Programmation en langage C d'un µcontrôleur PIC à l'aide du compilateur C-CCS

Sommaire

Programmation en C d'un µC PIC avec CCS - C compiler

1. Généralités sur le langage

1.1 Historique

Le langage "C" a fait son apparition en 1972 pour le développement du système d'exploitation Unix (Laboratoire BELL - Dennis Ritchie).
Il est devenu un standard de la norme ANSI en 1988 apres 5 ans de travail => Naissance de la norme « C-AINSI »
En 1988, normalisation du langage C => Norme C-AINSI
. Lui, ainsi que son petit frère le C++, sont très utilisés pour le développement d'applications sous station Unix et PC.
Depuis quelques années il a fait son entrée dans le monde des microcontrôleurs. Il permet de bénéficier d'un langage universel et portable pratiquement indépendant du processeur utilisé.
Il évite les tâches d'écritures pénibles en langage assembleur et élimine ainsi certaines sources d'erreurs.

Nous utilisons « **un des compilateurs C** » du commerce spécifique à la programmation des « microcontrôleurs PIC », le compilateur C de chez CSS.

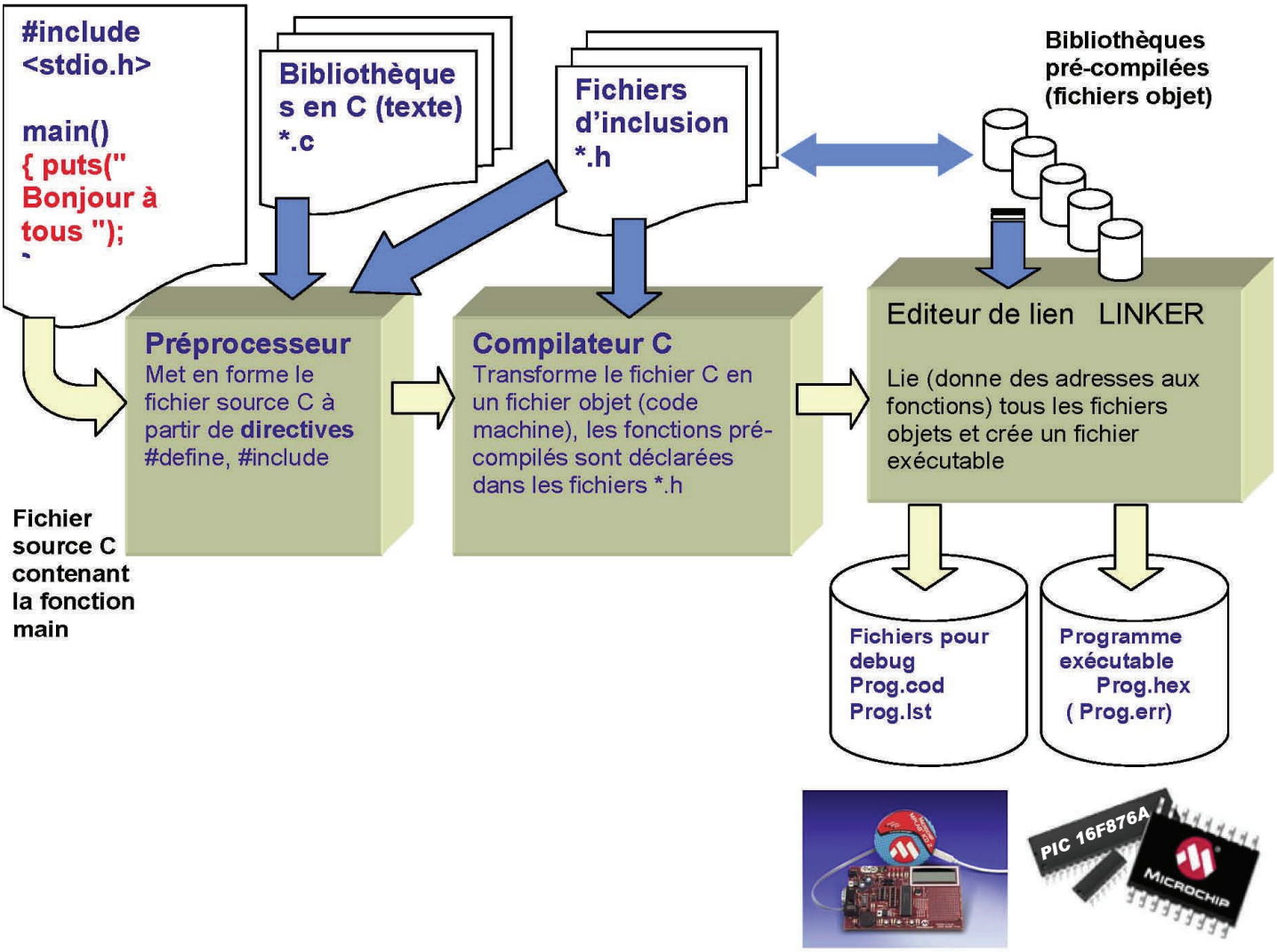
1.2 Qualités du langage :

Le langage C est un langage de programmation évolué, typé, modulaire et structuré :
-> Evolué : Le code est indépendant du processeur utilisé
-> Complet : Un code en C peut contenir des séquences de bas niveau (assembleur) proche du matériel.
-> Typé : Un type est l'ensemble des valeurs que peut prendre une variable (Entiers, réels, caractères etc … ou à définir par le programmeur)
Modulaire et structuré :Tout programme est décomposable en tâches simples (3 structures algorithmiques de base) qui seront regroupées sous forme de modules (fonctions) qui eux même regroupés de façon cohérente en tâches plus complexes (structurés) formeront le programme.
-> Souple : En C on peut tout faire… mais une grande rigueur s'impose.
-> Efficace : On réfléchis (devant sa feuille de papier) et on écrit (peu).

1.3 Filière de développement :

On désigne par filière de développement l'ensemble des outils qui rendre en œuvre pour passer du fichier texte (source codé en C) au code objet (code machine) téléchargé dans le microcontrôleur.
(voir représentation page suivante)

Fichiers (Extensions)	Description du contenu du fichier	Fichiers (Extensions)	Description du contenu du fichier
.C	Fichier source en langage C.	**.HEX**	Code objet (exécutable) téléchargé dans le µC
.H	Entête de définition des broches, Registres, Bits de Registres, Fonctions, et directives de pré-compilation.	**.TRE**	Montre l'organisation du programme sous forme d'arbre (découpages en fonction) et l'utilisation de la mémoire pour chaque fonction.
.PJT	Fichier de projet (pas obligatoire).	**.COF**	Code machine + Informations de débugage
.LST	Fichier qui montre chaque ligne du code C et son code assembleur associé généré.	**.ERR**	Erreurs éventuelles rencontrées durant la compilation.
.SYM	Indique la correspondance entre le nom des symboles (variables, bits, registres) et leur adresses hexadécimale en mémoire.		
.STA	Fichier statistique sur l'espace mémoire occupé. etc.		



1.3 Structure d'un programme en C

```
/* EXEMPLE DE PROGRAMME */
#include <p16f876A.h>
#define duree 10000

char c;
float pht;

void tempo(unsigned int compte);

void main(void)
  {   PORTB = 0x00;
      TRISB = 0x00;
      while(1)
          {   PORTB++;
              tempo(duree);
          }
  }

void tempo(unsigned int compte)
{   while(compte--);
}
```

Commentaires entre /* et */ ou apres //

Bibliothèque du composant utilisé, contient la définition des variables internes PORT A, PORT B ... Indispensable !

Equivalence : remplacées par leur valeur a la compilation

Variables globales (non utilisées ici) :
Char : Octet / Float : Réel.

Prototype de la fonction « tempo », indispensable car le cors de cette fonction est à la fin du programme.

Programme principal (main), Void indique qu'il n'y a pas de paramètre d'entrée.

Acolades qui ouvrent et terminent une fonction ou un ensemble d'instructions

Boucle inifnie

Représentation des nombres :
« 12 » codé en décimal représente 12
« 0xC » codé en hexadécimal représente 12
« 0b00001100 » codé en binaire représente 12

Fonction (sous programme), en C il n'y a que des fonctions
Un paramètre entier en entrée, pas de résultat retourné, du type y=sin(x)
compte est une variable locale car déclarée dans la fonction, elle n'existe que lors de l'exécution de la fonction.

Quelques informations d'écritures :

Un programme en C utilise deux zones mémoires principales :
-> La zone des **variables** est un bloc de RAM où sont stockées des données manipulées par le programme.
-> la zone des **fonctions** est un bloc de ROM qui reçoit le code exécutable du programme et les constantes.

Chaque ligne d'instruction se termine par un « **;** ».
Le début d'une séquence est précédé du symbole « **{** ».
La fin d'un séquence est suivie du symbole « **}** ».

Déclarer une variable ou une fonction, c'est informer le compilateur de son existence.

Les noms des variables que l'on utilise (variables, fonctions) sont des identificateurs. Leur écriture doit respecter les critères suivants :
-> Utiliser les caractères de l'alphabet, de a à z et de A à Z, les chiffres de 0 à 9 (sauf pour un premier caractère), le signe « _ » (underscore).
-> Ne doit pas contenir d'espaces ou de caractères accentués. Ne doit pas dépasser 32 caractères.
-> être représentatif de leur rôle dans le programme (pour une meilleure lecture et compréhension du programme).

ATTENTION : Respect de « la casse », l'identificateur « MaFonction » n'est pas la même chose que « mafonction ».

Constantes : Elles sont rangées dans la ROM et ne sont pas modifiables.
Exemple : *const int i=16569 , char c=ox4c ;*

Equivalences : Déclarés après la directive #define elles sont remplacées par leur valeur lors de la compilation.
Exemple : *#define pi 3.14*
#define A 0x0F *Attention : il n'y a pas de ; après une directive define.*

Notes : Spécificité du compilateur CCS
1. (déclaration #BIT)
Il est possible de déclarer un BIT associé a une variable ou une constante avec ce compilateur
Exemple : *#bit id=x.y*
#bit PB3=PortB.3
#bit LED_R=PortB.4

id = nom du bit (identificateur)
x = nom de la variable ou la constante
y = position du bit

1. (déclaration #BYTE)
Il est possible de déclarer un identificateur à un registre (ou case mémoire) situé en mémoire à partir de son adresse.
Exemple :
```
#byte PortA = 5 // adresse du PORT A
#byte PortB = 6 // adresse du PORT B
#byte PortC = 7 // adresse du PORT C
#byte Option_Reg = 0x81 // adresse du registre Option_Reg
```

Variables : Elles sont rangées dans la RAM soit à une adresse fixe (statique), soit dans une pile LIFO (dynamique)
Exemple : char a,b=28,c='A';

1.4 Les variables

Il existe différents types de variables reconnus par le compilateur :

Par défaut, tous ces types de données sont non signés, ils peuvent être signé en rajoutant le mot clé **signed** devant le type.

Type	Description	taille	domaine signé	domaine non signé
int1	Défini un nombre de 1 bit	1 bit	X	0 ou 1
int8	Défini un nombre de 8 bits	8 bits	-128 à + 127	0 à 255
int16	Défini un nombre de 16 bits	16 bits	-32768 à + 32767	0 à 65535
int32	Défini un nombre de 32bits	32 bits	[-2 147 483 648 à 2 147 483 647]	0 à 4 294 967 295
char	un caractère (codé en interne comme un octet)	8 bits	-128 à + 127	0 à 255
float	Nombre à virgule (réel) sur 32 bits	32 bits	3.4 x 10^-38 à 3.4 x 10^+38	X
short	Idem que int1	1 bit	X	0 ou 1
int	Idem que int8	8 bits	-128 à + 127	0 à 255
long	Idem que int16	16 bits	-32768 à + 32767	0 à 65535
void	type non spécifié			

Exemples de déclarations :

Exemple :
```
char a,b='A';
int1 led;
signed char c; // les variables est signée -> [-128 à 127]
```

1.5 Les vecteurs

Un vecteur est un tableau (matrice) de taille et de nombre de dimension quelconque.
Exemple un tableau permettant de stocker une phrase (chaine de caractères) est un tableau à une dimension (1 ligne de caractères)
les éléments (champs) du vecteur sont toujours du même type.

Exemple : char message[20] // message est un tableau de caractères : 20 positions.

remarque : les 20 éléments de ce vecteur sont : message[0], message[1],…., à message[19]

1.6 Les chaines de caractères :

Ce sont des vecteurs de 1 dimension dont les champs sont des caractères (type char). Le dernier élément est le caractère ascii « null » (ou \0) (a ne pas confondre avec le symbole du zéro ($30).

Exemple : char *chaine= "bonjour"

Remarque : La taille de la chaine n'est pas spécifiée

1.7 Les pointeurs :

Un pointeur est une adresse mémoire qui peut être utilisé pour accéder à une donnée.
Le pointeur est un nombre non signé de 8 bits. Manipuler un pointeur, c'est manipuler l'adresse d'une donnée (adressage indirect)
L'opération d'indirection est le : « * »

Exemple :

```
char *porta ;       // pointeur de 8 bits vers un octet
porta=0x05          // initialisation du pointeur
*porta=0x80;        // placer la valeur 0x80 à l'adresse 0x05
```

1.8 Les bases du compilateur CCS.

On a la possibilité d'écrire les chiffres de cette manière :

Le décimal : A = 10 ;
L'octal : A = 012 ;
L'hexadécimal A = 0x0A ;
Le binaire A = 0b00001010 ;

Pour les caractères : Exemple la lettre A code ASCII 65(Décimal) ou $41(Hexadécimal), peut s'écrire :
LETTRE = 65 ;
LETTRE = 0x41 ;
LETTRE = 'A' ;

1.9 Les opérateurs du langage C

Lors de son exécution, un programme est amené à effectuer des opérations qui peuvent être purement arithmétiques, de comparaison, d'affectation, etc…

Type	Symbole	Exemple
Opérateur d'affectation		
Affectation	=	x=10 ; y=a+b
Opérateurs arithmétiques		
addition	+	a = a+b ; x= 5 + a
soustraction	-	a= a-b ; y= c-51
moins unitaire	-	a = -b
multiplication	*	a = a * a ; b = y * 8
division	/	c = 9 / b ; d = a / b
Reste de la division entière (modulo)	%	r = a % b
Opérateurs logiques de comparaison => Le résultat de la comparaison peut prendre deux valeurs vrai ou faux		
ET logique	&&	c = a $$ b // c est vrai si a et b sont vrais, sinon c est faux
OU logique	||	c = a || b // c = 1 si a ou b sont non nuls, sinon c=0
Non logique	!	c = !c ; a=!b // c prend le complément de ce qu'il valait

Type	Symbole	Exemple
Opérateurs de comparaison => Le résultat de la comparaison peut prendre deux valeurs vrai ou faux		
égalité	==	if a==b ; if c==9
différent	!=	if c!=a
supérieur	>	if a>b ; if 8 >a
supérieur ou égal	>=	if a>=b
inférieur	<	if a>b ; if 8 >a
inférieur ou égal	<=	if a>=b
Opérateurs binaires de comparaison bit à bit => Le résultat de la comparaison peut prendre deux valeurs vrai ou faux		
ET binaire	&	x = a $ b // c est vrai bit à bit si bit à bit a et b sont vrais
OU binaire	|	x = a | b // c est vrai bit à bit si bit à bit a ou b sont vrais
OU exclusif binaire	^	x = a ^ b // c est vrai bit à bit si bit à bit a ou exclu b sont vrais
complément à 1	~	a = ~ b
décalage de n bits à droite	>>	x = y >> n // y = x décalé de n bits à droite
décalage de n bits à gauche	<<	x = y << n // y = x décalé de n bits à gauche

Exemples (a et b deux entiers) tel que :
a = 1100 0111 0101 0011 (0xC753)
b = 0001 1001 1010 1110 (0x19AE)

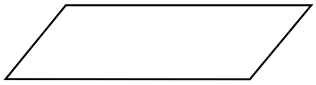
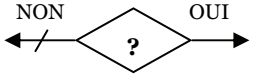
a & b = 0000 0001 0000 0010 (0x0102)
a | b = 1101 1111 1111 1111 (0xDFFF)
a ^ b = 1101 1110 1111 1101 (0xDEFD)
~ a = 0011 1000 1010 1100 (0x38AC)
~ b = 1110 0110 0101 0001 (0xE651)
a<<2 = 0001 1101 0100 1100 (0x1D4C)

2. Les structures algorithmiques

2.1 Présentation

-> Il existe différentes techniques pour représenter le fonctionnement d'un système automatisé : GRAFCET, Algoritmes, Organigramme.
-> Ces techniques sont également utilisées **pour la recherche et la mise au point de programmes**, qui ne se font en général pas directement dans le langage de programmation.
-> On passe par des ***étapes de dégrossissement*** du problème posé.
-> Pour cela on utilise des ***Algorithmes*** ou des ***Organigrammes***.
-> Ces techniques permettent de trouver une solution sur le papier qu'il faudra ensuite traduire dans le langage de programmation utilisé.

Définition d'un Algorithme	Définition d'un Organigramme
Suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème	Représentation schématique d'une solution à un problème

Eléments représentatifs d'un algorithme	Eléments représentatifs d'un Organigramme
Début : Mot délimiteur qui annonce l'introduction d'un algorithme	
Action : Opération sur des données : LIRE, AFFICHER, FAIRE, ECRIRE	
Test : Embranchement ; Ce symbole est utilisé pour représenter une décision ou un aiguillage.	
Fin : Mot délimiteur qui annonce la sortie d'un algorithme	

2.2 les structures algorithmiques

On peut trouver les trois familles algorithmiques suivantes :

A. Structure linéaire

B. Structure alternative

C. Structure itérative.

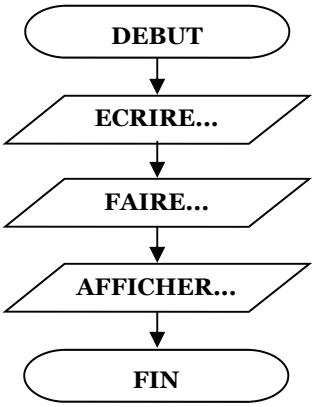
STRUCTURE LINEAIRE

Une structure linéaire est une suite d'actions à exécuter successivement dans l'ordre de leur énoncé. C'est UNE SÉQUENCE.
On utilise des verbes d'action caractéristiques des différentes phase de traitement.

Algorithme

```
DÉBUT
    ECRIRE...
    FAIRE...
    AFFICHER...
FIN
```

Algorigramme



Langage C

```
void main()
{
        print(...);
        PortA=0x55;
        ...
}

/* Remarque : la traduction en langage C est immédiate */
```

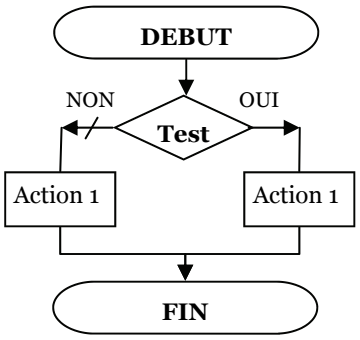
STRUCTURE ALTERNATIVE

Il y a **UN CHOIX** dans les actions à exécuter. Ce choix est fait en fonction d'un résultat d'un test.

Algorithme

```
DÉBUT
    SI « test vrai »
        Alors « action 1 »
        Sinon « action 2 »
    Fin de SI
FIN
```

Algorigramme



Langage C

```
void main()
{
        if (condition vraie)
        {
                Action 1;
        }
        else
        {
                Action 2;
        }
}
```

Remarques :

1/ La clause else peut etre omise

exemple :

```
If (compteur<1000)
{
        compteur = compteur+1 ;
}
```

2/ On peut emboiter des if (placer des if en cascade : if... If.... else if... else... else if

Regle : toujours associer le « else » au « if » sans « else » le plus proche

soigner l'indentification pour préserver la lisibilité du programme

Exemple :

```
if (x > 80)
        if (y > 25)
                break;
        else
                y++;
else
x++;
```

STRUCTURE ALTERNATIVE

Remarques (suite) :
Alternative Multiple : « **switch** » et « **case** »

Une seule variable entraîne des choix multiples :

Exemple :
```
switch (choix)
        case c1 :  < sequence 1>   /* exécutée si la variable choix ==  C1  */
        case c2 :  < sequence 2>   /* exécutée si la variable choix ==  C2  */
        case c3:  < sequence 3>   /* exécutée si la variable choix ==  C3  */
        ....
        case cn:  < sequence n>   /* exécutée si la variable choix ==  Cn */
        default:  < sequence_par_defaut>   /* exécutée si aucune des conditions ci-dessus est vrai */
```
Note : - Si une séquence est exécutée, toutes les suivantes le seront également sauf si elle se termine par l'instruction break.
- La séquence par défaut n'a de sens que si la séquence n (au moins) se termine par break.
- la variable de contrôle est de type entier, les conditions **ci** sont des constantes numériques.

STRUCTURE ITERATIVE

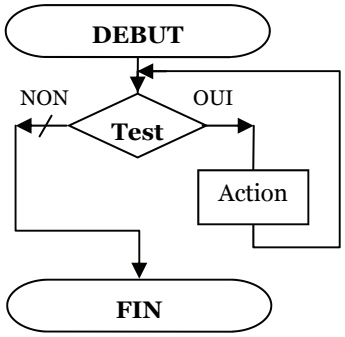
On rencontre 3 formes différentes qui permettent **LA RÉPÉTITION D'ACTIONS**.

1/ STRUCTURE **TANT QUE** (Répetition tant que le test est vrai)

Algorithme

```
DÉBUT
        TANT QUE « test vrai »
                « action »
        Fin de TANT QUE
FIN
```

Algorigramme



Langage C

```
void main()
{
        while (condition vraie)
        {
                Action 1;
        }
}
```

Remarques :
-> l'action n'est pas obligatoirement effectuée au moins une fois.
-> tant que la condition est vraie, la séquence d'action est exécutée.
-> la condition est évaluée avant exécution.
exemples :
```
while (x < x)
        {
                y=0;   // cette action ne sera jamais exécutée
        }

while (3) ;  // Provoque un bouclage infini  - remarquer que la séquence d'action est vide

while (x )       // calcule = = y * 10 * x
        {
                x-- ;
                y*=10 ;
        }
```

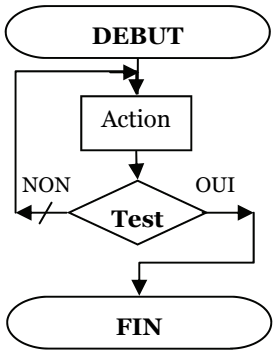
STRUCTURE ITERATIVE

2/ STRUCTURE **REPETER** (Répétition jusqu'à que le test soit vrai)

Algorithme

```
DÉBUT
REPETER
    « action »
jusqu'à que « test vrai »
Fin de tant que
FIN
```

Algorigramme



Langage C

```
void main()
{
    do
    {
        Action ;
    }
    while (condition vraie)
}
```

Remarques :

-> l'action est effectuée au moins une fois.

-> tant que la condition est vraie, la séquence d'action est exécutée.

-> les mots clés do et while sont des délimiteurs de la séquence : on peut se passer des accolades.

exemples :

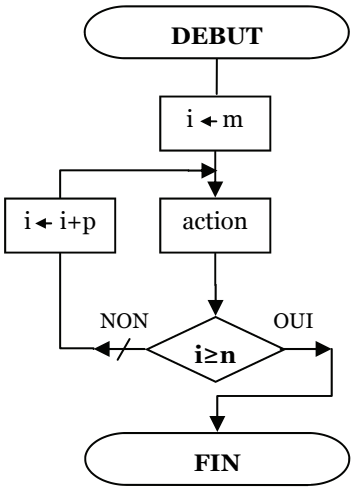
```
do
    {
        compteur=compteur+1 ;   // cette action sera exécutée au moins une fois
    }
while (!(compteur==100) ;  //tant que le while() est vrai soit (compteur==100) faux, on incrémente le compteur
```

3/ boucle **POUR** (Répétition un nombre de fois déterminées par un compteur)

Algorithme

```
DÉBUT
POUR compteur i
    variant de m à n
    par pas p
    « action »
Fin de POUR
FIN
```

Algorigramme



Langage C

```
void main()
{
    For (i=m ; i<=n ; i=i+p)
    {
        Action ;
    }
}
```

Remarques :

-> la séquence est exécutée tant que la condition est vraie. En général, l'opération agit directement ou indirectement sur la condition.

exemples :

```
for (a=0 ; a<=10; a++)      /* pour 0 ≤ a ≤ 10 */
for ( ; a<=10; a++)         /* pas d'initialisation de  a */
for (a=0 ; ; a++)           /* pas de condition (tjrs fausse) : boucle infinie */
for (a=0 ; a<=10; )         /* pas d'opération finale : a doit etre incrémenter dans les actions */
```

3/ boucle **POUR** (Répétition un nombre de fois déterminées par un compteur)

Remarques (suite…):

```
/************** Affiche les nombres de 0 à 10 **************/
main ()
{
char i;
        for ( i = 0; i <= 10; i++)
        {
              printf("i = %d\n", i);      /* i affiché en décimal et retour à la ligne */
        }
}
```

Autre écriture :

```
/************** Affiche les nombres de 0 à 10 **************/
main ()
{
char i;
        for ( i = 0; i <= 10;)
        {
              printf("i = %d\n", i++);            /* i affiché en décimal et retour à la ligne et incrémenter i*/
        }
}
```

Ou encore :

```
/************** Affiche les nombres de 0 à 10 **************/
main ()
{
char i;
i=0;              /* Initialiser i */
        for (; i <= 10;)
        {
              printf("i = %d\n", i++);            /* i affiché en décimal et retour à la ligne et incrémenter i*/
        }
}
```

Pour n'afficher que des valeurs paires :

```
/************** n'affiche que les nombres pairs  de 0 à 10 **************/
main ()
{
char i;
i=0;              /* Initialiser i */
        for (; i <= 10;i++)                       /*incrémenter i*/
        {
              printf("i = %d\n", i++);            /* i affiché en décimal et retour à la ligne et incrémenter i*/
        }
} // remarque : par boucle i est incrémenté deux fois.
```

3/ boucle **POUR** (Répétition un nombre de fois déterminées par un compteur)

Remarques (suite…):

L'instruction « break »

Elle permet de sortir immédiatement d'une boucle au moyen d'un saut inconditionnel à l'instruction qui suit la boucle.
Exemple :

```
main ()
{
char i;
        for ( i = 0; i <= 10; i++)
        {
                if (i==5) break;                /* lorsque i==5 on quitte la boucle / seuls les nombres 0 à 5 seront affichés */
                printf("i = %d\n", i);          /* i affiché en décimal et retour à la ligne */
        }
}
```

L'instruction « continue »

Elle permet de ne pas exécuter la séquence comprise entre « continue » et la fin de boucle tout en poursuivant le bouclage. Il s'agit là aussi d'un saut inconditionnel.
Exemple :

```
main ()
{
char i;
        for ( i = 0; i <= 10; i++)
        {
                if (i==5) continue;             /* lorsque i==5 on n'exécute plus la fin de boucle */
                printf("i = %d\n", i);          /* i affiché en décimal et retour à la ligne */
        }
}
/* dans ce programme, les nombres 0 à  4 et 6 à 10 sont affichés. La boucle est cependant exécutée 11 fois */
```

3. LES FONCTIONS ET PROCEDURES

3.1 Définition

On appelle procédure ou fonction, un sous-programme qui est appelé depuis le programme principal.
Une fonction (ou procédure) est une série d'instructions.

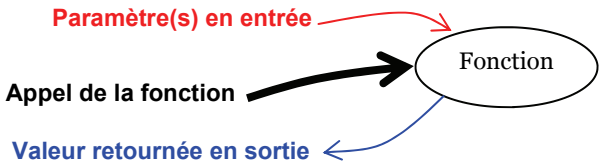
Une fonction peut être appelée par une autre fonction et doit retourner une valeur sauf si elle est du type void.

Elle peut prendre des arguments dont les noms et types sont définis entre les parenthèses.
Si la fonction n'attend aucun paramètre, on l'indique par le mot réservé void entre parenthèses.

Un paramètre d'entrée est la référence à une variable manipulée par la procédure ou la fonction.
Un paramètre de sortie est une valeur renvoyée par une fonction.

3.2 Syntaxe

La syntaxe d'une fonction est la suivante:




```
Type_de_la_valeur_de_retour Nom_de_la_fonction (type noms_des_paramètres)
    {
        liste d'instructions;        // Une ou plusieurs instructions séparées par des ;
        return (valeur)              // Valeur à renvoyer à la fin de la fonction
    }
```

Remarque :
-> Si la fonction ne renvoi pas de valeur (type de la valeur de retour = VOID), la ligne return (valeur) est inutile.
-> Tous les paramètres d'entrées deviennent des variables locales de la fonction.

a/ Exemple de fonction sans paramètre d'entrée et de sortie :

Pour appeler la fonction dans un programme :

```
initialisation();
```

Déclaration de la fonction :


```
void initialisation(void)
{
    TrisB=(0);
    TrisA=(0);
    TrisC=(0x8F);
    Adcon1=7;        // Toutes les broches du PortA en numérique
}
```

b/ Exemple de fonction avec des paramètres d'entrée et pas de valeur retournée en sortie :

Pour appeler la fonction dans un programme :

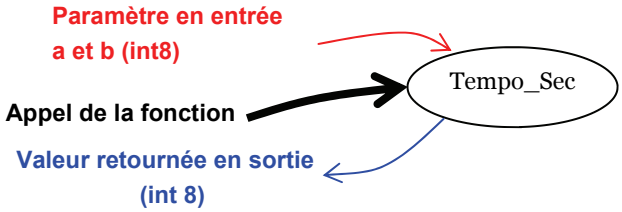

```
Tempo_Sec(10);
```

Déclaration de la fonction :

```
void Tempo_Sec(int8 duree)
{
    delay_ms(1000*duree); /* Appel d'une autre fonction Delay_ms
                             (avec comme paramètres 1000 * duree) */
}
```

c/ Exemple de fonction avec des paramètres d'entrée et une valeur retournée en sortie :

Pour appeler la fonction dans un programme :




```
c=Max(10,30);
```

Déclaration de la fonction :

```
int8 Max(int8 a, int8 b)
{
    if(a>b)         //si 'a' est supérieur à 'b'
        return a;   //on retourne 'a'
    else            //sinon
        return b;   //on retourne 'b'
}
```

4. Les bibliothèques standards

Il existe un ensemble de fonctions déjà existantes que l'on peut utiliser dans nos programmes.
Ces fonctions sont stockées dans des librairies (tel que stdio.h par exemple) que l'on doit faire référence en début de programme.
Exemple :

#include <stdio.h>

Fonctions de temporisation

Le compilateur intègre des fonctions très pratiques pour gérer les délais, à savoir :

```
delay_cycles(valeur) ;    // temporisation en NB de cycles
delay_us(valeur) ;        // temporisation en µS
delay_ms(valeur) ;        // temporisation en mS
```

Remarque :
Pour pouvoir utiliser ces fonctions, il faut indiquer par la ligne ci-dessous la fréquence du Quartz de votre application.
Cette ligne doit être définie au début du programme.
#use delay (clock=fréquence_du_quartz)

Exemples :
#use delay (clock=4000000) // Quartz de 4Mhz
#use delay (clock=20000000) // Quartz de 20Mhz

Fonctions de temporisation

Le compilateur intègre des fonctions très pratiques pour gérer les délais, à savoir :
delay_cycles(nbcycle) *; // temporisation en NB de cycles*
delay_us(valeur) **;** *// temporisation en µS*
delay_ms(valeur) **;** *// temporisation en mS*

valeur *peut être un int16 (de 0 à 65535) ou une constante de de 0 à 65535.*
nbcycle *est une constante de de 1 à 255.*
Remarque :
Pour pouvoir utiliser ces fonctions, il faut indiquer par la ligne ci-dessous la fréquence du Quartz de votre application.
Cette ligne doit être définie au début du programme.
#use delay (clock=fréquence_du_quartz)

Exemples :

```
/**** #use delay (clock=4000000)       Quartz de 4Mhz ****/
#use delay (clock=20000000)    // Quartz de 20Mhz

void main ()
{
        delay_us(1000);          // Créé une pause de 1 mS
        delay_cycles(10);        // Créé une pause de 10 x Tcycl = 10x200 ns = 2µS
        delay_ms(1000);          // Créé une pause de 1 Seconde.
}
```

La communication via la liaison série RS232

Il est possible de rediriger les opérations de lecture et d'écriture vers la liaison série que l'ont peut récupérer via l'hyperterminal de Windows. Avantage : Afficher des valeurs issue du µC ou lui en envoyer.
Les fonctions que nous pouvons utiliser :
- printf() ;
- putc() ;
- puts() ;
- getc() ;
- gets() ;
- kbhit() ;

Remarque :
Pour pouvoir utiliser ces fonctions, il faut indiquer par la ligne ci-dessous que l'on souhaite utiliser la liaison RS232 comme périphérique d'E/S. Cette ligne doit être définie au début du programme après la ligne qui définie la vitesse du quartz.

```
#use delay (clock=20000000)                          // Quartz de 20Mhz
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)      //Câblage liaison + vitesse de transfert en bauds
```

Ecriture formatée de données : printf()

Cette fonction permet d'envoyer une chaine de caractère formatée sur la liaison RS232.
Syntaxe :
printf (chaine)
ou
printf (Cchaine, valeurs...)

chaine *peut être une constante de type chaine ou un tableau de char terminé par le caractère null.*
Cchaine *est une chaine composée (voir ci-dessous).*
valeurs *est une liste de variables séparées par des virgules.*
La « Cchaine » doit etre composée de chaines de caractères constante et d'arguments de mise en forme des valeurs représenté par le symbole % (formatage)
Le formatage des variables :
%c : caractère ascii
%d : entier (int8 ou int 16) signé affiché en base décimale
%o : entier (int8 ou int 16) signé affiché en base octale
%u : entier (int8 ou int 16) non signé affiché en base décimale
%lu : entier long (int32) non signé affiché en base décimale
%ld : entier long (int32) signé affiché en base décimale
%x : entier (int8 ou int 16) affiché en base hexadécimale en minuscules
%X : entier (int8 ou int 16) affiché en base hexadécimale en majuscules
(pour les autres arguments voir la documentation de CCS-Compiler page 196)

Rappel *: Il faut avoir initialiser la gestion de la liaison RS232 avec la directive #use RS232() (voir précédemment)*

Exemple :

```
#use delay (clock=20000000)    // Quartz de 20Mhz
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)
void main ()
{       printf("Bonjour !");
        printf("\r\nMin: %2X  Max: %2X\n\r",min,max);  // \n=LF (saut de ligne), \r=CR (retour à la première colonne)
        printf("%dh %dm %ds »,heu,min,sec);
}
```

Ecriture formatée de données : putc()

Cette fonction permet d'envoyer un caractère formatée sur la liaison RS232.
Syntaxe :
putc (char)
char *est un caractère 8 bits*

Rappel *: Il faut avoir initialiser la gestion de la liaison RS232 avec la directive #use RS232() (voir précédemment)*

Exemple :

```
#use delay (clock=20000000)    // Quartz de 20Mhz
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)
void main ()
{
        putc('A');
        for(i=0; i<10; i++)
                putc(buffer[i]);
        putc(13);
}
```

Ecriture formatée de données : getc()

Cette fonction permet de recevoir un caractère formatée sur la liaison RS232.
Syntaxe :
variable=getc ()
***variable** est un caractère 8 bits*

***Rappel** : Il faut avoir initialiser la gestion de la liaison RS232 avec la directive #use RS232() (voir précédemment)*

Exemple :

```
#use delay (clock=20000000)    // Quartz de 20Mhz
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)
void main ()
{       printf("Continuer (O,N)?");
        do
        {       reponse=getch();
        }
        while (reponse!='O' && reponse!='N');
}
```

Ecriture formatée de données : puts()

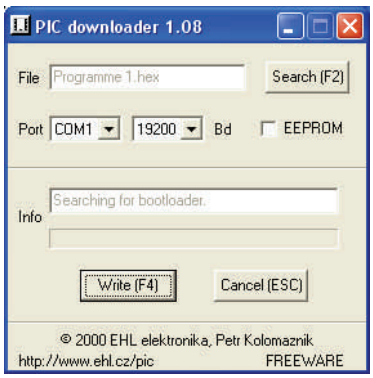
Cette fonction permet d'envoyer une chaine de caractères sur la liaison RS232, terminée par le passage a la ligne (LF)
Syntaxe :
variable=puts(chaine)
***chaine** est une chaine de caractères constante terminé par le caractère null*
***Rappel** : Il faut avoir initialiser la gestion de la liaison RS232 avec la directive #use RS232() (voir précédemment)*

Exemple :

```
#use delay (clock=20000000)    // Quartz de 20Mhz
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)
void main ()
{
        puts( " ------------ " );
        puts( " | Salut ! | " );
        puts( " ------------ " );
}
```

5. Le bootloader

Le bootloader est un programme « programmé » une première fois dans le microcontrôleur qui s'exécute au reset du µC. Apres s'être exécuter au démarrage, il donne « la main » au reste du programme (votre programme).



Son rôle est de gérer la liaison série et de permettre de télécharger par cette liaison votre code exécutable dans le pic IN SITU en utilisant sur le pc le logiciel « Pic downloader ».

Le PIC donné aux élèves contient déjà ce morceau de programme.
Attention : Il est impératif de protéger en mémoire le bootloader en incluant dans votre programme la ligne suivante :
#org 0x1F00, 0x1FFF {} *// Zone programme Protection Bootloader 16F876A*

LES FUSIBLES pour le PIC16F876A (notation CSS)	
Note : Le fonctionnement du microcontrôleur est défini par des « fusibles » qui sont en fait des valeurs définies dans des registres spécifiques du PIC. Pour configurer le PIC à la compilation il suffit de rajouter la directive ci-dessous avec les fusibles choisis ci-dessous. *#fuses HS,NOWDT,NOPROTECT,NOLVP*	
Configuration de l'Horloge	
HS	Oscillateur externe à très grande vitesse (Foscillation > 4 MHz)
XT	Oscillateur externe à grande vitesse (200KHz<Foscillation< 4MHz)
RC	Oscillateur RC Interne à 8 MHz (broche OSC1=E/S normale , OSC2=Fosc/4)
RC_IO	Oscillateur RC Interne à 8 MHz (broche OSC1 et OSC2 =E/S normales)
LP	Faible consommation : quartz à 32 KHz.
Sécurité	
PROTECT	Protection activée : Impossible de lire le code dans le microprocesseur.
NOPROTECT	Protection désactivée : possibilité de lire le code dans le microprocesseur.
CPD	Protection du code mémoire dans l'EEPROM : Seulement le CPU peut accéder à l'EEPROM
NOCPD	Pas de protection du code mémoire dans l'EEPROM.
WRT	Protection du code programme totale à la lecture (interdiction de lire) : 100% du code protégé
WRT_50%	Protection lecture : 50% des cases mémoires sont protégées
WRT_25%	Protection lecture : 25% des cases mémoires sont protégées
WRT_5%	Protection lecture : Seul les 255 dernières cases mémoire sont protégées [1F00 à 1FFF]
NOWRT	Pas de protection de la mémoire programme
Reset	
WDT	Activation du chien de garde (provoque le reset du µC si un bit d'un registre spécifique est réinitialisé périodiquement par le programme)
NOWDT	Pas de chien de garde
BROWNOUT	Activation du BROWN ON RESET : Reset sur une défaillance d'alim
NOBROWNOUT	Fonction BOR désactivé.
PUT	TIMER activé au RESET
NOPUT	TIMER désactivé au RESET
Débogage	
DEBUG	Autorise le mode débogage avec la sonde de programmation et de débogage In-Situ ICD
NODEBUG	mode débogage désactivé
Programmation	
LVP	Programmation avec tension de programmation faible sur la broche B3 (PIC16) ou B5 (PIC18) [PGM]
NOLVP	Pas de programmation base tension : broche B3 (PIC16) ou B5 (PIC18) utilisée comme une E/S normale