Apprendre le C avec le LearnCbot

Le texte équivalent pour Energia/Msp430G se trouve sous www.pyr.ch/coursera/LC7-msp.pdf

Chap 7 – Moteurs, servos, capteurs, multitâche

Les sources des exemples sont à disposition sous www.didel.com/coursera/LC7ino.zip

7.1 Commande de moteurs

Les moteurs à courant continu existent à partir de 4mm de diamètre et tournent à 100-200 tours par seconde. On les utilise avec des réducteurs. Ils fonctionnent dans une large gamme de tension, entre un minimum qui permet au moteur de lutter contre les frottements et la limite floue d'un échauffement excessif qui réduit la durée de vie. Entre ces extrêmes, il y a des lois simples entre courant, tension, couple, vitesse de rotation. La résistance de bobine et le facteur de réduction caractérise au mieux les moteur 3-12V utilisés dans les applications robotiques simples. L'à-coup de courant au démarrage dépend de cette résistance. Les moteurs jouet ont souvent une tension de démarrage élevée, qui empêche de bons asservissements avec du PWM. Le PFM permet des vitesses très lentes si chaque impulsion PFM est suffisante pour "décoller" le moteur.

Intéressons-nous aux robots; le LearnCbot va enfin mériter la dernière partie de son nom.

Moustaches et obstacles

Pour penser robot en programmant, il faut définir des noms appropriés, par exemple il faudra prendre des décisions quand les moustaches toucheront des obstacles.

```
#define ObsGOn Pous1On
```

permettra d'écrire `if(ObsGOn){..}` pour décider ce qu'il faut faire s'il y a obstacle à gauche.

Moteurs en tout ou rien

Pour les moteurs, il faut agir sur 2 sorties à la fois.

```
#define AvG Led2On; Led1Off
```

Cette macro active la pin 5 (led verte) et désactive la pin 4. Imaginons un moteur câblé entre ces 2 pins, la led verte du LCbot dit alors que le moteur gauche fait avancer le robot. Une macro peut lister plusieurs instructions, à la suite sur une même ligne, la dernière instruction n'a pas de `;` final.
Les 4 macros AvG AvD RecG RecG StopD et StopG commandent le robot en tout-ou-rien.

Exemple 7.11

Le robot démarre si on touche une moustache. Ensuite il avance et recule en tout-ou rien à l'infini

```
//A711 avance et recule
#include "LcDef.h"
#define AvG Led2On; Led1Off
#define RecG Led2Off; Led1On
#define StopG Led2Off; Led1Off
#define AvD Led3On; Led4Off
#define RecD Led3Off; Led4On
#define StopD Led3Off; Led4Off
#define ObsG  Pous1On
#define ObsD  Pous2On
```

```
void setup () {
  LcSetup ();
}
void loop () {
  while (!(ObsG | ObsD)) {}
  while (1) {

    AvD; AvG; delay (1000);

    RecD; RecG; delay (1000);

  }
}
```

PWM

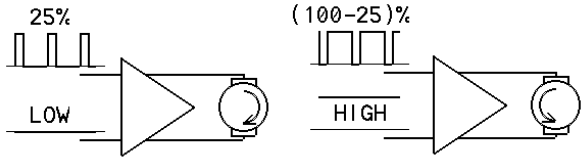
Pour une vitesse proportionnelle sur le moteur droite, la macro

```
#define PwmD(vv) analogWrite(6,vv)
```

permettra d'écrire `PwmD (128);` pour une vitesse 128, moitié de la vitesse maximale (255). Ceci évite de traîner des `analogWrite` non portables dans le programme principal (et c'est plus lisible, non?).

On ne peut évidemment pas mélanger des ordres en tout ou rien et en PWM. La macro Stop; couperait les moteurs le temps d'une instruction seulement. Le timer interne continuer ait à activer la pin selon la valeur PWM mémorisée. Pour arrêter le moteur, il faut programmer une vitesse nulle.

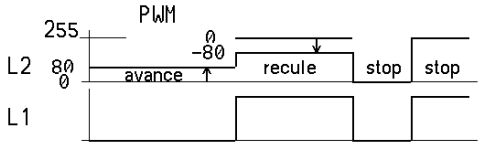
Avancer-reculer en PWM à 25% de vitesse n'est pas évident, puisque les leds rouges (recule) sur les pins 4 et 7 n'ont pas de PWM. Il faut astucer et la solution est d'inverser les signaux sur les deux pins, comme le montre la figure.



Pour avancer, les impulsions sont "vertes". Pour reculer, il faut des impulsion négatives sur le vert et avoir le rouge allumé en permanence. C'est troublant, mais le truc est inévitable si on n'a qu'un canal PWM par moteur.

Exemple 7.12

Le robot avance et recule lentement.




```
//A712.ino Avance et recule
#include "LcDef.h"
#define AvG  Led2On;  Led1Off
#define RecG Led2Off; Led1On
#define StopG Led2Off; Led1Off
#define AvD  Led3On;  Led4Off
#define RecD Led3Off; Led4On
#define StopD Led3Off; Led4Off
```

```
#define PwmD(vv) analogWrite(6,vv)
#define PwmG(vv) analogWrite(5,vv)

  LcSetup ();
}
#define Vit 80
void loop () {
  AvD;  PwmD(Vit);  // on avance
  AvB;  PwmG(Vit);
  delay (2000);
  StopD; StopG;
  RecD; PwmD(256-Vit); // on recule
  RecG; PwmG(256-Vit);
  delay (2000);
  StopD; StopG;
}
```

A noter que l'on pourrait écrire `-Vit` car `256-Vit = 0-Vit = -Vit`, mais cela induirait en erreur. Le paramètre est toujours un nombre positif 8 bits, il n'y a pas de bit de signe.
La fonction Arduino analogWrite(pin,pwm) et gérée par un timer et ne prend que quelques instructions à chaque changement. On peut programmer un PWM bloquant sur n'importe quelle pin. Le PFM et toutefois mieux adapté.

7.2 Commande bidirectionnelle en PFM

Une 3e façon de commander un moteur est par PFM, avec l'avantage de pouvoir tourner très lentement.

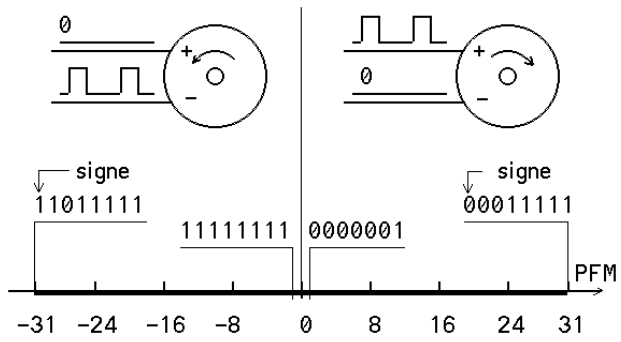
Choisissons 32 niveaux de vitesse. Les vitesses négatives sont en complément et on va saturer les vitesses à +31 et –31, c'est à dire que toute valeur plus grande sera ramenée à +31 ou –31, selon son signe. Le type pour la vitesse est byte. Il n'a pas toutes les propriétés d'un nombre négatif et le compilateur ne doit pas le considérer comme négatif.

```
if (vit&(1<<7)) {  // négatif
  if  (vit > Nniv) { vit = Nniv; }
}
else {
  if  (vit < -Nniv) { vit = -Nniv; }
}
```

Pour des vitesses positives, le cœur de la fonction DoPfm, dérivée du programme vu en 6.71 est:

```
if ((pfmCnt += pfm) > Nniv) {
   pfmCnt -= Nniv;
   AvD;
}
else { StopD; }
```

Pour les vitesse négative, il suffit de permuter les signaux de sortie.




```
// Fonction DoPfm();
byte pwmD;  // paramètre global
void DoPfmD () {
  if (!(pwmD &(1<<7))) {  // positif
    if  (pwmD > Nniv) { pwmD = Nniv; }
    if ((pfmCnt += pwmD) > Nniv) {
      pfmCnt -= Nniv;
      AvD;
    }
    else { StopD; }
  }
  else {  // négatif
    if  (pwmD < -Nniv) { pwmD = -Nniv; }
    if ((pfmCnt -= pwmD) > Nniv) {
```

La fonction `DoPfm();` pour un moteur est donnée ci-contre. Elle doit être appelés toutes les 2ms pour un moteur de 10 à 20mm.

```
      pfmCnt -= Nniv;
      RecD;
    }
    else { StopD; }
  }
}
```

Il faut tester la fonction PFM avec le programme le plus simple possible et vérifiel les conditions limites: PWM = 0; PWM=1; PWM =30; pulse encore PWM=31 32 ...127 continu. PWM = -1 pulse faiblement PWM = -30; pulse encore. PWM= -31, -32, .. –128 continu.

Exemple 7.21

DoPfm est appelé toutes les 2ms. La vitesse augmente toutes les 0.2s. Au bout de 3.2s, elle sature pendant 10 secondes et devient brusquement négative saturée.
On voit l'intensité la led rouge diminuer à partir de –32.
Un temps d'arrêt est programmé avant de recommencer.

Créez le programme A721b qui commande l'autre moteur.

```
//A721.ino Test DoPfmD()
. . . définitions, setup
. . . fonction DoPfmD
byte vitD=0;
void loop () {
  for (int i; i<100; i++) {
    delay (2);
    DoPfmD ();
  }
  vitD++;
  if (vitD++==0) {
    Led1On; delay (500); Led1Off;
  }
}
```

Cette partie PFM étant bien comprise, il faut la cacher dans un #include et de nouveau faire quelques programmes simple pour tester la fonction DoPfmDG() qui agit sur les 2 moteurs.

Exemple 7.22

Le fichier Robot.h sera notre référence dorénavant.
Il contient la fonction DoPfmDG() et les variables globales pfmD, pfmG associées.
Les valeurs pfmD, pfmG vont de –31 à +31 et sont saturées à ces valeurs.
La durée de l'impulsion PFM dépend de la taille (inertie en fait) du moteur. Avec 2ms, un petit moteur démarre pour une fraction de tour.
Les à-coups que l'on voit sur les Leds à faible vitesse sont absorbés par le réducteur et l'inertie du moteur.
Essayez une valeur PFM de 1, 16 et 31.
Secouez le LearnCbot en observant les leds et interprétez.

```
//A722.ino test simple
#include "LcDef.h"
#include "Robot.h"
void setup () {
  LcSetup ();
}
void loop () {
  delay (2);
  pfmD=10; pfmG=20;
  DoPfmDG ();
}
```

On peut imaginer maintenant un programme qui, dans la boucle de 2ms, active le PFM, teste des capteurs, lit des poussoirs, balaye un affichage, chaque tâche non bloquante communiquant par des variables et des flags. Cette boucle dans le programme principal peut être appelée par interruption.

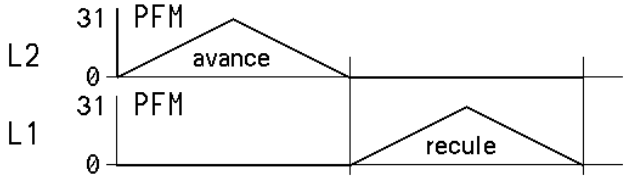
PFM par interruption

Un timer peut appeler la fonction DoPfm par interruption, toutes les 2ms. On voit que c'est très simple et le programme principal peut maintenant avoir des instructions bloquantes sans perturber le PFM. Les variables PFM sont globales, mais il faut les déclarer avec l'attribut volatile.

Exemple 7.23

Le programme A723 passe brusquement de la vitesse max à la vitesse min.
Modifier pour que la vitesse diminue et s'inverse progressivement, comme sur la figure ci-dessous. Le robot fera un aller-retour régulier.

```
//A723.ino PFM par interruption
#include "LcDef.h"
#include "Robot.h"
void setup () {
  LcSetup ();
  TCCR2A = 0; //default
  TCCR2B = 0b00000111; // clk/1024  16kKz  div 32→500Hz
  TIMSK2 = 0b00000001;  // TOIE2
  sei();
}
```




```
ISR (TIMER2_OVF_vect) {
  TCNT2 = -32 ;  // 2ms
  DoPfmDG();
}
void loop () {
  delay (200);
  pfmD++; pfmG++;
  if (pfmD==31) {
    pfmD=-31;  pfmG=-31;  }
}
```

7.3 Moteur pas à pas

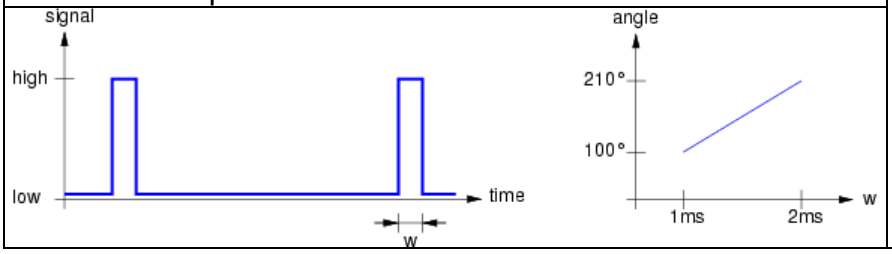
Les moteurs pas-à-pas ont différents câblages et séquences de pas. La séquence d'un moteur donné est mise dans une table, balayée à la vitesse voulue en sens direct ou inverse. Plusieurs librairies existent pour Arduino, et pour les moteurs Lavet utilisés dans les horloges et indicateurs de tableau de bord, voir www.didel.com/kidules/CKiClock.pdf
La programmation génère des séquences à partir d'une table. Modifier une vitesse, un sens de rotation, commander deux moteurs, ne pose pas de problème particulier, et est expliqué dans www.didel.com/kidules/CKiPas2Aig.pdf et www.didel.com/kidules/CKiDelta.pdf .

7.4 Encodeurs

Le principe de l'encodeur incrémental (quadrature encoder) a été vu en 4.7.2. Sur un moteur, la résolution de l'encodeur doit être bonne, pour permettre une mesure assez fine de la vitesse, donc l'utilisation d'un algorithme de réglage de la position sans oscillations. Les encodeurs qui remplacent les potentiomètres sont faciles à gérer; leur résolution et fréquence est faible.

7. 8 Servo de télécommande

Un servo est formé d'un moteur fortement réducté, avec un potentiomètre sur l'axe final. La valeur analogique lue est comparée avec la longueur du signal (durée 1 à 2 ms), reçu toutes les 20 ms. L'angle de rotation varie selon le modèle, et certains servos travaillent de 0.7 à 2.5ms.
S'il y a plusieurs servos, il sont décalés dans le temps pour minimiser les pointes de courant et utiliser un seul timer.



Exemple 7.81

Ce programme teste un servo sur la pin 4 (L1) en passant d'une position extrême à l'autre. On voit sur la Led1 un léger changement d'intensité (la durée `on` est de 1/20 à 2/20 du temps).
La Led13 indique le sens de déplacement du servo.

```
//A781.ino Teste un servo
#include "LcDef.h"
#define ServoOn Led1On
#define ServoOff Led1Off
void setup () {
  LcSetup ();
}
#define MinServo 1000
#define MaxServo 2000
int durServo=MinServo;
void loop () {
  Led13On;
  while (durServo<MaxServo) {
    ServoOn;
    delayMicroseconds (durServo);
    ServoOff;
    durServo +=2; // selon vitesse angulaire
    delayMicroseconds (20000-durServo);
  }
  Led13Off;
 . . . retour très similaire
}
```

Exemple 7.82

Arduino offre une librairie Servo gérée par interruption qui permet de câbler 12 servos vus comme des objets en C++, auxquels on peut donner un nom explicite.
Evidemment cela prend nettement plus de place en mémoire (>2000 bytes).

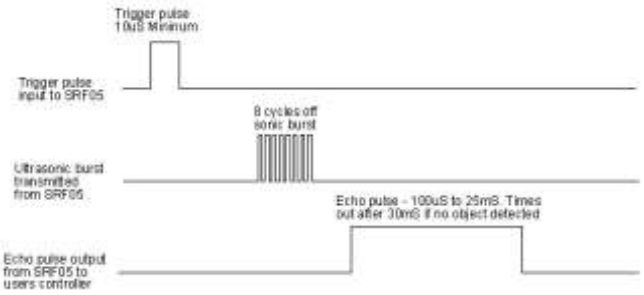
```
//A782.ino Teste la librairie Servo
#include <Servo.h>
Servo brasRobot;
void setup () {
  brasRobot.attach (4);  // sur Pin 4 = L1
}
void loop () {
  brasRobot.writeMicroseconds(1000);
  delay (1000);
  brasRobot.writeMicroseconds(2000);
  delay (1000);
}
```

7.9 Capteur ultrasons

Le capteur de distance à ultrason est facile à comprendre et utiliser.
Les circuits SR05/SR04 ont 4 broches actives:

`Gnd` `Vcc` Alimentation 4-5V
`Trig` Impulsion de 10 us. Déclenche l'envoi
`Echo` Impulsion positive pendant le temps de vol.

Une impulsions sur `Trig` déclenche l'envoi de 8 impulsions à 40 kHz et active le signal `Echo`, qui est désactivé quand l'écho sonore revient. Le min de distance est 2cm (~100 us), le max ~3m. La durée d'un aller retour pour une distance de 25cm est environ 1,4 ms. Si la distance est trop grande, le signal reste à 1 pendant 0.2s, ce qui est fort gênant.
La directivité est très mauvaise. Il faut s'écarter de 20 degrés d'un petit obstacle pour que l'écho vienne de derrière l'obstacle.

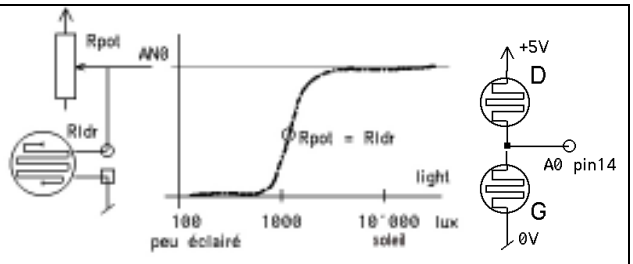



```
// Fonction GetSonar
int GetSonar ()
{
  int dist;
  digitalWrite(Trig,HIGH
  delayMicroseconds(10);
  digitalWrite(Trig,LOW);
  dist=pulseIn(Echo,HIGH);
  return dist;
}
```

La fonction Arduino `pulseIn(pin,level)` rend la durée en microsecondes de l'impulsion sur la `pin`. La polarité est le 2e paramètre.
Cette fonction est bloquante. Si le capteur ne répond pas, on attend éternellement.
Il y a donc une 2e fonction `pulseIn` avec comme 3eme paramètre la durée d'un timout. S'il y a timout, la valeur rendue est 0.

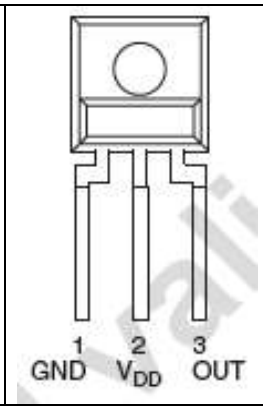
7.10 Capteur de lumière

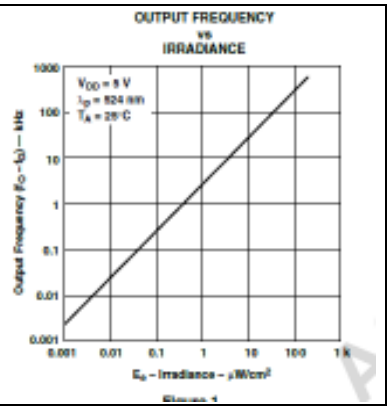
Les photorésistances ou LDR (Light Dependant Resistors) sont faciles à mettre en œuvre. On câble un diviseur de tension et on est heureux de voir sur le terminal une valeur qui change. Utiliser cette valeur dans une application oblige de travailler dans un domaine étroit de luminosité que l'on peut adapter avec un potentiomètre.



Deux LDR en diviseur de tension donnent la différence d'intensité, indépendamment de cette intensité dans une large gamme.

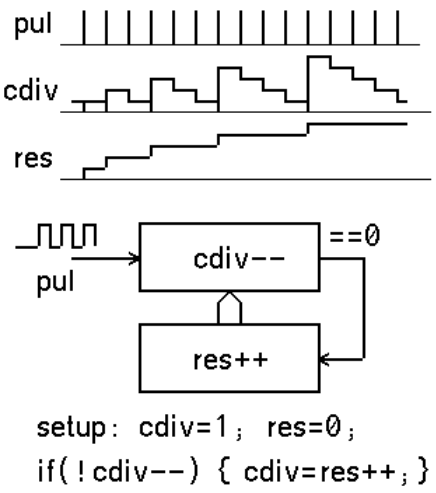
Un capteur intéressant mais relativement cher est le TSL237, qui est un convertisseur lumière-fréquence qui couvre 5 décades. Dans l'obscurité, la fréquence est de quelques Hz et on mesure la période. En plein soleil, on mesure une fréquence quelques centaines de kHz.

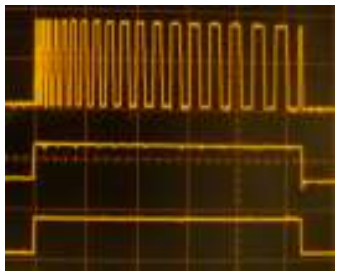




Compteur logarithmique

Notre oeil est sensible au logarithme de l'intensité lumineuse. Une variable intensité lumineuse doit aussi être en valeur logarithmique. Une solution simple est de compter les impulsions du TSL237 pendant 0.1 seconde et avoir une valeur entre 1 (obscurité totale) et ~240 (plein soleil). Le prédiviseur est le nombre d'impulsions déjà comptées.

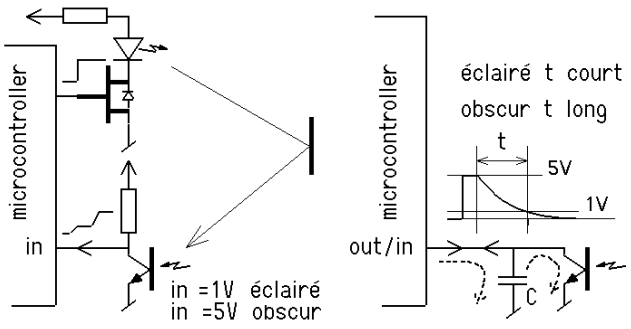




7.11 Capteur Infrarouge

En éclairant avec une diode infrarouge et en mesurant la lumière réfléchie, on s'affranchit partiellement de la lumière ambiante.

La diode émettrice est pulsée pour augmenter la puissance. Le phototransistor récepteur peut être associé à une résistance et lu en analogique (gamme de bonnes mesures étroite).
Il est plus efficace de mesurer le temps de décharge d'un condensateur à travers le phototransistor. Le condensateur est chargé en mettant la pin du microcontrôleur un instant en sortie. La mesure d'un temps permet une gamme de mesure beaucoup plus large, voir www.didel.com/xbot/DistIr.pdf et www.pyr.ch/coursera/CaptDistIrDoc.pdf et l'exemple A7121 plus loin.



La décharge d'une capacité s'applique à tous les capteurs qui ont une résistance variable vers le Gnd, en particulier les LDR vues précédemment.

7.12 Autres capteurs

Les capteurs performant; accéléromètre, boussole, température, humidité, pression, etc. ont de plus en plus des interfaces I2C, avec parfois une option SPI. La mise en oeuvre est facile, mais les librairies on souvent une dimension excessive.

7.13 Afficheurs

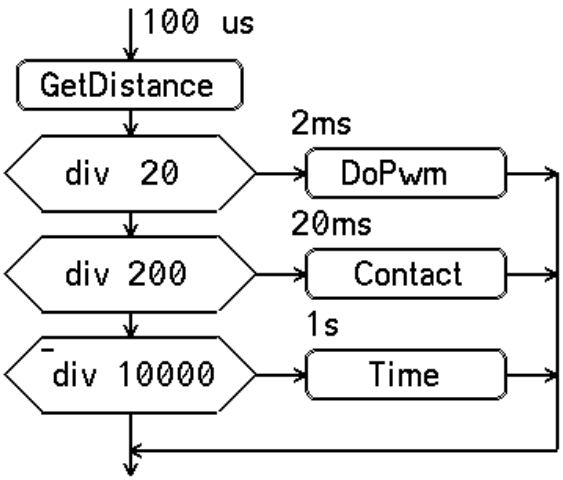
Il y aurait trop à dire sur les différents types d'afficheurs, 7 segments, LCD, Oleds. Ils sont utiles pour l'interaction et la mise au point s'ils ne perturbent pas par la lenteur des mises à jour de l'affichage le processus à analyser.

7.14 Interruptions et temps réel

Gérer plusieurs tâches "simultanément" se fait par interruption. L'appel d'une interruption dure 1-2 microsecondes et la routine exécutée doit être très courte pour que l'on reste en "temps réel", c'est-à dire avec des temps de réponse compatibles avec chaque tâche. Les communications compliquent la gestion des interruptions car les interruptions peuvent intervenir par bouffée et il faut ajouter des mémoires-tampons.
Le microncotrôleur AVR328 a une vingtaine de vecteurs d'interruption (on a vu le Timer2 et l'external interrupt) associés aux périphériques internes: pin change: timers, UART, SPI, I2C et convertisseurs A/D.
Ce n'est pas toujours le plus efficace de passer par les interruptions. Par exemple, pour mesurer une tension analogique, qui dure ~120 microsecondes, on peut lancer la conversion dans le programme principal et activer l'interruption qui va se produire dans 120 us. Que fera-t-elle? En général activer un flag pour dire que le résultat est prêt. Le programme principal, s'il a demandé une valeur analogique, c'est qu'il en a besoin. Il peut éventuellement faire une ou deux choses pour s'occuper en attendant le résultat, mais il va très vite se mettre dans une boucle pour tester le flag et attendre que la conversion soit terminée. Le gain par rapport à la solution bloquante du analogWrite() n'est pas évident.

Le programme communique avec les interruptions par des variables globales et des flags. S'il faut commander un actuateur, il suffit se mettre à jour ses paramètres.
Pour un capteur, un flag signale qu'une nouvelle mesure est prête. Si une interruption régulière effectue la mesure et met à jour la variable, il suffit dans le programme principal de lire la variable préparée par l'interruption précédente.

Une solution élégante pour gérer plusieurs tâches et de les confier à un arbre de décision qui est parcouru toutes les 100 microsecondes par exemple.
Le timer 2 appelle la routine d'interruption et on y traite d'abord les événements rapides. Des compteurs post-diviseurs s'occupent des tâches plus lentes www.didel.com/diduino/LibX.pdf .
Chaque tâche se fait en une dizaine d'instructions au plus. C'est le plus souvent une machine d'état.
Pour la mise au point, on teste chaque routine dans une boucle à 100 microseconde avant de les mettre progressivement ensemble. Cette approche modulaires avec des fichiers .h faciles à adapter est plus efficaces que les librairies Arduino écrites en C.



Le programme A7121 montre comme application un robot qui évite les obstacles sans les toucher. La vitesse du robot (`DoPfmDG()`) et la mesure de distance par infrarouge (`DoDistIR()`) sont géré par interruption. L'action d'une moustache stoppe le robots.

Les distances ont des valeurs entre 0 et 50. Les vitesses entre 0 et 128, saturées à 31. On peut directement envoyer sur le moteur droit la distance gauche, et symétriquement pour l'autre moteur.

```
//A7121.ino Evite murs
#include "LcDef.h"
#include "Robot.h"  // avec DoPfmDG()
#include "DistIr.h"

void setup () {
  LcSetup ();
  SetupDistIr();
  TCCR2A = 0;
  TCCR2B = 0b00000010;   // clk/8 2MHz 100 us  div200
  TIMSK2 = 0b00000001;
  sei();
}
volatile ;
ISR (TIMER2_OVF_vect) {
  TCNT2 = -198 ; //100 us
  DoDistIr();
```

Si la distance diminue, la vitese du moteur éloigné diminue aussi. Le robot va donc tourner et s'éloigner de l'obstacle. Dans un goulet d'étranglement, il va ralentir fortement sa vitesse par rapport à un espace libre.
 	
https://www.youtube.com/watch?v=AULplRbHatU
	`  if (cnt1++ >= 30) { // toutes les 2ms`
	`    cnt1 = 0;`
	`    DoPfmDG();`
	`  }`
	`}`
	`void loop () {  // programme test`
	`  pfmD = distIrG;`
	`  pfmG = distIrD;`
	`  if(ObsD || ObsG) {`
	`    pfmD=0; pfmG=0;`
	`    Led13On;`
	`    for (;;);  // stop`
	`  }`
	`}`

"Projet robot"
Le projet de la dernière vidéo de ce MOOC, décrit sous www.didel.com/coursera/ProjetRobot.pdf n'utilise pas d'interruptions. Le programme se décompose en une successions de tâches simples. Par exemple le robot doit avancer ou tourner par saccades. Une solution était de gérer le PFM par interruption et calibrer des délais pour que les distances et angles soient corrects. Des fonctions Tourne() et Avance() ont été préférées, le paramètre étant un nombre d'impulsions en tout ou rien pour les moteurs. Ré-écrire ce programme en utilisant des machines d'état augmenterait sa lisibilité.

Derniers rappels et compléments
Attribut **volatile**: A mettre pour une variable qui peut être changée par une interruption ou que le compilateur pourrait supprimer car elle n'a pas d'effet.
Attribut **static**: A mettre pour une variable locale qui ne doit pas être perdue d'un appel à l'autre. Elle reste une variable locale, non visible par le programme principal et les autres fonctions.

Un `#define Debug` permet de laisser dans le programme les parties utilisées pour tester, et les remettre en service facilement.
Dans l'exemple ci-contre, si on supprime la ligne
`#define FRENCH` active les lignes `#else`
	`#define FRENCH`
	`  . . .`
	`#ifdef FRENCH`
	`    Serial.print "Bonjour";`
	`#else`
	`    Serial.print "Hello";`
	`#endif`

Conclusion
Ces 7 chapitres et les quelques autres documents cités consolident les notions présentées dans les vidéos du MOOC EPFL "Comprendre les Microcontrôleurs". Ces vidéos présentent de nombreuses imperfections qu'il n'est pas envisageable de corriger.
Nos explications et exercices montrent comment programmer en étant proche du processeur, donc efficace. Lire et comprendre la documentation des fabricants de microcontrôleurs doit être plus facile. Les applications pour gérer des capteurs, animer des leds, commander des mouvement précis sont innombrables.
Le fossé se creuse avec la programmation d'application qui communiquent, en particulier par BlueTooth, et font appel à des librairies complexes et des environnement de développement long à maîtriser. C'est comme en sport: le spectre des engins roulants est très large, du skate board à la formule1. Engagez-vous dans ce qui vous fascine le plus.

jdn 140530/150512/151104