Apprendre le C avec le LearnCbot

Le texte équivalent pour Energia/Msp430G se trouve sous www.pyr.ch/coursera/LC4-msp.pdf

Chap 4 – Fonctions, Arduino, machines à état

4.1 Librairies

Mettons dorénavant toutes les définitions dans un fichier avec l'extension h. La procédure d'adjonction d'une librairie locale, qui permet de couper un programme en parties indépendantes est documenté sous www.didel.com/coursera/FichiersInclus.pdf
Créer des librairies générales suppose des connaissances de C++ et une grande expérience. Nous n'aborderons pas ce sujet, mais nous aurons l'occasion d'utiliser, donc documenter, quelques librairies Arduino.

Les sources des exemples sont à disposition sous www.didel.com/coursera/LC4ino.zip

Exemple 4.11

```
//A411.ino Clignote et bippe
#include "LcDef.h"
void setup () {
 LcSetup ();
}
// Tous nos fichier utilisant le LeanCbot auront ce début, avec comme différence pour le MSP un #include "LcMSPDef.h" et souvent la structure C avec main () et while ().
```

```
void loop () {
    Led1On; delay(300); Led1Off;
    Led2On; delay(300); Led2Off;
    Led3On; delay(300); Led3Off;
    Led4On; delay(300); Led4Off;
    for (int i=0; i<1000; i++) {
      HPToggle; delayMicroseconds (600);
    }
}
```

Rappelons que le grand avantage d'inclure toutes les définitions dans un fichier qui dépend du matériel est que les programmes sont portables. Vous trouvez dans www.didel.com/coursera/LC4ino.zip trois fichiers de définition:

1) `LcDef.h` Pour Arduino utilisant un AVR328 (Uno, Due, Diduino, Crowduino, etc)
2) `LcDefLeonardo.h` Pour cartes Arduino utilisant un Atmega 32U4
3) `LcArduiDef.h` utilisant les pinMode, digitalWrite, etc, donc peu efficace mais portable sur tous les environnement Arduino

Les exercices qui utilisent directement un PORT ne sont en général pas utilisables sur Leonardo.

4.2 #define et fonctions

Avant d'apprendre à écrire des fonctions, comprenons bien comment on peut se référer à des actions en C. La possibilité de faire la même chose de façon différente est toujours troublante pour le débutant.
Pour allumer nos leds, on a abondamment utilisé l'instruction `Led1On;`, résultant d'un

```
#define Led1On digitalWrite (L1,1)
```

Avec cette définition (appelée souvent "macro", le précompilateur fait une simple substitution de caractères (dictionnaire) pour que le programme soit lisible et plus facilement portable.

On peut aussi définir une fonction sans paramètre qui a le même effet:

```
void DoLed1On () {
    digitalWrite (L1,1);  // ou  bitSet (PORTD,bL1); sur Uno/Due avec AVR328
}
```

Mais dans ce cas, l'appel doit mettre en évidence qu'il n'y a pas de paramètre: il faut écrire `DoLed1On();` pour allumer la Led1.
Attention, si on oublie la parenthèse dans un appel de fonction, le compilateur ne signale rien et l'exécution ne se fait pas!

Les instructions de la fonction sont mises dans un coin de la mémoire, à côté du programme, et à chaque appel, le programme saute dans la fonction (routine) et revient à la boucle principale (ou dans une fonction appelante) avec la tâche effectuée.

On peut définir une fonction plus générale pour allumer l'une des quatre Leds, avec un paramètre qui est le no donné à la Led

```
void DoLedOn (byte nn) {
    if (nn==1) { digitalWrite (L1,1); }
    if (nn==2) { digitalWrite (L2,1); }
    if (nn==3) { digitalWrite (L3,1); }
    if (nn==4) { digitalWrite (L4,1); }
}
```

Pour allumer la Led1, il faut écrire `DoLedOn(1);` Une parenthèse vide est signalée comme erreur.

On peut créer une fonction encore plus générale avec 2 paramètres, le 2e paramètre disant si la Led est On (1) ou Off (0). Pour allumer la Led1 il faut écrire `DoLed(1,1);`
En ajoutant la ligne `#define ON 1` on peut écrire `Doled(1,ON);`

Ceci doit vous faire penser au `digitalWrite (pin, niveau);` La fonction Arduino en librairie utilise une vingtaine d'instructions pour agir sur le bon bit du bon port. Vous pouvez écrire cette fonctions avec ce que vous savez, mais il faudra une quarantaine de if !

4.3 Variables locales et globales

Il faut bien comprendre la différence entre les variables globales et les variables ou paramètres locaux. Une variable globale est définie pour toute la suite du programme. Une bonne pratique est de les regrouper devant le "loop",
Une variable locales ne peut être utilisée que dans la fonction ou le bloc où elle est définie. Leur intérêt est d'économiser de la place en mémoire en réutilisant les mêmes positions en mémoire physique pour des variables qui ne sont pas utilisées en même temps.

On a vu par exemple dans une boucle `for` que le compteur appelé traditionnellement `i` a un type déclaré dans la boucle: On écrit `for (int i=0; i< 4; i++) { .. }`.
Le compilateur réserve une position mémoire de 16 bits pour ce compteur, seulement pendant que les instructions entre { } sont exécutées, et cette position mémoire sera affectée à une autre variable locales plus loin. Le nom `i` pourra aussi être réutilisé, c'est un nom local qui n'a pas besoin d'être expressif.
Par contre les noms globaux, que l'on retrouve dans différentes pages du programme, doivent être expressifs; des mois plus tard, on doit savoir immédiatement ce qu'ils signifient.

4.4 Fonction avec des paramètres en entrée

La forme générale est `void FaireCeci (type1 param1, type2 param2) {instructions;}`
Le nombre de paramètres n'est pas limité mais s'il dépasse 2, on ne sait en général plus dans quel ordre ils sont sans aller consulter la documentation!
Une majuscule commence le nom d'une fonction, et ce nom doit exprimer au mieux la fonctionnalité, un verbe est utilisé de préférence. Arduino ne respecte pas toujours cette règle, le C ne fait aucune vérification. Le type de chaque paramètre doit être précisé, même si c'est le même que le précédent.

Exemple 4.41

La fonction delay(); doit être programmée sur un microcontrôleur qui n'a pas la librairie Arduino. Le passage à travers une boucle for prend 5-10 instructions pour s'exécuter, on répète 840 fois pour avoir 1 ms.
La fonction utilise deux boucles imbriquées, la boucle interne est calibrée à 1ms.
Les variables i et j sont locales, mais comme la boucle j n'exécute aucune instruction, le compilateur optimiserait en la supprimant si on n'ajoutait pas l'attribut "volatile".

Chronométrez 60 clignotements.

```
//A441.ino  Fonction FaireDelay (ms);
#include "LcDef.h"
void  setup () {
  LcSetup ();
}
#define Pour1ms 840   // calibré avec un chronomètre
void FaireDelai (int dd) {
   for (int i=0; i<dd; i++) {
      for (volatile int j=0;j<Pour1ms;j++){
      }
   }
}
int duree=500;   // période 1s
void loop () {
  FaireDelai (duree);
  Led2Toggle;
}
```

Exemple 4.42

On veut voir sur la Led1 des salves de clignotement de plus en plus longues.
On utilise la fonction `ClignoteNFoisLed1()` avec comme paramètre le nombre de clignotements.

Modifier ce programme pour que la durée soit une variable globale, comme dans l'exemple précédent, et pas une constante

```
//A442.ino| Salves de clignotement
#include "LcDef.h"
void setup () {
 LcSetup ();
}
#define DurCligno 200  //  ms
void ClignoteNFoisLed1 (byte nn) {
  for (byte i=0; i<nn; i++) {
    Led1On; delay (DurCligno);
    Led1On; delay (DurCligno);
  }
}
byte nbFois = 1 ;
void loop () {
    ClignoteNFoisLed1 (nbFois++);
    delay(500);
}
```

Exemple 4.43

La fonction `CliNFoisDurLed1(x,y)` a maintenant deux paramètres, le nombre de fois et la durée.
Vous remarquez que le nom choisi donne l'ordre des paramètres pour éviter les erreurs d'inattentions.

Pour stopper les clignotements, le dernier `if` mets le processeur en attente dans une boucle while qui ne fait rien. Une solution plus propre serait une boucle `for` suivie du `while(1){}`

```
//A443.ino| On varie la durée et le nombre
#include "LcDef.h"
void setup () {
 LcSetup ();
}
void CliNFoisDurLed1 (byte nn, int dd) {
  for (byte i=0; i<nn; i++) {
    Led1On; delay (dd);
    Led1On; delay (dd);
  }
}
byte nbFois = 1 ;
void loop () {
   CliNFoisDurLed1 (nbFois++,200);
   if (nbFois) > 10 { while (1){} }
}
```

4.5 Fonction avec un paramètre en sortie

La fonction va rendre un paramètre. Par exemple une fonction qui lit un capteur rend la valeur de ce capteur et il n'a pas de différence avec lire une variable:

```
mesure = LitTempérature();
```

La forme générale est

```
type PrendreCeci (type param1, ... ) {
   type var ;
   instructions ;
   return var ;
 }
```

On comprend enfin ce que "`void`" voulait dire: jusqu'à présent, il n'y avait pas de paramètre en sortie à annoncer!
A l'appel de la fonction, le compilateur doit réserver de la place en mémoire, un ou plusieurs bytes, Il faut donc annoncer le type de la variable à rendre.
Ensuite, une variable `var` de même type doit être déclarée pour préparer et rendre le résultat. L'instruction clé est à la fin: `return var;` qui définit ce qui est rendu par la fonction, une valeur numérique qui sera transférée ou testée.

On peut retourner une valeur sans déclarer une variable pour rendre le résultat: l'exemple bien connu est la fonction multiplication:

```
long Multiplie (int aa, int bb) {
   long cc;
   cc = (aa * bb) ;
   return cc ;
 }
```

```
long Multiplie (int aa, int bb) {
   return (aa * bb) ;
 }
```

Plus simple, mais pas très utile:

```
byte Quatre () {
  return 4;
}
```

permet d'écrire ensuite

```
toto = Quatre(); // ne pas oublier les (), c'est une fonction, et pas une définition
```

Exemple 4.51

Arduino documente une fonction `min(a,b)` qui rend le minumum entre a et b. Le type est `int` ou `long`.
Cette fonction est facile à écrire.

Pour la tester, utilisons le terminal dans un mode ou le clavier du PC permet de taper deux nombre décimaux séparés par une virgule et terminés par un retour à la ligne. Cliquer dans le rectangle en haut de page avant de taper les 2 nombres. Ils sont stockés dans une tables et envoyés quand il y a le retour à la ligne. Pour les détails, voir www.didel.com/C/Terminal.pdf

```
// A451 Min (a,b)
void setup() {
  Serial.begin(9600);
}
// fonction GetMin (a,b)
long GetMin (long a, long b) {
  long rr;
  if (a>b) {rr=b;}
  else {rr=a;}
  return rr;
}
byte valA, valB, valMin;
void loop() {
  while (Serial.available() > 0) {
    valeurA = Serial.parseInt();
    valeurB = Serial.parseInt();
    Serial.print(valA); Serial.print("  ");
    Serial.print(valB); Serial.print(" min = ");
    valMin = GetMin (valeurA, valeurB);
    Serial.println(valMin);
  }
}
```

Arduino propose aussi les fonctions suivantes faciles à programmer.
`max (a,b)` rend la valeur la plus grande entre a et b (pour un type signé, -3 < 1)
`constrain (x,a,b)` qui force x dans l'intervalle a,b, donc `x` (pas de changement) si `x>a ET x<b`, `a` si `x<a` et `b` si `x>b`

La documentation sur les fonctions Arduino se trouve en tapant "Arduino nom de la fonction" dans un browser. Une liste résumée se trouve dans notre aide-mémoire www.didel.com/C/Resume.pdf

Exemple 4.52

La fonction Arduino `random ()` génère un nombre au hasard, mais il faut comprendre que ce hasard est calculé, donc donne toujours la même séquence après le reset.
La fonction `randomSeed(valeur)` permet de partir d'une valeur qu'il faut changer à chaque démarrage.
La lecture d'une pin analogique en l'air est différente d'un reset à l'autre et c'est ce que l'on utilise en général. Ici, la valeur est prédéfinie (=22) et a été choisie pour que la combinaisons à reconnaître apparaisse quelques fois au début.
Le programme affiche un motif sur les 4 leds. Il faut agir sur P2 dans la seconde si et seulement si les deux leds vertes sont allumées

```
random (max);        // valeur rendue entre 0 et max-1
random (min,max);    // valeur rendue entre min et max-1
```

Changez la graine (seed) et observez.

```
//A452 Jeu de réflexe
#include "LcDef.h"

void  setup () {
  LcSetup ();
  Serial.begin (9600);
  randomSeed (22);
}
  byte motif;
void loop () {
  motif = random (16) ;
  Serial.println (motif);
  PORTD = motif << 4;
  delay (1000);
  if (PORTD == 0b01100000)  { // 2 leds vertes
    if (Pous2On) { // on clignote si on a fait juste
       Serial.println ("  bravo");
       for (byte i=0 ; i< 10; i++) {
         Led2Toggle; Led3Toggle; delay (200);
       }
    }
    else {
       Serial.println ("  rate");
    }
  }
}
```

Exemple 4.53

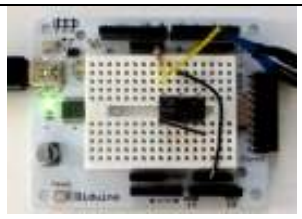
La fonction Arduino `millis();` donne le temps comme une montre. C'est un compteur qui part à zéro au reset et rend une valeur 32 bits quand on l'interroge.(on peut aussi ne lire que 16 bits).
`micros ()` existe aussi, mais est peu précis en dessous de 10 us,.
C'est donc facile par soustraction de connaître la durée d'un événement.
Le programme mesure la durée du premier contact du poussoir P1. Si c'est un rebond, on verra un temps court. Après ce premier contact, un délai de 1s laisse le temps de relâcher la touches, en ignorant les contacts suivants.
Le problème est de trouver un poussoir qui a des

```
//A453 Mesure du 1er rebonds
#include "LcDef.h"
void setup () {
  LcSetup ();
  Serial.begin (9600);
}
long tempsDeb ;
long duree;
void loop () {
  while (!Pous1On) { }   // attend  on
  tempsDeb = micros();

  while (Pous1On) { }   // attend  off
  duree = micros() - tempsDeb;
```

rebonds! Les poussoirs du LearnCbot n'en ont que rarement si on presse normalement.

```
  Serial.println (duree);
  delay (1000);
}
```

Il faut connecter à la place du LearnCbot un gros commutateur ou un microswitch avec une résistance pull-up. Celui de la photo donne assez systématiquement un premier rebond de 8 à 20 microsecondes avec des dizaines qui suivent sur 0.5 ms.



4.6 Fonctions bloquantes et non bloquantes

Avec une fonction bloquante, le programme ne va pas continuer tant que l'on n'est pas sorti de la fonction, ce qui est gênant si on doit en parallèle surveiller des capteurs ou asservir la vitesse d'un moteur.
Les fonctions de clignotement que l'on a vue précédemment sont bloquante, Le `for` et le `while` sont bloquants. Le `loop()` Arduino joue le rôle d'un `while(1)` qui bloque dans une boucle éternelle. A noter que `while (1)` a un équivalent que certains préfèrent: `for (;;)`
Pour clignoter sans bloquer, il faut une fonction où à chaque passage on progresse avec par exemple un comptage de temps. On change l'état de la led quand le comptage est terminé.
Dans une application qui surveille des capteurs et actionneurs, on doit enchaîner des fonctions non bloquantes.

Exemple 4.61

On veut clignoter deux leds à des vitesse différentes. Ecrivons deux fonctions qui seront exécutées toutes les 20ms. Pour une durée de une seconde, on traverse 25 fois la fonction avant de changer l'état de la Led.

La variable `cntLed1` pourrait être déclarée comme variable globale avant la définition de la fonction. Pour la déclarer à l'intérieur de la fonction, il faut lui ajouter l'attribut (qualifier) `static,` qui force le compilateur à la garder en mémoire. Autrement la valeur du compteur serait perdue, donnée à un autre variable quand le programme passe d'une fonction à l'autre.

```
//A461 Clignote L1 et L2
#include "LcDef.h"
void  setup () {
  LcSetup ();
}
void CliLed1 (byte pp) {
  static byte cntLed1;
  if (cntLed1++ > pp) {
    cntLed1 = 0;
    Led1Toggle;
  }
}
void CliLed2 (byte pp) {
  static byte cntLed2;
  if (cntLed2++ > pp) {
    cntLed2 = 0;
    Led2Toggle;
  }
}

void loop () {
  delay (20);
  CliLed1 (1000/20); // 1Hz
  CliLed2 (200/20);  // 5HZ
}
```

Exemple 4.62

Si on ne veut pas la contrainte des 20ms, on peut utiliser l'horloge `millis();`
On mémorise le temps. On remesure le temps chaque fois que l'on passe dans la fonction. Si le temps écoulé est supérieur à la durée de clignotement, on clignote et réactive la mémorisation.

Au démarrage du programme la variable `debCli` vaut 0 et `millis()` a déjà augmenté. Les deux leds passent à un. Il y a autoinitialisation. Une initialisation correcte devrait se faire dans le setup.

A noter que s'il y a un retard dans la boucle, la durée de clignotement sera influencée. Essayez avec le delay (300) et comprenez.

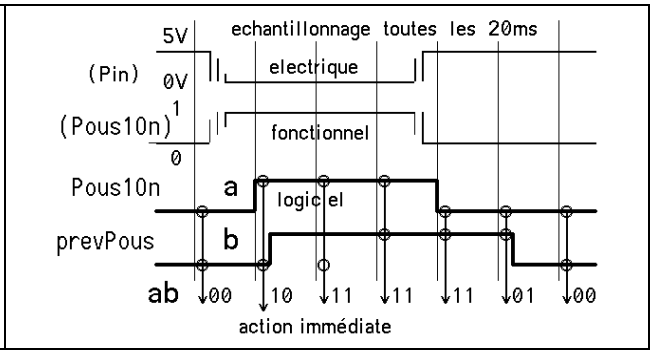
```
//A462 Clignote via millis()
#include "LcDef.h"
void setup () {
  LcSetup ();
}

void CliMilliLed1 (int dd) {  // duree en ms
  static int debCli1;
  if (millis() > debCli1+dd) {
    Led1Toggle;
    debCli1 = millis();
  }
}
void CliMilliLed2 (int dd) {   // duree en ms
  static int debCli2;
  if (millis() > debCli2+dd) {
    Led2Toggle;
    debCli2 = millis();
  }
}
```

```
void loop () {
  CliMilliLed1 (100);
  CliMilliLed2 (1000);
  //  delay (300);
}
```

On peut encore clignoter par interruption. Le programme active une variable durée. Si cette durée est nulle, cela ne clignote pas. On fera l'exercice quand on comprendra les interruptions et les timers.
La fonction Arduino `millis();` utilise un timer par interruption pour mettre à jour la valeur rendue, mais peu connaissent les détails. Il ne faudrait par ailleurs surtout pas la modifier.

Action selon des poussoirs
Pour compter les actions sur un poussoir, dans les exercices A272 et A291 on attendait sur le relâchement avec un while bloquant. L'exercice A292 demandait une modification délicate.

C'est plus propre de suivre de façon non bloquante l'évolution du signal. On l'a fait dans l'exemple A3111 pour surveiller les deux poussoirs en même temps.
Le signal `Pous1On` est échantillonné toutes les 20ms et on compare avec sa copie `prevPous1` faite 20ms avant. Si on a la combinaison 10, cela veut dire "touche nouvellement pressée". La condition ne se reproduira plus jusqu'à une nouvelle pression.



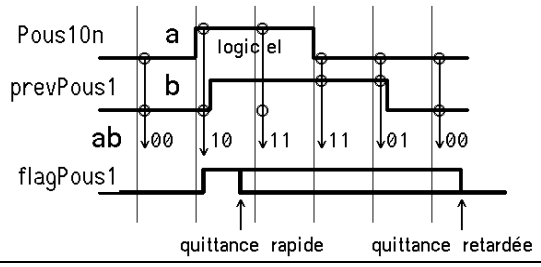
Exemples 4.63

Incrémentons un compteur à chaque action sur P1 comme on l'a déjà fait.
La variable `prevPous` est initialisée à zéro par le C. Pous1On est aussi à zéro, il n'y a pas un comptage +1 au démarrage.
La variable `prevPous` , déclarée comme byte, est vu comme une variable booléenne; le C ne distingue pas les booleens (0,1) et les bytes (0,≠0).

Une action on-off sur le poussoir est plus lente qu'un transfer Serial.print. Si c'était l'inverse, il y aurait problème. Pourquoi?

```
//A463 Compte à chaque pression
#include "LcDef.h"
void setup () {
  LcSetup ();
  Serial.begin (9600);
}
byte prevPous1; int compte;
void loop () {
  delay (20);
  if (Pous1On && (prevPous1==0)) {
      compte++;
      Serial.println (compte);
  }
  prevPous1 = Pous1On;
}
```

Dans l'exemple ci-dessus, la réaction est immédiate; cela ne peut pas toujours être le cas. Il faut alors mémoriser que la touche a été pressé dans un flag, un sémaphore qui reste actif.
Comme le train qui laisse un sémaphore activé même quand il a fini de passer sur le capteur. Ce flag est remis à zéro par un autre signal quand la voie est libre.



Exemples 4.64

Avec le programme ci-contre, P1 allume L1 et compte. P2 quittance.
On voit bien 3 tests indépendants:
- s'il y a transition on active le flag
- si le flag est actif on allume
- si on presse on éteint et clear le flag

(diviser pour régner est une méthode informatique)

Peut-on permuter les if?

```
//A464.ino  P1 allume L1, P2 quittance
#include "LcDef.h"
void setup () {
  LcSetup ();
}
byte prevPous1;
byte flagPous1;
void loop () {
  delay (20);
  if (Pous1On && (prevPous1==0)) {
     flagPous1=1;
  }
  prevPous1=Pous1On;
  if (flagPous1) {
```

```
      Led1On;
   }
   if (Pous2On & flagPous1) {
      Led1Off;
      flagPous1=0;
   }
}
```

4.7 Switch - case

La construction "switch – case" permet d'écrire des programmes faciles à écrire et à dépanner. On l'a déjà vue en section 2.8 dans sa forme la plus simple.

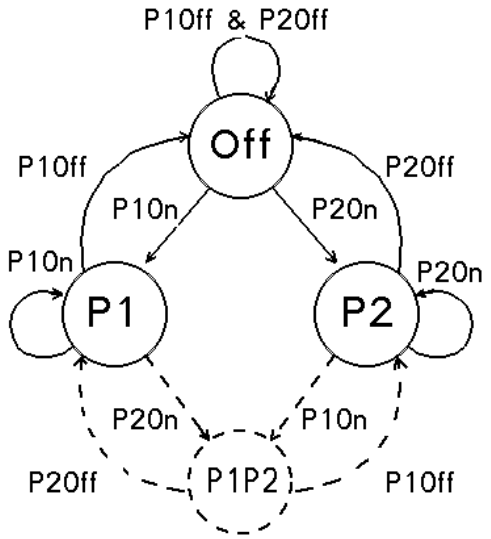
Le principe est de se débrancher selon l'état d'une variable, et cela remplace une série de if consécutifs avec l'avantage d'une mise en évidence des cas possibles.
Les cas possibles sont numérotés, le numéro du cas intéressant est stocké dans une variable.
A droite, on va commencer par le cas 2.

```
byte cas=2;
switch (cas) {
  case 5:     // la variable cas a la valeur 5
    . . . instructions à exécuter;
    break;
  case 2:
    . . .
  default:
}
```

On remarque que la syntaxe est particulière. Tous les cas sont dans une même accolade. et chaque cas est terminé par le cas suivant. L'instruction `break;` fait sortir du `switch`, donc de l'analyse des cas. Quand le processeur entre dans le `switch`, il regarde la valeur de la variable associée et saute pour exécuter ce cas. S'il n'existe pas, il passe aux instructions sous `default`.
La construction `enum` permet de donner à ces cas numérotés des noms qui clarifient la documentation.
`enum {toto,titi,tata} var=titi;` donne aux trois noms les valeurs 0,1,2 et initialise la variable var à la valeur titi (cela ne nous intéresse plus de savoir que c'est 1).
Notez que les noms sont entre accolade.
Par exemple pour un robot on va déclarer
`enum {Stop,Avance,Recule,Tourne} etatRobot=Avance;`
Le programme passe à travers le `switch` pour décider de la commande des moteurs. Un obstacle va changer la valeur `etatRobot` et le prochain passage par `switch(etatRobot)` appellera les instructions pour éviter l'obstacle.

Exemple 4.71

Reprenons l'exemple des 2 poussoirs qui commandent des clignotements.
Avec une approche top-down, on voit un système à 3 état: `Poff`, `P1` pressé, `P2` pressé; (on ignore en première étape la simultanéité).

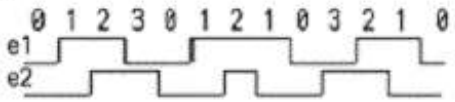


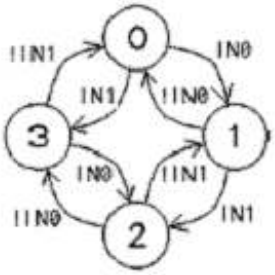
Complétez le programme selon l'état en pointillé. Si P1 et P2 sont pressé ensemble, mettre le compteur à zéro. Notez l'utilisation du mot "ensemble" et pas "simultané".

```
//A471.ino Clignotes selon P1 P2
#include "LcDef.h"
void setup () {
  LcSetup();
}
. . fonctions CliLed1 CliLed2  voir ex A461

enum {Poff,P1,P1} etat=Poff;
void loop () {
  delay (20);
  switch (etat) {
    case Poff:
      if (Pous1On) { etat = P1; }
      if (Pous2On) { etat = P2; }
      break;
    case P1:
      CliLed1(1000/25);
      if (!Pous1On) {
        Led1Off;
        etat = Poff;
      }
      break;
    case P2:
      CliLed2(200/25);
      if (!Pous2On) {
        Led2Off;
        etat = Poff;
      }
      break;
  }
}
```

Exemple 4.72

Prenons l'exemple d'un encodeur que l'on trouve dans les souris mécaniques et dans les encodeurs rotatifs qui ressemblent et remplacent de plus en plus les potentiomètres. Les deux signaux d'un encodeur sont déphasés de + ou – 90 degrés.





2 signaux, 2 bits, 4 états. Dans chaque état, on peut tourner dans un sens ou dans l'autre, l'état futur dépendra du bit qui change. Le graphe d'état ci-dessus documente tous les cas. Il n'y a plus qu'à écrire la structure switch.
Pour tester avec le LearnCbot, on utilisera P1 et P2, activés selon le diagramme des temps plus haut. Le compteur/décompteur est affiché sur le terminal.

```
//A472.ino Encodeur testé avec P1 P2

#include "LcDef.h"
void setup () {
  LcSetup ();
  Serial.begin (9600);
}

enum {e0,e1,e2,e3} next = e0;
int pos=0 ;
void loop() {
  switch(next)
  {
  case e0:
    if (P1On) {pos++; next = e1; }
    if (P2On) {pos--; next = e3; }
    break;
  case e1:
    if (P2On)  {pos++; next = e2; }
    if (!P1On) {pos--; next = e0; }
    break;
  case e2:
    if (!P1On) {pos++; next = e3; }
    if (!P2On) {pos--; next = e1; }
    break;
  case e3:
    if (!P2On) {pos++; next = e0; }
    if (P1On)  {pos--; next = e2; }
    break;
  }
  Serial.println (pos);
}
```

Que penser des rebonds de contact (faciles à simuler)? Ils font osciller entre deux états et ne perturbent pas le comptage global.

jdn 140518/151202