www.didel.com -- info@didel.com

Bureau d'ingénieur
Pierre-Yves Rochat
pyr@pyr.ch

www.didel.com/lc/LearnC01.pdf

Apprendre à programmer avec le LearnCbot

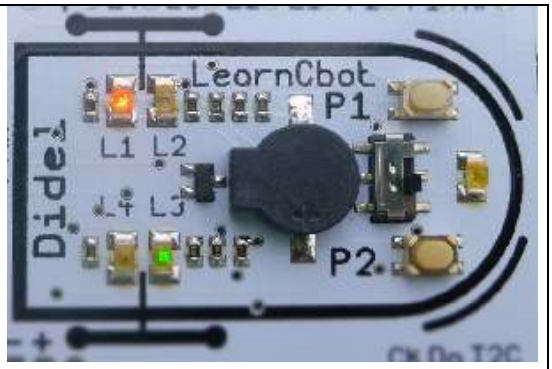
1 Notions de base

Note: Ce document est adapté et complété dans le cadre du MOOC EPFL "Microcontrôleurs". Les documents du cours se trouvent sous www.didel.com/coursera/lc1.pdf /lc2.pdf etc

Bienvenue pour découvrir comment un micro-contrôleur interagit avec son environnement.
Vous avez une carte Arduino ou Diduino, vous avez chargé l'environnement et vu comment clignoter une led et lire un poussoir.
Allons plus loin avec le LearnCbot, un shield super-efficace pour apprendre à programmer en Arduino/C, comprendre les capteurs et se préparer à des applications robotique.

1.1 Carte LearnCbot

La carte a 2 poussoirs, 4 leds et un haut-parleur, ce qui permet quantités d'exercices pour apprendre les instructions du C, comment définir, comment structurer avec des fonctions, en bref, comment programmer.
On peut câbler les composants sur un breadboard, voir www.didel.com/coursera/MatMOOC.pdf
Ignorons les connecteurs Grove pour le moment.
Presque tous les systèmes pilotés par microcontrô-leur ont des poussoirs et des leds. Le LCbot a 2 poussoirs , 4 leds et un petit haut-parleur; cela va permettre des exercices de toute complexité.



1.2 Le processeur Arduino

Arduino simplifie la vision du microcontrôleur: On a des pins numérotées que l'on définit "entrées" ou "sorties", sur lesquelles on peut lire et écrire. On a 5 fonctions Arduino pour interagir :

`pinMode (pin, 1);` // ou (pin, OUTPUT)	la pin est définie en sortie
`pinMode (pin, 0);` // ou (pin, INPUT)	la pin est définie en entrée
`digitalWrite (pin,1);` // ou (pin,HIGH)	la sortie est à l'état 1 (~5V)
`digitalWrite (pin,0);` // ou (pin,LOW)	la sortie est à l'état 0 (~0V)
`digitalRead (pin)` // valeur 0 ou 1	on lit la pin

A noter que `digitalRead (pin)` informe sur un état, et n'est pas une action comme un digitalWrite. On comprendra mieux plus loin; jouons d'abord avec les leds en sortie.

1.3 Fonctions d'attente sur Arduino

Le processeur est trop rapide pour nous quand on veut observer ce qu'il fait. Il faut le ralentir et Arduino met à disposition deux fonctions

`delay (DureeEnMilliSec);`	Attente en ms
`delayMicroseconds (Duree);`	Attente en us

Il y a d'autres fonctions (groupe d'instruction pour exécuter une tâche précise) qui l'on verra selon les besoins. Et on apprendra à faire nos propres fonctions.

1.4 Clignoter

On a tout ce qu'il faut maintenant pour le fameux "Blink": Allumer, attendre, éteindre, attendre, recommencer. Mais tout d'abord il faut configurer les entrées-sorties.
La led L1 que nous voulons utiliser est sur la pin 4.

```
//LcCligno1.ino  clignoter la pin 4
void setup() {
  pinMode (4,1);
}
void loop () {
  digitalWrite (4,1); delay (500);
  digitalWrite (4,0); delay (500);
}
```

Il faut respecter strictement les contraintes du C avec les parenthèses, accolades, points-virgules. Toute erreur de syntaxe génère des messages d'erreur le plus souvent incompréhensibles.

Le compilateur C/Arduino fait partie d'une chaîne qui crée du binaire que seul le processeur comprend, via l'assembleur qui code les instructions dans un premier langage compréhensible par les programmeurs expérimentés, puis par les langages évolués plus proche de nos habitudes logiques et arithmétiques. Mais quand on voit dans le programme précédent `pinMode (4,1);` est-ce mieux que du langage machine? Le C/Arduino donne des noms clairs aux fonctions de base, il nous faut aussi donner des noms clairs aux objets manipulés par ces fonctions.

1.5 #define

La pin4 commande la led L1, qui a son nom écrit sur le LCbot, on va garder ce nom et dire au compilateur que L1 et 4 sont équivalents.

```
#define L1  4    // ! sans signe =, sans ; final
```

Les #define construisent un dictionnaire que le compilateur lit pour voir que si on a écrit L4, il faut remplacer par un 4, qu'une partie Arduino du compilateur va assigner à un bit dans un port du microcontrôleur, que des aiguillages mettront en contact avec la mémoire ou l'unité arithmétique selon l'instruction.
La doc Arduino remplace les #define par une définitions de variable qui évite d'expliquer le #define Ecrire int L1=4; bloque une variable 16 bits pour une définition. Ce n'est pas élégant.
Par ailleurs, le #define permet de définir des actions qui s'expriment par une instruction, éventuellement 2 (on parle alors de macros), qui augmentent la lisibilité des programmes. Les fonctions seront l'étape suivante vers la complexité.

Allumer la led L1 peut s'écrire maintenant `digitalWrite (L1,1);`, mais on est encore au niveau du langage de notre machine: il faut un 1 (5V) pour allumer, dans d'autres montages il faut un 0.
Définissons ce câblage et utilisons des instructions fonctionnelles avec un nom clair: `L1On; L1Off;`

```
#define  Led1On digitalWrite (L1,1)     // Allume
#define  Led1Off digitalWrite (L1,0)     // Eteint
```

Attention, l'instruction dans une macro ne se termine pas par un `;`

Le programme qui clignote va avoir maintenant une partie principale compréhensible par tous, et une partie définitions et mise en route qui nécessite d'avoir compris le câblage. Mais une fois compris et codé, plus besoin d'y réfléchir.

```
//LcCligno2.ino  clignoter la led L1 sur la pin 4
#define L1  4
#define  Led1On digitalWrite (L1,1);        //  Allume
#define  Led1Off digitalWrite (L1,0);       //  Eteint
void setup() {
  pinMode (L1,1);
}
void loop () {
  Led1On;  delay (500);
  Led1Off; delay (500);
}
```

1.6 Constantes

Dans le programmes précédent, si on veut un clignotement plus rapide, il faut changer deux instruction pour modifier la constante 500. On pourrait définir cette constante et changer à un seul endroit:

```
//LcCligno3.ino    dans www.didel.com/coursera/lc1.zip
. . . définitions et set-up comme avant
#define DemiPer 500
void loop () {
  Led1On;  delay (DemiPer);
  Led1Off; delay (DemiPer);
}
```

Il faut plus écrire, mais quand on lit le programme, on voit ce que signifie le contenu de ( ) .

Dans les programmes Arduino, on définit souvent les constantes en écrivant

```
int demiPer=500;   // variable 16 bits qui vaut 500
```

Ceci utilise une variable; pour éviter ce gaspillage il faut écrire

```
const int DemiPer=500;   // constante 16 bits qui vaut 500
```

Ce qui est strictement équivalent au `#define DemiPer 500`
On pourrait encore définir une macro qui fait tout:

```
  #define AttenteDemiPer delay (500)
ce qui permet d'écrire dans le programme principal L1On;  AttenteDemiPer;
```

1.7 Variables

Une variable est une case mémoire 8bit (type byte ou char) 16 bits (type int) et il y a d'autres types. On expliquera chaque type quand on en aura besoin.
La valeur du délai peut être une variable, et dans ce cas on ne peut pas utiliser un #define. On déclare

```
int demiPer;     // la minuscule au début montre que c'est une variable et pas une constante
```

ce qui réserve en mémoire une case de 16 bits "vide" (le compilateur mets en général la valeur zéro). On peut donner une valeur initiale en écrivant

```
int demiPer = 100;
```

C'est important de bien distinguer les variables et les constantes. Une bonne habitude est d'avoir la première lettre d'une variable en minuscule, et les constantes toute en majuscule. Pour une raison de lisibilité nous n'avons pas écrit, quand DemiPer était une constante, DEMIPER ou DEMI_PER. Un puriste C l'aurait fait!

Revenons aux variables. On peut varier la vitesse de clignotement si on a défini la demi période comme variable. Le programme suivant diminue cette période. Le double signe. -- est une notation C pour soustraire 1 (++ ajoute 1). On aurait aussi pu écrire

```
demiPerCligno = demiPerCligno-1;   ou   demiPerCligno -=1;
```

```
//LcCligno4.ino  dans www.didel.com/coursera/lc1.zip
. . . définitions et set-up comme avant
  int demiPer = 100; // on commence avec 0.1 seconde
void loop () {
  Led1On;  delay (demiPer);
  Led1Off; delay (demiPer);
  demiPer--;
}
```

Donc la période va diminuer 99, 98, .. A partir de 20ms (50Hz) on ne voit plus le clignotement et à zéro, cela s'arrête! Non cela ne s'arrête pas s'il n'y a pas un test spécial. La variable 16 bits a passé à sa valeur maximum, comme le compteur d'une voiture neuve qui ferait un km en arrière et afficherait 99'999 km.
Le délai a passé à 65335, la led se ralumera dans 65 secondes et cela mettra des jours avant de repasser rapidement par le clignotement rapide.

1.8 Croquis et dossier

On sera amené à créer beaucoup de programmes. Les .ino sont encapsulés dans des croquis. Ce n'est pas du C et quelques explications et conseils sont donnés sous www.didel.com/lc/GestionCroquis.pdf

1.9 Résumé

On vient de voir beaucoup de choses importantes :
- il faut respecter les règles de syntaxe du C et les règles de bonne écriture avec des noms clairs
- il faut bien distinguer les variables et les constantes
- il faut mettre dans un bloc de définition initial tout ce qui concerne le câblage
- les variables sont des nombres binaires et ne réagissent pas toujours comme les nombres auxquels nous sommes habitués.
- pour bien comprendre un programme et avoir la syntaxe "dans le sang", il faut jouer à faire beaucoup de variantes de programmes.

1.10Tester une variable – if (condition vraie) { faire; }

Reprenons le programme 4. On voudrait que le clignotement pulse avec une demi-période qui reste entre deux valeurs Min et Max. Il faut pour cela comparer et exécuter une ou plusieurs instruction si la condition est vraie: On écrit dans la boucle du programme LcCligno4.

```
if (demiPer < Min) { demiPer=Max; }
```

```
//LcCligno5.ino
. . . définitions et set-up comme avant
  int demiPer = 100; // on commence avec 0.1 seconde
#define Min 50;
void loop () {
  Led1On;  delay (demiPer);
  Led1Off; delay (demiPer);
  demiPer--;
  if (demiPer < Min) { demiPer=Max; }
}
```

On peut comparer avec les signes > < >= <= != (différent) == (identique)

Attention, on prononce "égal" quand on compare, mais en C = est l'assignation arithmétique "je remplace", différente de la comparaison "est-ce la même chose", signe ==.
Dans le programme précédent remplacez < par ==; il y a une boucle de plus avant réinitialisation. Enlevez un = Le programme ne fonctionne plus!

1.11 Le haut-parleur

Le HP est sur la pin1, qui est utilisée pour les transferts série, mais cela ne gêne pas.
On a vu comment clignoter une led, utilisez des microsecondes avec le haut-parleur.

1.12 Les poussoirs

Les poussoirs P1 P2 sur les pins 2 et 3 doivent être initialisés en entrée dans le setup

```
#define P1  2
#define P2  3
pinMode (P1,INPUT);
pinMode (P2,INPUT);
```

Quand on lit une pin, on se pose la question "quel est son état?"
`digitalRead (P1)` est un état binaire, 0 ou 1, vrai ou faux.
Il faut maintenant tenir compte du câblage. Le poussoir, comme c'est le plus souvent le cas, est câblé pour court-circuiter une résistance "pull-up" et imposer 0V sur l'entrée du processeur; si on ne pèse pas, la pull-up impose du 5V.
Donc pour allumer la L1 quand on presse P1, il faut écrire

```
if (digitalRead (P1)==0) {
  digitalWrite (L1,1)
}
```

Attention aux parenthèses: Le compilateur repère un `if (condition)` et la condition est `digitalRead (P1)==0` avec un point d'interrogation sous-entendu. Est-ce vrai ou faux?
Si on presse, `digitalRead (P1)==0` est vrai (valeur 1). Si on ne presse pas la condition est fausse (valeur 0). On voit le risque de confusion entre états électriques et états logiques.
Quand on écrit le programme on doit penser l'état du poussoir, pressé ou relâché, 1 ou 0, vrai ou faux, On ou Off. On a choisi le nom PousOn pour cet état, mais il faut bien remarquer que l'état `PousOn` n'est pas de même nature que l'instruction `LedOn;`

```
#define Pous1On  (digitalRead (P1)==0) // électriquement actif à 0V
#define Pous1Off  (digitalRead (P1)==1) // électriquement actif à 5V
```

Le programme devient très lisible, et si vous testez une autre carte ou le poussoir est actif à 1 ou les leds actives à zéro, il suffit de changer les définitions, et rien dans le programme. De plus, le programme est maintenant écrit dans un C et en adaptant les définitions, il tournera sur une carte qui a un compilateur C sans facilités Arduino ou Energia.

```
//LcCopy.ino  Copie P1 sur L1
. . . définitions et set-up
void loop ()  {
    if (Pous1On) {
      Led1On;
    }
    if (Pous1Off) {
      Led1Off;
    }
}
```

1.13 Résumé

Une condition est une variable booléenne avec deux valeurs 0 (faux) et 1 (vrai).
Une variable 8 ou 16 bits peut aussi avoir deux états faux si égal à 0, vrai si différent de zéro.
Dans une instruction conditionnelle, si la condition est vraie, une ou plusieurs instuctions sont exécutées. Elles doivent être mises entre accolade; pour la clarté des programmes et réduire le risque d'erreur, l'accolade fermée doit être alignées avec l'instruction conditionnelle.
"Ctlr-T" sous Arduino formate avec l'accolade sous l'instruction.
L'usage d'espace et de retours à la ligne est assez libre avec le C.

Remarque: Les tutoriaux Arduino raisonnent avec les états électriques, et dans le programme, il faut sans cesse se souvenir "le bouton est pressé, je vois un zéro". Editez les programmes que vous trouvez pour qu'il soient plus clairs.
Un programme bien documenté n'est pas un programme qui répète sur chaque ligne ce que fait l'instruction! Il définit des noms clairs, et fait apparaître des blocs d'instruction dont la fonctionnalité est commentée avant le bloc, si elle n'est pas évidente.