

Principe de l'utilisation d'un tampon circulaire audio et d'un signal PWM.
Utilisation d'un squelette d'application audio fourni pour tester de la synthèse sonore

Matériel et logiciel requis pour cette séance :

Une **machine école**, une paire d'écouteurs personnels à brancher sur cette machine. Vérifiez que la sortie son est opérationnelle en réglant le volume à un niveau moyen (pas trop fort) sur vos écouteurs.
Les plate-formes Arduino n'étant pas disponibles pour cette séance, le fonctionnement du dispositif se fera en simulation sur **Proteus/Isis**.

Adaptations à la simulation : pas d'interactions

Le simulateur sur PC (même performant) n'a pas la puissance pour simuler l'ensemble Arduino+DangerShield en temps réel : il faut plusieurs secondes de simulation pour une seconde de temps simulé. Quelques essais utilisant la possibilité pour Isis de répercuter une sortie audio sur la carte son du PC pendant la simulation en mode interactif n'ont pas été concluants : le son est trop discontinu pour être exploitable. Nous ferons donc en 2 temps :

- Etape simulation et enregistrement dans un graphe "Audio" → quelques dizaines de secondes de calcul.
- Playback du son enregistré → 5 secondes du son généré par la simulation.

On peut utiliser Isis en mode interactif (appuyer sur des boutons virtuels, bouger des potentiomètres...) même si la puissance de calcul ne permet pas le temps réel (on interagit "au ralenti"). Malheureusement il devient alors difficile d'enregistrer un signal pour faire après un playback sonore¹

D'autre part, que ce soit en mode simulation pour le mode graphe (que nous utiliserons), ou le mode interactif, l'utilisation des entrées analogiques (position des potentiomètres) sur l'ATMega accroît la demande de calcul à un point tel qu'il faut 10 minutes pour simuler 5 secondes de son...²

Pour ces raisons pratiques nous allons considérer un schéma sans boutons et sans potentiomètres : le programme se contentera de générer une séquence audio figée, non interactive.

Squelette d'application audio sous CodeBlocks

Un projet CodeBlocks comportant de nombreuses fonctions pré-écrites³ est disponible juste (immédiatement) sous le lien de l'énoncé du TP2 sur la home-page archi : ArduinoAudio.zip

Téléchargez et dézippez ce projet ArduinoAudio.zip sur Z: (ni bureau ni mesdocs... même sur Z...)
et ouvrez le dans CodeBlocks (ouvrir le .cbp)

Si vous n'aviez pas configuré la compilation CodeBlocks pour Arduino à la séance TP1 : 2 manips à faire

1/ A l'ouverture du projet : CodeBlocks vous demande la valeur de la variable `avr_gcc`
(si il ne le fait pas, vérifiez Settings→Global variables... → Current Variable: `avr_gcc` doit être présent
(ou créé si il n'est pas dans la liste : Bouton New → saisir `avr_gcc`) et avoir le chemin qui suit ...)
Cliquez sur bouton ... à coté de base (Builtin fields) et naviguer vers le chemin suivant (puis OK et Close)
`c:\Program Files (x86)\arduino-0021\hardwaretools\avr` (un seul `avr` pas `avravr`)

2/ Allez dans le menu Settings → Compiler and debugger... → Selected compiler : GNU AVR GCC compiler
Puis : Panneau Toolchain executables, saisir `$(#avr_gcc)` dans la boîte Compiler's installation directory

Après avoir fait **CTRL-F11 (Rebuild All)** vous devez obtenir des fichiers exécutables dans le répertoire de projet `arduinoAudio\bin\Debug`. Celui qu'on utilisera est `arduinoAudio.elf.hex` ...

1 Le graphe de type AUDIO n'enregistre pas les sessions interactives. Le graphe "INTERACTIVE" permet ce type d'enregistrement mais ne permet pas le playback sonore : on pourrait sauver avec "Exporter données graphe..." et utiliser un programme pour le convertir en .wav (il faudrait le sous-échantillonner à 22050Hz) et le jouer avec un mediaPlayer mais ça devient compliqué...

2 La simulation des parties numériques du montage (le µcontrôleur) utilise des algorithmes différents, plus efficaces, que les parties analogiques. La simulation mixte analogique/numérique, sur un même montage, est un problème très complexe. La présence d'un signal analogique en amont du processus numérique doit probablement rendre impossible certaines des optimisations numériques. Il existe peut-être un moyen de configurer pour améliorer ça (voir doc sur la simulation de situations ADC où le signal Analogique est pré-calculé ou pré-défini ?).

3 Certaines ont été vues en TD, il peut y avoir quelques variantes mais les mêmes principes restent valables...

Modèle Isis

Lancez Proteus → Isis puis ouvrez arduinoAudio\isisModel\arduinoPwmSound.DSN

Le programme qui sera exécuté (virtuellement) par l'ATMEGA328P sera arduinoAudio.elf.hex (re)généralisé à chaque compilation du projet sur CodeBlocks. Pour tester une modification du code il faudra donc recompiler (CTRL-F11) sur CodeBlocks, puis revenir à Isis pour relancer une simulation (voir ci-après).

Pour info: dans Isis le choix du fichier exécutable ainsi que la configuration du µContrôleur se fait dans la fenêtre d'édition de propriétés, accessible sur double click du composant.

Test préliminaire

Pour vérifier le process compilation/simulation on peut coller le code suivant dans mainloop.cpp (sans remplacer ni modifier le reste du code déjà en place) :

```
    dans setup() {
// Initialisation des broches leds en sorties et les éteindre
led_init();

    dans loop() {
static uint8_t blink_cpt, blink_state;

// Une fois sur 245...
blink_cpt++;
if (blink_cpt>= 245)
{
    blink_cpt=0;

    // Inverser état (et allumer ou éteindre en conséquence)
blink_state^=1;
if (blink_state)
    LED_ON(0);
else
    LED_OFF(0);
}
}
```

puis compiler sous CodeBlocks

puis lancer une simulation interactive sous Isis : en bas à gauche de de la fenêtre principale

vous trouvez des commandes play/pause/stop ...



(Ces commandes sont parfois cachées par la barre de tâche Windows, dans ce cas réduire la fenêtre Isis pour les rendre visible, ou docker la barre de tâche ailleurs...)

Et donc "play" → la led clignote à 0.5 Hz ... en temps simulé, donc beaucoup plus lentement sur votre écran.

Enlever ce code de test pour la suite...

Quelques précisions sur l'interruption utilisée par le projet audio

Le principe de la routine d'interruption audio a déjà été présentée succinctement en cours, le principe est qu'elle est automatiquement appelée au milieu de chaque cycle du PWM audio (31372.55 fois par seconde) et que son rôle est d'écrire dans le registre OCR2B le nouveau rapport cyclique (0→0% 255→100%) du prochain échantillon.

Vous pouvez ouvrir audio.c pour voir cette routine ISR(TIMER2_OVF_vect) ainsi que void audio_init() qui enclenche le PWM (et donc l'appel automatique à la routine, à intervalle fixe, indépendamment du programme principal) dès qu'elle est appelée par le setup.

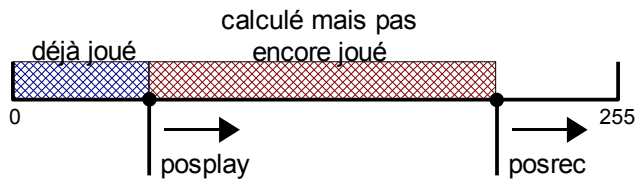
Tampon circulaire audio (ring buffer)

Pour une seule onde simple à générer en sortie audio on **pourrait** se contenter de faire le calcul (ou plutôt la consultation de la table d'onde) directement au niveau de l'interruption audio :

```
ISR(TIMER2_OVF_vect)    // NE TAPEZ PAS CE CODE (solution non retenue)
{
    // reprogrammer le rapport cyclique du PWM avec le nouvel échantillon
    // (ici on suppose que wave est un tableau en RAM avec un sinus dedans)
    OCR2B=wave[poswave];
    poswave++;
}
```

Mais pour des formes d'onde plus compliquées, 4 voix à gérer, une rythmique... il devient impossible de générer la valeur du nouvel échantillon directement dans la routine d'interruption en garantissant que ce soit toujours fait en moins de 1/31372 s. Notre application devra bien sûr générer ces 31372 nouveaux échantillons par seconde mais **en moyenne**. On aura parfois besoin, **punctuellement**, de plus de 1/31372 s avant de pouvoir calculer le prochain échantillon (aléas du programme, gestion de l'interface et des changements de notes...).

Il faut pouvoir **lisser** ces variations de charge CPU...
 Les échantillons de la sortie audio seront calculés par le programme principal un peu en avance dans un tampon intermédiaire, et récupérés à intervalle fixe par l'interruption qui les répercutera sur le rapport cyclique du PWM. Ce tampon prend la forme d'un tableau de 256 entiers non signés 8 bits → `uint8_t audio_buf[256]` ; déclaré (avec `extern`) dans `audio.h` et défini dans `audio.c`

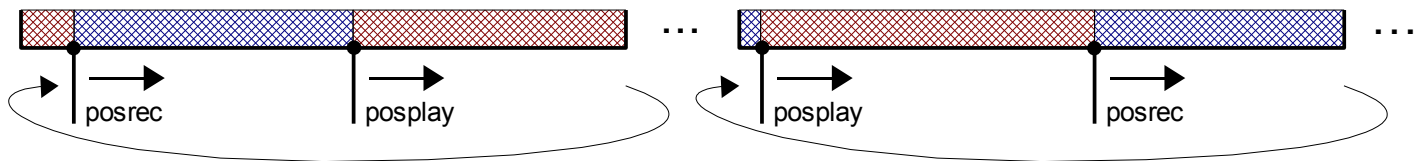


posplay avance dans l'interruption à chaque nouvel échantillon joué (**play**) en PWM (timing parfaitement régulier)

posrec avance dans la boucle principale du programme à chaque fois qu'on a calculé un nouvel échantillon (enregistrement: **record**)

Si posrec reste un peu bloqué par moment à cause d'un traitement occasionnel pendant lequel on ne peut pas encore s'occuper du calcul de l'échantillon suivant ce n'est pas grave : la partie "calculé mais pas encore joué" sert d'amortisseur, de **tampon**. La seule contrainte est que posrec avance aussi vite que posplay **en moyenne**. Le délai maximum entre posrec et posplay est de 255 échantillons donc $255/31372 =$ un peu moins de 1/100 s ce qui ne posera pas de problème de latence (**lag**) pour jouer de l'instrument.

Le tampon est "circulaire", c'est à dire que les indices posrec et posplay d'accès au tableau se font modulo 256. Ce comportement ne nécessite aucun traitement particulier grâce à l'utilisation d'un type 8 bits non signé (`uint8_t`) pour ces indices. Avec l'intervalle [0...255] faire 255+1 ramène à 0. On a un modulo 256 "naturel".



Voir les définitions de `audio_posplay` et `audio_posrec` et la lecture de `audio_buf` de l'interruption : `audio.c`

Reste un "petit détail" à régler : si posrec est en moyenne aussi rapide que posplay, disons juste un peu plus rapide pour se garantir un peu de marge, posrec fini par rattraper posplay ce qui casse le mécanisme. Mais il est difficile de tester si posrec reste bien derrière posplay car il est aussi devant (tampon **circulaire**)⁴ Une solution classique est de "couper" le tampon en 2 parties et d'imposer au programme principal de ne jamais écrire de données avec posrec dans le même demi-tampon (la même moitié) que là où se trouve posplay. Si posrec arrive dans le même demi-tampon que posplay alors le programme principal suspend son activité (boucle à vide) jusqu'à ce que posplay avance à son rythme (par l'interruption) sur le demi-tampon suivant.

Dans le programme vous disposez de 2 fonctions pour synchroniser votre programme principal (posrec) avec l'interruption de sortie audio (posplay) :

- `audio_waitplay()` qui bloque sur place tant que posplay est encore dans le même demi-tampon que posrec
- `audio_recording()` qui retourne un booléen vrai si posplay et posrec sont dans des demi-tampons différents

Repérer et essayer de comprendre le code de ces 2 fonctions dans `audio.h`

Repérer et essayer de comprendre le code de `audio_write`

Comprendre l'organisation des appels à ces fonctions audio au niveau du code loop de `mainloop.cpp` (pour l'instant l'appel à `audio_write` envoie la valeur constante 0, donc onde plate, pas de son)

⁴ Une alternative serait comme sur une piste Olympique de ne pas juste considérer la position sur la piste circulaire mais de tenir à jour un compteur du nombre de tours effectués pour chacun des deux concurrents...

Biiiiiiiiiiiiip !

Il est temps de faire du son, on va essayer un sinus de fréquence fixe.

Dans loop de mainloop.cpp déclarez une variable 8 bits non signée poswave :

```
static uint8_t poswave;
```

Le mot clé static indique que la variable conserve la valeur précédente à chaque appel de loop (ce qui n'est pas le cas d'une variable locale usuelle) et qu'elle se comporte comme une variable globale du point de vue de sa valeur initiale (ici aucune mention de valeur initiale donc 0).

Dans la boucle d'enregistrement while(audio_recording()) envoyer les valeurs de la table sinus (remplacer l'appel audio_write(0); par cet appel qui génère les valeurs de l'onde)

```
audio_write( (int8_t)pgm_read_byte_near(sinus+poswave) );
```

Puis avancer dans la table d'onde, avec modulo 256 naturel, toujours dans la boucle d'enregistrement

```
poswave++;
```

Pour tester après compilation : sous Isis on lance une simulation en mode graphe en appuyant sur la barre Espace avec le curseur souris au dessus de la fenêtre AUDIO ANALYSIS.

Dès que la simulation (5s de temps simulé, 1 minute ou plus de simulation) se termine, vous devez entendre un son sinusoïdal de fréquence fixe 122,55 Hz joué par le graphe.

Pour rejouer le même son (sans re-simuler) utiliser CTRL-Espace avec le curseur au dessus de la fenêtre AUDIO ANALYSIS.

Vous pouvez aussi sauvegarder un fichier audio .wav (double cliquer sur le graphe...) et l'écouter plus tard avec n'importe quel media-player.

Il est possible de voir la courbe du graphe plus en détails : double-cliquer sur la barre de la fenêtre AUDIO ANALYSIS ouvre une fenêtre détaillée (agrandir en plein écran) dans laquelle on peut zoomer (loupe +) et défiler (flèches à coté de loupe + en bas de la fenêtre détaillée). Reconnait-on bien une sinusoïde ?

Quelle est la fréquence des oscillations de haute fréquence qui viennent se superposer au signal attendu ? D'où viennent ces oscillations parasites (qui font que le son perçu n'est pas parfaitement doux et pur) ?

Remettre le graphe en pleine échelle (zoom sur la feuille entière, loupe carré) et fermer la fenêtre détaillée.

Lancer la simulation pour le graphe TRANSIENT ANALYSIS (barre espace avec la souris dessus)

Ceci enregistre 1s du signal numérique en sortie de l'Arduino (c'est le signal PWM) et du signal analogique Audio qui n'est en fait que le signal PWM filtré par un passe-bas de fréquence de coupure ~10kHz (les 3 passe-bas RC du 1er ordre en cascade⁵). Vous pouvez ouvrir une fenêtre détaillée pour ce graphe représentant les 2 signaux simultanément et en zoomant vous pouvez retrouver le type de chronogramme montré en cours pour illustrer l'utilisation d'un PWM à rapport cyclique variable pour générer un signal analogique.

Attention à bien interpréter les échelles verticales, même si il est représenté de manière très écrasée, le graphe numérique PWM a la même amplitude 0:5V que le graphe analogique Audio.

Il est également possible d'utiliser l'oscilloscope interactif : lancer la simulation en mode interactif (play), essayer de trouver les réglages trigger source et level et la base de temps permettant de bien voir le principe du PWM.

Si vous aviez fermé l'oscilloscope interactif vous pouvez le ré-afficher en lançant le mode interactif (play) puis en allant dans le menu déroulant d'Isis Mise au point (Debug ?) → Digital Oscilloscope

Biiiiiiiiiiiiip plus ou moins grave ou aigu

On a vu (Cf cours) que l'approche simpliste poswave++ ne permet pas de décider de vitesses de lectures réglables et donc de fréquences arbitraires → utilisation d'un format en virgule fixe Q8.8

Changer la déclaration de poswave en 16 bits non signé : static uint16_t poswave;

Diviser par 256 lors d'un accès à la table pour obtenir un indice dans [0...255] :

```
audio_write( (int8_t)pgm_read_byte_near(sinus+(poswave>>8)) );
```

Ajouter un incrément à poswave correspondant à 256x l'incrément à virgule représenté :

```
poswave+=384; // Par exemple ceci correspond à un incrément de 1.5
```

Tester quelques valeurs pour obtenir des hauteurs différentes

Finalement il y a des macros pour faciliter la consultation des tables à partir d'un indice en Q8.8 :

```
audio_write( SAMPLE_READ(sinus,poswave) ); // pareil mais plus lisible
```

Voir tables.h pour la définition de ces macros d'accès aux tables

5 Ce passe bas du 3ème ordre est très rudimentaire, il serait à améliorer pour atténuer davantage les oscillations résiduelles du PWM

Enveloppe d'amplitude

Le programme précédent génère une note de volume constant (amplitude fixe)

Une note d'instrument est généralement d'amplitude variable entre le début et la fin de la note.

En synthèse sonore on distingue souvent 4 phases : Attack Decay Sustain Release

On peut gérer ces différentes phases avec des formules de calculs (interpolations linéaires...) mais on peut aussi utiliser directement une table. Une table d'amplitude exponentielle décroissante est proposée dans tables.c

Essayez d'expérimenter et de comprendre le principe de fonctionnement du code suivant :

```
// "partition" mélodie
uint16_t incrwave_seq[8] = {546, 614, 689, 729, 819, 919, 1032, 1093};
// "partition" vitesses d'enveloppes
uint8_t increnv_seq[8] = { 50, 25, 50, 100, 50, 25, 50, 100};
// temporisation lecture partition
uint16_t tempo=1000;

void loop()
{
    static uint16_t poswave;
    static uint16_t incrwave;

    static uint8_t env;
    static uint16_t posenv;
    static uint8_t increnv;

    static uint8_t tick_cpt;
    static uint16_t tempo_cpt;
    static uint8_t pos_seq;

    audio_waitplay();
    while (audio_recording())
    {
        // Une fois sur 16...
        if ( tick_cpt == 0 )
        {
            // mettre à jour l'enveloppe
            env= ENVELOPE_READ(expdown,posenv);
            posenv+=increnv;
            // On ne veut pas dépasser la fin de l'enveloppe...
            // (commenter pour "rebouclage")
            if (posenv>0xFF00) posenv=0xFF00;

            // Une fois sur tempo...
            if (tempo_cpt==0)
            {
                // On arrive à une nouvelle note :
                posenv=0; // Relance l'enveloppe amplitude (attaque)
                poswave=0; // Redémarre au début de la table d'onde
                incrwave=incrwave_seq[pos_seq]; // Hauteur nouvelle note
                increnv=increnv_seq[pos_seq]; // Vitesse d'enveloppe nouvelle note
                pos_seq++; // Un cran plus loin sur la partition...
                if (pos_seq==8)
                    pos_seq=0;
            }

            tempo_cpt++;
            if ( tempo_cpt == tempo )
                tempo_cpt=0;
        }

        tick_cpt++;
        if ( tick_cpt == 16 )
            tick_cpt=0;

        // Une enveloppe est une modulation d'amplitude :
        // le signal modulant (env) multiplie le signal modulé (le sinus)
        audio_write( (env*SAMPLE_READ(sinus,poswave))>>6 );
        poswave+=incrwave;
    }
}
```

L'utilisation d'un code une fois sur 16 (à régler selon vos besoins) permet de ralentir la vitesse de lecture de la table d'enveloppe (beaucoup plus lente que la lecture d'une table d'onde) et d'alléger les temps de traitement liés à la gestion des enveloppes

Vous pouvez expérimenter une ou plusieurs de ces possibilités :

- ***Glissando : modifier la hauteur de note en cours de jeu, par exemple `incwave--` une fois sur 16***
- ***Avoir une vraie partition rythmique (la note suivante arrive plus ou moins vite)***
- ***Polyphonie : combiner 2 ou 4 lignes mélodiques simultanées, à sommer dans l'appel à `audio_write` (et non pas 2 ou 4 appels à `audio_write`). Attention aux intervalles en amplitude de la somme...***
- ***Utiliser le générateur aléatoire `[0...255] rnd_rand()` à la place de `SAMPLE_READ` → bruit blanc***
- ***Mettre en place d'autres formes d'onde (paragraphe suivant)***

Nouvelles tables d'onde

Classiquement une note est caractérisée par sa durée, sa hauteur (fréquence) et son timbre qui dépend de la forme de l'onde sonore. Pour l'instant les tests ont utilisé un seul timbre qui correspond au sinus (son le plus élémentaire, le plus "doux"). Pour ajouter de nouveaux timbres au projet il suffit d'ajouter de nouvelles tables d'ondes. Pour calculer les 256 valeurs d'une table il faudra faire un projet console DOS et copier le résultat obtenu dans le fichier `tables.c` avec une définition de tableau en mémoire programme (même format que pour le sinus) sans oublier de déclarer ce même tableau en extern dans `tables.h`

Ajouter au projet une table pour une onde de forme rampe décroissante (facile) et une onde de forme triangle (plus difficile). Tester les nouveaux timbre obtenus.

Pour info voici le code utilisé pour obtenir la table sinus (déjà en place dans `tables.c`)

Pour un bon comportement avec les enveloppes en amplitudes, la valeur moyenne de l'onde doit être ~nulle

```
int main()
{
    int i,total,val;

    total=0;
    for (i=0;i<256;i++)
    {

        val=round(sin(2.0*M_PI*i/256)*127.0);

        total+=val;
        printf("%4d,",val);
    }

    printf("\n\nvaleur moyenne: %f\n", (float)total/256);

    return 0;
}
```