Step Jeu Son
Gérer le son d'un bris de verre lors de l'impact

Documents utiles :
Cortex-M3 Programming Manual (PM0056.pdf)

Le point périph …
La PWM, éléments théoriques
La PWM, mise en oeuvre

1. Introduction

Avec cette étape, nous rentrons dans la phase projet qui sera de moins en moins guidée :

Projet :

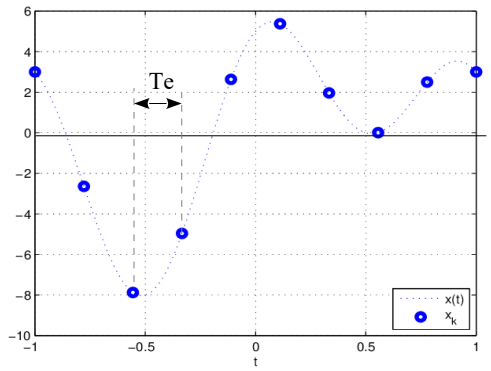
- **Step Son : la cible doit émettre un bruit de bris de verre si le joueur a touché une cible**
- Step DFT : mise en place de l'analyse DFT sur 6 canaux en format fractionnaire (virgule fixe)
- Step DFT Réel : ajout de la DMA pour une utilisation en contexte réel
- Step Intégration: fusion de tout les éléments

Plus particulièrement nous allons gérer l'aspect son du projet.

1.1. Comment générer un son ?

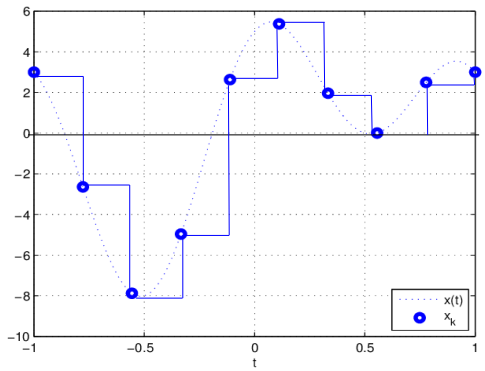
Le son qui va être joué a déjà été échantillonné dans une table. Chaque échantillon est signé 16 bits dans notre cas.

un son échantillonné... :



Te est la période d'échantillonnage fixée lors de l'enregistrement.
Pour jouer le son, il suffit de prendre les échantillons un par un à la même période d'échantillonnage et d'envoyer chacun des échantillons dans un ***DAC*** (*Digital to Analog Converter*). Le résultat est alors le suivant :

Le son restitué avec DAC... :



Finalement, pour jouer un son avec un microcontrôleur, il nous faut :

- disposer de la table d'échantillons,
- connaître la période d'échantillonnage Te du son,
- utiliser un ***timer*** dont le modulo vaut Te,
- associer un *callback* à ce timer,
- A chaque entrée dans le *callback*, lire l'échantillon à traiter (un seul!!) et l'envoyer sur le *DAC*, puis mettre à jour un index de table,
- stopper la lecture de table lorsque tous les échantillons sont lus (sinon on lit toute la mémoire du µcontrôleur!!)

Problème :

- Le *STM32F103* ne dispose pas de *DAC*, nous utiliserons donc un autre procédé subtil très souvent utilisé : la *PWM (Pulse Width Modulation)*
- la résolution de la *PWM* (équivalent au *DAC*) est de 720. Ce qui veut dire que les échantillons codés sur [-32768 ; + 32767] (16 bits signés) doivent être ramenés dans la plage [0 ; 719].

2. Préparation du projet KEIL

1. Déployer l'archive *PjtKEIL_StepSon.zip,* ouvrir le projet et choisir la cible simulation.

   Dans le répertoire */Src*, se trouvent deux fichiers source non inclus dans le projet, *bruitverre.asm* et *GestionSon.s*.

2. Ajouter ces 2 fichiers dans le projet KEIL :
   Pour cela (ce n'est pas une obligation, juste de l'organisation...) : créer un nouveau « pseudo répertoire » que vous appellerez *Son* (clic droit sur *CibleSondeST* dans la fenêtre *Project*, → *Manage Project Items* → colonne *Groups* : créer le groupe *Son* → Ajouter les deux fichiers).

3. Jouer le son sans DAC ni PWM ...

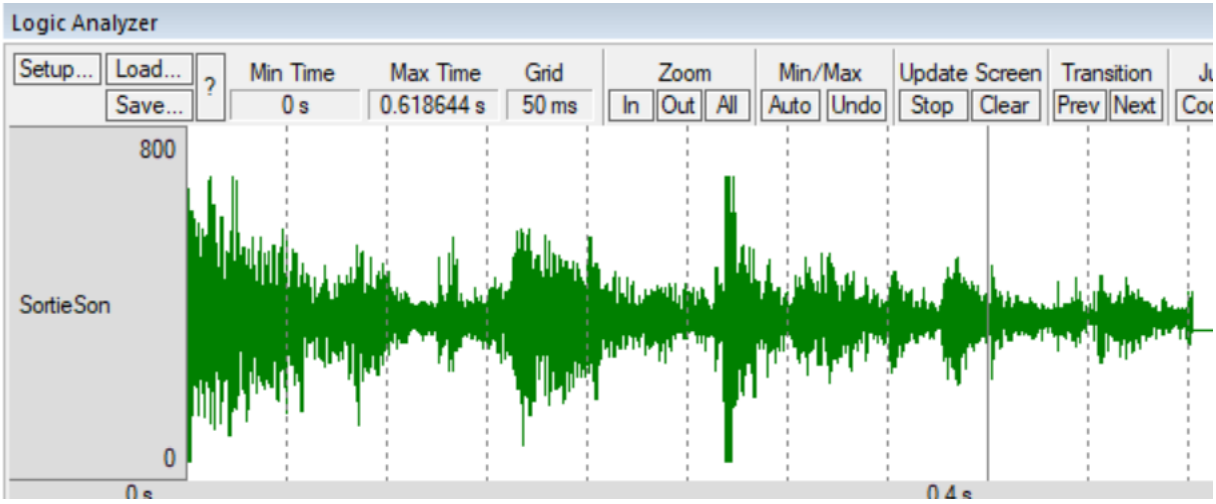
L'objectif ici est de mettre en place tout ce qui est nécessaire sauf la sortie *PWM* que nous verrons dans un second temps. Il s'agit donc rejouer le son enregistré dans le fichier *bruitverre.asm* et de le lire dans le *logic analyse*r dans un premier temps.

3. Travail dans *principal .c* : configurer le *timer 4* pour qu'il ait une périodicité correspondant à celle du son que l'on veut jouer (voir entête du fichier *bruitverre.asm*). Activer les interruptions pour lancer *CallbackSon* dont le corps sera écrit en assembleur dans le fichier *GestionSon.s*. (exactement le principe vu dans le step précédent). La priorité sera fixée à 2.

4. Coder la fonction *CallbackSon* en assembleur (attention à respecter les conventions d'appel *AAPCS*) :
   - prévoir une variable globale (pour l'analyseur logique) qu'on appellera *SortieSon* sur 16bits, elle contiendra la valeur mise à l'échelle de l'échantillon courant,
   - prévoir un index de table qui permettra de repérer l'échantillon à traiter dans la table,
   - utiliser l'instruction de chargement indexée (register offset) qui est dédiée au parcours des tableaux,
   - l'intervalle de valeurs possibles pour *SortieSon* est [0 ;719].

   NB : Le test ultime est résumé dans la question suivante. Néanmoins, ne pas hésiter à faire des tests progressivement, au fil du codage. Par exemple, le paramétrage du Timer 4 dans *principal.c* peut se vérifier avec un *Callback* vide (ou presque...). Simplement pour vérifier le timing...

5. Tester la solution : en visualisant *SortieSon* , on doit obtenir l'allure suivante :

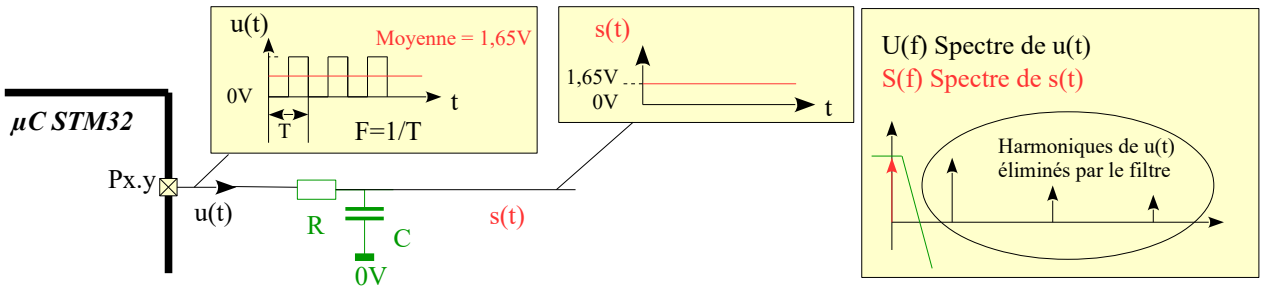


4. Et je mets le son …

4.1. Eléments de théorie PWM

Un microcontrôleur embarque rarement un *DAC*, par contre un microcontrôleur peut facilement générer un signal carré entre 0V et 3,3V. Un filtre passe-bas peut extraire la tension moyenne du carré et éliminer les harmoniques. On obtient alors une grandeur non plus logique, mais analogique !

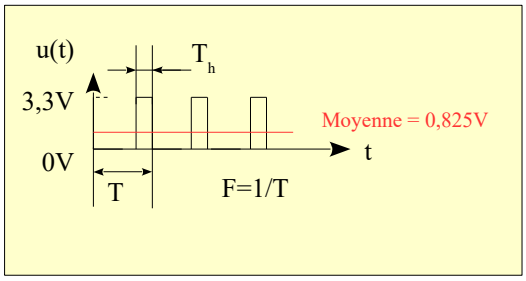
Voici le concept pour une valeur continue souhaitée de 1,65V :



Choix de la fréquence de coupure du filtre : $Fc = \frac{1}{2 \cdot \pi \cdot R \cdot C} \ll F$ (F fréquence du signal carré) pour un filtrage efficace des harmoniques tout en préservant la moyenne.

Pour obtenir une tension continue autre que 1.65V, il suffit de modifier la durée à l'état haut du signal carré.

On introduit alors le ***rapport cyclique (duty Cycle),*** α **= *Th/T***



De manière générale, la moyenne du signal de sortie s'écrit (en notant E = 3,3V) :

$$u_0 = \frac{1}{T}\int_T u(t)\,dt = \frac{1}{T}\int_T E\,dt = \frac{T_h}{T}\cdot E$$

soit $\boxed{u_0 = \alpha \cdot E}$

Le microcontrôleur doit donc être capable de fournir un signal numérique binaire (0V ou 3,3V) mais dont on contrôle la largeur d'impulsion (c'est à dire le rapport cyclique α). Ce type de signal est appelé signal *PWM* (***P**ulse **W**idth **M**odulalion*).

La mise en œuvre de la *PWM* est détaillée ici .

4.2. Observation de la PWM en simulation LTSpice

Le but de ce petit exercice est de comprendre comment on peut obtenir un signal analogique variable avec une *PWM*, et ceci grâce au logiciel *LTSpice*.

6. Télécharger le fichier *LTSpice PWM.asc*. Ouvrir le fichier. Lancer la simulation.
7. Observer le signal *Compteur*, puis *CompValue* en cliquant dessus pour les faire apparaître dans la fenêtre de simulation. Observer également le signal *PWM*. Faire un zoom sur quelques périodes.
8. Quelle condition faut-il pour que le signal *PWM* soit à l'état haut ? À l'état bas ?
9. Observer enfin le signal filtré accessible sur l'équipotentielle *Filtre*. Comparer ce signal à celui que l'on veut transmettre.

4.3. Ajout de la PWM dans le programme

4.3.1. Configuration PWM

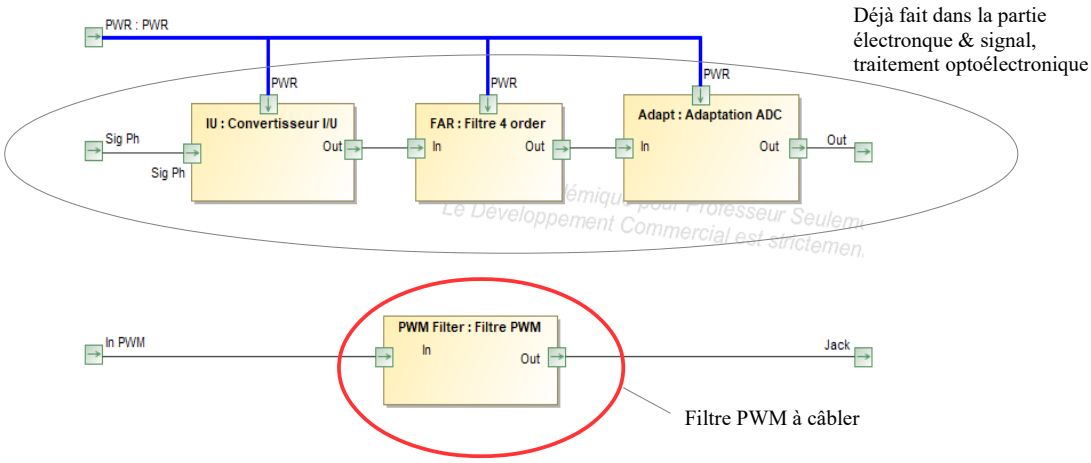
10. Ajouter dans la configuration du *main* la configuration de la *PWM* de manière à utiliser le *Timer 3*, canal 3. Fixer la *période_Ticks* à 720. Quelle est alors la fréquence de la *PWM* ?
11. Le canal 3 est en fait relié à la pin *PB.0*. Pour que le signal *PWM* soit disponible sur cette pin, il faut faire une configuration spéciale, qui va au delà de la simple configuration de sortie. Voici la ligne qu'il faut ajouter :
    char GPIO_Configure(GPIOB, 0, OUTPUT, ALT_PPULL);
    De cette manière, ce n'est plus le programme qui gère *PB0*, c'est la *PWM* qui prend directement la main dessus.

4.3.2. Utilisation de la PWM

12. Pour mettre à jour le rapport cyclique, il suffit d'utiliser la fonction présentée en fin de document à la place de la variable *SortieSon* (qui devient inutile, mais peut être conservée pour observer à la fois la valeur de l'échantillon et la *PWM* sur PB0). Pour cela penser à inclure le fichier *.inc* relatif à la librairie driver (si une erreur est générée, voir *Step 2* où la procédure similaire pour les *.h* est décrite).
13. Vérifier en simulation dans le *logic analyser* en observant *PB.0 (*ainsi que *SortieSon* si elle a été conservée).

4.4. Test du programme sur cible réelle

Rappelons l'organisation de votre plaque d'essais :



14. Connectez votre carte micro-contrôleur à votre plaque d'essais. Si c'est la carte de TP avec l'*Olimex,* c'est immédiat *(utiliser la petite nappe).* Si c'est une carte *Nucléo*, il faut repérer *PB0* et le *GND*.
15. Calculer un filtre passe-bas d'ordre 1 d'environ 4kHz de fréquence de coupure. On prendra une résistance entre 1kΩ et 4,7 kΩ.
16. Tester le son en visualisant sur l'oscilloscope ou en utilisant un haut-parleur amplifié.

4.5. Finalisation du module GestionSon.s

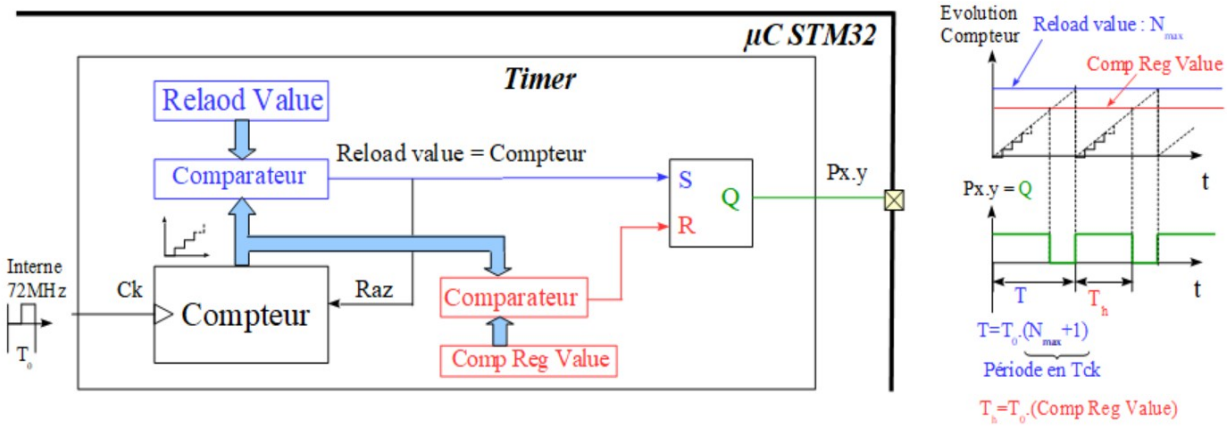
17. Ajouter une fonction *StartSon*, sans paramètre, qui permet de lancer la séquence sonore,
18. Vérifier son bon fonctionnement,

Et pour finir :

19. rédiger un *header file, GestionSon.h* à placer dans /src évidemment. Il contiendra les deux déclarations de fonction.

5. Point périphérique de micro-contrôleur : la PWM

5.1. Aspect matériel



Les constructeurs de microcontrôleur ont prévu une unité toute faite qui permet de générer une ***PWM*** très simplement : cette fonctionnalité prend place dans un **Timer**.

5.2. Aspect logiciel

5.2.1. Configuration

Fonction de configuration :
*unsigned short int PWM_Init_ff(TIM_TypeDef *Timer, char Voie, u32 Periode_ticks);*
(voir .h pour plus de détail)

Exemple : *PWM_Init_ff(TIM1, 4, 250);*
Une fois cette fonction exécutée, le timer 1 va compter modulo 250, c'est à dire que la période de la *PWM* sera égale à $T_{PWM} = T_{Ck} \cdot 250 = \frac{250}{72 \cdot 10^6} = 3{,}47\,\mu s$. Sa résolution est naturellement égale à 250. Le signal *PWM* sortira sur la voie **4** du Timer **1**.

NB : la fonction *PWM_Init_ff* appelle en interne la fonction déjà connue, *Timer_1234_Init_ff*.

5.2.2. Utilisation

fonction d'utilisation:
Elle est adaptée au projet, puisqu'elle ne permet l'usage que du *Timer 3*, canal 3.
void PWM_Set_Value_TIM3_Ch3(unsigned short int Thaut_ticks);

Exemple (on suppose que la *PWM* est configurée avec une résolution de 250)
PWM_Set_Value_TIM3_Ch3(125);
Le signal *PWM* aura donc un rapport cyclique de 50%

Retour