```c
#include "stm32l1xx.h"

#include <math.h>
#define pi 3.141592
#include <stdlib.h>

void TIM2_IRQ_Config();

int main(void)
{
    TIM2_IRQ_Config();

    // # LED sur PB7
    /* Activer GPIOB sur AHB */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB,ENABLE);
    /* Configurer PB7 comme sortie tout-ou-rien */
    GPIO_InitTypeDef gpio_b;
    GPIO_StructInit(&gpio_b);
    gpio_b.GPIO_Mode = GPIO_Mode_OUT;
    gpio_b.GPIO_Pin = GPIO_Pin_7;
    GPIO_Init(GPIOB,&gpio_b);

    while(1) { }
}

// callback pour l'interruption periodique associee a TIM2
void TIM2_IRQHandler() {
    if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
    {
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
        GPIO_ToggleBits(GPIOB, GPIO_Pin_7); // Inversion du pin 7
    }
}

// ### TIMER 2 + IRQ a 500 ms
// Configuration Timer 2 a 500 ms
// avec emission d'IRQ : execute periodiquement TIM2_IRQHandler()
void TIM2_IRQ_Config()
{
    /*Activer TIM2 sur APB1 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);
    /* Configurer TIM2 a 500 ms */
    TIM_TimeBaseInitTypeDef timer_2;
    TIM_TimeBaseStructInit(&timer_2);
    timer_2.TIM_Prescaler = 0; // Prescalaire et Période ont au final le même résultat
    timer_2.TIM_Period = 364-1; // Cependant, on utilise Prescalaire pour compter le temps et Période pour mesurer le temps.
// Fhorloge = 44 000 Hz ; CPU = 16*10^6
// Thorloge = Modificateur/16x10^6 <=> 1/44000 = TIM_Period/16x10^6 <=> TIM_Period = 16x10^6/44000 = 363.6 (on arrondira au supérieur)

// On retrouve 2kHz, la moitié de la fréquence prévue, à cause du fonctionnement de l'horloge
    TIM_TimeBaseInit(TIM2,&timer_2);
    TIM_SetCounter(TIM2,0);
    TIM_Cmd(TIM2, ENABLE);

    /* Associer une interruption a TIM2 */
    TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);

    NVIC_InitTypeDef nvic;
    /* Configuration de l'interruption */
```

```
60      nvic.NVIC_IRQChannel = TIM2_IRQn;
61      nvic.NVIC_IRQChannelPreemptionPriority = 0;
62      nvic.NVIC_IRQChannelSubPriority = 1;
63      nvic.NVIC_IRQChannelCmd = ENABLE;
64      NVIC_Init(&nvic);
65  }
66
```

```c
#include "stm32l1xx.h"
#include <math.h>

void DAC1_Config();
void DAC1_Set(uint16_t value);

void TIM2_IRQ_Config();

float* T;
int n=0;

int main(void)
{
    TIM2_IRQ_Config();
    DAC1_Config();

    // # LED sur PB7
    /* Activer GPIOB sur AHB */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB,ENABLE);
    /* Configurer PB7 comme sortie tout-ou-rien */
    GPIO_InitTypeDef gpio_b;
    GPIO_StructInit(&gpio_b);
    gpio_b.GPIO_Mode = GPIO_Mode_OUT;
    gpio_b.GPIO_Pin = GPIO_Pin_7;
    GPIO_Init(GPIOB,&gpio_b);

    T = malloc(100*sizeof(float));
    for(int k=0;k<100;k++) {
        T[k] = 511 * sin(2*3.14159*k/100) + 2047;
    }


    while(1) {

    }
}

// callback pour l'interruption periodique associee a TIM2
void TIM2_IRQHandler() {
    if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
    {
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
        GPIO_ToggleBits(GPIOB, GPIO_Pin_7);
        DAC1_Set(T[n%100]); // Converti la valeur en tension
        // On ne dépasse pas 99 valeurs : à n = 100, n%100 = 0
        n++; // Incrémentation
    }
}

// ### TIMER 2 + IRQ a 500 ms
// Configuration Timer 2 a 500 ms
// avec emission d'IRQ : execute periodiquement TIM2_IRQHandler()
void TIM2_IRQ_Config()
{
    /*Activer TIM2 sur APB1 */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);
    /* Configurer TIM2 a 500 ms */
    TIM_TimeBaseInitTypeDef timer_2;
    TIM_TimeBaseStructInit(&timer_2);
    timer_2.TIM_Prescaler = 0;
    timer_2.TIM_Period = 363;
    TIM_TimeBaseInit(TIM2,&timer_2);
```

```
63      TIM_SetCounter(TIM2,0);
64      TIM_Cmd(TIM2, ENABLE);
65
66      /* Associer une interruption a TIM2 */
67      TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
68
69      NVIC_InitTypeDef nvic;
70      /* Configuration de l'interruption */
71      nvic.NVIC_IRQChannel = TIM2_IRQn;
72      nvic.NVIC_IRQChannelPreemptionPriority = 0;
73      nvic.NVIC_IRQChannelSubPriority = 1;
74      nvic.NVIC_IRQChannelCmd = ENABLE;
75      NVIC_Init(&nvic);
76  }
77
78
79  // ### DAC1 (DAC Channel 1) sur PA4
80  // Configuration
81  void DAC1_Config()
82  {
83      /*Activer GPIOA sur AHB */
84      RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
85      /* Configurer PA4 en mode analogique*/
86      GPIO_InitTypeDef gpio_a;
87      GPIO_StructInit(&gpio_a);
88      gpio_a.GPIO_Mode  = GPIO_Mode_AN; // Mode Analogique
89      gpio_a.GPIO_Pin = GPIO_Pin_4; // Sortie sur PIN 4
90      GPIO_Init(GPIOA, &gpio_a);
91
92      /*Activer DAC sur APB1 */
93      RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);
94      /* Configurer DAC1 avec parametres par defaut */
95      DAC_InitTypeDef dac_1;
96      DAC_StructInit(&dac_1);
97      DAC_Init(DAC_Channel_1, &dac_1);
98      /* Activer DAC1 */
99      DAC_Cmd(DAC_Channel_1, ENABLE);
100 }
101
102 void DAC1_Set(uint16_t value)
103 {
104     DAC_SetChannel1Data( DAC_Align_12b_R, value );
105     DAC_SoftwareTriggerCmd( DAC_Channel_1, ENABLE );
106
107 }
108
```

```c
#include "stm32l1xx.h"

#include <math.h>
#include <stdlib.h>

void DAC1_Config();
void DAC1_Set(uint16_t value);
void GPIOA_PA0_Config();

void TIM2_IRQ_Config();

unsigned int note_periode[8] = {611,544,484,458,408,363,323,306};


float* T;
int n = 0;
int interrupteur = 0;

void buildSawTooth() {
    for(int k=0; k<100;k++) {
        T[k] = 2047-511 + (k*1022/100);
    }
}
void buildTriangle() {
    int k;
        for(int k=0; k<100/2;k++) {
            T[k] = 2047-511 + (k*1022*2/100);
        }
        for(; k<100;k++) {
            T[k] = 2047 + 511 -((k-100/2)*1022*2/100);
        }
}
void buildSinus() {
    for (int k = 0; k < 100; k++) {
        T[k] = 511 * sin(2 * 3.14159 * k / 100) + 2047;
    }
}

int main(void) {
    TIM2_IRQ_Config();
    DAC1_Config();
    GPIOA_PA0_Config();

    // # LED sur PB7
    /* Activer GPIOB sur AHB */
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);
    /* Configurer PB7 comme sortie tout-ou-rien */
    GPIO_InitTypeDef gpio_b;
    GPIO_StructInit(&gpio_b);
    gpio_b.GPIO_Mode = GPIO_Mode_OUT;
    gpio_b.GPIO_Pin = GPIO_Pin_7;
    GPIO_Init(GPIOB, &gpio_b);

    T = malloc(100 * sizeof(float));

    //buildSinus();
    //buildSawTooth();
    //buildTriangle();

    GPIOA_PA0_Config();

    int prev_switch_status = 0;
```

```
63
64      while (1) {
65          int switch_status = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0);
66          if (switch_status == Bit_SET && prev_switch_status == 0) {
67              // A REMPLIR : ce que l'on doit executer si le bouton est appuyé
68              interrupteur++; // Nb d'appuie sur l'interrupteur
69              TIM_Cmd(TIM2, DISABLE);
70              TIM_SetCounter(TIM2, 0);
71              TIM_TimeBaseInitTypeDef timer_2;
72              TIM_TimeBaseStructInit(&timer_2);
73              timer_2.TIM_Prescaler = 0;
74              timer_2.TIM_Period = note_periode[interrupteur%8]; // A 8, on redescend à 0
75              TIM_TimeBaseInit(TIM2, &timer_2);
76              TIM_Cmd(TIM2, ENABLE);
77          }
78          prev_switch_status = switch_status;
79      }
80  }
81
82  // callback pour l'interruption periodique associee a TIM2
83  void TIM2_IRQHandler() {
84      if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET) {
85          TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
86          GPIO_ToggleBits(GPIOB, GPIO_Pin_7);
87          DAC1_Set(T[n % 100]);
88          n++;
89      }
90  }
91
92  // ### TIMER 2 + IRQ a 500 ms
93  // Configuration Timer 2 a 500 ms
94  // avec emission d'IRQ : execute periodiquement TIM2_IRQHandler()
95  void TIM2_IRQ_Config() {
96      /*Activer TIM2 sur APB1 */
97      RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
98      /* Configurer TIM2 a 500 ms */
99      TIM_TimeBaseInitTypeDef timer_2;
100     TIM_TimeBaseStructInit(&timer_2);
101     timer_2.TIM_Prescaler = 0;
102     timer_2.TIM_Period = 363;
103     TIM_TimeBaseInit(TIM2, &timer_2);
104     TIM_SetCounter(TIM2, 0);
105     TIM_Cmd(TIM2, ENABLE);
106
107     /* Associer une interruption a TIM2 */
108     TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
109
110     NVIC_InitTypeDef nvic;
111     /* Configuration de l'interruption */
112     nvic.NVIC_IRQChannel = TIM2_IRQn;
113     nvic.NVIC_IRQChannelPreemptionPriority = 0;
114     nvic.NVIC_IRQChannelSubPriority = 1;
115     nvic.NVIC_IRQChannelCmd = ENABLE;
116     NVIC_Init(&nvic);
117 }
118
119 // ### DAC1 (DAC Channel 1) sur PA4
120 // Configuration
121 void DAC1_Config() {
122     /*Activer GPIOA sur AHB */
123     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
124     /* Configurer PA4 en mode analogique*/
```

```
125     GPIO_InitTypeDef gpio_a;
126     GPIO_StructInit(&gpio_a);
127     gpio_a.GPIO_Mode = GPIO_Mode_AN;
128     gpio_a.GPIO_Pin = GPIO_Pin_4;
129     GPIO_Init(GPIOA, &gpio_a);
130
131     /*Activer DAC sur APB1 */
132     RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);
133     /* Configurer DAC1 avec parametres par defaut */
134     DAC_InitTypeDef dac_1;
135     DAC_StructInit(&dac_1);
136     DAC_Init(DAC_Channel_1, &dac_1);
137     /* Activer DAC1 */
138     DAC_Cmd(DAC_Channel_1, ENABLE);
139 }
140
141 void DAC1_Set(uint16_t value) {
142     DAC_SetChannel1Data( DAC_Align_12b_R, value);
143     DAC_SoftwareTriggerCmd( DAC_Channel_1, ENABLE);
144
145 }
146
147 void GPIOA_PA0_Config() {
148     // switch PA0
149     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
150     GPIO_InitTypeDef switch_PA;
151     GPIO_StructInit(&switch_PA);
152     switch_PA.GPIO_Mode = GPIO_Mode_IN;
153     switch_PA.GPIO_Pin = GPIO_Pin_0;
154     GPIO_Init(GPIOA, &switch_PA);
155 }
156
```