Leçon N°4

INTRODUCTION AU LANGAGE MIKROC

I. STRUCTURE D'UN PROGRAMME EN MIKROC

La structure la plus simple d'un programme en mikroC, c'est le programme représenté dans le code-source 4.1, qui nous permettra de faire clignoter une LED connectée au PORTB (par exemple bit 0 du PORTB) du microcontrôleur PIC avec une période de 2 secondes (1 seconde allumée et une seconde éteinte).

1. Règles générale d'écriture en mikroC

- Les instructions propres au langage mikroC doivent être écrites en minuscule (**void** main (void)).
- Les instructions particulières aux microcontrôleurs doivent être écrites en majuscule (TRISB).
- Les retours à la ligne et les espaces servent uniquement à aérer le code
- Toutes instructions ou actions se terminent par un point virgule « ; ».

Code-source 4.1 (LED.c)

```
/************************************************************************
                                LED clignotante
                                ===============
Ce programme fait clignoter une LED connectée à la broche RB0 du PIC 16F84A
************************************************************************/
void main( )
{
    for( ; ; )                  // Boucle sans fin
    {
        TRISB = 0;              // Configuration du PORTB  en sortie
        PORTB.F0 = 0;           // RB0 = 0
        Delay_Ms(1000);         // Pause d'une seconde
        PORTB.F0 = 1;           // RB0 = 1
        Delay_Ms(1000);         // Pause d'une seconde
    }                           // Fin de la boucle
}
```

Examinons le fichier *LED.c* du code-source

2. Commentaires

En mikroC, les commentaires de programmes peuvent être de deux types : de ***longs commentaires,*** s'étendant sur plusieurs lignes, et de ***courts commentaires***, occupant une seule ligne.

Comme montre le **Code-source 4.1** les ***longs commentaires*** commencent par le caractère « /* » et se terminent par le caractère « */ ». De même, de ***courts commentaires*** commencent par le caractère « ***//*** » et il n'a pas besoin d'un caractère de terminaison.

3. Début et fin d'un programme

En mikroC, un programme commence avec les mots-clés :

void main()

Après cela, une accolade ouvrante est utilisée pour indiquer le début du corps de programme. Le programme se termine par une accolade fermante. Ainsi, comme indiqué dans le **Code-source 4.1**, le programme a la structure suivante :

```
void main()
{
        // Votre code ici
}
```

II. ELEMENTS DE PROGRAMMATION EN MIKROC

1. Les variables

Une variable est une portion réservée d'une mémoire à laquelle on a donné un nom. Toute variable utilisée dans un programme doit auparavant être définie.

La *définition* d'une variable consiste à la *nommer* et lui donnée un *type* et éventuellement lui donner une valeur initiale (*initialiser*). C'est cette définition qui réserve (*alloue*) la place mémoire nécessaire en fonction du type.

La position de la déclaration ou de la définition d'une variable détermine sa portée c'est-à-dire sa durée de vie et sa visibilité.

- Les variables *globales* sont déclarées en dehors de toute fonction.
- Les variables *locales* sont déclarées à l'intérieur des fonctions et ne sont pas visible à l'extérieur de la fonction dans laquelle celle-ci est définie.

Les noms des variables ne peuvent contenir que des lettres de *a* à *z* et à partir de *A* à *Z*, le trait de soulignement "_" et les chiffres de 0 à 9.

Les noms des variables dans mikroC n'est pas sensible à la casse, de sorte que *Som*, *som* et *soM* représente le même identifiant.

Certains noms sont réservés pour le compilateur lui-même et ne peut pas être utilisés comme noms de variables dans un programme. Le tableau 4.1 donne une liste alphabétique de ces noms réservés.

Tableau 4.1 Noms réservés en mikroC

ASM	***ENUM***	***SIGNED***	***AUTO***	***EXTERN***	***SIZEOF***
BREAK	***FLOAT***	***STATIC***	***CASE***	***FOR***	***STRUCT***
CHAR	***SWITCH***	***CONST***	***IF***	***TYPEDEF***	***CONTINUE***
GOTO	***INT***	***UNION***	***LONG***	***DEFAULT***	***UNSIGNED***
DOUBLE	***RETURN***	***VOLATILE***	***DO***	***REGISTER***	***VOID***
ELSE	***SHORT***	***WHILE***			

2. Les constantes

Les constantes représentent des valeurs fixes (numérique ou caractère) dans des programmes qui ne peuvent pas être changées. En mikroC, les constantes peuvent être entiers, flottants, caractères, chaînes ou des types énumérés.

2.1. Integer Constants

Les constantes entières (*Integer Constants*) peuvent être en décimal, hexadécimal, octal ou binaire. Le suffixe *u* ou *U* force la constante d'être non signé (***unsigned***) et le suffixe *l* ou *L* force la constante d'être longue (***long***). L'utilisation de *U* (ou *u)* et *L* (ou *l*) oblige la constante d'être *unsigned long*.

Les constantes sont déclarées en utilisant le mot-clé *const* et sont stockées dans le flash de mémoire du microcontrôleur PIC. Par exemple, *MAX* est déclaré comme la constante 100 :

const MAX =100;

Les constantes hexadécimales commencent par les caractères 0x ou 0X et peuvent contenir des données numériques de 0 à 9 et les caractères hexadécimaux de A à F. Dans l'exemple suivant, *TOTAL* est la constante de la valeur hexadécimale FF:

const TOTAL = 0xFF;

Les constantes octales ont un zéro au début du nombre et peuvent contenir des données numériques de 0 à 7. Dans l'exemple suivant, une constante *CNT* est affectée une valeur octale 17 :

const CNT = 017;

Les constantes binaires commencent par 0b ou 0B et ne peuvent contenir que 0 ou 1. Par exemple une constante nommée *Min* est déclarée comme ayant la valeur binaire 11110000 :

const Min = 0b11110000

2.2. Floating Point Constants

Les constantes à virgule flottante (*Floating Point Constants*) se compose de :

- ✓ Entier décimal
- ✓ Point décimal
- ✓ Partie décimale fractionnaire

✓ e ou E et l'ordre du signe

Dans l'exemple suivant, une constante nommée *TEMP* est déclarée comme ayant la valeur fractionnelle 37.50 :

const TEMP = 37.50 ou *const TEMP = 3.750E1*

2.3. Character Constants

Une constante de caractère (*Character Constants*) est un caractère renfermé dans des guillemets simples. Par exemple, une constante nommée *First_Alpha* est déclarée comme ayant la valeur du caractère 'A':

const First_Alpha = 'A';

2.4. String Constants

Les constantes de chaîne (*String Constants*) sont des séquences fixes de caractères stockées dans la mémoire flash du microcontrôleur. La chaîne doit commencer et se terminer par un guillemet « " ». Un exemple d'une constante de type chaîne est la suivante :

"Il s'agit d'un exemple de chaine constante"

2.5. Enumerated Constants

Les constantes énumérées (*Enumerated Constants*) sont de type entier et sont utilisées pour faire un programme plus facile à suivre. Dans l'exemple suivant, une constante nommée *couleur* stocke les noms de couleurs. Le premier élément de couleur a la valeur 0:

enum couleur {noir, marron, rouge, orange, jaune, vert, bleu, gris, white};

3. Séquences d'échappement

Les séquences d'échappement sont utilisées pour représenter les caractères ASCII non imprimables. Par exemple, la combinaison de caractères « \n » représente le caractère de nouvelle ligne.

4. Les Tableaux

Les tableaux sont utilisés pour stocker des éléments liés dans le même bloc de mémoire. Un tableau est déclaré en spécifiant son type, le nom, et le nombre d'éléments à stocker. Par exemple :

unsigned int Total [5] ;

Dans le langage de programmation mikroC, nous pouvons aussi déclarer des tableaux aux dimensions multiples. Tableaux unidimensionnels sont généralement appelés vecteurs, et des tableaux bidimensionnels sont appelés matrices. Un réseau bidimensionnel est déclaré en spécifiant le type de données de la matrice, le nom de tableau, et la taille de chaque dimension.

Dans l'exemple suivant, réseau bidimensionnel *Q* a deux rangées et deux colonnes, ses éléments diagonaux sont mis à 1, et de ses éléments non diagonaux sont remis à 0 :

unsigned char Q[2][2] = { {1,0}, {0,1} };

5. Les Pointeurs

5.1. Notion de pointeur

Les pointeurs (*Pointers*) sont une partie importante du langage mikroC, car ils occupent les adresses mémoire des autres variables. Les pointeurs sont déclarés de la même manière que d'autres variables, mais avec le caractère « * » en face du nom de variable.

Dans l'exemple suivant, un pointeur de caractère non signé du nom *pnt* est déclaré :

*unsigned char *pnt;*

Quand un nouveau pointeur est créé, son contenu est d'abord indéterminé et il ne tient pas l'adresse d'une variable. Nous pouvons attribuer l'adresse d'une variable à un pointeur à l'aide le « & » :

pnt = &Count;

Maintenant *pnt* affecte l'adresse de variable *Count*. La variable *Count* peut être affectée à une valeur en utilisant le caractère « * » en avant de son pointeur. Par exemple, le *Count* peut être attribuée à 10 à l'aide de son pointeur :

** pnt = 10; // Count = 10*

c'est la même chose que

Count = 10; // Count = 10

ou, la valeur du *Count* peut être copié à la variable *Cnt* en utilisant son pointeur :

*Cnt = *pnt; // Cnt = Count*

5.2. Pointeurs et tableaux

Dans le langage mikroC, le nom d'un tableau est aussi un pointeur de tableau. Ainsi, pour le tableau :

unsigned int Total[10];

Le nom *Total* est également un pointeur de ce tableau, et il contient l'adresse du premier élément de la matrice. Ainsi, les deux énoncés suivants sont égaux :

Total[2] = 0;

et

**(Total + 2) = 0;*

6. Les Structures

Une structure peut être utilisée pour recueillir des éléments connexes, qui sont ensuite traités comme un seul objet. Contrairement à un tableau, une structure peut contenir un mélange de types de données. Par exemple, une structure permet de stocker les données personnelles (nom, prénom, âge, date de naissance, etc.) d'un étudiant.

Une structure est créée en utilisant le mot-clé ***struct***, suivi d'une structure de nom et d'une liste des déclarations de membre. Éventuellement, des variables de même type que la structure peuvent déclarer à l'extrémité de la structure.

L'exemple suivant déclare une structure nommée *Personne* :

```
struct Personne
{
        unsigned char nom[20];
        unsigned char prenom[20];
        unsigned char nationalite[20];
        unsigned char age;
}
```

Nous pouvons attribuer des valeurs aux élements d'une structure en spécifiant le nom de la structure, suivi d'un point «.» et le nom de l'élément.

III. OPERATEURS ET EXPRESSIONS EN MIKROC

Les opérateurs sont appliqués aux variables et d'autres objets dans les expressions pour assurer certaines conditions ou des calculs.

Une expression est un objet syntaxique obtenu en assemblant des constantes, des variables et des opérateurs.

1. Opérateur d'affectation

L'opérateur la plus importante dans un langage de programmation est celle qui consiste à donner une valeur à une variable. Cette opération est désignée par le symbole « = ».

2. Les opérateurs arithmétiques

Opérateur	Nom	Notation
+	addition	x + y
-	soustraction	x - y
*	multiplication	x * y
/	division	x/y
%	modulo	x%y

Les opérateurs +, -, *, fonctionnent comme en arithmétique. Par contre, l'opérateur / (division) se comporte de manière différente selon que les opérandes sont des entiers ou des nombres flottants.

3. Autres opérateurs unaires d'affectation

Opérateur	Equivalent	Notation
++	x = x + 1	x++ ou ++x
--	x = x - 1	x-- ou --x

4. Les autres opérateurs binaires d'affectation

Opérateur	Equivalent	Notation
+=	x = x + y	x += y
-=	x = x - y	x -= y
*=	x = x * y	x *= y
/=	x = x / y	x /= y
%=	x = x % y	x %= y
=	x = x y	x = y
<<=	x = x << y	x <<= y
&=	x = x & y	x &= y
|=	x = x | y	x |= y
^=	x = x ^ y	x ^= y

5. Les opérateurs de comparaison

Opérateur	Nom	Notation
==	test d'égalité	x == y
!=	test de non égalité	x != y
<=	test inférieur ou égal	x <= y
>=	test supérieur ou égal	x >= y
<	test inférieur strict	x < y
>	test supérieur strict	x > y

En mikroC, le résultat d'une comparaison est 1 (!= 0) ou 0 selon que cette comparaison est *vraie* ou *fausse*. Il n'existe pas de type booléen en mikroC : la valeur entière 0 sera considérée comme équivalente à la valeur *faux* et toute valeur différente de 0 équivalente à la valeur *vraie*.

6. Les opérateurs logiques

Une variable booléenne est une variable pouvant prendre la valeur *vrai* ou *faux*. La valeur d'une expression booléenne est, comme le résultat des comparaisons, une valeur entière.

Opérateur	Nom	Notation
&&	ET	x && y
||	OU	x || y
!(unaire)	NON	!x

7. Les opérateurs de manipulation de bits

Opérateur	Nom	Notation
&	ET bit à bit	x & y
|	OU bit à bit	x | y
^	OU exclusif bit à bit	x ^ y
! (unaire)	NON bit à bit	! x
>>	décalage à droite	>> x
<<	décalage à gauche	<< x

8. Opérateur de dimension

Cet opérateur donne l'occupation mémoire (en octets) d'une variable ou d'un type de donné.

Opérateur	Equivalent	Notation
sizeof	opérateur de dimension	sizeof (e)

Exemple :

La valeur de l'expression **sizeof(c)** est 1 si **c** est une variable de type **char**.

IV. LES STRUCTURES DE CONTROLE

Les instructions sont normalement exécutées séquentiellement à partir du début à la fin d'un programme. Nous pouvons utiliser des instructions de contrôle pour modifier ce flux séquentiel normal dans un programme C. Les instructions de contrôle suivantes sont disponibles dans les programmes en mikroC :

- De sélection ***if* et *switch***
- D'itération ou bouclage ***for***
- Modifications inconditionnels d'exécution
- Instructions de sélection ***if* et *switch***

1. Instruction if

Cette instruction conditionnelle permet d'exécuter des instructions de manière sélective en fonction du résultat d'un test. La déclaration du format général de l'instruction ***if*** est la suivante :

```
if (expression)
        instruction1
else
        instruction2
```

Si ***l'expression*** est vraie, ***l'instruction1*** s'exécute sinon, dans le deuxième cas, c'est ***l'istruction2*** qui s'exécute.

Exemple:

```
if (x> 0 && x <10)
{
        Total + = Sum;
        Sum++;
}
else
{
        Total = 0;
        Sum = 0;
}
```

2. Instruction switch

L'instruction ***switch*** est utilisée pour assurer la commutation entre des différentes déclarations si un certain nombre des conditions est vrai ou faux.

La syntaxe de commutation :

```
switch (condition)
{
        case condition1: Instructions1; break;
        case condition2: Instructions2; break;
                . . . . . .
        case conditionN: InstructionsN; break;
        default:              InstructionsN+1;
}
```

Exemple:

```
switch (Cnt)
{
        case 1:    A=1;    break;
        case 10:   B=10;   break ;
        case 100: C=100; break;
        default:    D=1;
}
```

3. Instructions d'itération for, while, do, goto, continue et break

Les instructions d'itération nous permettent d'effectuer des boucles dans un programme, où une partie d'un code doit être répétée un certain nombre de fois. Dans mikroC, l'itération peut être effectuée de quatre façons :

- Utilisation de ***for***
- Utilisation de ***while***
- Utilisation de ***do***
- Utilisation de ***goto***, ***continue*** et ***break***

3.1. Instruction for

La syntaxe d'une instruction ***for*** est :

```
for (expression initiale; expression de condition; expression increment)
{
        Instructions;
}
```

Exemple:

```
for (i = 0; i < 3; i++)
{
        for (j = 0; j < 4; j++)       Sum = Sum + M[i][j];
}
```

3.2. Instruction while

La syntaxe d'une instruction while est la suivante :

```
while (condition)
{
        Instructions;
}
```

Ici, les instructions sont exécutées jusqu'à ce que la condition devienne fausse, ou les instructions sont exécutées de façon répétée aussi longtemps que la condition est vraie. Si la condition est fausse à l'entrée de la boucle, la boucle ne sera pas exécutée et le programme continue de l'extrémité de la boucle *while*. Il est important que la condition change à l'intérieur de la boucle, sinon une boucle sans fin sera formée.

Le code suivant montre comment mettre en place une boucle d'exécuter 10 fois :

```
// Une boucle qui s'exécute 10 fois
k = 0;
while (k <10)
{
        Instructions;
        k++;
}
```

Une boucle sans fin peut également être formée par réglage de la condition qui doit être toujours vrai :

```
// Une boucle sans fin
while (k == k)
{
        Instructions;
}
```

Il est possible d'avoir une déclaration while sans corps. Une telle déclaration est utile, pour par exemple, si nous attendons un port d'entrée pour changer sa valeur.

Voici un exemple où le programme va attendre aussi longtemps que le bit 0 de PORTB (PORTB.0) est au niveau logique 0. Le programme se poursuivra jusqu'au changement à la valeur logique 1 sur les broches du port.

```
while (PORTB.F0 == 0); // Attendre jusqu'a ce que PORTB.0 devient un
        ou
while (PORTB.F0);
```

3.3. Instruction do

Une déclaration **do** est similaire à une déclaration **while** sauf ce que la boucle s'exécute jusqu'à ce que la condition devienne fausse, ou, la boucle s'exécute tant que la condition est vraie. La condition est testée à la fin de la boucle. La syntaxe d'une déclaration **do** est la suivante :

```
do
{
        Instructions;
```

```
} while (condition);
```

La première itération est toujours effectuée si la condition est vraie ou fausse. Il s'agit de la principale différence entre une déclaration **while** et une déclaration **do**.

Le code suivant montre comment mettre en place une boucle d'exécuter 10 fois en utilisant la déclaration **do** :

```
/*Exécution 10 fois */
k = 0;
do
{
        Instructions;
        k++;
} while (k <10);
```

La boucle commence avec k = 0, et la valeur de *k* est incrémenté à l'intérieur de la boucle après chaque itération. À la fin de boucle *k* est testé, et si k n'est pas inférieur à 10, la boucle termine.

Une boucle sans fin peut également être créée si la condition est réglée pour être vrai tout le temps:

```
/* Une boucle sans fin */
do
{
        Instructions;
} while (k == k);
```

3.4. Instructions *goto*, *continue* et break

Une instruction **goto** peut être utilisée pour modifier le flux normal de contrôle dans un programme. Elle provoque le programme à sauter à une étiquette spécifiée.

Une étiquette peut être n'importe quel jeu de caractères alphanumériques commençant par une lettre et se terminant par le caractère « : ».

L'instruction **goto** peut être utilisée conjointement avec une instruction **if**.

L'exemple suivant montre comment mettre en place une boucle pour l'exécuter 10 fois en utilisant **goto** et **if** :

```
/ * Exécution 10 fois * /
k = 0;
Boucle:
        Instructions;
        k++;
if (k < 10) goto Boucle;
```

La boucle commence par l'étiquette **Boucle** et la variable k **= 0**. Les instructions sont exécutées à l'intérieur de la boucle et **k** est incrémenté de 1. La valeur de **k** est alors comparée à 10 et le programme retourne à l'étiquette **Boucle** si **k <10**. Ainsi, la boucle est exécutée 10 fois jusqu'à ce que la condition à la fin devienne fausse. À la fin de la boucle la valeur de **k** est 10.

Les instructions **continue** et **break** peuvent être utilisés à l'intérieur d'itérations pour modifier le flux de commande. Une instruction **continue** est généralement utilisée avec une instruction **if** et provoque le saut de la boucle d'itération.

Voici un exemple qui calcule la somme des nombres de 1 à 10, sauf le nombre 5 :

```
/* Calculer la somme des nombres 1, 2, 3, 4, 6, 7, 8, 9, 10 */
Sum = 0;
i = 1;
for (i = 1; i <= 10; i++)
{
        if (i == 5) continue;     // Passer le numéro 5
        Sum = Sum + i;
}
```

De même, une instruction **break** peut être utilisé pour mettre fin à une boucle à l'intérieur de la boucle.

Dans l'exemple suivant, la somme des nombres de 1 à 5 est calculée, même si les paramètres de la boucle sont mis à parcourir 10 fois :

```
/ * Calculer la somme des nombres 1, 2, 3, 4, 5 * /
Sum = 0;
i = 1;
for (i = 1; i <= 10; i++)
{
        if (i > 5) break; // Stopper la boucle si i > 5
}
```