La programmation des PIC en C

Les variables, les constantes, les calculs mathématiques

Réalisation : HOLLARD Hervé.
http://electronique-facile.com
Date : 24 octobre 2009
Révision : 1

Sommaire

Introduction

Les microcontrôleurs PIC de la société Microchip sont depuis quelques années dans le "hit parade" des meilleures ventes. Ceci est en partie dû à leur prix très bas, leur simplicité de programmation, les outils de développement que l'on trouve sur le NET.

Aujourd'hui, développer une application avec un PIC n'a rien d'extraordinaire, et tous les outils nécessaires sont disponibles gratuitement. Voici l'ensemble des matériels qui me semblent les mieux adaptés.

Ensemble de développement (éditeur, compilateur, simulateur) :
MPLAB de MICROCHIP **http://www.microchip.com**
Logiciel de programmation des composants :
IC-PROG de Bonny Gijzen **http://www.ic-prog.com**
Programmateur de composants:
PROPIC2 d'Octavio Noguera voir notre site **http://electronique-facile.com**

Pour la programmation en assembleur, beaucoup de routines sont déjà écrites, des didacticiels très complets et gratuits sont disponibles comme par exemple les cours de **BIGONOFF** dont le site est à l'adresse suivante **http://abcelectronique.com/bigonoff**.

Les fonctions que nous demandons de réaliser à nos PIC sont de plus en plus complexes, les programmes pour les réaliser demandent de plus en plus de mémoire. L'utilisateur est ainsi à la recherche de langages "évolués" pouvant simplifier la tâche de programmation.

Depuis l'ouverture du site **http://electronique-facile.com**, le nombre de questions sur la programmation des PIC en C est en constante augmentation. Il est vrai que rien n'est aujourd'hui disponible en français.

Mon expérience dans le domaine de la programmation en C due en partie à mon métier d'enseignant, et à ma passion pour les PIC, m'a naturellement amené à l'écriture de ce **didacticiel**. Celui-ci se veut **accessible à tous ceux qui possèdent** une petite expérience informatique et électronique (utilisation de Windows, connaissances minimales sur les notions suivantes : la tension, le courant, les résistances, les LEDs, les quartz, l'écriture sous forme binaire et hexadécimale.).

Ce cinquième fascicule vous permettra de programmer de façon plus structurée grâce aux fonctions. Vous pourrez enfin utiliser au mieux le PIC 16F84 grâce aux interruptions.

Structure de ce document

Ce document est composé de chapitres. Chaque chapitre dépend des précédents. Si vous n'avez pas de notion de programmation, vous devez réaliser chaque page pour progresser rapidement.

Le type gras sert à faire ressortir les termes importants.

Vous trouverez la définition de chaque **terme nouveau** en **bas de la page** où apparaît pour la première fois ce terme. *Le terme est alors en **italique**.*

La couleur **bleue** est utilisée pour vous indiquer que ce **texte** est **à taper** exactement sous cette forme.

La couleur **rouge** indique des **commandes informatiques** à utiliser.

Le matériel nécessaire

Les deux logiciels utilisés lors du premier fascicule.

Un programmateur de PIC comprenant un logiciel et une carte de programmation. Vous trouverez tout ceci sur notre site.

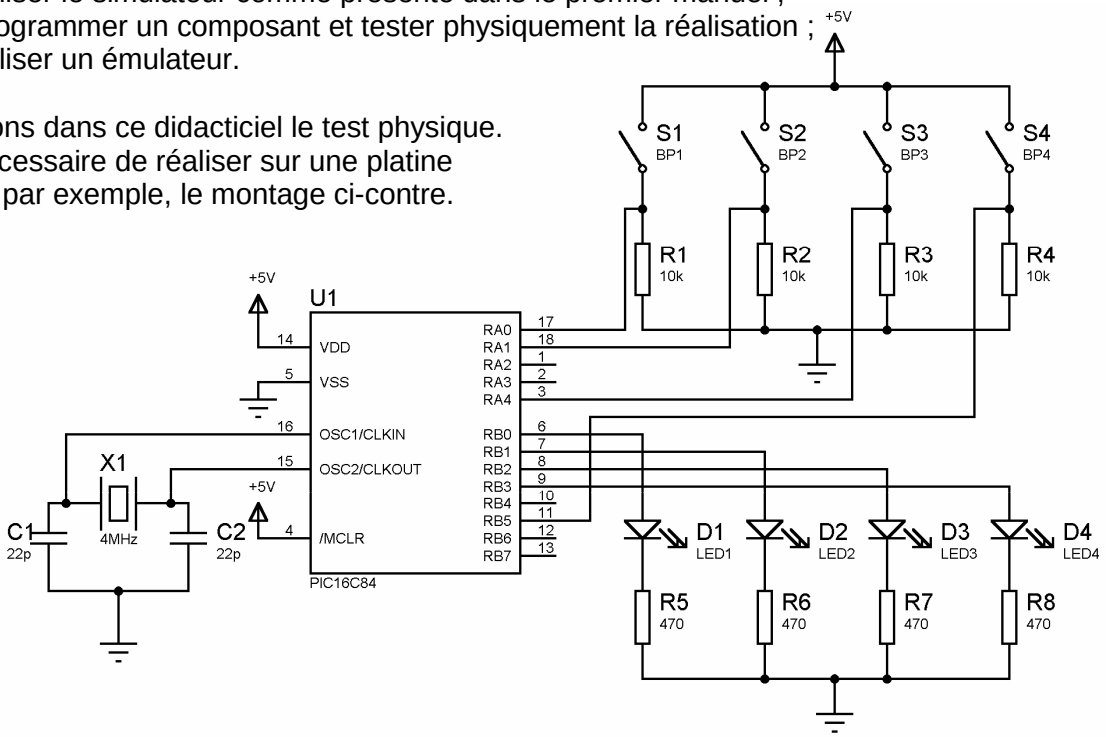
Un PIC 16F84, un quartz de 4MHz, deux condensateurs de 22pF, 4 leds rouges, 4 résistances de 470 ohms, 4 interrupteurs, 4 résistances de 10 Kohms. Une platine d'essai sous 5 volts.

La platine d'essai

Pour tester les programmes proposés, il est possible :

-utiliser le simulateur comme présenté dans le premier manuel ;
-programmer un composant et tester physiquement la réalisation ;
-utiliser un émulateur.

Nous utiliserons dans ce didacticiel le test physique.
Il est ainsi nécessaire de réaliser sur une platine
de type LAB, par exemple, le montage ci-contre.



Généralités sur les variables

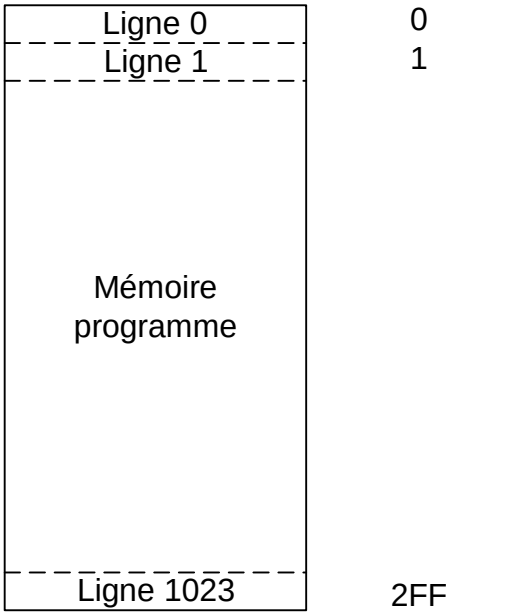
1- Généralités

Le choix du type de mémoire implantée dans un microcontrôleur est uniquement dû à une stratégie constructeur. Nous ne parlerons ici que du PIC 16F84. Celui-ci possède une mémoire Eeprom[1] de programme, une mémoire Eeprom de données, une mémoire RAM[2] de données.

Une variable est une portion réservée d'une mémoire à laquelle on a donné un nom. On peut stocker ou lire les informations de ces variables.

2 - La mémoire Eeprom de programme

Cet espace mémoire est destiné à **contenir le programme**. Les données sont conservées en absence d'alimentation. On ne peut y écrire que par le biais d'un programmateur externe. Il est possible d'y stocker **des constantes** lors de la programmation du composant Ex : message d'avertissement qui apparaîtra sur un afficheur LCD.



Caractéristiques :

-1024 lignes de 14 bits (1 ligne = 1 instruction en assembleur).

-Stockage du programme, de constantes.

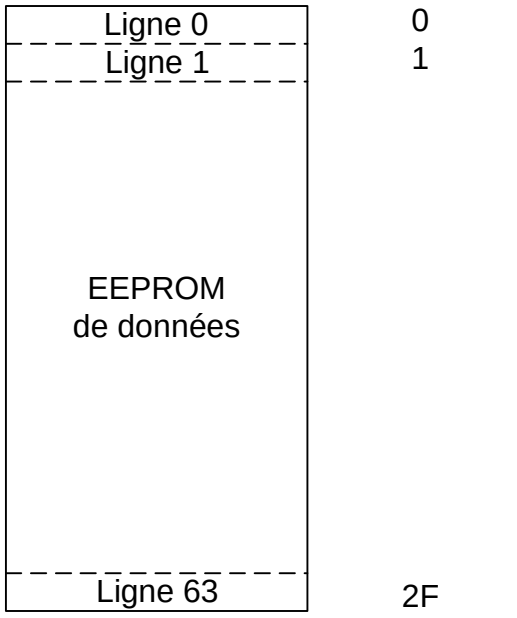
-Ecriture uniquement par programmateur.

-Lecture possible de constantes par le programme.

-Lecture lente de constantes.

3 - La mémoire Eeprom de données

Cet espace mémoire est destiné à **contenir des variables** ou des constantes. Les données sont conservées en absence d'alimentation. On peut y écrire par le biais d'un programmateur externe ou du programme.



Caractéristiques :

-64 lignes de 8 bits.

-Stockage de variables, de constantes.

-Ecriture possible par programmateur externe.

-Ecriture lente par le programme (10ms).

-Lecture rapide.

[1] Mémoire à accès lent, mais qui se conserve en absence d'alimentation.
[2] Mémoire à accès rapide, qui s'efface en absence d'alimentation.

4 - La mémoire RAM de données

Cet espace mémoire est destiné à contenir les registres, des variables. Les données sont perdues en absence d'alimentation. Seuls 68 octets sont libres d'utilisation pour des variables.

Caractéristiques :

-24 octets réservés aux registres (Ex : PORTA).

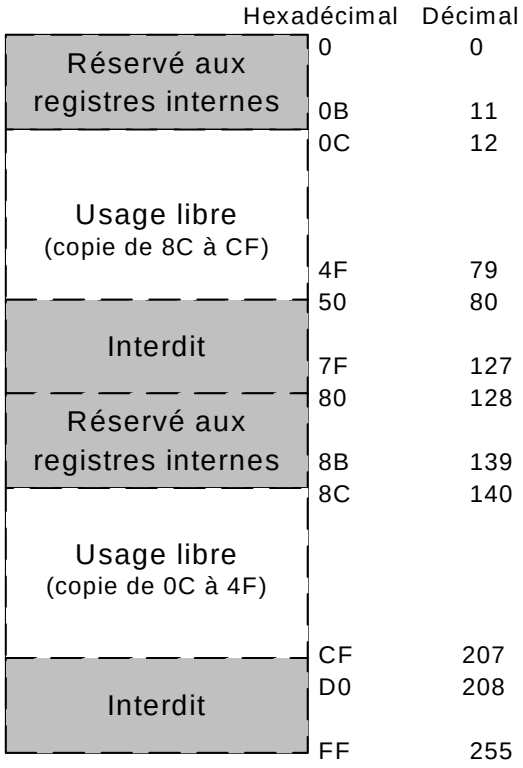
-68 octets pour les variables.

-Ecriture instantanée par le programme.

-Lecture instantanée par le programme.

Particularité de la zone à usage libre :

La zone partant de 0C à 4F a comme miroir la zone partant de 8C à CF. Le fait d'écrire par exemple en OC écrit donc aussi en 8C.



Les différents types de constantes et variables

Toute variable ou constante en C possède un type. Chaque compilateur utilise des déclarations différentes pour chaque type. Un même compilateur peut aussi accepter plusieurs déclarations pour un même type. Voici les types les plus utilisés et leur déclarations.

Type bit

- **Le type bit** occupe **un bit en mémoire.**
  Nous pouvons déposer dans cette variable un 0 ou un 1 logique.

Type entier non signé (entier naturel)

- **Le type char** ou **uns8** occupe **un octet en mémoire.**
  Nous pouvons déposer dans cette variable un nombre entre 0 et 255.
- **Le type uns16** occupe **deux octets en mémoire.**
  Nous pouvons déposer dans cette variable un nombre entre 0 et 65535.
- Il existe aussi les types uns24 et uns32 qui ne sont pas disponibles dans la version d'évaluation. Vous trouverez des informations sur ces variables dans la documentation du compilateur.

Type entier signé (entier relatif)

- **Le type int** ou **int8** occupe **un octet en mémoire.**
  Nous pouvons déposer dans cette variable un nombre entre -128 et +127.
- **Le type int16** occupe **deux octets en mémoire.**
  Nous pouvons déposer dans cette variable un nombre entre -32768 et 32767.
- Il existe aussi les types int24 et int32 qui ne sont pas disponibles avec la version d'évaluation. Vous trouverez des informations sur ces variables dans la documentation du compilateur.

Types à virgule fixe (décimal)

Dans ce type, un nombre d'octets est réservé pour la partie réelle ou relative et un autre nombre est réservé pour la partie décimale. Ce type n'existe pas dans la version d'évaluation de CC5X.

- **Le type fixedU8_8** occupe **deux octets en mémoire**.
  Nous pouvons déposer dans cette variable un nombre entre 0 et 255,996. La résolution pour ce type est de 0.00390625
- **Le type fixed8_8** occupe **deux octets en mémoire**.
  Nous pouvons déposer dans cette variable un nombre entre -128 et 127,996. La résolution pour ce type est de 0.00390625
- Il existe d'autres types à virgule fixe (une vingtaine). Vous trouverez des informations sur ces variables dans la documentation du compilateur.

Types à virgule flottante (décimal)

Dans ce type, les nombres sont mis sous une forme assez complexe, comportant une mantisse (chiffres) et un exposant.

Dans la version d'évaluation, seul le type **float** est disponible. Il est codé sous **trois octets**, et peut aller de +/- 1,1e-38 à +/- 3,4e38.

L'inconvénient de ce type est la longueur du code pour réaliser des opérations et le temps de traitement.

Codage des entiers naturels (codage binaire)

Nous allons comprendre le codage du type uns8, qui permet de stocker un nombre entre 0 et 255 dans un registre 8 bits.

Nombre	Registre 8 bits
0	00000000
1	00000001
2	00000010
3	00000011

Nombre	Registre 8 bits
4	00000100
253	11111101
254	11111110
255	11111111

Pour comprendre le codage il faut connaître la valeur décimale de chaque bit.

	Registre 8 bits							
Bits	8	7	6	5	4	3	2	1
Valeurs décimale de chaque bit	128	64	32	16	8	4	2	1

Tout nombre est décomposable en une addition de une ou plusieurs "valeur décimale de chaque bit".
Exemple : 253 = 128 + 64 + 32 + 16 + 8 + 4 + 1 (seule la valeur décimale du bit 2 n'apparaît pas)

Le code binaire peut être formé en mettant à 1 chaque bit dont la valeur décimale apparaît.

	Registre 8 bits							
Bits	8	7	6	5	4	3	2	1
Valeurs décimale constituant 253	128	64	32	16	8	4		1
253 codé en binaire	1	1	1	1	1	1	0	1

253 est donc égal à 11111101 en binaire.

Codage des entiers relatifs

Nous allons comprendre le codage du type int8, qui permet de stocker un nombre entre -128 et 127 dans un registre de 8 bits.

Codage	0	1	...	127	128	129	...	254	255
Nombre	0	1	...	127	-128	-127	...	-2	-1

Nous verrons dans le registre de stockage du microcontrôleur le code du nombre en binaire.

Ce codage s'appelle complément à 2. Le passage en complément à 2 est assez simple :

- Le complément à deux d'un nombre positif est égal au nombre lui même.
- Le complément à deux d'un nombre négatif s'obtiens en inversant tous les bits de la valeur absolue du nombre codé en binaire et en ajoutant 1.

Exemple :
3 en complément à 2 s'écrit 3.
-3 en complément à deux s'écrit 253 (3 s'écrit 00000011, l'inversion s'écrit 11111100, l'inversion + 1 s'écrit 11111101, ce nombre en décimal est égal à 253)

L'avantage de ce codage est le suivant :
Si un registre dans le microcontrôleur à pour valeur 255 et qu'on lui rajoute 1, le résultat prendra comme valeur 0. Nous avons fait l'addition suivante 0 = -1 + 1. Le processus d'addition "classique" permet ainsi de traiter des nombres négatifs.

Codage des décimaux non signés

Nous allons étudier le type fixedU8_8. Il permet de coder un nombre entre 0 et 255,996 avec une résolution de 0.00390625

Un registre 8 bits représente la partie entière, un registre 8 bits représente la partie décimale.
La partie entière est codée selon le codage binaire vu précédemment.
La partie décimale diffère du codage binaire par les valeurs attribuées à chaque bit.

	Registre 8 bits représentant la partie décimale							
Bits	8	7	6	5	4	3	2	1
Valeurs décimale de chaque bit	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256

Exemple de codage de la partie décimale: 0,5 est égal à 10000000
0.00390625 est égal à 00000001

Exemple général :

253,75 s'écrit 11111101 11000000

Les autres codages

Nous ne parlerons pas des décimaux signés qui maintenant doivent vous paraître évidents.
Nous ne parlerons pas des nombres à virgule flottante qui ont un codage complexe qui ne présente pas d'intérêt ici.

Visualiser des variables numériques sous MPLAB.

Taper sous MPLAB le programme suivant :

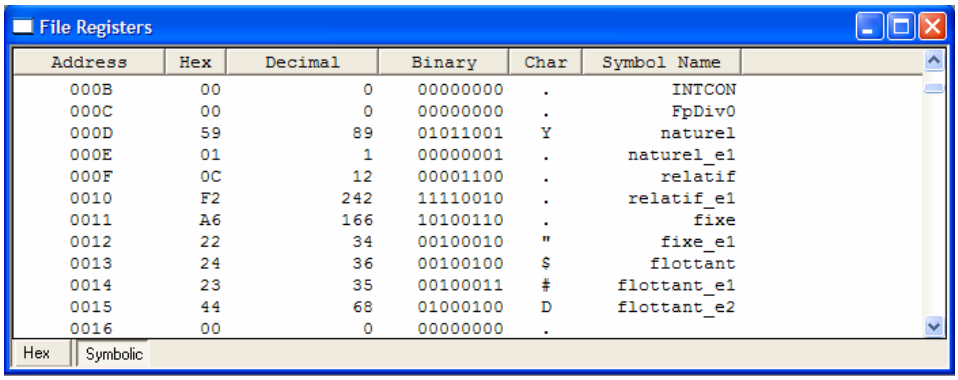
```
// Attention de respecter les majuscules et minuscules
//----------------------declaration de librairies-----------------------------
#include "MATH24F.H"

//-------------------------declaration des variables--------------------------
uns16 naturel;                  // reservaion de 2 octets pour un entier naturel
int16 relatif;                  // reservation de 2 octets pour un entier relatif
fixedU8_8 fixe;                 // reservation de 2 octets pour un nombre à virgule fixe
float flottant;                 // reservation de 3 octets pour un nombre a virgule flottante

//-----------------------------Fonction principale----------------------------

void main(void)
{
        naturel=345;
        relatif=-3572;
        fixe=34.65;
        flottant=652.56 ;
        float24ToIEEE754(flottant);   // permet de modifier la variable afin de la rendre
                                      //visualisable par Mplab
}
```

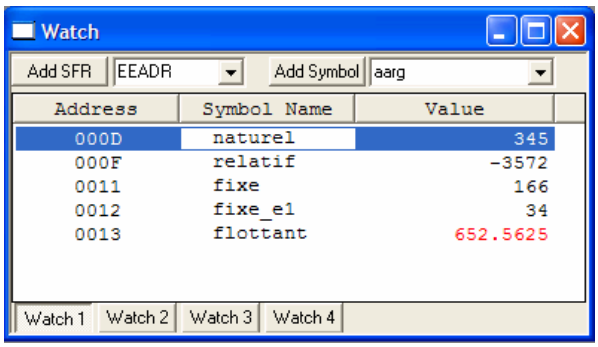
Ouvrir la fenêtre File Registers pour visualiser les contenus des différents registres constituants les variables déclarées.



Aucune variable n'est facilement évaluable puisque la fenêtre File Registers ne permet de voir que des char. Pour pouvoir les visualiser, il faut utiliser la fenêtre Watch comme suit :

- Sélectionner une variable par le menu déroulant de droite
- Cliquer sur Add Symbol
- Ouvrir la fenêtre propriété de la variable par un clic doit sur la variable
- Choisir le type de variable

Vous devez obtenir le résultat ci-contre



Attention : Mplab ne permet pas de visualiser des variables en virgule fixe.

Les variables en RAM

La déclaration d'une variable en ram se fait de deux façons:

- **"type" "nom" @ "adresse-de-la-portion_reservé"**
- **"type" "nom"** (le compilateur réserve une portion encore libre)

La place de la déclaration des variables dans le programme est importante et détermine la durée de vie de la variable.

- **Variable globale**: La variable **existe durant tout le programme**. N'importe quelle fonction peut y avoir accès en écriture ou en lecture. On la déclare au début du programme avant toute fonction.
- **Variable locale**: La variable **existe uniquement dans la fonction ou elle a été crée.** L'intérêt d'une telle variable réside dans l'optimisation de la mémoire. Un même emplacement mémoire peut alors être utilisé par plusieurs fonctions si celles-ci ne sont pas imbriquées

Il est important de connaître la place des variables dans la mémoire pour les visualiser lors de la simulation. Il est aussi indispensable de ne pas créer plus de variables que de mémoire disponible.
Pour cela, lors de la compilation, une table récapitulative de l'état de la mémoire RAM est affichée.

Voici un petit programme, qui ne fait rien, mais sert uniquement à comprendre le processus.

```
// Attention de respecter les majuscules et minuscules

//-------------------------declaration des variables----------------------------

char a @ 0x11;              // reservation d'un octet nomme a à l'adresse 0x11
bit b @ 0x12.2;             // reservation d'un bit nomme b à la place 2 de l'octet 0x012
char c, d;                  // reservation de deux octets nommes c et d
bit e;                      // reservation d'un bit nomme e
uns8 f;                     // reservation d'un octet contenant le nombre 9 et nomme f
int16 g;                    // reservation de 2 octets pour un nombre signe nomme g
float h;                    // reservation de 3 octets pour un nombre a virgule flottante

//-------------------------------Fonction a-----------------------------

void aa(void)
{
char loca;
        nop();              // instruction qui ne fait rien
}

//-------------------------------Fonction b-----------------------------

void bb(void)
{
char locb;
        nop();              // instruction qui ne fait rien
}


//-------------------------------Fonction principale-----------------------------

void main(void)
```

```
{
        aa();
        bb();
}
```

Taper ce programme, le compiler. Les messages d'information de la compilation nous informe de l'état de la RAM, nous allons étudier ces messages.

```
RAM: 00h : -------- ----=..7 .-7..... ********
RAM: 20h : ******** ******** ******** ********
RAM: 40h : ******** ********
RAM usage: 12 bytes (1 local), 56 bytes free
```

Explication des lignes 1, 2 et 3 :

- Chaque symbole représente l'état d'un octet. La première ligne, par exemple, récapitule l'état des octets 00h à 19h (le h signifie que nous parlons en hexadécimal).
- \- Octet réservé par le compilateur pour le fonctionnement du microcontrôleur (registre PORTA, TRISB, …) ou par l'utilisateur à une place précise (0X11).
- . Variable globale réservé par l'utilisateur, mais dont la place n'a pas d'importance.
- = Variable locale.
- 7 Octet où 7 bits sont disponibles.
- * Octet libre.

La ligne 4 récapitule l'état de la RAM.

Remarquez qu'il n'y a qu'un seul octet de réservé pour deux variables locales.

Les tableaux en RAM

Lorsque qu'il est nécessaire de créer plusieurs variables qui seront traitées les unes après les autres, il est intéressant de créer un tableau.

Nous allons créer un tableau de 6 variables de type int, que nous allons appeler t. Pour différencier les variables nous avons besoin d'un index. Les variables prennent le nom de t[index].

Index	0	1	2	3	4	5
Variable	t[0]	t[1]	t[2]	t[3]	t[4]	t[5]

La déclaration d'un tel tableau se fait par :

```
int t[6] ;
```

Déposons par exemple -24 dans t[3]

```
t[3]=-24 ;
```

Afin d'automatiser le traitement des variables du tableau, il est possible d'utiliser une variable de type char comme index que nous allons par exemple appeler i. Nous pouvons gérer par le programme i, nous gérons ainsi les variable du tableau.

Déposons -24 dans t[3] à l'aide de i :

```
i=3 ;
t[i]=-24 ;
```

Exemple

Nous allons réaliser un chenillard des leds 1, 2, 3 et 4 programmable par l'utilisateur. Le chenillard aura une période de 0,5 seconde.
A la mise sous tension, l'utilisateur actionne les interrupteurs 1,2,3 et 4 afin de définir la séquence de clignotement des leds 1, 2, 3 et 4. Inter1 représente led1, inter2 représente led2 …
Lorsque 10 états différents ont été programmés, le chenillard se met en fonction.

Nous allons donc créer un tableau de 10 variables que nous appellerons chen. Nous allons gérer ce tableau grâce à un index que nous appellerons i.

```
// Attention de respecter les majuscules et minuscules

//--------------------------------E/S-----------------------------------------
char sortie @ PORTB;
bit inter1 @ RA0;
bit inter2 @ RA1;
bit inter3 @ RA4;
bit inter4 @ RB5;
bit led1 @ RB0;
bit led2 @ RB1;
bit led3 @ RB2;
bit led4 @ RB3;

//-------------------------declaration des variables---------------------------
char i;                         // index du tableau des états du chenillard
char chen[10];          //etats du chenillard
char etat_inters;       //etats des inter
```

```
//-----------------delay en multiple de 10ms. Quartz de 4 MHz, erreur 0.16 %-------------
void delay_10ms (char n)
{
char i;
OPTION = 7;             //prescaler to 256
do
  {
  i =TMR0 +39;                  // 256us * 39 =10 ms
  while (i != TMR0);
  }
while (--n > 0);
}

//-----------------------Fonction lecture des interrupteurs----------------------
void lecture(void)
{
        etat_inters.0=inter1;   // Mise de l'etat des inter dans la variable etat_inters
        etat_inters.1=inter2;
        etat_inters.2=inter3;
        etat_inters.3=inter4;
}

//-------------------------------Fonction principale-----------------------------

void main(void)
{
i=0;                                    // Initialisation des registres
etat_inters=0;
sortie = 0;                             // Initialisation des pattes du microcontroleur
TRISB = 0b11110000;

do                                              // Programmation du chenillard
{
        while (etat_inters==0) lecture();       // attente action sur un inter
        delay_10ms(8);                                          // antirebond
        chen[i]=etat_inters;                            //mise a jour du tableau
        i++;
        while (etat_inters!=0) lecture();       // attente inters relaches
}
while (i<10);
i=0;
for(;;)                                 // Mise en fonction du chenillard
{
        etat_inters=chen[i];    // Mise a jour des leds
        led1=etat_inters.0;
        led2=etat_inters.1;
        led3=etat_inters.2;
        led4=etat_inters.3;
        delay_10ms(50);                 //attente 0,5s
        i++;
        if (i==10) i=0;
}
}
```

Les constantes

Il est possible de stocker des données dans l'eeprom de programme au moment de la programmation. Evidement ces données ne sont pas modifiables, ce sont des constantes.

Utilisation implicite des constantes

Nous avons déjà utilisé sans le savoir des constantes. Le fait d'écrire un nombre dans le programme utilise une constante. Par exemple mettons le registre OPTION à 55

```
OPTION = 55 ;
```

Nous avons déjà utilisé sans le savoir un mot clé permettant de définir une constante :

```
#define nom_de_la_constante valeur_de_la_constante
```

Le compilateur remplace tous les nom_de_la_constante qu'il rencontre dans le programme par valeur_de_la_constante. Mettons les registres OPTION et PORTB à 55

```
#define reglage 55
OPTION = reglage ;
PORTB = reglage ;
```

Utilisation voulue de constantes en Eeprom de programme

Il est aussi possible de déposer des données sous toute forme dans l'Eeprom de programme et de les récupérer au moment opportun. Nous disposons ainsi d'un espace de données important.

La déclaration d'une constante se fait comme suit :

```
const type nom_le_la_constante = valeur_de_la_constante
```

Le compilateur gère le lieu de stockage des données. Nous voulons par exemple créer le tableau suivant et le déclarer comme constante :

Tableau	t[0]	t[1]	t[2]	t[3]	t[4]	t[5]
Valeurs	-3	5	235	8	1249	-65

La déclaration prend la forme :

```
const int16 t[] = { -3, 5, 235, 8, 1249, -65} ;
```

Le processus de récupération de ces données est assez complexe, et nécessite des notions d'assembleur. Nous n'en dirons pas plus sur le processus, nous nous contenterons de l'utilisation de ces constante. Elle se fait comme pour un tableau en RAM.

Récupérons t[2] afin de le mettre dans la variable temp :

```
temp = t[2] ;
```

Exemple

Nous allons réaliser un chenillard ayant 2 séquences de 10 états préprogrammées. Le chenillard aura une période de 0,5 seconde.

A la mise sous tension, l'utilisateur actionne l'interrupteur 1 ou 2 afin de définir la séquence à utiliser. Le chenillard se met alors en fonction en utilisant la séquence choisie.

```
// Attention de respecter les majuscules et minuscules

//--------------------------------E/S----------------------------------------
char sortie @ PORTB;
```

```
bit inter1 @ RA0;
bit inter2 @ RA1;
bit led1 @ RB0;
bit led2 @ RB1;
bit led3 @ RB2;
bit led4 @ RB3;

//-------------------------declaration des variables-------------------------
char i;                         // index du tableau des états du chenillard
char etat_inters;       //etats des inter

//-------------------------declaration des constantes-----------------------------
const char un[]={0,1,2,4,8,0,1,2,4,8};
const char deux[]={0,1,2,3,4,5,6,7,8,9};

//-----------------delay en multiple de 10ms. Quartz de 4 MHz, erreur 0.16 %-------------
void delay_10ms (char n)
{
char i;
OPTION = 7;             //prescaler to 256
do
  {
  i =TMR0 +39;                  // 256us * 39 =10 ms
  while (i != TMR0);
  }
while (--n > 0);
}

//-----------------------Fonction lecture des interrupteurs----------------------
void lecture(void)
{
        etat_inters.0=inter1;   // Mise de l'etat des inter dans la variable etat_inters
        etat_inters.1=inter2;
}
//-------------------------------Fonction principale-----------------------------

void main(void)
{
i=0;                                    // Initialisation des registres
etat_inters=0;
sortie = 0;                             // Initialisation des pattes du microcontroleur
TRISB = 0b11110000;
while (etat_inters==0) lecture();       // attente action sur un inter
if (etat_inters==1)                                     //action sur inter 1
{
for(;;)
{
        etat_inters=un[i];
        led1=etat_inters.0;
        led2=etat_inters.1;
        led3=etat_inters.2;
        led4=etat_inters.3;
        delay_10ms(50);
        i++;
        if (i==10) i=0;
}
}
```

```
if (etat_inters==2)                    //action sur inter 2
{
for(;;)
{
        etat_inters=deux[i];
        led1=etat_inters.0;
        led2=etat_inters.1;
        led3=etat_inters.2;
        led4=etat_inters.3;
        delay_10ms(50);
        i++;
        if (i==10) i=0;
}
}
}
```

Les variables en EEPROM de données

La mémoire EEPROM présente sur la RAM, le gros avantage de ne pas s'effacer lors de l'arrêt de l'alimentation, mais l'inconvénient d'être difficile à utiliser. Cette mémoire sert le plus souvent à stocker des données d'acquisition. Il est aussi possible d'y stocker des constantes.

Contrairement à la RAM, CC5X ne gère pas cet espace mémoire, il faudra donc gérer les adresses nous même.

1 - Ecriture en Eeprom par le programmateur.

Il est possible d'écrire des octets dans cette mémoire lors de la programmation. Pour cela, nous allons utiliser certaines instructions dans le programme. Il est de plus nécessaire de savoir que pour le programmateur cet espace mémoire se situe de l'adresse 0X2100 à l'adresse 0x212F.

- Se placer à une adresse : #pragma cdata[adresse]
- Ecrire à partir de l'adresse courante : #pragma cdata[] = octet1, octet2, ...
- Ecrire à une adresse précise : #pragma cdata[adresse] = octet

Attention : ces instructions ne sont pas visibles dans l'Eeprom de programme. Elle seront uniquement utilisées par le programmateur.

Les instructions précédentes peuvent se placer n'importe où dans le programme, elles seront déplacées par le compilateur qui les mettra après le programme, et dans l'ordre dans lequel elle sont écrites.

Nous verrons un exemple un peu plus loin.

2 - Lecture en Eeprom par le programme

Pour le compilateur CC5X, la lecture est assez simple et identique à la méthode utilisée en assembleur :

- Ecriture de l'adresse à lire dans le registre EEADR. Les adresses valides sont 0x00 à 0x2F.
- Mise a 1 du bit RD du registre EECON1
- Lecture de la donnée dans le registre EEDATA

Nous verrons un exemple un peu plus loin

3 - Ecriture en Eeprom sans interruption par le programme

Pour le compilateur CC5X, l'opération d'écriture est plus complexe, et identique à la méthode utilisée en assembleur :

- Mise à 1 du bit WREN du registre EECON1 (1 seule fois dans le programme)
- Ecriture de l'adresse à où l'on désire stocker un octet dans le registre EEADR
- Ecriture de la donnée à sauvegarder dans le registre EEDATA
- Réalisation d'un sous programme de sécurité afin d'éviter les manipulations malencontreuses. Ce programme est écrit en assembleur
- Au bout d'un temps variable de 10 à 20ms, l'écriture est terminée, le bit WR passe à 0 (il a été mis en 1 par le sous programme en assembleur)

Ecriture du sous programme en assembleur :

```
#asm
    BSF STATUS,RP0
    MOVLW 0x55
    MOVWF EECON2
    MOVLW 0xAA
    MOVWF EECON2
    BSF EECON1,WR
#endasm
```

Nous verrons un exemple un peu plus loin.

4 - Ecriture en Eeprom avec interruption par le programme

L'opération d'écriture est presque identique à la précédente, nous allons attendre une interruption généré par la fin d'écriture au lieu de lire le bit WR.

- Mise à 1 du bit WREN du registre EECON1 (1 seule fois dans le programme)
- Mise à 1 du bit EEIE du registre INTERCON (bit homologue à TOIE, RBIE …)
- Mise à 0 du bit EEIF du registre EECON1 (bit homologue à TOIF, RBIF …)
- Ecriture de l'adresse à où l'on désire stocker un octet dans le registre EEADR
- Ecriture de la donnée à sauvegarder dans le registre EEDATA
- Désactivation des interruptions par le bit GIE (voir les interruptions)
- Réalisation d'un sous programme de sécurité afin d'éviter les manipulations malencontreuses. Ce programme est écrit en assembleur
- Activation des interruptions par le bit GIE
- Au bout d'un temps variable de 10 à 20ms, l'écriture est terminée, une interruption est générée, il faut alors lire le bit EEIF du registre EECON1 afin de savoir d'où vient l'interruption comme nous l'avons vu dans le fascicule précédent, puis remettre ce bit à 0.

Nous ne verrons pas d'exemple pour ce paragraphe.

5 - Exemple sur l'utilisation de l'Eeprom

Nous allons réaliser un chenillard des leds 1, 2, 3 et 4 programmable par l'utilisateur.

A la mise sous tension, les leds 1, 2, 3 et 4 clignotent selon une séquence définie par l'utilisateur avec une période de 0,5 seconde.

Le système se met en programmation si les inter 1, 2, 3 et 4 sont tous actionnés au moment de la mise sous tension. La sortie du mode programmation se fait en arrêtant l'alimentation.

En mode programmation, une led s'allume pour indiquer que la séquence s'efface, puis s'éteint. Par la suite les actions sur les interrupteurs 1, 2, 3 et 4 sont mémorisées afin de constituer la séquence du chenillard. Inter1 représente led1, inter2 représente led2 …

```
// Attention de respecter les majuscules et minuscules
//--------------------le compilateur met l'Eeprom à 0---------------------------
#pragma cdata[0x2100]
#pragma cdata[] = 1,2,3,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0
#pragma cdata[] = 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
#pragma cdata[] = 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
//--------------------------------E/S-----------------------------------------
char sortie @ PORTB;
bit inter1 @ RA0;
bit inter2 @ RA1;
bit inter3 @ RA4;
bit inter4 @ RB5;
bit led1 @ RB0;
bit led2 @ RB1;
bit led3 @ RB2;
bit led4 @ RB3;

//--------------------------declaration des variables---------------------------
char adresse;           //adresse en Eeprom
char etat_inters;       //etats des inter ou des leds

//-----------------delay en multiple de 10ms. Quartz de 4 MHz, erreur 0.16 %--------------
void delay_10ms (char n)
{
char i;
OPTION = 7;             //prescaler to 256
do
  {
  i =TMR0 +39;                  // 256us * 39 =10 ms
  while (i != TMR0);
  }
while (--n > 0);
}


//-----------------------Fonction lecture des interrupteurs----------------------
void lecture_inters(void)
{
        etat_inters.0=inter1;   // Mise de l'etat des inter dans la variable etat_inters
        etat_inters.1=inter2;
        etat_inters.2=inter3;
        etat_inters.3=inter4;
}

//-----------------------Fonction lecture de L'EEprom de donnée--------------------
char lecture_eeprom(char adr)
{
EEADR=adr;
RD=1;
return (EEDATA);
}

//-----------------------Fonction ecriture dans L'EEprom de donnée--------------------
void ecriture_eeprom(char adr, char donnee)
{
while (WR==1);
WREN=1;
```

```
EEADR=adr;
EEDATA=donnee;
#asm
        BSF STATUS,RP0
        MOVLW 0x55
        MOVWF EECON2
        MOVLW 0xAA
        MOVWF EECON2
        BSF EECON1,WR
#endasm
}

//------------------------------Fonction principale-----------------------------
void main(void)
{
adresse=0;                                  // Initialisation des registres
etat_inters=0;
sortie = 0;                                 // Initialisation des pattes du microcontroleur
TRISB = 0b11110000;
lecture_inters();
if (etat_inters==0b00001111)                // Mode programmation
{
        PORTB=1;
        while (etat_inters!=0) lecture_inters();        // attente inters relaches
                                                        //effacement de l'Eeprom
        do ecriture_eeprom(adresse,0); while (++adresse<64);
        PORTB=0;
        adresse=0;
        for(;;)
        {
                while (etat_inters==0) lecture_inters();        // attente action sur un inter
                delay_10ms(8);                                  // antirebond
                ecriture_eeprom(adresse,etat_inters);           //mise a jour du tableau
                adresse++;
                while (etat_inters!=0) lecture_inters();        // attente inters relaches
        }
}

for(;;)                                                         // Mise en fonction du chenillard
{
        etat_inters=lecture_eeprom(adresse);        //lecture de la combinaison
        if (etat_inters==0) adresse=0;              //fin de la zone programmation
        else                                        // Mise a jour des leds
        {
                led1=etat_inters.0;
                led2=etat_inters.1;
                led3=etat_inters.2;
                led4=etat_inters.3;
                delay_10ms(50);                                 //attente 0,5s
                adresse++;
        }
}
}
```

Les 5 opérations : +,-,*,/,%

Nous avons à plusieurs reprises mis en oeuvre des additions et des soustractions de nombres de type char.

La mise en oeuvre des multiplications, des divisions, des modulos, des autres types est aussi simple. Elle nécessite uniquement et dans certains cas de déclarer une bibliothèque en début de programme.

Afin de traiter les notions d'addition, de soustraction, de multiplication, de division, de modulos sur tout type de données, nous allons utiliser le programme ci-dessous, que nous simulerons.

```
// Attention de respecter les majuscules et minuscules

//-----------------------declaration de librairies-----------------------------
#include "MATH24F.H"

//-------------------------declaration des variables---------------------------
float operande1;                    // operande 1
int8 operande2;                     // operande 2
float resultat;                     // resultat

//-----------------------fonction principale-------------------------------------
void main(void)
{
operande1=35.46;
operande2=-1;
resultat=operande1+operande2;
}
```

Afin de réaliser différents tests, il est possible de modifier :

- Les types de variables
- Les valeurs des opérandes
- Le type d'opération
- La mise en oeuvre de librairies. (Il est nécessaire de lire la documentation du compilateur pour comprendre l'utilité de chaque librairie).

Pour pouvoir visualiser les deux opérandes et le résultat, vous pouvez utiliser la fenêtre Watch. Il vous faudra lire la page 10 de ce didacticiel pour utiliser efficacement cette fenêtre.

La page suivante vous permettra de comprendre les différences entre les principales opérations.

1 - Opérations sur types identiques

Type d'opération	Bibliothèques nécessaires	Temps de réalisation approximatifs	Nombre de lignes utilisées approximativement
Addition et soustraction sur char, int	Aucune	20us	20
Addition et soustraction en virgule fixe	Aucune	20us	20
Addition et soustraction en virgule flottante	"MATHXXF.H"	200 à 600us	100
Multiplication division sur char, int	Aucune	100us	25
Multiplication, division en virgule fixe	"MATHXXX.H"	200 à 2000us	60
Multiplication, division en virgule flottante	"MATHXXF.H"	200 à 600us	100

Observation :

- L'utilisation de nombres à virgule flottante est assez délicate car les opérations prennent beaucoup de place en Eeprom de programme (100 lignes occupent 1/10ème de l'espace) et prennent beaucoup de temps.

Attention

- La plupart des compilateurs ne gèrent pas les dépassements. Si le résultat d'une opération de type char+char = char peut donner un nombre >255, l'opération sera réalisée, mais sera fausse.

2 - Opérations sur types différents

Pour la plupart des opérations sur types différents, il est nécessaire de réaliser une transformation de type. Cette transformation est invisible pour l'utilisateur.

Type d'opération	Bibliothèques nécessaires	Temps de réalisation	Nombre de lignes utilisées
char + char = uns 16	Aucune	10us	10
int8 + int8 = int16	Aucune	20us	20
char + char = float char - char = float	"MATH24F.H"	100us	100
char + float = float char - float = float	"MATH24F.H"	200us	250

Observation :

- L'opération char + float = float occupe 25% de l'Eeprom de programme.

Conclusion :

L'utilisation des types à virgule flottante doit être faite en dernier recours.

Petite astuce pour manipuler des décimaux en utilisant des variables de type entier :

Admettons que vous vouliez utiliser des décimaux avec deux chiffres après la virgule. Au lieu d'utiliser des virgules fixes ou flottantes, vous pouvez utiliser des nombres entiers tels que ces nombres entiers soient égaux aux nombres décimaux multipliés par 100.

Ex : Au lieu de réaliser l'opération 12,54 * 51 = 639,54 vous pouvez réaliser 1250 * 51 = 63954

Il est alors possible de facilement récupérer la partie entière en divisant le résultat par 100. Nous trouverons ici 639.

Cette astuce a ses limites car il est rapidement nécessaire d'utiliser des entiers de 24 ou 32 bits.

Et comment peut-on afficher des nombres ?

Admettons qu'un registre contienne le nombre 124. En réalité, ce registre contient 01111100.

Si nous voulons utiliser des afficheurs 7 segments, Il faut décomposer ce registre en trois registres contenant respectivement 1, 2 et 4 (codage BCD) puis diriger ces 3 registres vers 3 afficheurs 7 segments. Il existe des algorithmes qui réalisent ce travail et que l'on peut trouver rédigés en assembleur chez Microchip (AN526, AN544).

Si nous voulons utiliser un afficheur LCD, il faut rédiger des fonctions qui permettent de dialoguer avec les LCD. Vous en trouverez dans une des réalisations de ce site : le thermomètre.

Malheureusement, ces problèmes dépassent largement le cadre de ce didacticiel.

Pour aller plus loin

Ayant de moins en moins de temps à consacrer à l'électronique, ce didacticiel sera le dernier de cette série.

N'oubliez pas l'existence de l'association "Electronique facile" qui propose une aide personnalisée à vos projets. Pour cela, vous serez mis en relation avec un bénévole. Les bénévoles de l'association sont pour la plupart des professionnels de l'électronique. Cette personne vous consacrera quelques minutes par semaine pour répondre à vos questions.