LE LANGAGE "C" ADAPTÉ AU MICROCONTRÔLEURS

1. PRÉSENTATION.

Le langage "C" a fait son apparition en 1972 pour le développement du système d'exploitation Unix. Il est devenu un standard de la norme ANSI en 1983. 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.

2. ARCHITECTURE D'UN PROGRAMME C POUR µC.

La saisie d'un programme en "C" répond pratiquement toujours à la même architecture. On peut noter que le symbole "#" est suivi d'une directive de compilation, le symbole "//" est suivi d'un commentaire.

```
#include <sdtio.h>          // Directive de compilation indiquant d'inclure la bibliothèque E/S standard
#include <reg_uc.h>         // Directive de compilation indiquant d'inclure la bibliothèque spécifique au µC

#define clear=0x00          // Directive de compilation indiquant des équivalences
...

char val1=0xA5;             // Déclaration d'une variable "caractère" avec valeur initiale
int val2;                   // Déclaration d'une variable "nombre entier"
...

void tempo(char temps)
        {
        ...
        }
                            // Fonctions et procédures appelées plusieurs fois dans le programme principal
int bintobcd(char bin)
        {
        ...
        return ...;
        }

void main(void)             // Programme principal
        {
        DDRBA=0xFF          // initialisation et configuration
        ....
        while (1)           // Boucle principale
                {
                ...
                tempo(100);
                ...
                val2=bintobcd(val1);
                ...
                }
        }

void nmi(void)interrupt 0                   // Sous programme d'interruption
        {
        ...
```

```
}
```

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 "}".

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

3. LES TYPES DE DONNÉES 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.

3.1 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	DÉFINITION	TAILLE	DOMAINE NON SIGNÉ	DOMAINE SIGNÉ
char	Variable de type caractère ascii mais 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
float	Variable de type nombre réel	32 bits	$+/- 3,4.10^{-38}$ à $3,4.10^{38}$	X

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

3.2 TYPES STRUCTURÉS.

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 données.
- 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".

4. LES OPÉRATEURS.

4.1 LES OPÉRATEURS ARITHMÉTIQUES.

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

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

4.2 LES OPÉRATEURS D'AFFECTATION.

L'opérateur indispensable au langage "C" est l'affectation défini 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, effectue une opération arithmétique.

OPÉRATEUR	FONCTION	EQUIVALENCE
=	Affectation ordinaire	X=Y
+=	Additionner de _	X+=Y équivalent à X=X+Y
-=	Soustraire de _	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

4.3 LES OPÉRATEURS LOGIQUES.

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

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

4.4 LES OPÉRATEURS LOGIQUES BIT À BIT.

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

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

4.5 OPÉRATEURS DE COMPARAISON.

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

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

5. LES STRUCTURES RÉPÉTITIVES.

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

5.1 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.

```
while (int x!=0)
      {
      ...
      }
```

La structure précédente répète la suite d'instruction comprises entre crochets tant que la variable entière "x" est différente de 0.

5.2 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 exécuter une fois même si la condition n'est jamais vérifiée.

```
do {
...
}while (int x!=0);
```

5.3 STRUCTURE "FOR" : POUR <VARIABLE> ALLANT DE <VALEUR INITIALE> À <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'instruction comprise entre crochets. La variable "i" prendra les valeurs successives de : 0, 1, 2, 3 et 4.

1. LES STRUCTURES ALTERNATIVES.

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

6.1 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.

```
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".

6.2 STRUCTURE "SWITCH ... CASE".

Cette structure remplace une suite de "if ... else if ...else" et permet une de réaliser différentes séquences appropriées à la valeur de la variable testée.

```
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=4" etc.

2. 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 principal d'un programme "C" est définie grâce au mot clé "**main**".

Les fonctions du langage "C" peuvent renvoyées des valeurs de même qu'elles peuvent prendre en compte des arguments provenant de la procédure d'appel. Si il n'y 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 permettant de positionner à 1 le bit de rang "n" d'un registre 8 bits "reg".

```
void setbit(char *reg,char n)
     {
     char m=1<<n;                  // variable m = 1 décalé de n fois à gauche
     *reg=*reg | m;                // opération bit à bit OU entre *reg et m
     }

void main(void);                   // Programme principal
     {
     ...
     While(1)                      // boucle principale
           {
           ...
           setbit(&DRB,5);         // mise à 1 du bit 5 de DRB
           ...
           }
```

Afin qu'un résultat soit renvoyé, nous sommes obligés de passer par un pointeur en ce qui concerne "reg".

2) Fonction permettant de tester le bit de rang **"**n" d'un registre "reg" en renvoyant la valeur "1" si le bit est à l'état haut ou "0" dans le cas contraire.

```
char testbit(char reg,char n)
     {
     char m=1<<n;                  // variable m = 1 décalé de n fois à gauche
```

```
     return (reg & m);              // renvoi opération bit à bit ET entre reg et
m
     }

void main(void);                    // Programme principal
     {
     ...
     While(1)                       // boucle principale
           {
           ...
           if (testbit(DRA,0)) setbit(DRB,5);  //mise à 1 du bit 5 de DRB si
DRA0=1
           ...
           }
```

3. FONCTION PRÉDÉFINIE.

Il existe dans tous les compilateurs "C" des bibliothèques de fonctions prédéfinies. La plus utilisée parmi elles est <**stdio.h**> qui est propre au 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.

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.

4. DÉCLARATION DE VARIABLES ET DE CONSTANTES SPÉCIFIQUES AU MICROCONTRÔLEUR.

La déclaration de variables correspond au fait de réserver un bloc, plus ou moins grand, dans la mémoire de donnée. Comme la taille de cette espace est limitée en ce qui concerne les microcontrôleurs, il est important de pouvoir contrôler parfaitement certains paramètres comme par exemple l'adresse.

Pour cela le cross - compilateur dispose de mots clés supplémentaires permettant de définir le type ainsi que l'adresse du bloc de mémoire utilisé.

MOT CLÉ	SIGNIFICATION
at	Permet de définir l'adresse de la variable
code	Permet de réservé un bloc dans la mémoire programme
data	Permet de réservé un bloc dans la mémoire de données
register	Permet de réservé un bloc dans les registres de la CPU si cela est possible

5. EXEMPLE.

Le programme suivant réalise une bascule bistable. Il permet de mettre à "1" le bit 5 du port B si le bit 1du port A est à "1" . La remise à "0" s'effectue en positionnant à "1" le bit 1 du port A.

```
#include <st6265b.h>

char testbit(const char reg,const char n)
     {
     char m=1<<n;              // variable m = 1   décalé de n fois à gauche
     return (reg & m);         // renvoi opération bit à bit ET entre reg et m
     }

void setbit(char *reg,const char n)
     {
     char m=1<<n;              // variable m = 1   décalé de n fois à gauche
```

```
      *reg=*reg | m;            // opération bit à bit OU entre reg et m
      }

void resetbit(char *reg,const char n)
      {
      char m=1<<n;              // variable m = 1   décalé de n fois à gauche
      *reg=*reg & ~m;           // opération bit à bit OU entre reg et le complément de m
      }

void main(void)                 // Programme principal
      {
      DDRA=0x00;                // Port A en entrée
      ORA=0x00;                 // Sans pull up
      DRA=0x00;                 // Ni interruption

      DDRB=0xFF;                // Port B en sortie
      ORB=0xFF;                 // Symétrique

      while (1)                 // boucle principale
            {
            if (testbit(DRA,0)) setbit(&DRB,5);    // mise à 1 du bit 5 de DRB si DRA0=1
            if (testbit(DRA,1)) resetbit(&DRB,5);  // mise à 0 du bit 5 de DRB si DRA1=1
            }
      }
```