PROGRAMMATION DES MICROCONTROLEURS EN C

TRUCS ET ASTUCES

Christian Dupaty
Lycée Fourcade 13120 Gardanne
c.dupaty@aix-mrs.iufm.frv

1. Imposer une adresse physique pour une variable

Les registres des microcontrôleurs sont à des adresses imposées par le concepteur or le linker d'un compilateur C a le contrôle total des adresses il choisit ou sont rangées variables et constantes. Définir une adresse physique permet par exemple de définir comme une variable l'adresse d'un registre (adresse imposée par le micro contrôleur)

Exemple, on veut associer au mot mamem la mémoire RAM à l'adresse 0x80.

Ceci est un nombre : `0x80`
Un CAST transforme ce nombre en un pointeur sur un octet `(unsigned char *) 0x80`
Ceci représente le contenu de ce pointeur `*(unsigned char *)0x80`
Afin d'éviter cette écriture lourde, on crée une équivalence : `#define mamem *(unsigned char *)0x80`

```
#define mamem *(unsigned char *)0x80
char c;
void main (void)
{
mamem = 0xAA;    // on met 0XAA dans la mémoire 0x80
c=mamem;         // c= 0xAA
while(1);
}
```

2. Créer une variable de plus de 256 octets

C18 est configuré pour optimiser le temps d'accès aux données, il travaille par défaut sur les bank de 256 octets.
La déclaration d'une variable de plus de 256 octets (un tableau par exemple déclanche une erreur de link :

```
MPLINK 3.90.01, Linker Copyright (c) 2005 Microchip Technology Inc.
Error - section '.udata_varlarges.o' can not fit the section.
Section '.udata_varlarges.o' length=0x0000012c Errors : 1
```

On peut cependant déclarer un linker des espaces mémoires supérieurs. Ils s'en accommode, modifie automatiquement les noms des banks lors des accès. Le temps d'accès est seulement plus long

```
// Variables de plus de 256 octets
unsigned char c[300]; // c ne rentre pas dans une bank de 256 octest !!!

void main (void)
{
     while(1);
}
```

Pour cela il faut modifier la config du linker, exemple : p18f452.lkr comme ceci

```
-----------------------------------------------------------------------------
-----------
// Sample linker command file for 18F452i used with MPLAB ICD 2
// $Id: 18f452i.lkr,v 1.2 2002/07/29 19:09:08 sealep Exp $
LIBPATH  .
FILES c018i.o
FILES clib.lib
FILES p18f452.lib

CODEPAGE   NAME=vectors  START=0x0        END=0x29       PROTECTED
CODEPAGE   NAME=page     START=0x2A       END=0x7DBF
CODEPAGE   NAME=debug  START=0x7DC0     END=0X7FFF     PROTECTED
```

```
CODEPAGE   NAME=idlocs   START=0x200000   END=0x200007   PROTECTED
CODEPAGE   NAME=config   START=0x300000   END=0x30000D   PROTECTED
CODEPAGE   NAME=devid    START=0x3FFFFE   END=0x3FFFFF   PROTECTED
CODEPAGE   NAME=eedata   START=0xF00000   END=0xF000FF   PROTECTED

ACCESSBANK  NAME=accessram  START=0x0      END=0x7F
DATABANK    NAME=gpr0       START=0x80     END=0xFF
DATABANK    NAME=grossebank      START=0x100    END=0x3FF // GROSSE BANK
DE 768 octets (on peut mettre plus)
DATABANK    NAME=gpr4       START=0x400    END=0x4FF
DATABANK    NAME=gpr5       START=0x500    END=0x5F3
DATABANK    NAME=dbgspr     START=0x5F4    END=0x5FF    PROTECTED
ACCESSBANK  NAME=accesssfr  START=0xF80    END=0xFFF    PROTECTED
SECTION    NAME=CONFIG     ROM=config
STACK SIZE=0x100 RAM=gpr4
-----------------------------------------------
```

grossebank représente maintenant un espace de 768 octets, la directive #pragma udata permet de forcer le linker à utiliser grossebank (cela est facultatif, MPLINK recherche automatiquement une bank adaptée et ici il n'y en a qu'une) il n'y a plus d'erreur de link.

```
// Variables de plus de 256 octets
#pragma udata grossebank    // facultatif
unsigned char c[300];
#pragma udata                // facultatif

void main (void)
{
     while(1);
}
```

3. Gérer les bits

Le C prévoit la gestion indépendantes des bits à travers une structure. Il est souvent pratique de disposer de "drapeaux" indiquant qu'un processus, doit, s'est ou est en train de se dérouler. Afin d'éviter d'utiliser la plus petite variable du C, le type char pour déclarer un drapeau mais plutôt un bit (0 ou 1), il faut déclarer une structure "champ de bits"

Dans l'exemple ci dessous on crée le nouveau type "chbits" et deux structures de bits : drap1 et drap2 qui seront des octets puisque le type chbits comporte 8 bits.

On accède ensuite aux éléments d'une structure à l'aide du point (.). nomstructure.element

```
struct chbits {
                unsigned bit0:1;     // on aurait pu appeler ce bit
moteur_on ou led_eteinte etc...
                unsigned bit1:1;
                unsigned bit2:1;
                unsigned bit3:1;
                unsigned bit4:1;
                unsigned bit5:1;
                unsigned bit6:1;
                unsigned bit7:1;
              } drap1,drap2;        // et ici moteur ou affichage ...

char c;
void main (void)
{
    drap1.bit2=1;                   // le bit 2 de drap1 est mis à 1
    c=drap2.bit5;                   // c prend la valeur du bit 5 de drap2
( o ou 1)
    if (drap1.bit5) drap2.bit4=0;    // le bit 4 de drap2 est mis à 0 si le
bit 5 de drap1 est à 1
```

```
    while(1);
}
```

4. Attendre qu'un bit change d'état

Pour cela on utilise une structure champs de bit comme ci dessus ou comme celles décrites dans p18f452.h par exemple (accès aux registres du PIC par nom)
Le principe consiste à rester dans une boucle tant que le bit concerné est dans l'état opposé à celui que l'on attend.

Exemple : attendre que le bit 0 du port A passe à 1

```
while (PORTAbits.RA0==0);    // on reste dans cette boucle tant que RA0=0
et on en sort dès qu'il passe à 1
```

Exemple : attendre que le bit 0 du port A passe à 0

```
while (PORTAbits.RA0==1);    // on reste dans cette boucle tant que RA0=1
et on en sort dès qu'il passe à 0
```

Les ==1 et ==0 sont facultatifs puisque la boucle while fonctionne en terme de "vrai" ou "faux", on peut écrire :
`while (!PORTAbits.RA0);` ou **`while (PORTAbits.RA0);`**

5. Accéder à un bit par des macros

Les #define du C permettent de déclarer des équivalence avec des valeurs numériques mais également avec des fonctions

```
#define mamem (*(volatile unsigned char *) 0x10) // definit mamem à
l'adresse 0x10
#define BIT(x) (1<<(x)) //                       // equivalence de décalage
#define setbit(p,b) (p)|=BIT(b)                  // positionne le bit b de
l'octet p à 1
#define clrbit(p,b) (p)&=~BIT(b)                 // positionne le bit b de
l'octet p à 0

void main (void)
{
        mamem |= BIT(3);                 // le bit 3 de mamem passe à 1
        mamem &= ~BIT(3);                // le bit 3 de mamem passe à 0
        mamem ^= BIT(3);                 // le bit 3 de mamem bascule
        if (mamem & BIT(3)) {};          // un test
        setbit(mamem,5);                 // le bit 5 de mamem passe à 1
        clrbit(mamem,5);                 // le bit 5 de mamem passe à 0
        while(1);
}
```

6. Tableaux ou Pointeurs

Les faibles espaces mémoires des microcontrôleurs peuvent parfois être résolus grâce à une meilleur gestion de a RAM et de la ROM La déclaration d'un tableau réserve la taille mémoire (RAM ou ROM) déclarée. La déclaration d'un pointeur ne réserve rien.
Exemple :a fonction char *mois(char i); qui retourne un pointeur sur une chaîne de caractères.

a) écriture avec un tableau

```
char *mois(char i)
{
    const char nomdumois[12 ][10] =
{"janvier","février","mars","avril",....};
    return nomdumois[i];
}
```

Occupation mémoire 12 mois de 10 caractères max = 120 octets, on réserve de la place inutilement, exemple pour le mois de mai le tableau contient

m	a	i	\0	\0	\0	\0	\0	\0	\0

donc 6 octets inutilisés

b) écriture avec un tableau de pointeurs

```
char *mois(char i)
{
    const char *nomdumois[ ] =  { "janvier","février","mars","avril",
.....};
    return nomdumois[i];
}
```

nomdumois ⇨ janvier\0 8 octets
nomdumois +1 ⇨ février\0 8 octets
nomdumois +2 ⇨ mars\0 5 octets
nomdumois +3 ⇨ avril\0 6 octets
nomdumois +4 ⇨ mai\0 4 octets
....

Occupation mémoire : 8 + 8 + 5 +6 + 4 +.... L'occupation mémoire correspond à la somme des tailles des chaînes, la gestion mémoire est bien meilleure.

7. Variables locales ou globales

L'utilisation de **variables locales** améliore considérablement la **lisibilité et la sécurité** des données dans un programme en C. Les variables locales sont par défaut "**dynamiques**" donc crées à l'entrée de la fonction qui les déclare et détruite à la sortie. Pour cela elles sont rangées dans la "**pile**" , zone mémoire particulière, destinée primairement au stockage des données de retour des sous programme.
Il peut arriver que l'on souhaite **conserver une variable** entre deux appels à une fonction, il faut alors la déclarer **"static"**, elle sera rangée dans la RAM à une adresse fixe comme une variable globale, mais ne sera "visible" que dans la fonction qui l'a déclaré.
L'**inconvénient** des variables globales "dynamiques" est leur **temps d'accès** . Le principal **avantage** est l'**optimisation** de l'utilisation de l'espace mémoire RAM souvent petit sur les microcontrôleurs.
Voici trois exemples du même programme qui met en évidence la nécessité d'un compromis "vitesse d'exécution" "taille du code" "sécurité des données"
Cadre de gauche , le source en C, cadre de gauche, l'assembleur généré par le compilateur (C18)

Addition sur des variables globales - Un programmeur en assembleur aurait produit le même code

```
char a;//3 globales
char b;
char c;

void main (void)
{
   a=0;
   b=2;
   c=3;
   a=b+c;
   while(1);
}
```

```
 void main (void)
 {
 a=0;
0000E2 0100 MOVLB 0
0000E4 6B8A CLRF 0x8a, BANKED      ; a
a été placé en 0x8a par le linker
 b=2;
0000E6 0E02 MOVLW 0x2
0000E8 6F8B MOVWF 0x8b, BANKED     ; b
a été placé en 0x8b
 c=3;
0000EA 0E03 MOVLW 0x3
0000EC 6F8C MOVWF 0x8c, BANKED     ; c
a tét placé en 0x8c
 a=b+c;
0000EE 518B MOVF 0x8b, W, BANKED   ; b
dans w
```

```
                                0000F0 258C ADDWF 0x8c, W, BANKED  ;
b+c dans w
                                0000F2 6F8A MOVWF 0x8a, BANKED     ; w
dans a
                                 while(1);
                                0000F4 D7FF BRA 0xf4
                                 }
```

Addition sur des variables locales dynamiques , cela devient beaucoup plus compliqué en raison des accès par pointeurs dans la pile

```
                                            void main (void)
                                0000CA CFD9 MOVFF 0xfd9, 0xfe6
                                0000CC FFE6 NOP
                                0000CE CFE1 MOVFF 0xfe1, 0xfd9
                                0000D0 FFD9 NOP
                                0000D2 0E03 MOVLW 0x3
                                0000D4 26E1 ADDWF 0xfe1, F, ACCESS
                                             { char a=0;
                                0000D6 6ADF CLRF 0xfdf, ACCESS      ;
a est dans 0xfdf
                                               char b=2;
                                0000D8 52DE MOVF 0xfde, F, ACCESS
                                0000DA 0E02 MOVLW 0x2
                                0000DC 6EDD MOVWF 0xfdd, ACCESS     ;
void main (void)                b dans 0xfdd
{                                              char c=3;
    char a=0;                   0000DE 0E03 MOVLW 0x3
    char b=2;                   0000E0 6EF3 MOVWF 0xff3, ACCESS
    char c=3;                   0000E2 0E02 MOVLW 0x2
    a=b+c;                      0000E4 CFF3 MOVFF 0xff3, 0xfdb      ;
    while(1);                   c dans 0xfdb
}                               0000E6 FFDB NOP
                                             a=b+c;
                                0000E8 CFDB MOVFF 0xfdb, 0xfe6      ;
                                c dans fe6
                                0000EA FFE6 NOP
                                0000EC 0E01 MOVLW 0x1
                                0000EE 50DB MOVF 0xfdb, W, ACCESS   ;
                                c dans w
                                0000F0 52E5 MOVF 0xfe5, F, ACCESS
                                0000F2 24E7 ADDWF 0xfe7, W, ACCESS  ;
                                w+ ? dans s w
                                0000F4 6EDF MOVWF 0xfdf, ACCESS     ;
                                w dans a
                                             while(1);
                                0000F6 D7FF BRA 0xf6
```

Addition sur des variables locales statiques on retrouve le même code assembleur que pour les variables globales

```
                                 void main (void)
                                 {
                                 static char a;    // trois variables
void main (void)                locales statiques
{                                static char b;
static char a;                   static char c;
static char b;                   a=0;
static char c;                  0000E2 0100 MOVLB 0
    a=0;                        0000E4 6B8A CLRF 0x8a, BANKED       ;
    b=2;                        0 dans a
    c=3;                         b=2;
    a=b+c;                      0000E6 0E02 MOVLW 0x2
    while(1);                   0000E8 6F8B MOVWF 0x8b, BANKED      ;
}                               2 dans b
                                 c=3;
                                0000EA 0E03 MOVLW 0x3
```

```
0000EC 6F8C MOVWF 0x8c, BANKED       ;
3 dans c
 a=b+c;
0000EE 518B MOVF 0x8b, W, BANKED     ;
b dans w
0000F0 258C ADDWF 0x8c, W, BANKED    ;
w+c dans w
0000F2 6F8A MOVWF 0x8a, BANKED       ;
w dans a
 while(1);
0000F4 D7FF BRA 0xf4
 }
```

Les variables locales statiques sont gérés comme les variables globales, elles restent cependant invisibles en dehors de la fonction qui les déclare.