

A2I12 - TP 1 : Compilateur C sur micro-contrôleur**I) Les chenillards****1°) Petite arithmétique des bits**

On rappelle qu'en C le "ou booléen" se fait par `||`, le "et booléen" par `&&`. Nous aurons besoin du "ou bit à bit" `|` et du "et bit à bit" `&`. Soit le contenu d'un registre B sur huit bits,

<i>b7</i>	<i>b6</i>	<i>b5</i>	<i>b4</i>	<i>b3</i>	<i>b2</i>	<i>b1</i>	<i>b0</i>
<i>1</i>	<i>1</i>	<i>1</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>1</i>	<i>1</i>

vous désirez mettre à 1 le bit b2 sans changer les autres bits, comment faites-vous ?

Vous désirez mettre à 0 le bit b6 sans changer les autres bits comment faites-vous ?

2°) Exemple

On vous donne un programme C qui fait clignoter une led (poids faible mais à gauche) sur le port B.

```
#include <hcl1.h>
#undef PORTB
#define PORTB *(unsigned char *) (_IO_BASE + 0x60)
void wait(int cnt);
void main(void) {
    int x;
    PORTB=0;
    x= 30000;
    for(;;) {
        PORTB = 0x01;
        wait(x);
        PORTB = 0x00;
        wait(x);
    }
}
void wait(int cnt){
    for (;cnt>0; cnt--);
}
```

Écrire ce programme pour l'essayer, le compiler, le charger et l'exécuter. Modifier-le pour faire clignoter la led juste à côté.

3°) Exercices

- Écrire un chenillard simple : une led se déplaçant de la droite vers la gauche et en utilisant le même type de temporisation que dans le programme exemple.
- Écrire un chenillard double : un chenillard de la gauche vers la droite et simultanément un autre de la droite vers la gauche.
- Écrire un chenillard à entassement (1 led se déplaçant de la droite vers la gauche et s'accumulant à gauche).

II) L'afficheur sept segments

b7	b6	b5	b4	b3	b2	b1	b0
DP	g	f	e	d	c	b	a

B6=0 => allume g ...

Le registre M sert à sélectionner l'afficheur M=1 celui de gauche... M=4 celui de droite.

1°) Exemple

```
#include <hc11.h>
#undef PORTB
#define PORTB *(unsigned char *) (_IO_BASE + 0x60)
#define PORTM *(unsigned char *) (_IO_BASE + 0x62)

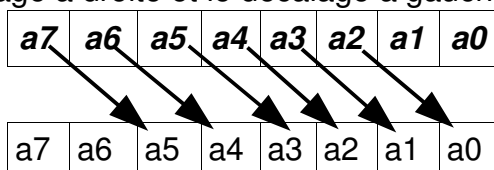
void wait(int cnt);
void main(void) {
    int x;
    x= 30000;
    for(;;) {
        PORTM = 0x01; /* afficheur gauche */
        PORTB = 0x00;
        wait(x);
        PORTM = 0x02; /* afficheur centre */
        wait(x);
        PORTM = 0x04; /* afficheur droite */
        wait(x);
    }
}
void wait(int cnt){
    for (;cnt>0; cnt--);
}
```

Écrire ce programme pour l'essayer, le compiler, le charger et l'exécuter. Modifier-le pour afficher un "A" au lieu d'un "8".

2°) Exercices

- Écrire un sous-programme qui affiche sur l'afficheur de droite une valeur hexadécimale comprise entre "0" et "F". La valeur à afficher sera prise comme un type char passé en paramètre. On testera bien sûr que sa valeur ne dépasse pas 15. Écrire le programme entier qui permet le test.
- Écrire un sous-programme qui affiche sur l'afficheur de droite et celui du milieu, une valeur hexadécimale comprise entre "00" et "FF". La valeur à afficher sera prise comme un type char passé en paramètre. Écrire le programme entier qui permet le test.

Indications : les opérateurs de décalage sont >> et << pour respectivement le décalage à droite et le décalage à gauche. a=a>>2 signifie :



- Modifier le programme précédant pour qu'il affiche en continu le contenu d'un compteur qui compte sans arrêt (avec une temporisation pour que l'on puisse lire l'affichage).

- Écrire un sous-programme qui affiche sur l'afficheur de droite et celui du milieu et celui de gauche une valeur décimale comprise entre "000" et "999". La valeur à afficher sera prise comme un type int passé en paramètre. Écrire le programme entier qui permet le test.
- Mettre au point un affichage de secondes en utilisant une temporisation adéquate sans utiliser de timer (sera vu en 2° année).

A2I12 - TP 2 : Compilateur C sur micro-contrôleur

I) L'affichage des entiers

Vous disposez d'une librairie de bas niveau pour utiliser l'afficheur LCD. Elle s'appelle pilotlcd.c et si vous voulez l'utiliser cela nécessitera un #include <pilotlcd.h> dans vos "include" habituels. On l'initialise avec un lcdinit();

Un programme contenant un printf comme ci-dessous est volumineux, il suffit de constater un temps de chargement un peu long :

```
#include <hc11.h>
#include <pilotlcd.h>
#include <stdio.h>
void main(void) {
    lcdinit();
    printf("Bonjour a tous");
}
```

On compile en allant dans otpion, nomfichier.mak et en ajoutant le fichier pilotlcd.c qui se trouve dans c:\cc11\libsrc.

Remarque : plutôt que changer le fichier make, il est possible d'écrire

```
#include "c:\cc11\libsrc\pilotlcd.c"
```

en fin de fichier, après le main. Cela apporte un avantage, si l'afficheur ne fonctionne pas correctement, la commande "d lcdtamp" affiche alors le contenu du buffer d'écran LCD.

On utilisera ainsi plutôt la fonction int putchar(char c) qui affiche le caractère c en décalant ce qui est déjà affiché sur l'écran :

```
#include <hc11.h>
#include <pilotlcd.h>
#include <stdio.h>
void main(void) {
    char i,toprint[]="Bonjour a tous";
    lcdinit();
    for (i=0;i<16;i++) putchar(toprint[i]);
}
```

Exercice 1

1°) Écrire un sous-programme qui affiche en clair une valeur numérique passée dans un unsigned char (autrement dit entre 0 et 255). Écrire le programme complet qui

permette le test de ce sous-programme.

2°) Écrire un sous-programme qui affiche en clair une valeur numérique passée dans un `signed char` (autrement dit entre -128 et +127). Écrire le programme qui permette le test de ce sous-programme.

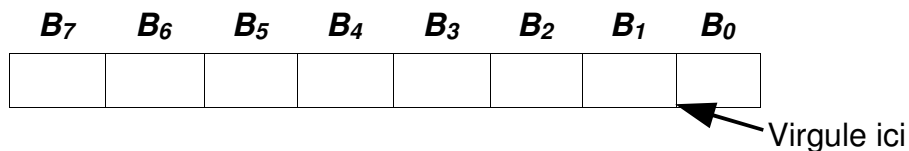
Indications : la transformation de valeurs numériques en codes ASCII correspondant peut facilement se faire avec un tableau du type

```
char conversion[]={'0','1','2','3','4','5','6','7','8','9'};
ou même
char conversion[]="0123456789";
```

II) L'affichage des nombres à virgule

1°) Le format virgule fixe

On s'intéresse à un octet B avec la virgule entre B_1 et B_0 comme indiqué sur le dessin :



Si l'on ne considère que des nombres positifs, quelles sont les valeurs possibles de la partie entière (avant la virgule) ?

Quelles sont aussi les possibilités après la virgule.

Exercice 2

Écrire un sous-programme qui affiche en clair une valeur numérique passée dans un `unsigned char` en format à virgule fixe. Écrire le programme qui permette le test de ce sous-programme. Par exemple $B=3$ affichera 1,5 et $B=7$ affichera 3,5 est-on d'accord ?

2°) L'affichage de température d'un DS1620

Le composant DS1620 est un thermomètre intégré qui s'interface en général à un micro-contrôleur par une liaison série. Les données de température sont sur 9 bits entre -55° et $+125^\circ$ par incréments de $0,5^\circ$ conformément à la documentation du composant :

TEMPERATURE/DATA RELATIONSHIPS

Table 1

<i>TEMP</i>	<i>DIGITAL OUTPUT (Binary)</i>	<i>DIGITAL OUTPUT (Hex)</i>
$+125^\circ$	$(011111010)_2$	00FAh
$+25^\circ$	$(000110010)_2$	0032h
$+1/2^\circ$	$(000000001)_2$	0001h
0°	$(000000000)_2$	0000h
$-1/2^\circ$	$(111111111)_2$	01FFh

<i>TEMP</i>	<i>DIGITAL OUTPUT (Binary)</i>	<i>DIGITAL OUTPUT (Hex)</i>
-25°	(111001110) ₂	01CEh
-55°	(110010010) ₂	0192h

Exercice 3

Écrire un sous-programme qui affiche en clair une valeur numérique passée dans un int en format à virgule fixe sur 9 bits (voir DS1620). Écrire le programme qui permette le test de ce sous-programme.

Indication : On vous donne -25 en complément à deux sur 9 bits, il vous faut le calculer en complément à deux sur 16 bits pour voir le rapport entre les deux, en particulier ce qu'il faut faire des huit bits de poids fort.

A2I12 - TP 3 : Interfaçage C - Assembleur

I) Introduction

Il y a deux aspects essentiels à l'interfaçage du langage C avec de l'assembleur :

- on écrit une partie en C et une partie en assembleur (difficile)
- on écrit du code assembleur dans le code C.

Dans ce TP, on va s'intéresser essentiellement au deuxième aspect qui est bien plus facile.

Directive ASM : la directive asm permet d'introduire directement des instructions assembleur dans un programme C.

Par exemple l'instruction assembleur dans le programme ci-dessous met la valeur 30h dans A et 50h dans B (on peut utiliser l'une ou l'autre des méthodes)

<i>1ere méthode</i>	<i>2eme méthode</i>
asm("ldaa #\$30 "); //accA<-30h asm("ldab #\$50 "); //accB<-50h	asm("ldaa #\$30\n" //accA<-30h "ldab #\$50 "); //accB<-50h

Les lignes écrites en assembleur sont exclues d'optimisation. Le compilateur utilise le registre IX pour adresser les paramètres et les variables locales des sous-programmes ou fonctions. Les fichiers générés (*.lis, *.lst, *.s) permettent de trouver où se trouvent ces données. Par exemple pour le début du sous-programme C suivant :

```
int toto(int p1, int p2)
{ int i,j;
  ... }
```

on trouvera après compilation en assembleur :

```
; IX -> 0,x
; j -> 2,x
; i -> 4,x
; p2 -> 12,x
; p1 -> 8,x
_toto::
```

Un code écrit en assembleur peut donc accéder aux variables d'un sous-programme C. Il est alors possible de manipuler à loisir ces variables. La technique est intéressante puisqu'elle permet d'écrire un bout de code assembleur très rapide et très court (par rapport à l'équivalent C).

Une bonne technique de programmation consiste donc à écrire un programme en C et de le tester. Cette phase doit permettre de détecter les portions "lentes" du programme. La réécriture en assembleur de ces portions permet alors d'obtenir un code optimal (dans le sens compromis entre le temps d'écriture et vitesse d'exécution). Cette méthode qui consiste à faire cohabiter des portions de code d'origine différentes est applicable bien évidemment dans d'autres cas (ex : code MATLAB qui appelle une routine en C).

II) Première approche : recopie de char

Le premier travail à faire consiste à faire une recopie du contenu d'une variable vers une autre.

```
char recopie(char nb) {
    char temp ;
    ???
    return(temp);
}
```

- Insérer la fonction ci-dessus dans un code C : écrire la fonction main() qui appelle cette fonction recopie(). En reprenant le travail de la séance précédente, écrire une fonction afficheChar(char) qui affiche sur le LCD la valeur d'un char passé en paramètre.
- Après compilation consulter le fichier *.lis et chercher les lignes qui correspondent à la fonction recopie() pour identifier la position de nb et temp.
- Écrire le code assembleur équivalent à temp=nb (à la place des "?"). Ne pas oublier que les variables sont de type char (un octet)
- Compiler et tester le programme. Vérifier qu'en affichant la variable renvoyée par la fonction recopie(), on obtient la bonne valeur à l'affichage.

III) Sous programme de calcul en assembleur (avec des char)

Écrire une fonction qui effectue la somme de deux char passés en paramètres :

```
char somme(char nb1, char nb2) {
    char temp ;
    ???
    return(temp);
}
```

Insérer la fonction ci-dessus dans un code C : écrire la fonction main() qui appelle cette fonction somme(). En reprenant le travail de la séance précédente, utiliser la fonction afficheChar(char) qui affiche sur le LCD la valeur d'un char passé en paramètre pour tester votre ensemble.

IV) Sous-programme de calcul en assembleur (avec des int)

La recherche de la position en mémoire des paramètres dans les fichiers *.lis peut être fastidieuse. On peut également adresser les variables locales et les paramètres dans une directive asm par leur noms %<nom de la variable> :

```
asm( "ldd %p1\n"
     " std %i");
```

Écrire une fonction qui effectue la somme de deux int passés en paramètres.

V) Le format virgule flottant

Le format d'un nombre flottant est normalisé par la norme ANSI IEEE standard 754 :



dont la valeur est $(-1)^S \times 1, M_{22} \dots M_0 \times 2^{E-127}$

On désire afficher une valeur donnée par un capteur de température pouvant varier entre +50° et -50°C avec une partie décimale. Cette valeur se trouve dans un float. Pour faciliter son affichage on vous demande d'envisager de transformer cette température dans un format à virgule fixe sur un octet. Étant donné que la partie entière varie entre 0 et 50 combien de bits vous faut-il pour coder celle-ci ? Combien vous en reste-t-il pour la partie décimale (après la virgule) ?

On vous donne le squelette d'un programme C permettant de manipuler le format flottant pour un affichage. On vous demande de l'écrire, de vérifier à l'aide du fichier `lst` le positionnement des variables et surtout d'écrire une partie en assembleur permettant de réaliser l'instruction C : `p=&valeur`; L'objectif de cette instruction est de pouvoir manipuler un flottant (valeur) sur 32 bits comme un tableau de 4 octets.

```
/* affichage flottant */
#include <stdio.h>
#include <hcl1.h>
#include <pilotlcd.h>
void main(void)
{
    float valeur; // valeur flottante valeur -> 27,x
    char i, afficher[16], conversion[]="0123456789";
    unsigned char *p, exposant, mantisse; // p -> 31,x
    short v, aff;
    lcdinit();
    valeur = 25.25;
    //p=&valeur; ce compilateur ne veut pas !!!
}
```

Ecrire ici la partie assembleur qui permette de remplacer `p=&valeur` que ce compilateur ne veut pas faire !

```
    exposant=(p[0]<<1)+((p[1]&0x80)>>7);
    mantisse=p[1]<<1 + ((p[2]&0x80)>>7);
    // conversion virgule fixe
    v = mantisse | 0x0100;
    if (exposant-127 >0)
        v = (v << (exposant-127))>>6;
    else
        v = (v>>(6-exposant +127));
    // affichage de la virgule fixe
    aff = v>>2;
    if (valeur < 50 && valeur > -50) {
        afficher[0]=conversion[aff/10];
        afficher[1]=conversion[aff%10];
        afficher[2]=',';
        aff = v & 0x03;
        switch (aff) {
            case 0 : afficher[3]='0';afficher[4]='0';break;
            case 1 : afficher[3]='2';afficher[4]='5';break;
            case 2 : afficher[3]='5';afficher[4]='0';break;
            case 3 : afficher[3]='7';afficher[4]='5';break;
        }
    } else { //msg erreur : !Err!
```

```
    afficher[0]='!';afficher[1]='E';afficher[2]='r';  
    afficher[3]='r';afficher[4]='!';  
}  
for (i=0;i<5;i++) putchar(afficher[i]);  
}  
#include "c:\cc11\libsrc\pilotlcd.c"
```

Indication : on a réalisé ce travail en 9 instructions en sauvegardant les valeurs contenues dans X (absolument obligatoire) dans B et dans Y.

Correction partielle du TP3

Les corrections sont données car ce TP n'est jamais réalisé par les étudiants. En tout cas ceux qui le font sont ceux qui cherchent par eux même.

Exercice recopie :

```
#include <hc11.h>
#include <stdio.h>
#include <pilotlcd.h>
#undef PORTB
#define PORTB *(unsigned char*) (_IO_BASE + 0x60)
char recopie (char c);
void affichechar(char x);

void main (void) {
    char nb='z',val;
    val=recopie(nb);
    lcdinit();
    putchar(' ');
    affichechar(nb);
}
char recopie (char c){
    char temp;
    asm(" ldaa 7,x "); // accA <-c
    asm(" staa 3,x "); // temp <- accA
    return (temp);
}
void affichechar(char x){
    char cent, diz, unit;
    cent = x/100;
    diz = (x/10)%10;
    unit = c%10;
    putchar(cent+0x30); // 0x30 code ascii de '0'
    putchar(diz+0x30);
    putchar(unit+0x30);
}
}
```

Exercice calcul somme de char :

```
#include<hc11.h>
#include<pilotlcd.h>
char somme(char nba,char nbb) {
    char temp;
    asm("ldaa 7,x");//nba- >accA
    asm("ldab 11,x");//nbb -> accB
    asm("aba"); // accA <- accA + accB
    asm("staa 3,x");//accA -> temp
    return temp;
}
void affichechar(char cara) {
    char cent, diz, unit;
    cent = x/100;
    diz = (x/10)%10;
    unit = c%10;
    putchar(cent+0x30); // 0x30 code ascii de '0'
    putchar(diz+0x30);
    putchar(unit+0x30);
}
void main(void) {
```

```

char nb1=1,nb2=4,res;
lcdinit();
res=somme(nb1,nb2);
putchar(' ');
affichechar(res);
}

```

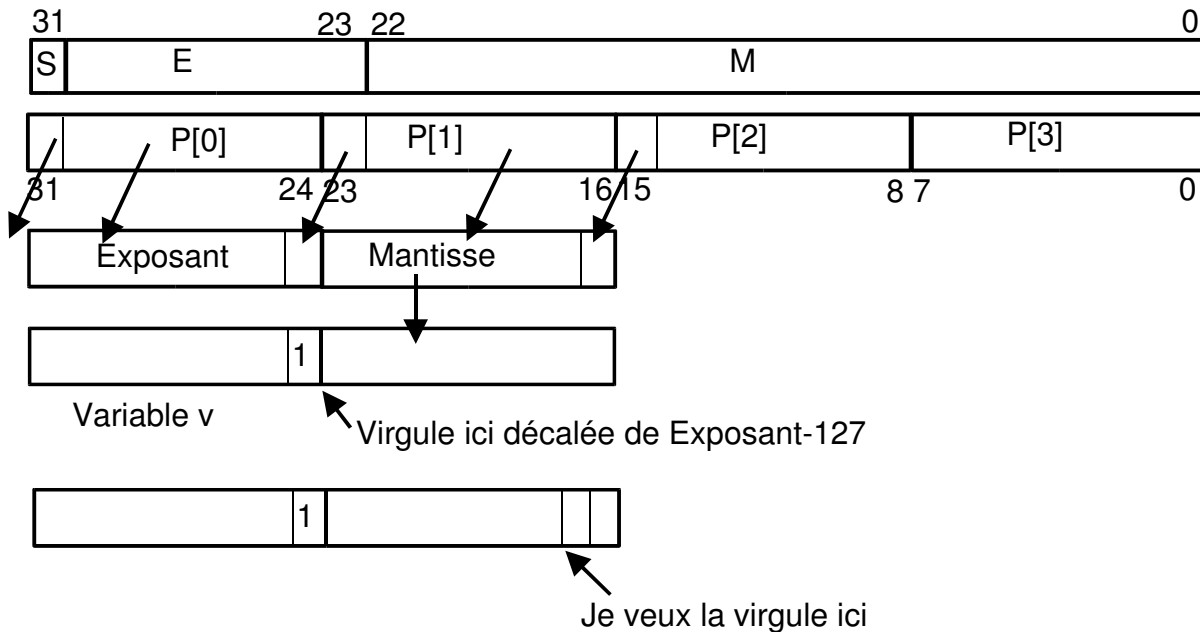
V) Format virgule flottante -> virgule fixe

On donne le programme testé pour 69HC11 pour la norme IEEE 754 : qui permet de convertir un float en format virgule flottante sur un octet avec deux bits après la virgule. Donc seules les valeurs .00, .25, .50 et .75 sont possibles après la virgule.

```

/* affichage flottant */
#include <stdio.h>
#include <hc11.h>
#include <pilotlcd.h>
void main(void)
{
    float valeur; // valeur flottante valeur -> 27,x
    char i,afficher[16],conversion[]="0123456789";
    unsigned char *p,exposant,mantisse; // p -> 31,x
    short v,aff;
    lcdinit();
    valeur = 25.25;
    //p=&valeur; ce compilateur ne veut pas !!!
    asm(" pshy"); // pour conserver Y
    asm(" pshx");
    asm(" puly"); // Y<-X
    asm(" pshb"); // B -> pile pour conserver B
    asm(" ldab #27"); //B <- 27
    asm(" aby"); // Y <- Y+B=Y+27 : adresse de valeur
    asm(" pulb"); //pile -> B pour restaurer B
    asm(" sty %p"); //y -> 31,X c'est a dire p
    asm(" puly"); // pour restaurer Y
    exposant=(p[0]<<1)+((p[1]&0x80)>>7);
    mantisse=p[1]<<1 + ((p[2]&0x80)>>7);
    // conversion virgule fixe
    v = mantisse | 0x0100;
    if (exposant-127 >0)
        v= (v << (exposant-127))>>6;
    else
        v = (v>>(6-exposant +127));
    // affichage de la virgule fixe
    aff = v>>2;
    if (valeur < 50 && valeur > -50) {
        afficher[0]=conversion[aff/10];
        afficher[1]=conversion[aff%10];
        afficher[2]='.';
        aff = v & 0x03;
        switch (aff) {
            case 0 : afficher[3]='0';afficher[4]='0';break;
            case 1 : afficher[3]='2';afficher[4]='5';break;
            case 2 : afficher[3]='5';afficher[4]='0';break;
            case 3 : afficher[3]='7';afficher[4]='5';break;
        }
    } else { //msg erreur : !Err!
        afficher[0]='!';afficher[1]='E';afficher[2]='r';
        afficher[3]='r';afficher[4]='!';
    }
    for (i=0;i<5;i++) putchar(afficher[i]);
}
#include "c:\cc11\libsrc\pilotlcd.c"

```



Quelques essais sous LINUX :

```
main() {
    float r,valeur;
    register short v;
    unsigned char *p,i,exposant,mantisse;
    p=&r; /**** produira normalement un warning***/
    r=16.75; // donne 16.75 par reconstruction
    // recherche exposant (1 octet) et mantisse (1 octet)
    exposant=(p[0]<<1)+((p[1]&0x80)>>7);
    mantisse = p[1]<<1+((p[2]&0x80)>>7);
    // conversion en virgule fixe ....
    v = mantisse | 0x0100;
    if (exposant-127 > 0)
        v = (v << (exposant-127))>>6;
    else
        v= (v >> (6 - exposant +127));
    printf("octet virgule fixe : %x\n",v);
    return 0;
}
```

Ne pas oublier de tester le bit complètement à gauche pour le signe, ce qui n'est pas fait ici.

Le problème c'est que le compilateur CC11 n'accepte pas "p=&r;"

Le programme complet (C sur PC) qui m'a servi aux tests est :

```
// float.c
#include <math.h> // a cause du pow
main() {
    float r,valeur;
    register short v;
    unsigned char *p,i,exposant,mantisse;
```

```

p=&r;
r=16.75; // donne 16.75 par reconstruction
// parceque little INDIAN sous LINUX x86
i=p[1]; // inutile sous 68HC11 qui est BIG INDIAN
p[1]=p[2];
p[2]=i;
i=p[3];
p[3]=p[0];
p[0]=i;
// fin little INDIAN
printf("decomposition : ");
for (i=0;i<4;i++) printf(" %d",p[i]);
printf("\n");
exposant=(p[0]<<1)+((p[1]&0x80)>>7);
mantisse = p[1]<<1;
// reconstruction de la valeur bit par bit...
valeur = (1+((mantisse & 0x80)>>7) *0.5
          +((mantisse & 0x40)>>6) *0.25
          +((mantisse & 0x20)>>5) *0.125
          +((mantisse & 0x10)>>4) *0.0625
          +((mantisse & 0x08)>>3) *0.03125
          +((mantisse & 0x04)>>2) *0.015625
          +((mantisse & 0x02)>>1) *0.0078125
          +(mantisse & 0x01) *0.00390625) * pow(2,exposant-127);
printf("mantisse : %d exposant %d",mantisse,exposant);
printf("\n");
printf("valeur : %f\n",valeur);
// conversion en virgule fixe
v = mantisse | 0x0100;
if (exposant-127 > 0)
    v = (v << (exposant-127))>>6;
else
    v= (v >> (6 - exposant +127));
printf("octet virgule fixe : %x\n",v);
return 0;
}

```