Fonctions en C

Le document plus récent www.didel.com/coursera/LC4.pdf est beaucoup plus clair et complet

Les fonctions en C, aussi appelées procédures, correspondent aux routines de l'assembleur et elle sont tout aussi indispensables pour écrire un code lisible.
Comme dans tous les langages, les noms doivent être bien choisis, et rappelons l'usage :
- constantes en majuscule (mots courts) **CR PORTA DODIEZE**
  (ce document suit rarement cette règle pas préférence esthétique, DeDieze est plus lisible)
- variables en minuscules, majuscule pour séparer (noms explicites) **distanceGauche**
- variables locales temporaires, mots courts **i j temp**
- fonctions avec majuscule initiale dans un verbe **FaireCeci LireClavier**

Chercher sous Google ''règles d'écriture en C''

#define
Une programmation claire utilise des noms clairs, qui évoquent leur fonctionnalité. Les caractéristiques technologiques doivent être bien séparées.
Le #define donne à un objet, à une ou plusieurs instructions un nom choisi avec soin pour être explicite et non ambigü.
Par exemple,

```
#define Led 13 // on a branché une Led sur la pin 13
                  cette led est allumée s'il y a du 0V sur cette pin
```

permet ensuite d'écrire `digitalWrite (Led,LOW) ;` pour allumer la led
Les #define doivent être utilisés pour définir le matériel et faire en sorte que le programme principal ne dépendent ni du câblage ni du compilateur .
Par exemple `#define LedOn digitalWrite (Led,LOW)`
permet d'écrire dans le programme `LedOn ;`
Si on utilise un compilateur C qui n'utilise pas les digitalWrite, il suffut de changer les définitions initiales, pas le programme.
Abuser des #define pour remplacer des fonctions n'est pas recommandé.
Pour les définitions simples, on peut écrire

```
const int Led13 = 13 ;
```

ou, ce qui est fréquemment vu, mais utilise inutilement une position mémoire de 16 bits

```
int Led13 = 13 ;
```

Une fonction simple
La fonction Clignote s'écrit

```
void Clignote ()  {
  LedOn;
  delay(1000);
  LedOff;
  delay(1000);
}
```

Dans le programme, on appelle cette fonction en écrivant

```
Clignote ();
```

La parenthèse est vide car il n'y a pas de paramètre.qui influencent cette action d'avancer.
Certains écrivent `void Clignote (void)` par cohérence avec le void expliqué plus loin.

Fonction avec un paramètre
L'instruction `delay (val);` est une fonction prédéfinie avec un paramètre et il faut savoir que ce paramètre est une variable ou une constante 32 bits de type `long`. .Si c'est une variable, il faut avoir déclaré avant avec la ligne `long val;` (`int val;` est accepté).
Sauf erreur, si on déclare la variable comme non signé et que l'on décompte, le délai n'est plus nul à la valeurs 0, mais maximum, et on se trouve dans le cas du compteur d'une voiture, qui indique 0 pour 100000 km, et si on fait un km en arrière, 99999 km

Exemple 1 **Un flash**

On veut faire des flashs de différentes longueur, avec une durée "logique" de 1,2,3. Donc 1 pour court, 2 pour plus long, etc. La durée "physique" est une constante en millisecondes. Si on veut changer les durées en gardant leur proportion, le changement se fait à un seul endroit.
`MinDur = 100 milliseconds.`
Si on écrit `Fash(2);` on a un flash de 200 ms

```
// Flash (duration in multiple of MinDur)
// test with Flash.ino (complete program)
void Flash (byte dd)  {
  MinDur = 100 ; // 100 millisec
  digitalWrite (Led13,HIGH);
  delay (dd*MinDur);
  digitalWrite (Led13,LOW);
  delay (200);
}
```

La durée du flash peut être une variable et elle peut être nulle suite à une erreur
`flash(0);` est un flash of 65536*100 ms !
Modifions la procédure pour éviter ce cas.
On pourrait aussi dire que plus de 10 flashs n'a pas de sens. et ajouter une instruction:
`if (dd>10) dd=10;`

Dans le programme, on définit une variable `byte durFlash;` et l'instruction d'appel est `FlashV (durFlash) ;`
La variable dd est locale à la procédure.

```
// FlashV (duration in multiple of 100ms, no flash for 0)
// test with Flash0.ino
void FlashV (byte dd)  {
  if (dd != 0) // different of 0  {
    MinDur = 100 ; // millisec
    LedOn;
    delay (dd*MinDur);
    LedOff;
    delay (200);
  }
}
```

Pourquoi void ? Il n'y a pas de paramètre en sortie, on verra plus loin un exemple avec paramètre.
Pourquoi (byte dd) avec le nom de la fonction ? Il faut bien comprendre que dd est un paramètre local, un tiroir vide, rempli à chaque appel par la valeur à traiter.

Exemples avec deux paramètres en entrée
Une note simple est une répétition d'impulsions sur le haut-parleur. Le nombre d'impulsions sera le 2e paramètre de la procédure `note (periode, nbDeRepet) ;`
Le produit des deux paramètres donne la durée, mais notre propos ici n'est pas de faire de la musique, mais expliquer les fonctions. La fonction `note` s'écrit

```
//  note (période, nbDeRépetitions) ;
void note (byte pp, byte nr)  {
      for (int i=0; i<n; i++) {
            HpOn ;
            delayMicroseconds (pp/2) ; // demi-période
            HpOff ;
            delayMicroseconds (pp/2) ;
      }
}
```

Pourquoi `pp/2` ? Le paramètre dans note est tout naturellement la période, que l'on obtient par deux retards d'une demi-période. Donc il faut diviser le paramètre par 2. Heureusement, le compilateur remplace cette instruction par pp>>2 , qui décale le mot binaire à droite, ce qui divise par 2 en une seule instruction.
A noter que si on veut la fréquence en paramètre, il faut une division, et là le compilateur va se faire vraiment plaisir (en pratique on passe par des tables de conversion plus rapides).

Fonction avec plusieurs paramètres en entrée
Il n'y a pas de limitation dans le nombre de paramètres en entrées, qui sont déclaré comme des variables locales. Si on veut que ces variables locales soient conservées d'un appel à l'autre, il faut ajouter le mot static. Une fonction peut aussi agir sur des variables globales qui ne sont pas mentionnées dans l'appel, mais doivent être bien documentées.

En résumé

```
void AgirSur (int var1, int var2) {
   byte i ;   //variable temporaire
  // les instructions de la procédure qui
```

On donne la liste des variables utilisées (paramètres formels), qui n'ont de signification qu'à l'intérieur de la fonction. La fonction peut aussi agir

// utilisent var1 var2 i ;
}
Appel: `AgirSur (3,toto) ;`
	sur des variables globales, déclarées au début du programme.
	.

Fonction avec un paramètre en sortie

`byte LirePot ()`
`{`
`byte pot ;`
`pot = analogRead (A0) / 4 ;`
`return pot ;`
`}`
	byte dit qu'il que le paramètre en sortie sera de type byte, mais il faut re-déclarer ce paramètre au début en le nommant et à la fin, après l'avoir calculé, pour dire que c'est la valeur demandée. Ici, la parenthèse vide dit qu'il n'y a pas de paramètres en entrée ou qu'il est global.

La ligne return peut donner directement la valeur :

`byte LirePortAlow ()`
`{`
`return (PORTC& 0x0F) ;`
`}`
Utilisation : `EtatPous = LirePortAlow () ;`
	Un define serait possible dans ce cas
	`#define PortALow (PORTC & 0x0F)`
	
	Utilisation : `EtatPous = PortAlow ;`
	! il n'y a pas de `()`

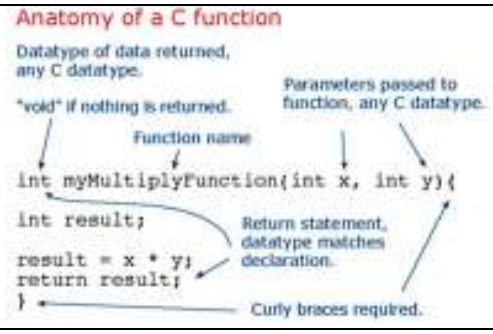
Fonction avec des paramètres en entrée et un seul paramètre en sortie

Les fonctions mathématiques sont les seules expliquées dans les cours sur le C.
On ne peut avoir qu'un seul paramètre en sortie, mais ce paramètre peut être un pointeur qui donne accès à une structure.

`unsigned int Mult (unsigned int nb1, nb2) {`
`unsigned int result ;`
`result = nb1 * nb2 ;`
`return result ;`
`}`
`Ou a la place de ces 3 lignes`
`return nb1 * nb2 ;`
	Exemple :
	
	`calcul = Mult (2,3) ;`
	// donc calcul prend la valeur 12
	Attention aux types de données et débordements

`byte CalculePfm (int position, byte consigneVitesse) {`
`byte pfm ;`
`int i ;` //variable temporaire supplémentaire
`. . .` // les instructions de la procédure ;
`return pfm ;`
}
	`pfmMot1 = CalculePfm (dist,VitNom);`

Pour toute fonction, it faut bien réfléchir aux types de variable, et passer du temps pour choisir les noms, et ne pas hésiter à changer les noms qui deviennent ambigus.
Un nom bien choisi évite une tartine d'explications.

Résumé Ci-contre, ce qu'il suffit de dire pour tout savoir (quand on sait tout) !	

Une fonction est un paquet d'instructions exécutables. Ce paquet porte un nom, ce qui permet de l'appeler, c'est à dire provoquer l'exécution des instructions qu'il contient.
On utilise dans les fonctions des variables locales, qui renaissent chaque fois que le bloc est appelé, et meurent à la fin de l'exécution. Pour conserver ces variables, il faut précéder leur type du mot statique, ou les déclarer comme variable globale.