LE LANGAGE "C" POUR MICROCONTROLEURS

1. ORGANISATION D'UN PROGRAMME C POUR µC.

La saisie d'un programme en « C » répond pratiquement toujours à la même architecture. Le symbole « # » est suivi d'une directive de compilation, le symbole « // » est suivi d'un commentaire. Le texte compris entre le symbole « /* » et « */ » est pris pour un commentaire également.

```
#include <mega16.h>              // Directive de compilation indiquant d'inclure le fichier de définition du µC ATmega16
#include <delay.h>               // Directive de compilation indiquant d'inclure la bibliothèque delay.h

#define DATA_LCD PORTB           // Directive de compilation indiquant des équivalences
#define RS PORTC.0               // exemple : bit RS correspond au bit 0 du PORT C

Unsigned char cap=0xA5;          // Déclaration d'une variable "caractère non signé" avec valeur initiale exprimée en hexa
int wind;                        // Déclaration d'une variable de type « entier » initialisée à 0 par défaut
...
void Wr_LCD(unsigned char data ,unsigned char LCD_RS)
{
RS=LCD_RS;                        //déclaration d'une fonction qui sera appelée une ou plusieurs fois par un autre
…                                 // programme. Les paramètres entre parenthèses sont passés à la fonction. Void
delay_ms(2);                      //signifie que la fonction ne retourne rien. Les prototypes de fonctions doivent
…                                 //être placés avant le programme principal (main)
}

unsigned int read_adc(unsigned char adc_input)
{
ADMUX=adc_input|ADC_VREF_TYPE;    //le paramètre entre parenthèses unsigned char adc_input est passé à la
ADCSRA|=0x40;                     // fonction. La fonction renvoie une valeur de type unsigned int
while ((ADCSRA & 0x10)==0);       //
ADCSRA|=0x10;
return ADCW;                      // return signifie qu'une valeur doit être retournée par la fonction
}

interrupt [TIM1_COMPA] void fonction(void) //sous-programme d'interruption à placer avant le main
{
…
PORTA=~PORTA;                     //void fonction(void) signifie que la fonction ne reçoit et ne renvoie pas
}                                 //de valeur


void main(void)                   // Programme principal
        {
        DDRA=0xFF ;               // initialisation d'un port
        ....
        while (1)                 //Boucle principale (les instructions entre les accolades sont exécutées indéfiniment)
                {
                a= read_adc (0);
                a= a>>2;
                b=(50*a)/255;
                …
                }
        }
```

Remarques : Dans la déclaration de certaines fonctions, on emploie le type "**void**" qui signifie que la fonction ne renvoie ou n'exige aucune valeur.

Chaque ligne d'instruction se termine par un ";".

Le début d'une suite d'instructions est précédé du symbole "{".

La fin d'une suite d'instructions est suivie du symbole "}".

La notation des nombres peut se faire en décimal de façon normale, en hexadécimal avec le préfixe "0x" ou en binaire avec le préfixe « 0b ».

2. LES TYPES DE DONNEES DU LANGAGE "C".

Il existe, dans le langage "C", plusieurs "types" de données classés selon leurs tailles et leurs représentations. On ne détaillera que ceux utilisés dans le cadre des microcontrôleurs.

2.1 TYPES DE BASE.

On en rencontre généralement trois types qui peuvent être signés ou non signés. Dans ce dernier cas la déclaration sera précédée du mot clé "**unsigned**".

TYPE	DEFINITION	TAILLE	DOMAINE NON SIGNE	DOMAINE SIGNE
char	Variable de type caractère utilisée pour les nombres entiers	8 bits	0 à 255	-128 à +127
int	Variable de type nombre entier	16 bits	0 à 65536	-32768 à 32767
Long int	Variable de type nombre entier	32 bits	0 à +4294967295	-2147483647 à +2147483647
float	Variable de type nombre réel	32 bits	$+/- 3,4.10^{-38}$ à $3,4.10^{38}$	X

2.2 TYPES STRUCTURES.

Les types structurés sont en fait une association de plusieurs variables de base de même type. Il en existe deux types :

- Les types **tableau** dont la taille est définie - *ex : **int** tableau**[**10**]** ; tableau de 10 entiers.*
- Les types **pointeurs** dont la taille n'est pas définie - *ex : **char ***chaine ; pointeur de cacactères.*

Les chaines de caractères peuvent être définies par les 2 types. On préfère cependant le pointeur pour sa taille indéfinie.

Remarque :

- Le pointeur est comparable à un registre d'index qui contient non pas une donnée mais l'adresse de cette donnée.

L'adresse d'une variable affectant un pointeur, s'obtient en précédant son nom du symbole "**&**".

- Une chaîne de caractère se termine toujours par le caractère nul "\0".

3. LES OPERATEURS.

3.1 LES OPERATEURS ARITHMETIQUES.

Ces opérateurs permettent d'effectuer les opérations arithmétiques traditionnelles : Addition, soustraction, multiplication et division entière.

OPERATEUR	FONCTION
+	Addition
-	Soustraction
*	Multiplication
/	Division entière
%	Reste de la division entière

3.2 LES OPERATEURS D'AFFECTATION.

L'opérateur indispensable au langage "C" est l'affectation définie principalement par le signe "=". Il permet de charger une variable avec la valeur définie par une constante ou par une autre variable. Il en existe d'autres qui, en plus de l'affectation, effectuent une opération arithmétique.

OPERATEUR	FONCTION	EQUIVALENCE
=	Affectation ordinaire	X=Y
+=	Additionner à _	X+=Y équivalent à X=X+Y
-=	Soustraire à_	X-=Y équivalent à X=X-Y
*=	Multiplier par _	X*=Y équivalent à X=X*Y
/=	Diviser par _	X/=Y équivalent à X=X/Y
%=	Modulo	X%=Y équivalent à X=X%Y
--	Soustraire de 1 (Décrémentation)	X-- équivalent à X=X-1
++	Ajouter 1 (Incrémentation)	X++ équivalent à X=X+1

REMARQUE

Bien que les instructions « variable » ++ et « variable » – produisent les mêmes effets sur une variable que les instructions « variable » +=1 et « variable » -=1, les premières peuvent, selon la performance du compilateur, être exécutées plus rapidement (simple incrémentation dans le premier cas et addition dans le second).

3.3 LES OPERATEURS LOGIQUES.

Ces opérateurs s'adressent uniquement aux opérations de test conditionnel. Le résultat de ces opérations est binaire : "0" ou "1".

OPERATEUR	FONCTION
&&	ET logique
||	OU logique
!	NON logique

3.4 LES OPERATEURS LOGIQUES BIT A BIT.

Ces opérateurs agissent sur des mots binaires. Ils effectuent, entre deux mots, une opération logique sur les bits de même rang.

OPERATEUR	FONCTION
&	ET
|	OU
^	OU exclusif
~	NON
>>	Décalage à droite des bits
<<	Décalage à gauche des bits

3.5 OPERATEURS DE COMPARAISON.

Ces opérateurs renvoient la valeur “0” si la condition vérifiée est fausse, sinon ils renvoient “1”.

OPERATEUR	FONCTION
==	Egale à
!=	Différent de
>	Supérieur à
<	Inférieur à
>=	Supérieur ou égal à
<=	Inférieur ou égal

4. LES STRUCTURES REPETITIVES.

Le langage “C” possède des instructions permettant de répéter plusieurs fois une même séquence en fonction de certaines conditions.

4.1 STRUCTURE “WHILE” : TANT QUE … FAIRE …

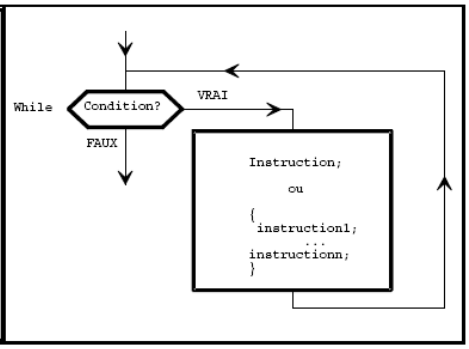
Avec ce type d’instruction le nombre de répétitions n’est pas défini et dépend du résultat du test effectué sur la condition. Si cette dernière n’est jamais vérifiée, la séquence n’est pas exécutée.

```
Syntaxe:

while (condition) instruction;

ou

while (condition)
{
     instruction1;
          ...
     instructionn;
}
```




```
while (THETA==0)
      {séquence d’instructions
      ...
      }
```

La structure ci-dessus répète la suite d’instructions comprise entre crochets tant que la variable entière “ THETA ” est égale à 0.

4.2 STRUCTURE “DO … WHILE” : FAIRE … TANT QUE…

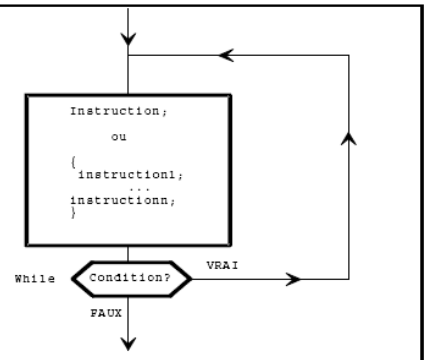
Cette structure ressemble fortement à la précédente à la seule différence que la séquence à répéter est au moins exécutée une fois même si la condition n’est jamais vérifiée.

```
      do
      {
      ...
      }
while (THETA==0);
```

```
Syntaxe:
do instruction;
while (condition) ;

ou

do
{
     instruction1;
          ...
     instructionn;
}
while (condition);
```



4.3 STRUCTURE "FOR" : POUR <VARIABLE> ALLANT DE <VALEUR INITIALE> A <VALEUR FINALE> FAIRE…

Cette instruction permet de répéter, un nombre de fois déterminé, une même séquence.

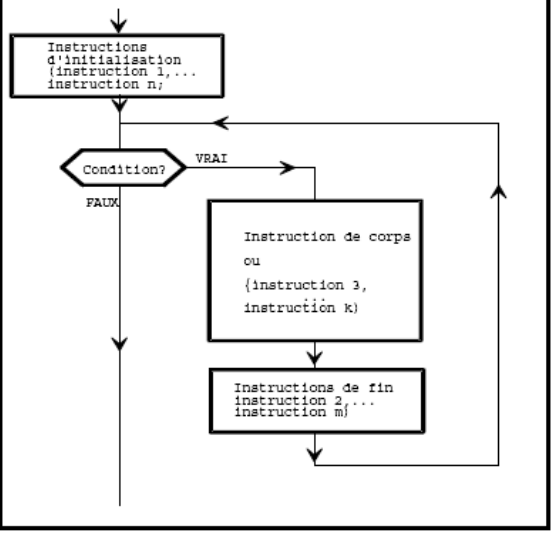
```
for (i=0;i<5;i++)
      {
      ...
      }
```

La structure précédente répète 5 fois la suite d'instructions comprise entre crochets. La variable "i" prendra les valeurs successives de : 0, 1, 2, 3 et 4.

```
Syntaxe:
for
(instr_1,...,instr_n d'init;
   condition ;
instr_2,...,instr_m de fin)
 instruction de corps;

ou

for
(instr_1,...,instr_n d'init;
   condition ;
instr_2,...,instr_m de fin)
{
         instruction_3;
           ...
         instruction_k;
}
```



5. LES STRUCTURES ALTERNATIVES.

Ces structures permettent d'exécuter des séquences différentes en fonction de certaines conditions.

5.1 STRUCTURE "IF … ELSE" : SI <CONDITION> FAIRE … SINON FAIRE …

Avec cette structure on peut réaliser deux séquences différentes en fonction du résultat du test sur une condition.

Syntaxe:
if (condition) instruction;
ou :
if (condition)
{
instruction1;
…
instructionn;
}

Ou encore:

if (condition) instruction1; else instruction2;

ou:
if (condition)
{
instruction1;
...
}
else
{
instruction2;
...
}
Exemple:

```
if (a<b) c=b-a;
else c=a-b;
```

La structure précédente affecte la valeur “b-a” à “c” si “a” est inférieur à “b” sinon “c” est affecté par la valeur “a-b”.

5.2 STRUCTURE “SWITCH … CASE”.

Cette structure remplace une suite de “if ... else if ...else” et permet de réaliser différentes séquences appropriées à la valeur de la variable testée. Si aucune valeur n'est trouvée, alors ce sont les instructions qui suivent **default** qui sont exécutées .
Syntaxe:
switch (identificateur)
{
case valeur1 :
instruction_1 ; ou {instructions_1...} ;
break;
case valeuurm :
instruction_2 ; ou {instructions_2...} ;
break;
case valeurk:
instruction_j ; ou {instructions_j...} ;
break;
default :
instruction_i ; ou {instructions_i...} ;
break;
}

```
switch (a)
      {
      case ‘1’ : b=16;
      case ‘2’ : b=8;
      case ‘3’ : b=4;
      case ‘4’ : b=2;
      }
```

Dans la structure précédente “b=16” si “a=1”, “b=8” si “a=2” etc.

6. LES FONCTIONS.

Afin de réduire la taille du programme il est souvent préférable de définir sous forme de fonction une même suite d’instructions appelée plus d’une fois dans le programme.

La fonction principale d’un programme “C” est définie grâce au mot clé “**main**”.

Les fonctions du langage “C” peuvent renvoyer des valeurs de même qu’elles peuvent prendre en compte des arguments provenant de la procédure d’appel. S’il n’y a pas de renvoi ou aucun argument, on saisit le mot clé “**void**” en remplacement.

La valeur renvoyée est définie après le mot clé “**return**”.

Lorsque l'on veut, dans une fonction, modifier une variable passée en argument il est obligatoire d'utiliser un pointeur.

Une fonction doit toujours être définie avant sa procédure d'appel. Dans le cas contraire une simple déclaration doit être faite dans l'en tête du programme.

Exemples :

1) Fonction faisant appel au convertisseur analogique numérique

```
unsigned int read_adc(unsigned char adc_input)
{
ADMUX=adc_input|ADC_VREF_TYPE;    //fixe la voie d'entrée
ADCSRA|=0x40;                     // démarre la  conversion
while ((ADCSRA & 0x10)==0);       // attente fin conversion
ADCSRA|=0x10;
return ADCW;                      // renvoie le résultat
}
void main(void);                  // Programme principal
        {
        ...
        While(1)                  // boucle principale
                {
                a= read_adc (0);  //appel de la fonction
                }
```

7. FONCTIONS PREDEFINIES.

Il existe dans tous les compilateurs "C" des bibliothèques de fonctions prédéfinies. La plus utilisée parmi elles est <**stdio.h**> qui est propre aux organes d'entrées / sorties standards. Dans le cas des ordinateurs ces organes sont le clavier et l'écran. Dans le cas d'un microcontrôleur ces organes sont généralement les interfaces séries du composant (interface RS232C).

Dans cette bibliothèque on trouve les fonctions suivantes :

Printf() : écriture formatée de données.
scanf() : lecture formatée de données.
Putchar() : écriture d'un caractère.
getchar() : lecture d'un caractère.

8. LES POINTEURS.

Ce sont des **variables** contenant **l'adresse** d'autres variables. Ces variables sont déclarées en rajoutant le signe « ***** » devant.
Exemple pour le pointeur a :
unsigned int *a ;
Pour pouvoir accéder à **la valeur d'un pointeur** il suffit de rajouter une étoile ***** devant ***l'identificateur de pointeur***.
Exemples : *a=c; // le contenu de c est stocké à l'adresse située dans a
d=*a; // d prend la valeur du contenu pointé par a

8.1 L'opérateur d'adresse &.

Il permet de connaître l'adresse de n'importe quelle variable ou constante.
Exemple : b=&c ; // b prend la valeur de l'adresse de la variable c.

```
c=30;
a=0x160;
b=&tab[2];      //b prend la valeur de l'adresse de la variable tab[2]
 *a=c;         // le contenu de c est stocké à l'adresse située dans a (le contenu de l'adresse 0x160 prendra la valeur 30)
d=*a;           // ici d prendra la valeur du contenu de l'adresse 0x160 soit 30
```