C sur dsPIC™

Digital Signal Controller

MICROCHIP

CREMMEL Marcel

Lycée Louis Couffignal

STRASBOURG

Adaptée de la
brochure C-booklet
d'Elektor

Initiation au C sur dsPIC

Historique

Aucun langage de programmation n'a pu se vanter d'une croissance en popularité comparable à celle de C.
Le langage C n'est pas un nouveau né dans le monde informatique :

- 1972 : développement par ATT Bell Laboratories.

Le langage C par rapport à d'autres langages pour comparaison :

- C++ : 1988, 1990
- Fortran : 1954, 1978 (Fortran 77), 1990, 1995 (Fortran 1995)
- Cobol : 1964, 1970
- Pascal : 1970

C ANSI

Nécessité la définition d'un standard actualisé et plus précis.
1983 : l'*American National Standards Institute* (ANSI) charge une commission de mettre au point une définition explicite et indépendante machine pour le langage C : naissance du standard C-ANSI
1988 : seconde édition du livre *The C Programming Language* respectant le standard - C-ANSI. C'est la bible du programmeur en C (Le langage C. K&R. Editions Masson).

C++

1983 : création du langage C++ par un groupe de développeurs de AT&T
But : développer un langage qui garde les avantages de ANSI-C mais orienté objet.
Depuis 1990, il existe une ébauche pour un standard ANSI-C++.
Remarque : C++ est un sur-ensemble du C si bien que 80 % des gens qui déclarent développer en C++ développent en fait en C !

Avantages du langage C

C est un langage :
(1) universel : C n'est pas orienté vers un domaine d'applications spéciales
(2) compact : C est basé sur un noyau de fonctions et d'opérateurs limités qui permet la formulation d'expressions simples mais efficaces.
(3) moderne : C est un langage structuré, déclaratif et récursif. Il offre des structures de contrôle et de déclaration comme le langage Pascal.
(4) près de la machine : C offre des opérateurs qui sont très proches de ceux du langage machine (manipulations de bits, pointeurs…). C'est un atout essentiel pour la programmation des systèmes embarqués.
(5) rapide : C permet de développer des programmes concis et rapides.
(6) indépendant de la machine : C est un langage près de la machine (microprocesseur) mais il peut être utilisé sur n'importe quel système ayant un compilateur C.
(7) portable : En respectant le standard ANSI-C, il est possible d'utiliser (théoriquement ☺) le même programme sur tout autre système (autre microprocesseur), simplement en le recompilant. Il convient néanmoins de tenir compte des spécificités des périphériques intégrés.
(8) extensible : C peut être étendu et enrichi par l'utilisation de bibliothèque de fonctions achetées ou récupérées (logiciels libres…).

Inconvénients du langage C

(1) efficacité et compréhensibilité : C autorise d'utiliser des expressions compactes et efficaces. Les programmes sources doivent rester compréhensibles pour nous-mêmes et pour les autres.
Sans commentaires ou explications, les fichiers sources C peuvent devenir incompréhensibles et donc inutilisables (maintenance ?).
(2) portabilité et bibliothèques de fonctions : Le nombre de fonctions standards ANSI-C (d'E/S) est limité. La portabilité est perdue si l'on utilise une fonction spécifique à la machine de développement.
(3) discipline de programmation : C est un langage près de la machine donc dangereux bien que C soit un langage de programmation structuré.

Table des matières

Ce document se limite aux éléments fondamentaux du langage C. Nous avons délibérément fait l'impasse sur les structures complexes du C telles que *pointer*, *structure*, *union* etc. Son but est de servir de guide au débutant. Il n'a pas l'ambition de remplacer un cours approfondi.

1. Les bases du C

1.1 La structure des programmes en C

Tout programme en C se compose de différentes parties comme les commentaires, les instructions de pré-traitement, les déclarations, définitions, expressions, assignations et les fonctions. Voici un exemple simple :

```
/*****************************************************************
 ***                     Nom de l'application                  ***
 ***                         dsPIC30F2010                      ***
 ***                                                           ***
 ***  Auteur : xxxxxx                                          ***
 ***  Version V2 à incrément de phase                          ***
 ***  Date de création : 17/08/2006                            ***
 ***  Dernière mise à jour : 15/09/2006                        ***
 *****************************************************************/

#include <p30f2010.h>

/*-------------------------------------------------------
                    Câblage du LCD
-------------------------------------------------------*/
// Connexions LCD Nokia
#define Lcd_DC_Pin   PORTBbits.RB0  //  RB0 = D/C
#define Lcd_CE_Pin   PORTBbits.RB1  //  RB1 = SCE
#define Lcd_DIN_Pin  PORTBbits.RB2  //  RB2 = SDIN
#define Lcd_CLK_Pin  PORTBbits.RB4  //  RB4 = SCLK
#define LED1         LATCbits.LATC13
#define LED2         LATCbits.LATC14
#define BP           PORTDbits.RD1  // Bouton poussoir

/*-------------------------------------------------------
                  Définitions de type
-------------------------------------------------------*/
typedef enum
{
    Lcd_Cmd  = 0,
    Lcd_Data = 1
} LcdCmdData; // Type de transfert : commande ou données

/*-------------------------------------------------------
             Déclarations de variables
-------------------------------------------------------*/
/*-----------------
        LCD
-------------------*/
char Adr_X ; //Copie logicielle de l'adresse X du PCD8544 (0 à 83)
char Adr_Y ; //Copie logicielle de l'adresse Y du PCD8544 (0 à 5)

/*-------------------------------------------------------
             Déclarations des fonctions
-------------------------------------------------------*/
/*-------------------------------------
 void LcdSend(char Data, LcdCmdData CD)
-------------------------------------
 Nom         :  LcdSend
 Description  :  Envoi d'un octet vers le controleur du LCD
 Arguments    :  data -> Donnée ou commande à transmettre
                 cd   -> Type de transfert :
                           - Lcd_Cmd  -> commande
                           - Lcd_Data -> donnée
 Valeur renvoyée : aucune. */
```

Commentaires informant sur l'application, l'auteur et la version

Instruction au compilateur

Déclarations d'équivalences liées au câblage du dsPIC

Déclaration de "types" nouveaux (non-standards comme "char" et "int")

Déclaration des variables globales (accessibles dans tout le programme)

Déclaration d'une fonction

Commentaires obligatoires de description de la fonction.

```
void LcdSend(char Data, LcdCmdData CD)
{
  int i;

  // Sélection du controleur (actif à "0").
  Lcd_CE_Pin=0; // CE = "0"
  if (CD == Lcd_Data) Lcd_DC_Pin=1;
                 else Lcd_DC_Pin=0;
  // Transmission série de l'octet
  for (i=0; i<8; i++)
  {
   if ((Data & 0x80)==0) Lcd_DIN_Pin=0;
                    else Lcd_DIN_Pin=1;
   Data <<= 1;
   Lcd_CLK_Pin=1;
   Lcd_CLK_Pin=0;
  }
  // Dé-sélection du controleur
  Lcd_CE_Pin=1;
}
/*-------------------------------------------------------
        Déclarations des fonctions d'interruption
-------------------------------------------------------*/
/*-----------------------------
 void _ISR _T1Interrupt (void)
-----------------------------
 Nom         : _T1Interrupt
 Description : Activé toutes les 200mS par le "Timer 1"
                Traitement :
                    - Inversion de LED1 si RD1=1
                    - Inversion de LED2 si RD1=0
void _ISR _T1Interrupt (void)
{
 IFS0bits.T1IF=0; //Acquittement interruption
 if (BP) // Test du BP sur RD1
  {
   LED1 ^= 1; LED2  = 0; // Inverser LED1 et éteindre LED2
  }
 else
  {
   LED1  = 0; LED2 ^= 1; // Éteindre LED1 et inverser LED2
  }
 }
/*-------------------------------------------------------
                  Fonction principale
-------------------------------------------------------*/
int main(void)
{
 // Initialisations des périphériques du dsPIC
 ADPCFG=0xFFFF;// Pas d'entrée analogique
 TRISB=0xFFE0; //RB0 à RB4 en sortie
 Lcd_CE_Pin=1; //Dé-sélection du controleur

 // Initialisation des variables
 Adr_X=Adr_Y=0;

 // Boucle sans fin du programme principal
 while (1)
 {
  LcdSend(0x55,Lcd_Cmd);
  LcdSend(0xAA,Lcd_Data);

 }
}
```

Déclaration d'une variable locale (accessible uniquement dans la fonction)

Corps de la fonction

Veiller à l'indentation des lignes pour une bonne lisibilité du programme

Déclaration d'une fonction d'interruption

Acquitement obligatoire de l'interruption

Fonction principale exécutée immédiatement après un "reset"

Appels de la fonction "LcdSend" avec ses 2 paramètres

Les fonctions d'interruption sont appelées par des événements matériels (chgt d'états sur les entrées, fin de délai, réception d'un caractère série, etc.)

1.2 La fonction principale

Tout programme en C doit comporter au moins une fonction, la principale, dite ***main function*.** Elle s'exécute toujours en premier et est appelée après un reset du µC.

Dans un programme bien **structuré**, la fonction principale ne réalise pas directement les tâches de l'application. Celles-ci sont confiées à autant de fonctions réalisées par des *functions*. Ce travail de division en tâches élémentaires est souvent la phase essentielle dans l'élaboration d'un bon programme. Il permet aussi de répartir le développement en équipes et de faciliter son évolution.

La fonction principale comporte toujours 4 phases exécutées dans l'ordre :

- Initialisations des périphériques du dsPIC
- Initialisations des variables
- Validations des demandes d'interruption
- Boucle sans fin :
  - Appels successifs des fonctions non activées par une interruption (peu nombreuses en général)
  - Les fonctions activées par une interruption (les plus nombreuses en général) sont exécutées uniquement en cas de besoin : quand le périphérique associé le demande par une interruption. La boucle sans fin est alors interrompue le temps de traiter la fonction.

On déclare une *main function* comme n'importe quelle autre fonction, mais l'identificateur *main* est obligatoire. Elle n'a pas de paramètre (*void* = vide) et renvoie un entier (*int*) même si la fonction "*main*" ne se termine jamais ! Dans le "source", elle se situe généralement à la fin, car le compilateur exige que les déclarations des fonctions précèdent leurs appels.

1.3 Les commentaires en C

Les *comments* (commentaires) sont des textes ou des paragraphes qui n'entrent pas dans la programmation proprement dite, que le compilateur ne traduira pas et qui n'occuperont donc pas d'espace en mémoire du µC. Ils revêtent pourtant une grande importance parce qu'ils expliquent aux autres personnes le sens du programme ou de ses parties essentielles. Ils sont aussi d'un grand secours au rédacteur, parce qu'après quelques jours, on ne se souvient pas forcément pourquoi on a formulé le code de la sorte plutôt qu'autrement. Le temps gagné à les négliger risque de se perdre après en recherches fastidieuses.

```
/*
Les commentaires sont délimités par
barres obliques et astérisques...
*/
```

```
// Sauf pour une fin de ligne de commentaires
```

Les commentaires sur une seule ligne débutent par deux barres obliques et s'arrêtent automatiquement en fin de ligne.

Attention : en assembleur, on utilise le point-virgule (**;**) comme indicatif de texte explicatif. Mais en langage C, le point-virgule termine une instruction !

1.4 L'instruction #include

La norme ANSI du langage C ne retient pas de nombreuses déclarations et fonctions qui, sans être indispensables, sont cependant très utiles. On les regroupe dans ce que l'on appelle des bibliothèques (ou librairies). Dès lors, pour que le compilateur les trouve, au cours de la traduction des codes sources C, il faut les déclarer comme fichiers d'en-tête, appelés "header files" à inclure (*include* …). On reconnaît ces fichiers à leur suffixe «.h ».

Exemples : `#include <stdio.h>` : fonctions d'E/S standard (peu utilisées dans un µC)
`#include <p30f2010.h>` : obligatoire dans tous les programmes car le fichier définit les noms et adresses de tous les registres du dsPIC

1.5 Les mots-clés en C

Dans la norme ANSI du langage C, il existe 32 mots-clés définis, appelés *keywords*. Ces mots sont réservés au compilateur. Tous les *keywords* doivent être écrits en minuscules et on ne peut pas les utiliser à d'autres fins, comme nom de variable, par exemple.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Plusieurs compilateurs C ajoutent aux définitions ANSI d'autres mots-clés pour mieux mettre à profit les possibilités du processeur.

2. Les constantes et variables

2.1 Les systèmes de numération

Le langage C connaît différents systèmes de numération (number base , base de numération) : le nombre décimal, binaire, octal ou hexadécimal.
Les données chiffrées sans spécification de la notation sont interprétées par défaut comme des nombres décimaux. Toutes les autres bases doivent être identifiées. Les nombres en octal sont précédés de **0** (zéro), ceux en hexadécimal commencent par **0x** et les binaires par **0b**.

Base	**Formulation**	**Chiffres permis**	**Exemples**
Décimal (10)		`0123456789`	`5`
Octal (8)	`0`…	`01234567`	`05`
Hexadécimal (16)	`0x`…	`0123456789ABCDEF`	`0x5A`
Binaire (2)	`0b`…	`01`	`0b10101010`

2.2 Les types de données

Avant d'utiliser des données en C, il faut en déclarer le type. Sans quoi, le compilateur ne peut pas savoir combien de mémoire il doit leur réserver. En principe, on choisit toujours le type qui suffit pour un usage donné, sans occuper inutilement de place en mémoire. Voici les principaux types de données :

Nombres entiers (les nombres signés sont codés C2) :

Type	**Taille (bits)**	**Min**	**Max**
`char, signed char`	8	-128	+127
`unsigned char`	8	0	255
`short, signed short`	16	-32768	+32767
`unsigned short`	16	0	65535
`int, signed int`	16	-32768	+32767
`unsigned int`	16	0	65535
`long, signed long`	32	-2^{31}	$+2^{31}-1$
`unsigned long`	32	0	$2^{32}-1$
`long long, signed long long`	64	-2^{63}	$+2^{63}-1$
`unsigned long long`	64	0	$2^{64}-1$

Nombres réels ou "flottants" :

Type	**Taille (bits)**	**Emin**	**Emax**	**Nmin**	**Nmax**
`float`	32	-128	+127	2^{-126}	2^{128}
`double`	32	-128	+127	2^{-126}	2^{128}
`long double`	64	-1022	+1023	2^{-1022}	2^{1024}

En C, il faut faire attention à la notation américaine des nombres. Alors qu'en français nous utilisons systématiquement la virgule pour séparer la partie entière de la partie décimale, les nombres en virgule flottante (float) s'écrivent avec un point décimal. La virgule (*comma* en C, sert de séparateur dans les listes entre nombres ou entre variables.

Exemples de déclaration de variables :

```
unsigned bouton_stop; // Bouton à 2 positions marche/arrêt (valeurs 0 ou 1)
unsigned int annee;   // Assez pour un millésime 0 à 65 535
float volume;         // Réel, virgule flottante pour calculs
```

2.3 Les constantes

Une constante (*constant*) est une donnée que le programme ne peut modifier.
Il faut distinguer 2 types de constantes :

1. **Les constantes non mémorisées**
   Il s'agit en fait de déclarations d'équivalence utilisant la directive #define. Par exemple :
   ```
   #define vrai 1
   #define faux 0
   #define PI 3.1415
   #define phrase "Bonjour à tous"
   ```
2. **Les constantes mémorisées**
   Il s'agit généralement de tableaux de nombres (table de conversion Sinus par exemple) ou de chaînes de caractères mémorisés dans la mémoire flash ou EEPROM. Ces données sont disponibles dès la mise sous tension du µC.
   Exemples de déclaration :

   ```
   const int Table_BN_BR[]={0,1,3,2,6,7,5,4};
   ```

   Le compilateur reconnaît le mot clef *const* et affecte 8 cases mémoire consécutives de la mémoire flash avec les valeurs 0, 1, 3, 2, 6, 7, 5 et 4 codées sur 16 bits.
   Dans le programme, l'accès aux éléments du tableau se fait comme si la table était en RAM :

   ```
   i=Table_BN_BR[3];
   ```

   Dans cet exemple, l'entier i est affecté avec la valeur 2.
   L'architecture Harward du dsPIC ne permet pas normalement ce type d'accès : le bus "données" n'accède qu'à la RAM. Le bloc "PSV" du CPU du dsPIC (voir le document "Présentation du dsPIC") permet de placer une partie de la mémoire programme dans les 32Ko supérieur de l'espace RAM. Heureusement pour le programmeur en C, le compilateur de MPLAB s'occupe de configurer le PSV en fonction des besoins.

2.4 Les variables

Par variable, on entend une place en mémoire pour un nombre, une lettre ou un texte que le programme pourra modifier. En C, toutes les variables doivent avoir été déclarées avant l'emploi.

La déclaration des variables fait partie des «statements» et se termine par un point-virgule. Voici comment déclarer une variable :

```
type  <label>;
```

Exemples de déclarations :

```
int i,j;                        //2 variables entières 16 bits codées C2
unsigned char Un_caractere;       //1 octet pour un caractère
unsigned char Phrase[20]="ABCDEF";//20 octets pour une chaîne de caract
```

Dans le dernier cas, le compilateur ajoute du code pour initialiser le tableau avant le lancement de "main"

Exemples d'affectations et de manipulations dans le programme :

```
i=j=0;
Un_caractere='A';
for (i=0; i<5; i++) Phrase[i]='-';
```

3. Les opérateurs en C

3.1 Les opérateurs arithmétiques

Les signes des opérations arithmétiques correspondent à ceux bien connus des calculettes (sauf %) :

```
+   Addition          // Exemples : y = x+3;
-   Soustraction      //            y = x-b;
*   Multiplication    //            y = a*b;
/   Division          //            y = a/b;
%   Modulo            //            y = a%b;
```

Le signe égal n'a pas la même fonction en C que dans les mathématiques traditionnelles, il sert d'opérateur d'affectation. Cela veut dire que le membre à droite du signe égal représente un calcul à effectuer, dont le résultat sera attribué à la variable inscrite à gauche. Aussi, les expressions suivantes sont-elles admises en C, mais n'auraient pas la même portée en mathématique normale :

```
x =x+y;     // Calculer x+y et mémoriser la réponse dans x
x =-x;      // Changer le signe de la variable x
```

3.2 Les opérateurs de logique combinatoire

Ces opérateurs sont très utilisés dans le contrôle de processus.

```
&   ET                // Exemples : y = a & b;
|   OU                //            y = a | b;
^   XOR               //            y = a ^ b;
~   Inversion         //            y = ~b;
>>  Décalage à droite //            y = a >> x;
<<  Décalage à gauche //            y = a << x;
```

Les opérations logiques sont traitées "bit à bit" comme l'illustre l'exemple ci-dessous :

```
R = A & ~B;
```

Variables de type "char" (8 bits)
A=155=0x9B et B=120=0x78

Bit	7	6	5	4	3	2	1	0
A	1	0	0	1	1	0	1	1
~B	1	0	0	0	0	1	1	1
R	1	0	0	0	0	0	1	1

Le résultat donne R=0x83=131

3.3 Les opérateurs relationnels

Les opérateurs relationnels servent à comparer des variables avec d'autres variables ou des constantes. Ils fournissent ensuite un résultat vrai (valeur 1) ou faux (valeur 0).

>	plus grand que	==	égal
>=	plus grand ou égal	!=	non égal
<	plus petit que		
<=	plus petit ou égal		

Le C autorise l'affectation d'une variable avec un résultat de test :

```
x = A < B;  // 1 si vrai et 0 si faux
```

Piège :
Ne pas confondre == (test d'égalité) et = (affectation) ☺
Exemple :

```
if (x == 2) {...
vs if (x = 2) { ...
```

La première teste l'égalité de la valeur contenue dans la variable *x* et la valeur 2 , alors que la deuxième teste la valeur de l'affectation x=2 qui vaut 2 quelle que soit la valeur de *x* (dans cet exemple).

3.4 Les opérateurs logiques

Il ne faut pas les confondre avec les opérateurs de logique combinatoire : les opérateurs logiques combinent les résultats vrai ou faux des opérateurs relationnels. Elle ne manipulent pas les bits des variables.

```
&&  ET                  // Exemples : if (a==b)&&(a==c) {  };
||  OU                  //            if (a==b)||(a==c) {  };
!   Complémentation     //            if (!(b>c))  {  };
```

Le résultat des opérateurs logiques est toujours vrai ou faux.

3.5 Les raccourcis

Les États-uniens sont passés maîtres dans la composition d'abréviations (*shortcuts*). La remarque s'applique particulièrement aux concepteurs du langage C, Dennis Ritchie et Brian Kerningham. Voilà qui permet de se simplifier la tâche à la saisie du codage mais le rend moins lisible pour le néophyte.

Shortcut	Normal	Shortcut	Normal
a*=b	a=a*b	a<<=b	a=a<<b
a/=b	a=a/b	a>>=b	a=a>>b
a+=b	a=a+b	a&=b	a=a&b
a-=b	a=a-b	a|=b	a=a|b
a%=b	a=a%b	a^=b	a=a^b
a++ ou ++a	a=a+1		
a–ou –a	a=a-1		

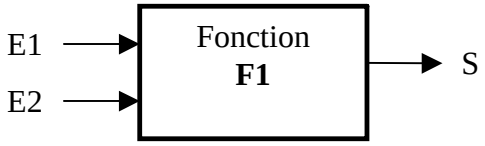
Exemples : recherche de l'indice d'un élément dans une table

```
while (Table[i++]==55);// La variable i est incrémentée après être utilisée comme indice
while (Table[++i]==55);// La variable i est incrémentée avant d'être utilisée comme indice
```

3.6 Les fonctions en C

3.6.1 La notion de fonction

Avant d'écrire la première ligne de programme, le programmeur doit élaborer ou disposer du schéma fonctionnel de l'objet technique dont le dsPIC est le cœur. Ce schéma est une association de fonctions principales, elles-mêmes décomposées en fonctions secondaires.
Certaines de ces fonctions sont réalisées par le "hardware", d'autres par le "software". Ces dernières sont l'objet de ce paragraphe.
Comme pour les fonctions électroniques, une fonction logicielle comporte des entrées, des sorties et un traitement entre celles-ci.



Important : la fonction doit être parfaitement définie **avant** d'écrire son programme en C.

Une étude fonctionnelle menée correctement permet :
- de réaliser des simulations de faisabilité préalables
- de répartir les tâches de développement entre les membres d'une équipe : chacun est chargé d'un certain nombre de fonctions et il n'a pas besoin de tout maîtriser.
- de faciliter la maintenance (une fonction peut être corrigée sans toucher aux autres)
- de faciliter l'évolution du produit

→ Le langage C est adapté à cette démarche grâce aux ***function***.

Exemple : transcription en C de la fonction F1 :

```
int Fonction_F1(int E1, int E2)
{
 return((E1+E2)/2);// Calcul de la moyenne de E1 et E2
}
```

Appel de la fonction :

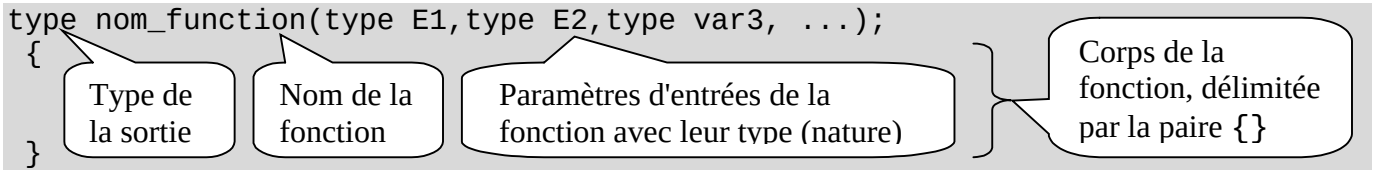
```
i=Fonction_F1(7,5);
```

A l'appel de la fonction, E1 et E2 prennent respectivement les valeurs 7 et 5. La valeur renvoyée correspond à la sortie S et affecte la variable i avec le résultat du traitement (ici, la moyenne des 2 entrées).

Conclusion : l'électronicien programmeur est chargé d'écrire ou de modifier des *function* à partir de leurs descriptions fonctionnelles. Il doit aussi les tester par simulation, sur maquette et sur l'application.
Ces fonctions seront alors utilisées par les informaticiens programmeur pour élaborer le programme complet.
Cela impose que les sources des *functions* soient très bien commentés pour être utilisés par un autre membre de l'équipe sans avoir besoin de les analyser. Il est obligatoire d'indiquer au moins la nature des entrées et sortie et le traitement effectué (voir l'exemple de source du §1.1).
Des commentaires judicieux faciliteront aussi la maintenance, soit pour détecter les "bugs", soit pour améliorer la fonction.

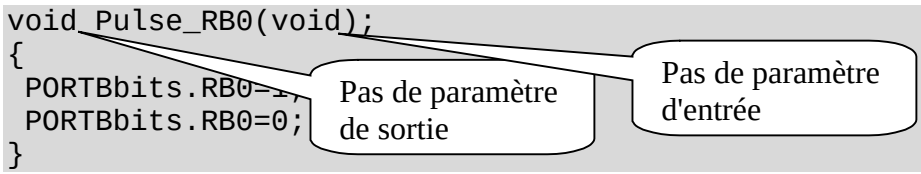
3.6.2 Déclaration d'une fonction

La forme générale d'une fonction C est :



Exemples :

- Une fonction sans paramètre d'entrée ni de sortie :

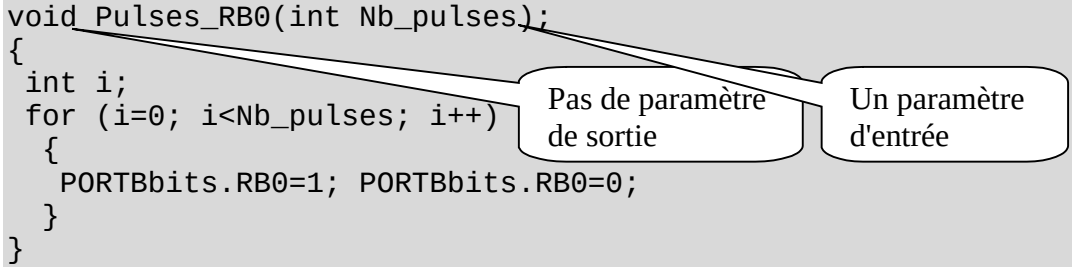


Appel de la fonction :

```
Pulse_RB0();
```

Le mot *void* (vide) indique au compilateur que la fonction *Pulse_RB0* n'a pas de paramètre d'entrée et qu'elle ne renvoie pas de résultat non plus.
Note : l'absence de paramètre ne signifie pas que la fonction n'a pas d'entrée et de sortie (une fonction de ce type n'existe pas !). Ici l'entrée est l'instant d'appel de la fonction et la sortie est le signal RB0 sur lequel est produit l'impulsion.

- Une fonction avec un paramètre d'entrée et aucun en sortie :



Appel de la fonction :

```
Pulses_RB0(10); // Production de 10 impulsions sur la broche RB0
```

- Une fonction avec deux paramètres d'entrée et un en sortie :

```
int Moyenne(int N1, int N2);
{
 int S;
 S=(N1+N2)/2;
 return(S);
}
```

Nature du paramètre de sortie

Deux paramètres d'entrée

Appel de la fonction :

```
X=Moyenne(A,10);
```

Au retour de la fonction X est affecté par le résultat de (A+10)/2
Note : la déclaration de S est optionnelle. La formule de calcul, dans des cas simples comme celui-ci, peut être placée en paramètre de l'instruction *return* (voir §3.6.1).

3.6.3 Appels imbriqués de fonctions

On peut appeler une fonction au sein d'une fonction, qui elle-même appelle une autre fonction et ainsi de suite. Cette possibilité s'est imposée car les schémas fonctionnels sont également imbriqués.

Le nombre d'appels imbriqués n'est limité que par la taille de la pile. En effet, à chaque appel d'une fonction, un grand nombre d'informations est empilé dans la pile :

- l'adresse de retour vers le programme appelant
- les variables locales
- le résultat en cours de calcul

Dans un dsPIC, la pile est placée en RAM, comme les variables globales. La taille de la RAM est très limitée dans un µC (1Ko dans un dsPIC30F2010) et la limite peut être rapidement atteinte. Dans ce cas, le **programme plante** !

Exemple :
La fonction "Factorielle" présentée ci-dessous s'appelle elle-même : elle est **récursive**. Il est très dangereux d'utiliser la récursivité sans expérience, toutefois l'exemple simple proposé met bien en évidence les problèmes de débordement de la pile.

```
int Factorielle(int N)
{
 int Resultat;
 if (N!=1) Resultat=N*Factorielle(N-1);
      else Resultat=1;
 return(Resultat);
}
```

Appel de la fonction dans le programme principal :

```
X=Factorielle(5);
```

Avec ce simple appel, la fonction Factorielle est en fait appelée 5 fois avant de revenir la première fois. Ces 5 appels imbriqués nécessitent déjà 50 octets de pile (pour mémoriser les adresses de retour et les résultats intermédiaires de *Resultat*).

3.7 Les structures de contrôle de programme

3.7.1 if

Il arrive souvent qu'une instruction (*statement*) ou un bloc d'instructions ne doive s'exécuter que si (*if*) une certaine condition est remplie.
Une condition est considérée comme remplie si le test de la condition renvoie la valeur «vrai». Un zéro pour cette valeur signifie «faux» (*false*), n'importe quel autre chiffre est pris pour «vrai» (*true*).

La forme générale est :

```
if (condition) statement;
```

S'il faut plusieurs instructions pour décrire le traitement, alors on doit les grouper entre accolades :

```
if (condition)
 {
  statement_1;
  statement_2;
  statement_3;
  //...
 }
```

Exemples :

```
if (A==2) LED1=1; // Une seule instruction : les "{}" sont inutiles
if (A==3) { LED1=0; LED2=0;}
```

3.7.2 if-else

S'il faut faire exécuter une ou plusieurs instructions quand la condition est réalisée, mais aussi d'autres instructions si elle ne se réalise pas, l'instruction adaptée est la paire: "if – else" (si - sinon).

La forme générale en est :

```
if (condition)
   {
    //...
   }
 else
   {
    //...
   }
```

Exemple :

```
if (PORTDbits.RD1) // Test du BP sur RD1
   {
    LATCbits.LATC13^=1; // Inverser LED1
    LATCbits.LATC14 =0; // Éteindre LED2
   }
 else
   {
    LATCbits.LATC13 =0; // Éteindre LED1
    LATCbits.LATC14^=1; // Inverser LED2
   }
```

3.7.3 Switch

Si, dans une décision, il y a plus de deux options à envisager, l'utilisation de la commande if-else devient laborieuse. On préfère alors faire appel à l'alternative multiple "*switch – case*".

La forme générale est :

```
switch (variable)
{
 case constante1: // Commentaires
  {
   …
   break;
  }
 case constante2: // Commentaires
  {
   …
   break;
  }
 case constante3: // Commentaires
  {
   …
   break;
  }
 default:        // Commentaires
  {
   …
   break;
  }
}
```

La fonction switch compare le contenu de la *variable* à celle de la constante dans les différents cas (*case*) prévus. Quand le résultat de la comparaison est vrai, la ou les instructions correspondantes sont exécutées. Quand le mot-clé *break* (arrêt) est atteint, l'exécution du programme sort de la structure *switch*. Si aucun des cas prévus n'a été rencontré, c'est le bloc d'instructions sous *default* qui s'exécute. Si celle-ci n'est pas nécessaire, on peut omettre cette partie.

Exemple :

```
switch(Buffer_Cmd[1]) // Code de la commande
 {
  case 'B' : // Avance/retard de phase (à multiplier par 16)
   {
    Offset_Angle=Buffer_Cmd[2]<<8;
    break;
   }
  case 'C' : // Type de consigne (dans Buffer_Cmd[2])
   {
    switch(Buffer_Cmd[2])
    {
     case 'P' : // Poignée
     {
      Flags.Consigne=0;
      break;
     }
     case 'T' : // Réglage "Tension" depuis PC via RS232
     {
      Flags.Consigne=1;
      break;
     }
    }
    break;
   }
```

```
  case 'T' : // Consigne "Tension" (0 à 1023)
   {
    POT_Pos=Buffer_Cmd[2]<<8;
    break;
   }
  default : break;
 }
```

Cet exemple illustre une imbrication de *switch*.

3.7.4 for

Si un bloc d'instructions doit s'exécuter plusieurs fois, on utilise la boucle (*loop*) "*for*" .

En voici la forme générale :

```
for (<affectation initiale>;<condition d'arrêt>;<modification compteur>)
   {
    //...
   }
```

Cette structure utilise et manipule une variable "compteur" pour effectuer le bloc d'instructions un certain nombre de fois.
A l'appel de "*for loop*", le compteur est affecté par la valeur initiale. A chaque itération de la boucle, l'état du compteur est modifié (incrémentation le plus souvent) jusqu'à atteindre la condition d'arrêt.
Le compteur peut être n'importe quelle variable de type unaire.

Exemples :

```
int i;

for (i=0; i<10; i++) Cligno_led();// La led clignote 10 fois
```

```
int a, b, c, i;

a = 2; b = 10; c = 4;

for (i=a; i<b; i+=c)
 {
  Led_rouge = 1;
  Led_rouge = 0;
 }
```

Cette dernière boucle sera parcourue 2 fois.

3.7.5 While

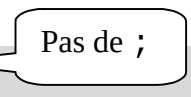
On se sert de la boucle "*while*" quand l'exécution des instructions incluses est soumise à une condition.

Forme générale :

```
while (<condition>)
 {
  //...
 }
```

Lors de l'appel de la boucle "*while*", la condition est d'abord testée. Si la réponse est affirmative (*true*) les instructions du bloc s'exécutent aussi longtemps que le test ne renvoie pas un résultat faux.

Exemple :




```
while (i)
{
 i=BP();
 Cligno_led();
}
```

Tant que la fonction "*BP()*" retourne la valeur 1 (vrai), la fonction "*Cligno_led()*" est exécutée.

3.7.6 Do-While

Aucune boucle "*while*" ne sera exécutée si au moment de son lancement la condition n'est pas remplie. Si au moins une des instructions doit être exécutée, l'instruction de test doit la suivre. Dans ce cas, on utilise la boucle "*do-while*".

Sa forme générale est :

```
do
 {
  //...
 }
while (<condition>);
```

Exemple :

```
do
 {
  i=BP();
  Cligno_led();
 }
while (i);
```

Dans ce cas-ci, la fonction "*Cligno_led*" s'exécute au moins une fois.