Génération d'un signal par modulation de largeur d'impulsion

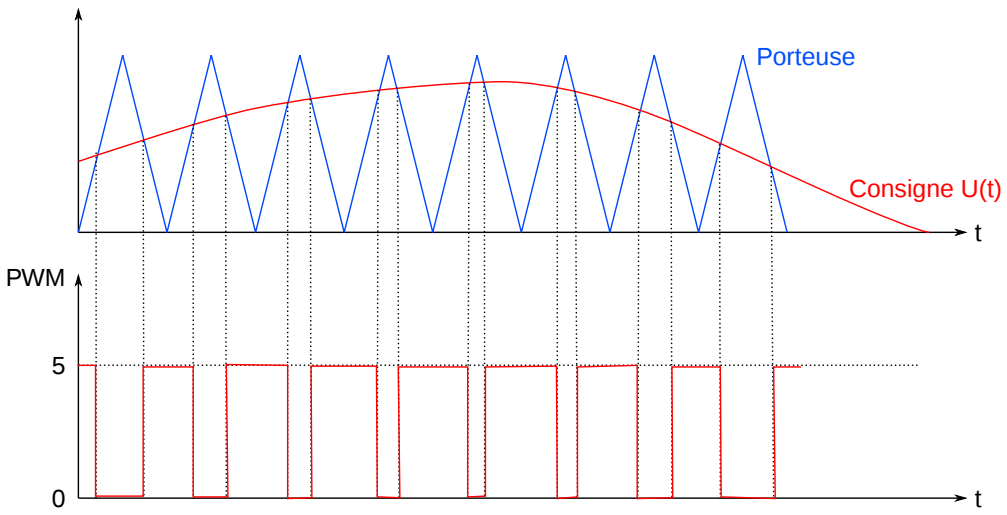
1. Introduction

La technique de modulation de largeur d'impulsion (Pulse Width Modulation PWM) consiste à générer un signal carré avec un rapport cyclique modulé en fonction d'un signal de commande. Le signal généré peut servir à commander un circuit de puissance à découpage (pont en H), associé à un filtrage passe-bas inductif, pour générer une onde sinusoïdale ou d'une autre forme. La technique est utilisée dans les onduleurs monophasés, diphasés ou triphasés [1]. Le même principe est utilisé dans les amplificateurs Audio de classe D.

Cette page montre comment générer des signaux par MLI avec un Arduino Due 32 bits. Pour faire la même chose avec un Arduino 8 bits (Uno, Mega ou autre), consulter Génération d'un signal par modulation de largeur d'impulsion. L'arduino Due permet d'atteindre des fréquences plus de 10 fois supérieures à un Arduino 8 bits. L'objectif est d'alimenter un haut parleur (ou une autre charge inductive) avec un signal audio, par l'intermédiaire d'un pont en H piloté par le signal PWM.

2. Modulation de largeur d'impulsion

La figure suivante montre le fonctionnement de la modulation de largeur d'impulsion (MLI). Une porteuse triangulaire est comparée à un signal de consigne, par exemple une sinusoïde. Le signal de consigne doit avoir une fréquence bien plus petite que la porteuse. Le signal de sortie est au niveau haut (disons 5 V) lorsque la consigne est supérieure à la porteuse, au niveau bas (0 V) dans le cas contraire. On considère le cas d'un signal de consigne à valeurs positives. Pour traiter un signal alternatif, il suffira de lui appliquer un décalage.



Le signal PWM obtenu doit subir un filtrage passe-bas pour en extraire le signal de consigne. Pour comprendre le principe de cette restitution, considérons le cas d'une consigne constante égale à $U(t) = U_0$. Le signal PWM est alors un signal carré dont le rapport cyclique est $\alpha = U_0/m$, où m est la valeur maximale de la porteuse. La moyenne de ce signal carré est précisément égale à U_0.

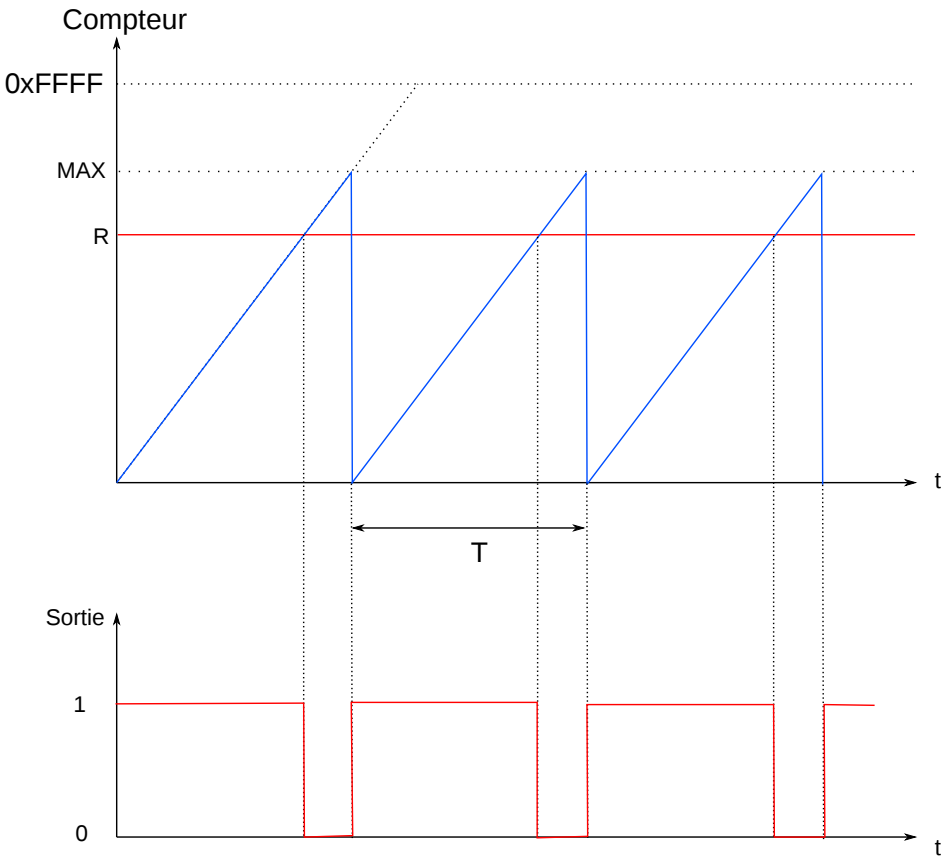
Lorsque la consigne est lentement variable par rapport à la porteuse, il faudra appliquer un filtrage passe-bas pour restituer les variations de basses fréquences de la consigne. En pratique, le signal PWM est utilisé pour commander un circuit de puissance travaillant en commutation, et le filtrage passe-bas est assuré par une bobine en série avec la charge. Pour commander un pont en H, il faudra aussi disposer du signal complémentaire, obtenu avec une porte NON. Le générateur PWM de l'Arduino Due nous permettra d'obtenir directement ce signal complémentaire.

3. Programmation du module PWM

Le microcontrôleur SAM3X8E dispose d'un module PWM comportant 8 voies indépendantes. Chacune peut générer un signal PWM et son complémentaire, pour piloter un circuit de découpage comme un pont en H.

Le fonctionnement de ce module est détaillé dans la documentation du microcontrôleur. L'utilisation directe des registres étant complexe, nous allons utiliser l'API Atmel. Les fonctions de configuration du PWM sont définies dans le fichier `sam/system/libsam/source/pwmc.c`. Un exemple d'utilisation de ces fonctions apparaît dans la fonction `analogWrite` de l'API Arduino, définie dans `sam/cores/arduino/wiring_analog.c`.

Un générateur PWM fonctionne avec un compteur 32 bits et une horloge. À chaque top d'horloge, le compteur est incrémenté d'une unité. Lorsqu'il atteint une valeur maximale MAX, il revient à 0, puis le cycle recommence. Le rapport cyclique est fixé par un registre dont la valeur R est inférieure à MAX. À chaque incrémentation, le compteur est comparé à R. Si le compteur est égal à R, la sortie bascule au niveau bas. Lorsque le compteur est remis à zéro, la sortie bascule au niveau haut. On obtient ainsi un signal carré dont le rapport cyclique est R/MAX.



La fréquence d'horloge du compteur est la fréquence de base du microcontrôleur (84 MHz), ou bien un sous-multiple. Pour notre application, on souhaite générer un signal PWM dont la fréquence est de l'ordre de 100 kHz. La fréquence de base sera donc utilisée. Si f est la fréquence du signal, la fréquence de l'horloge doit être $f * FMAX$. Pour atteindre des fréquences de l'ordre de 100 kHz, il faut donc se limiter à une valeur de FMAX faible, par exemple :

```
#define MAX_PWM_VALUE 400
```

Cette valeur permet d'atteindre la fréquence $84/400 = 0,210\ MHz$. La résolution sur les valeurs de rapport cyclique est égale à 1/400. Si l'on veut une meilleure résolution, il faut augmenter MAX et donc se contenter d'une fréquence maximale moins grande.

La première étape de la configuration est l'activation du périphérique PWM :

```
pmc_enable_periph_clk(PWM_INTERFACE_ID);
```

Si `frequence` contient la fréquence souhaitée en Hz, la configuration de l'horloge se fait de la manière suivante :

```
PWMC_ConfigureClocks(frequency*MAX_PWM_VALUE,0,VARIANT_MCK);
```

La macro `VARIANT_MCK` définit la fréquence de base, ici 84 MHz. Il y a en fait deux horloges A et B. Ici, on utilise seulement l'horloge A.

Chaque voie génère plusieurs sorties. La table des sorties figure dans la documentation du SAM3X8E. Il faut aussi consulter la table de correspondance de l'Arduino Due, pour connaitre les bornes de la carte reliées aux sorties. Nous allons générer deux signaux PWM sur les sorties suivantes :

- ▷ Signal PWML6 (voie 6 Low) : sortie PC23 -> borne 7
- ▷ Signal PWMH6 (voie 6 High) : sortie PC18 -> borne 45
- ▷ Signal PWML5 (voie 5 Low) : sortie PC22 -> borne 8
- ▷ Signal PWMH5 (voie 5 High) : sortie PC19 -> borne 44

Pour chaque voie, les signaux Low et High sont complémentaires. On pourra donc utiliser les bornes 7 et 45 pour piloter un pont en H.

La configuration des deux sorties pour une voie se fait de la manière suivante

```
uint32_t ulPin1 = 7;
uint32_t ulPin2 = 45;
PIO_Configure(g_APinDescription[ulPin1].pPort,
                    g_APinDescription[ulPin1].ulPinType,
                    g_APinDescription[ulPin1].ulPin,
                    g_APinDescription[ulPin1].ulPinConfiguration);
PIO_Configure(g_APinDescription[ulPin2].pPort,
                    g_APinDescription[ulPin1].ulPinType,
                    g_APinDescription[ulPin2].ulPin,
                    g_APinDescription[ulPin2].ulPinConfiguration);
```

Si `chan` contient la voie PWM à utiliser (ici 6), la configuration de la voie se fait par :

```
PWMC_ConfigureChannel(PWM_INTERFACE, chan, PWM_CMR_CPRE_CLKA, 0, 0);
```

On précise ici que l'horloge A doit être utilisée.

La période est définie en donnant la valeur MAX :

```
PWMC_SetPeriod(PWM_INTERFACE, chan, MAX_PWM_VALUE);
```

Si `ulvalue` contient le rapport cyclique R, celui-ci est défini par :

```
PWMC_SetDutyCycle(PWM_INTERFACE, chan, ulValue);
```

Pour finir, on active la voie pour déclencher la génération des signaux :

```
PWMC_EnableChannel(PWM_INTERFACE, chan);
```

Lorsque qu'on commande un circuit de commutation comme un pont en H, il peut être nécessaire d'introduire un temps mort (dead time) entre l'ouverture des deux portes et la fermeture des deux portes complémentaires, afin d'éviter un court-circuit transitoire par les transistors, qui peut se produire lorsque le temps de commutation des transistors n'est pas négligeable devant la période. Voici comment configurer les temps morts (à faire avant l'activation de la voie) :

```
PWM_INTERFACE->PWM_CH_NUM[chan].PWM_CMR |= PWM_CMR_DTE; // dead time enable
uint32_t dead_time = MAX_PWM_VALUE/20;
PWMC_SetDeadTime(PWM_INTERFACE, chan, dead_time, dead_time);
```

On définit le même temps mort pour les signaux High et Low, égal à 1/20 de la période.

Si l'on utilise un pont en H intégré comme le L6203, les temps morts sont introduits par le circuit logique interne du pont, donc il n'est pas nécessaire de le placer ici.

4. Génération du signal périodique de consigne

Pour générer le signal périodique de consigne $U(t)$, on utilise la méthode développée dans Synthèse numérique d'un signal périodique. Les échantillons du signal sur une période sont stockés dans une table, auquel on accède avec un accumulateur de phase. Un Timer est programmé pour générer des interruptions à la fréquence d'échantillonnage. À chaque interruption, la valeur du rapport cyclique (R) est modifiée.

5. Programme Arduino

Voici tout d'abord les constantes et les variables globales :
ArduinoDueGenerateurPWM.ino

```
#include "Arduino.h"
#define MAX_PWM_VALUE 400
// pin 7 = PC23 = PWML6
// pin 45  = PC18 = PWMH6
// pin 8 = PC22 = PWML5
// pin 44 = PC19 = PWMH5

#define NECHANT 128
#define SHIFT_ACCUM 25
uint32_t table_onde[NECHANT];
uint32_t accum1,accum2,increm;
volatile void (*TC0_function)();
char pwm_chan[2] = {6,5};
char pwm_pin_1[2] = {7,8};
char pwm_pin_2[2] = {45,44};
```

La fonction suivante configure et déclenche le Timer TC0 pour les interruptions. `ticks` est le nombre de tops d'horloge pour la période d'échantillonnage (horloge 42 MHz). `function` est la fonction à exécuter lors de l'interruption.

```
void declencher_timer(uint32_t ticks, volatile void (*function)()) {
   uint8_t clock = TC_CMR_TCCLKS_TIMER_CLOCK1; // horloge 84MHz/2=42 MHz
   uint32_t channel = 0;
   TC0_function = function;
   pmc_set_writeprotect(false);
   pmc_enable_periph_clk((uint32_t)TC0_IRQn);
   TC_Configure(TC0, channel, TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC | clock);
   TC0->TC_CHANNEL[channel].TC_RC = ticks;
   TC_Start(TC0, channel);
   TC0->TC_CHANNEL[channel].TC_IER=TC_IER_CPCS;
   TC0->TC_CHANNEL[channel].TC_IDR=~TC_IER_CPCS;
   NVIC_EnableIRQ(TC0_IRQn);
}
```

Voici le gestionnaire d'interruption :

```
void TC0_Handler()
{
   TC_GetStatus(TC0, 0);
   (*TC0_function)();
}
```

La fonction suivante configure et déclenche la génération PWM pour une voie définie dans les tableaux `pwm_chan`,`pwm_pin1`,`pwm_pin2`. La fréquence du signal PWM est donnée en Hz. `ulvalue` est la valeur initiale du rapport cyclique, qui sera modifiée lors des interruptions.

```
void init_pwm(int i, uint32_t frequency, uint32_t ulValue) {
   pmc_enable_periph_clk(PWM_INTERFACE_ID);
   PWMC_ConfigureClocks(frequency*MAX_PWM_VALUE,0,VARIANT_MCK);
   uint32_t chan = pwm_chan[i];
   uint32_t ulPin1 = pwm_pin_1[i];
   uint32_t ulPin2 = pwm_pin_2[i];
   PIO_Configure(g_APinDescription[ulPin1].pPort,
                 g_APinDescription[ulPin1].ulPinType,
                 g_APinDescription[ulPin1].ulPin,
                 g_APinDescription[ulPin1].ulPinConfiguration);
   PIO_Configure(g_APinDescription[ulPin2].pPort,
                 g_APinDescription[ulPin1].ulPinType,
                 g_APinDescription[ulPin2].ulPin,
                 g_APinDescription[ulPin2].ulPinConfiguration);

  PWMC_ConfigureChannel(PWM_INTERFACE, chan, PWM_CMR_CPRE_CLKA, 0, 0);
```

```
  PWM_INTERFACE->PWM_CH_NUM[chan].PWM_CMR |= PWM_CMR_DTE; // dead time enable
  PWMC_SetPeriod(PWM_INTERFACE, chan, MAX_PWM_VALUE);
  PWMC_SetDutyCycle(PWM_INTERFACE, chan, ulValue);
  //uint32_t dead_time = MAX_PWM_VALUE/20;
  //PWMC_SetDeadTime(PWM_INTERFACE, chan, dead_time, dead_time);
  PWMC_EnableChannel(PWM_INTERFACE, chan);
}
```

Voici la fonction qu'il faut appeler lors des interruption. Elle se charge de lire la table avec les accumulateurs de phase et de modifier les rapports cycliques des deux voies utilisées :

```
volatile void synthese_table() {
    accum1+= increm;
    accum2+= increm;
    PWM_INTERFACE->PWM_CH_NUM[pwm_chan[0]].PWM_CDTYUPD = table_onde[accum1 >> SHIFT_A
    PWM_INTERFACE->PWM_CH_NUM[pwm_chan[1]].PWM_CDTYUPD = table_onde[accum2 >> SHIFT_A
}
```

Pour aller plus vite, on accède directement au registre `PWM_CDTYUPD` au lieu d'appeler la fonction `PWMC_SetDutyCycle`.

La fonction suivante remplit la table avec une sinusoïde dont l'amplitude est donnée :

```
void set_sinus_table(float amp) {
  int i;
  float dt = 2*3.1415926/NECHANT;
  for(i=0; i<NECHANT; i++) {
    table_onde[i] = MAX_PWM_VALUE*0.5*(1.0+amp*sin(i*dt));
  }
}
```

Voici enfin la fonction `setup`, qui définit la fréquence du PWM (100 kHz), la fréquence d'échantillonnage (42 kHz), la fréquence de la sinusoïde (1 kHz). Elle calcule l'incrément des accumulateurs de phase. Le second accumulateur est initialisé de manière à donner une sinusoïde en quadrature avec la première.

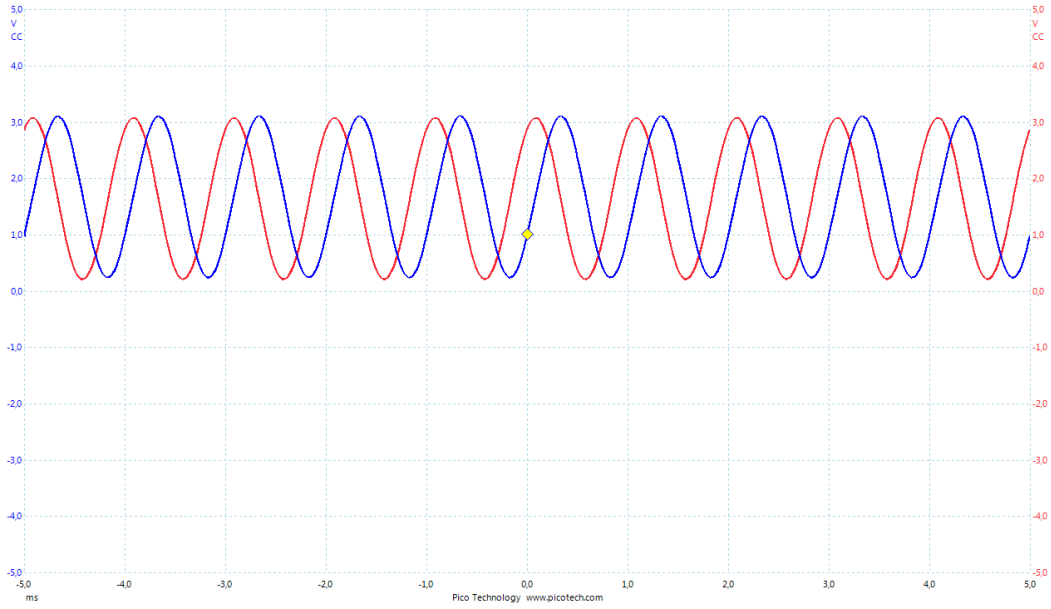
```
void setup() {
  set_sinus_table(1.0);
  uint32_t pwm_freq = 100000; // fréquence du PWM en Hz
  init_pwm(0,pwm_freq,0);
  init_pwm(1,pwm_freq,0);
  uint32_t frequence = 1000; // Fréquence de la sinusoïde en Hz
  uint32_t fechant = 42000; // Fréquence d'échantillonnage en Hz
  uint32_t techant = 1.0/fechant; // en secondes
  uint32_t ticks = 42000000/fechant; // nombre de tops d'horloges à 42 MHz pour la pé
  increm = (uint32_t) (((float)(0xFFFFFFFF))*((float)(frequence)*techant)); // incrém
```

```
  accum1 = 0;
  accum2 = ((uint32_t)(NECHANT * 0.25)) << SHIFT_ACCUM;
  declencher_timer(ticks,synthese_table);
}


void loop() {


}
```

6. Test avec un filtre RC

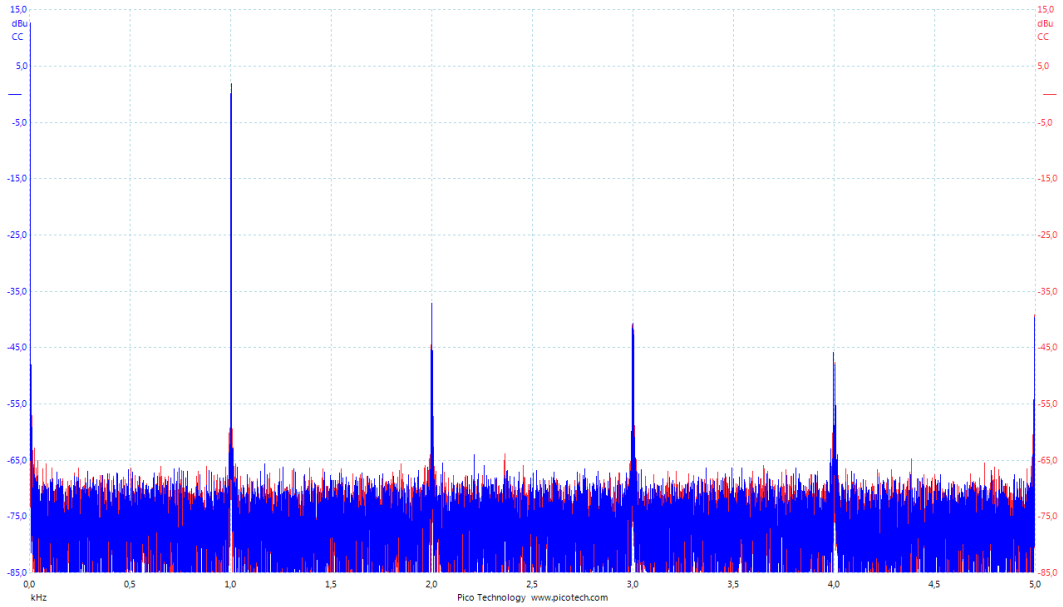
Le filtrage du signal est testé avec un filtre RC. Les valeurs $R = 8200\ \Omega$ et $C = 10\ nF$ donnent une fréquence de coupure de $1,9\ kHz$, qui nous permet de générer des signaux jusqu'à $1000\ Hz$ tout en assurant une très bonne atténuation des harmoniques du PWM, dont le fondamental est à $100\ kHz$.

Voici les deux signaux après filtrage :



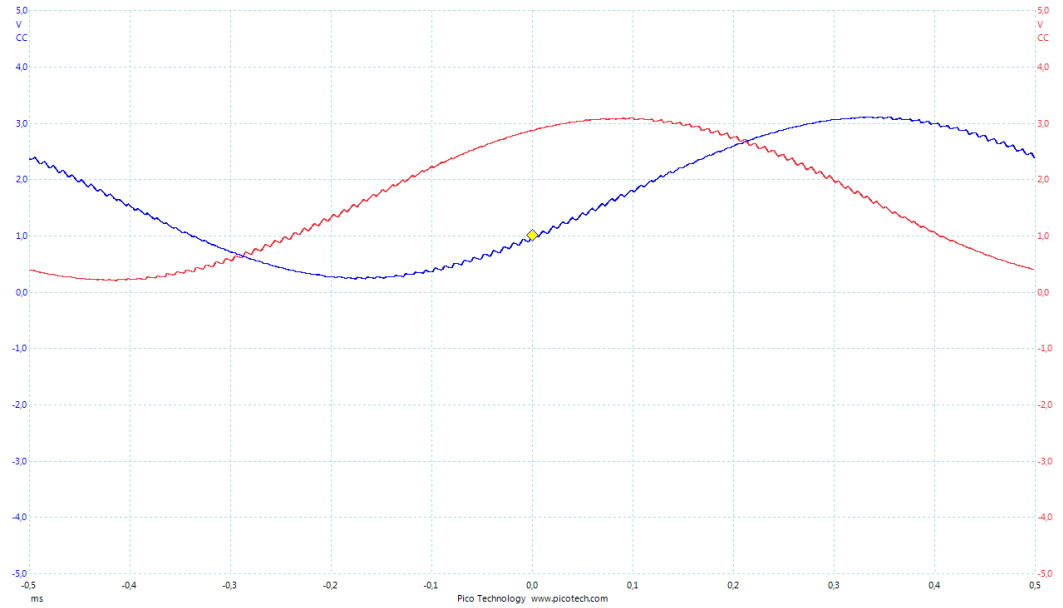
L'amplitude maximale des tensions de sortie s'étend de 0 à $3,3\ V$. Sur cet exemple, la sinusoïde a l'amplitude maximale mais le gain du filtre RC est notablement inférieur à 1.

Une analyse spectrale permet de vérifier la fréquence :

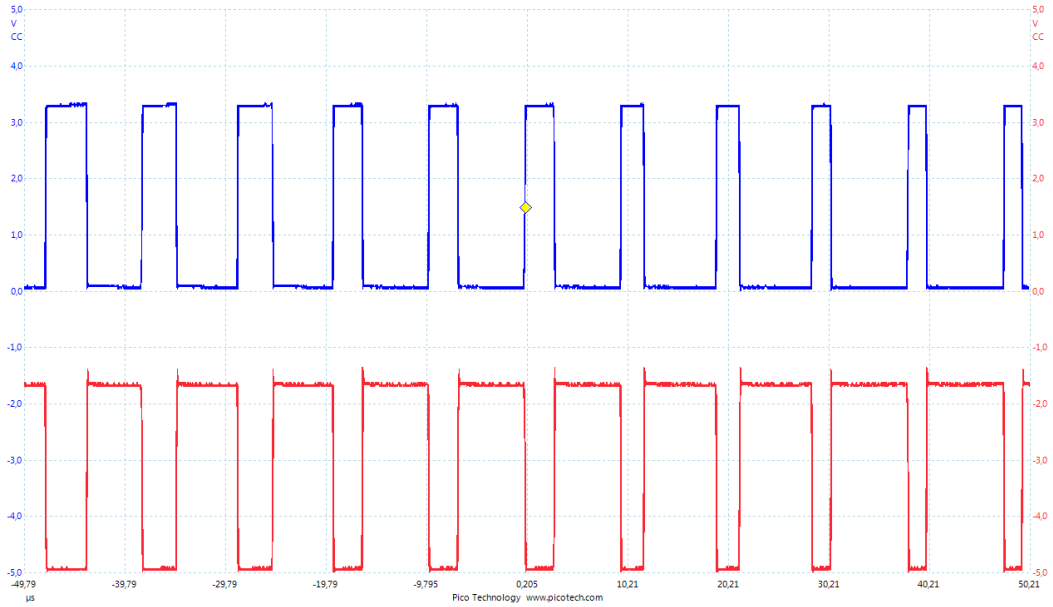


La distorsion est très faible, puisque l'harmonique de rang 2 se trouve à environ $-35\ dB$.

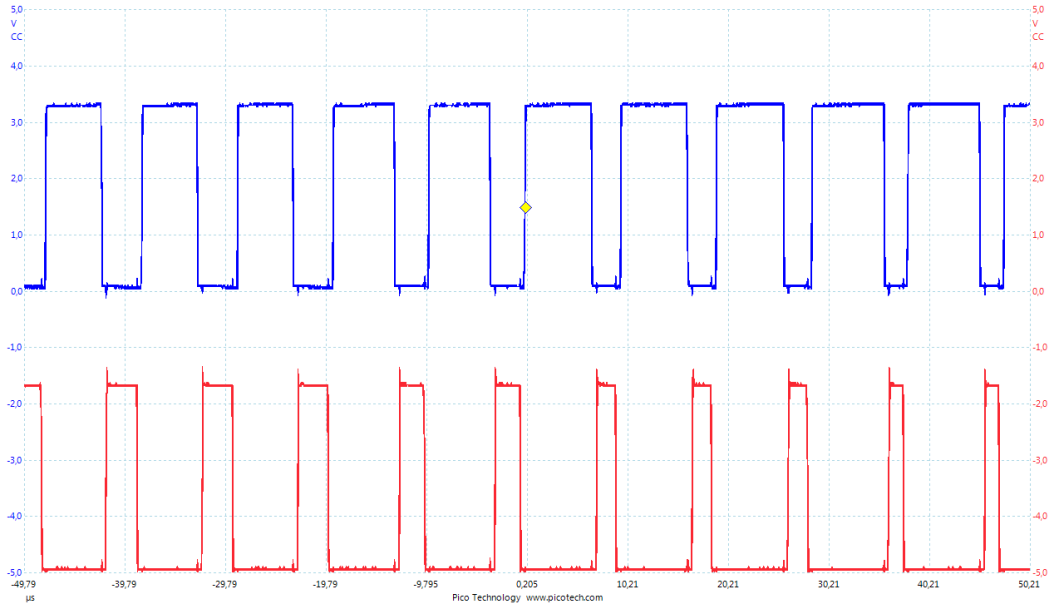
Voici un détail qui permet de voir l'ondulation résiduelle :



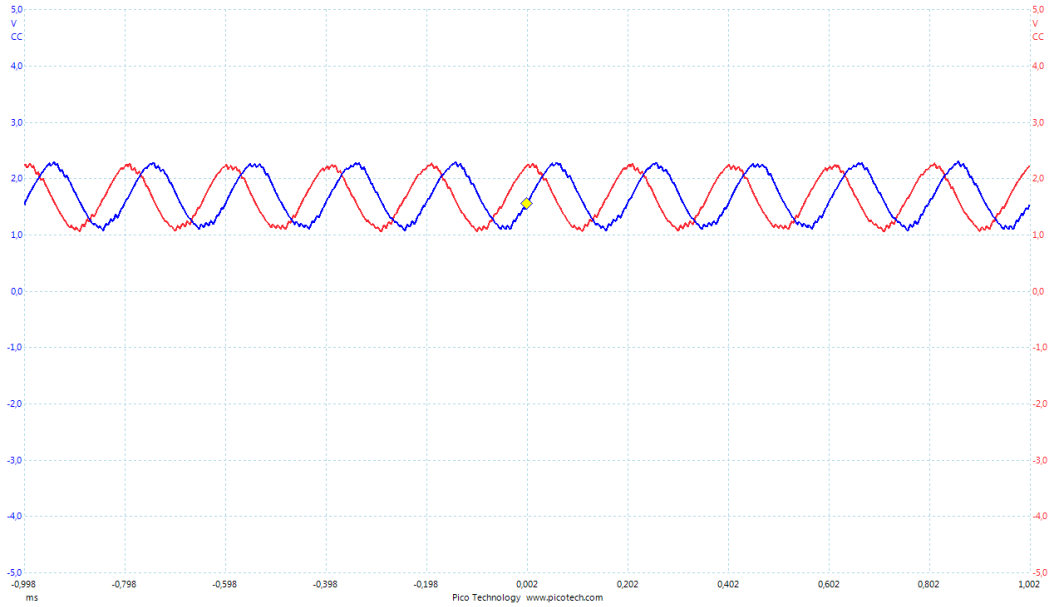
Voici un détail des deux signaux complémentaires (High et Low) PWM non filtrés, pour une des deux voies, sans temps mort :



Voici un autre détail avec un temps mort égal à 1/20 ième de la période :



Voici les signaux générés à 5000 Hz :



L'amplitude est plus petite à cause du filtre RC. La sinusoïde et le signal PWM sont dans la bande atténuante du filtre. Les ondulations sont bien visibles. Pour les réduire, il faudrait utiliser un filtre plus sélectif.

Pour un signal audio où de telles fréquences sont relativement rares, le résultat est acceptable. En pratique, le haut-parleur serait associé à un filtre LC d'ordre 2, ce qui permet d'augmenter la fréquence de coupure.

7. Conclusion

L'Arduino Due permet de générer un signal par modulation de largeur d'impulsion. Avec une fréquence de porteuse de 100 kHz et une fréquence d'échantillonnage de 42 kHz, il est possible de générer des signaux audio, à condition d'accepter une atténuation des composantes supérieures à 1000 Hz. Cette atténuation pourra être compensée partiellement par un filtrage numérique préalable du signal. On peut alimenter directement un haut-parleur avec un pont en H pouvant fonctionner à 100 kHz. Un filtrage LC d'ordre 2 devrait permettre d'augmenter la fréquence de coupure du filtre passe-bas tout en assurant l'élimination de la fréquence 100 kHz du signal PWM.

Références

[1] L. Lasne, *Electronique de puissance*, (Dunod, 2015)