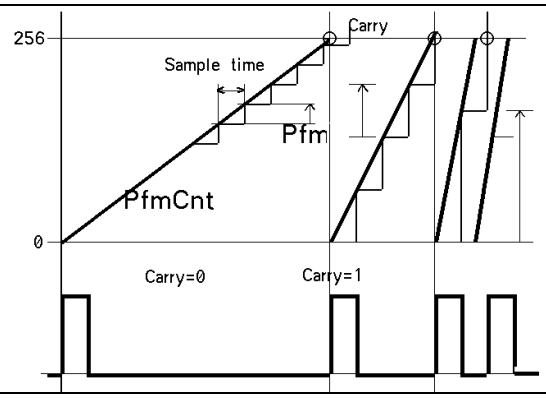
Première version de www.didel.com/diduino/PfmPratique.pdf

PFM par interruption

Le PFM permet des vitesses lentes, voir très lentes avec des moteurs ayant un mauvais rendement, comme les moteurs jouet. Si on veut le même résultat avec du PWM, il faut asservir la vitesse, donc avoir un encodeur qui mesure précisément cette vitesse.
Le principe du PWM (Pulse Width Modulation), du PFM (Pulse Frequency Modulation) et du BCM (Binary Coded Modulation) est expliqué sous www.didel.com/kidules/PwmPfm.pdf . Le PPM (Pulse Position Modulation) www.didel.com/kidules/Servos.pdf a d'autres objectifs.
Ce document se concentre sur la programmation du PFM en C et le test avecArduino.
Un document similaire moins complet montre l'utilisation avec Pinguino
www.didel.com/kidules/KiPwm.pdf
La transposition avec d'autres cartes et d'autres compilateurs est facile pour celui qui connaît sa carte et son compilateur.

Algorithme

La figure ci-contre montre comment se calcule du PFM 8 bits. Toutes les 5ms (pour un moteur de 10 à 20 mm) on ajoute la valeur Pfm à une variable PfmCnt. Si cette valeur dépasse 256, on génère une impulsion de 5ms.
Une précision de 4 bits nous suffit Si le résultat est supérieur à 16, on génère l'impulsion et on masque pour ne garder que 4 bits. Si Pfm=0, PfmCnt ne varie pas et il n'y a jamais d'impulsion. Si Pfm = 16, il y a dépassement à chaque cycle, et la sortie est continument à 1.



Notre algorithme PFM se contente de 16 niveaux (4 bits plus le signe). Pour toutes les applications, 3 bits seraient mêmes suffisants étant donné le manque de précision pour les capteurs, les moteurs, le glissement sur les sol.
On pourrait, toutes les 5 ms, de calculer PfmCnt += Pfm et décider ce qu'il faut faire d'autre (lire les capteurs, modifier les Pfms selon les valeurs lues). Ces instructions sont en général rapides et on attend 4-5 ms avant de recommencer cette boucle.
C'est plus élégant d'utiliser une interruption toutes les 5ms. L'interruption génère les impulsions PFM et le programme principal est libre pour gérer les moustaches, que l'on peut mettre ensuite dans la routine d'interruption.

PFM sans interruptions

L'étape suivante est de tester le PFM, avec le programme le plus simple et familier qui permet en plus de voir comment le moteur réagit. Le moteur est branché sur les pins 4 et 5 (le moteur en 6,7 est aussi sélectionné) On raisonne avec le port D dont les bits 4,5,6,7 commandent les moteurs avec les mêmes numéros de pin Arduino.

Dans le setup, on force les bits 7..4 à l'état 1 avec un OU, pour ne pas modifier les bits 3..0. Si on écrivait DDRD = 0xF0 ; on forcerait en entrée les pins Rx Tx d'Arduino, que l'on ne doit pas toucher.

Chaque fois que l'addition dépasse 15, on masque pour rester en 4 bits et on active le moteur.

```
// Pfm0.ino| test simple
#define MotAv 0b0110000
#define MotStop 0x00
char  pfm = 7;  // 0 .. 16
char pfmCnt = 0;
#define DelPfm 8   // durée min pour débloquer moteur

void setup() {
  DDRD = DDRD | 0xF0 ; // out
}
void loop() {
  delay (DelPfm);
  pfmCnt += pfm ;
  if (pfmCnt > 15) // 4 bits 0 ... 15 {
    pfmCnt &= 0b1111 ;
    PORTD = MotAv ;
  }
  else  PORTD = MotStop ;

}
```

On a donc des valeurs pfm entre 0 et 16 pour avoir entre 0 et 100% de puissance. le délai dans la boucle dépends des moteurs: l'énergie reçue doit être suffisante pour que le rotor fasse une fraction de tour. Cette durée est de l'ordre de 2ms pour un petit moteur pager (Bimo) et de 6 à 10 ms pour les moteurs du Ddr qui sont sous-alimenté (moteurs 12V).

Que prévoir si on demande un PFM supérieur à 16? Le plus naturel est de saturer, donc de remplacer par la valeur 16 toutes valeurs supérieures.

On peut imaginer plusieurs solution pour dire que l'on recule. Une variable binaire avanceRecule peut coder la direction, et le paramètre pfm est la valeur absolue.de la vitesse.
Le plus naturel est de définir des vitesses négatives, ce qui implique de bien comprendre le codage des nombres négatifs et de programmer le bon type.
Utilisons le type char, 8 bits signés (byte est non signé), pour le pfm. Donc de 0 à 16 on a la vitesse qui augmente, de 16 à 127, elle est saturée à 16. De –1 à –16, on a une vitesse en sens inverse qui augmente, et est saturée à -16 jusqu'à -128.
Faut-il rappeler que –1 est codé 0xFF et –128 est codé 0x80. Tous les vitesses négatives ont le bit de poids fort à 1, c'est ce que le processeur teste pour savoir si c'est négatif. Avec le type byte non signé, 0x80 est plus grand que 0x7F. Avec le type char, 0x80 est plus petit, puisqu'il est négatif.

Ecrivons la boucle pour le pfm signé 4 bits. On teste ici séparément les deux cas. Commander les deux moteurs en même temps nous simplifie la vie.

```
 void loop ()---- ne marche pas???
{
   delay (8) ;   // test avec 8 ms
    j++;  if  (j>15) j=1 ;
    if (pfmD >=0) // pfmD  positif
    {
      if (pfmD > 15)  pfmD = 15 ;   // sature
      if (pfmD >= j) PORTD |= MotDAv ; // active
      else PORTD &= !MotDAv ; // efface
    }

    if (pfmD <0) // pfmD  négatif
    {
      if (pfmD < -16)  pfmD = -16 ;  // sature
      if (-pfmD >= j) PORTD |= MotDRec ;
      else PORTD &= !MotDRec ;
    }
}
```

Commande de 2 moteurs
Les instructions Pfm ont été regroupées dans une procédure DoPfm appelée par l'interruption. Cette procédure peut être mise dans un fichier .h importé.
On remarque que le test du Pfm négatif se fait à deux endroits, pour dédoubler le moins d'instructions possibles.

```
//PfmInt  moteurs droite et gauche
#define MotGRec 0x80
#define MotGAv 0x40
#define MotGStop 0b00111111 //~0xC0
#define MotDAv 0x20
#define MotDRec 0x10
#define MotDStop 0b11001111  //~0x30

// global variables
char pfmD, pfmG  ;  // 0 .. 16  -1..-16

// used inside procedureDoPfm only
char ptempD, ptempG ;
char pfmCntD, pfmCntG;
byte cnt ;
```

```
void DoPfm (char pG,char pD)
{
   if (pD <0) ptempD = -pD ;// pD  négatif
   else ptempD = pD ;
   if (ptempD > 16)  ptempD = 16 ;  // sature
     pfmCntD += ptempD ;
     if (pfmCntD > 15) // 4 bits 0 ... 15
     {
        pfmCntD &= 0b1111 ;
        if (pD <0) PORTD |= MotDRec ;
        else PORTD |= MotDAv ;
     }
     else  PORTD &= MotDStop ;

   if (pG <0) ptempG = -pG ;// pG  négatif
   else ptempG = pG ;
   if (ptempG > 16)  ptempG = 16 ;  // sature
     pfmCntG += ptempG ;
     if (pfmCntG > 15) // 4 bits 0 ... 15
     {
        pfmCntG &= 0b1111 ;
        if (pG <0) PORTD |= MotGRec ;
        else PORTD |= MotGAv ;
     }
     else  PORTD &= MotGStop ;
}  // fin DoPfm
```

```
void setup()
{
  cli();
  TCCR2A = 0; //default
  TCCR2B = 0b00000010;
  TIMSK2 = 0b00000001;
  DDRD = DDRD | 0xFC ;
   sei();
}

ISR (TIMER2_OVF_vect)
{
  TCNT2 = 50;  //  100 us
  if (cnt++ > 40)  // 4ms
  {
    cnt = 0;
    DoPfm (pfmD,pfmG);  // 10u
  }
}

void loop ()
{
   pfmD = 3 ;
   pfmG = -6 ;
   delay (1000) ;
pfmD = 100;
```

```
  delay (1000) ;
}
```

Vitesses –16 à +16 corrigées

```
//PfmAvRecTable.ino  moteurs droite et gauche
//\\xBotDef.h

#define MotGRec 0x80
#define MotGAv 0x40
#define MotGStop 0b00111111 //~0xC0
#define MotDAv 0x20
#define MotDRec 0x10
#define MotDStop 0b11001111  //~0x30
char pfmD ;
char pfmG ;  // 0 .. 16  -1..-16

  char ptempD, ptempG ;
  char pfmCntD, pfmCntG;
  byte cnt ;
 char vitD [] = {1,2,2,3, 4,4,5,6, 8,10,12,14
,16,20,28,32};
 char vitG [] = {1,2,2,3, 4,4,5,6, 8,10,12,14
,16,20,28,32};
void DoPfm (char pG,char pD)
{
    if (pD <0) ptempD = -pD ;// pD  négatif
    else ptempD = pD ;
    if (ptempD > 16)  ptempD = 16 ;  // sature
      pfmCntD += ptempD ;
      if (pfmCntD > 15) // 4 bits 0 ... 15
      {
         pfmCntD &= 0b1111 ;
         if (pD <0) PORTD |= MotDRec ;
         else PORTD |= MotDAv ;
      }
      else  PORTD &= MotDStop ;

    if (pG <0) ptempG = -pG ;// pG  négatif
    else ptempG = pG ;
    if (ptempG > 16)  ptempG = 16 ;  // sature
      pfmCntG += ptempG ;
      if (pfmCntG > 15) // 4 bits 0 ... 15
      {
         pfmCntG &= 0b1111 ;
         if (pG <0) PORTD |= MotGRec ;
         else PORTD |= MotGAv ;
      }
      else  PORTD &= MotGStop ;
}   // fin DoPfm

void setup()
{
  cli();
  TCCR2A = 0; //default
  TCCR2B = 0b00000010;  // clk/8
  TIMSK2 = 0b00000001;  // TOIE2
  DDRD = DDRD | 0xFC ;  // 7..4 out
  sei();
}
```

```
ISR (TIMER2_OVF_vect)
{
  TCNT2 = 62;  // 256-200 --> 250x8x50ns = 100 us
  // on ajoutera ici des timers, des servos, etc
  if (cnt++ > 40)  // 4ms
  {
    cnt = 0;
    DoPfm (pfmD,pfmG);  //durée 10us
  }
}

void loop ()  {
  for (byte i=0; i<16; i++)  {
    pfmD = vitD[i]; pfmG = vitG[i];
    delay (100);
  }
  for (byte i=16; i>-16; i--)  {
    pfmD = i; pfmG = i;
    delay (100);
  }
  for (byte i=-16; i<=0; i++)  {
    pfmD = i; pfmG = i;
    delay (100);
  }
   delay (1000) ;
}


idée  10 vitesses
#define MaxVit 32
char vitD [] = {1,2,3,5,7,10,14,19,25,32};
char vitG [] = {1,2,3,5,7,10,14,19,25,32};
    // diff      1 1 2 2 3 4  5  6  7
```

Incomplet, pas vérifié ! -- voir www.didel.com/xbot/LibX.pdef (juin 2015)

```
#include "PfmInt.h"

ISR (TIMER2_OVF_vect)
{
 TCNT2 = 50;  // 256-200 --> 250x8x50ns = 100 us
 // on ajoutera ici des timers, des servos, etc
 if (cnt++ > 40)  // 4ms
 {
   cnt = 0;
   DoPfm (pfmD,pfmG);  //durée 10us
 }
}
```

```
#include "PfmInt.h"


void loop() {
 delay (4);
 DoPfm (pfmD,pfmG);  //durée 10us

// do any non blocking task
}
```