Introduction au langage C pour µC

Introduction

Le C est le langage le plus fréquemment utilisé pour la programmation des microcontrôleurs. Sa syntaxe ressemble beaucoup à celle de Java.

Ce document n'est pas un cours sur le C, il a pour but de mettre en évidence les principales différences entre C et les langages que vous avez déjà étudiés (Python et Java).

Parmi les différences fondamentales, citons :

- C est un langage compilé, alors que Python est un langage interprété et que JAVA est un mixte des deux puisqu'il est compilé pour une machine virtuelle. La compilation rend le programme non portable, ce qui n'est en général pas une contrainte dans le cas des systèmes à microcontrôleur, puisque les programmes sont de toute façon développés pour une plateforme spécifique. Par contre, cela rend l'exécution du programme plus rapide.
- C ne possède pas de gestion dynamique de la mémoire du type *Garbage Collector*. L'utilisation de variables dynamiques est possible, mais doit être entièrement gérée par le programme.
- C n'a pas de gestion des exceptions (comme l'instruction *try* de Python). A nouveau, ces tests doivent être géré par le programme (division par zéro, débordement de variables, …).

Les variables

Les variables en C sont typées, les types sont globalement les mêmes que ceux de Java (cf. *Programmation d'une carte à µC*).

Les variables doivent être explicitement définies avant d'être utilisées.

Les variables définies en dehors des fonctions sont globales.

Les variables définies dans une fonction sont locales à la fonction.

Si une variable locale porte le même nom qu'une variable globale, elle la "masque", rendant l'accès à la variable globale impossible dans la fonction où elle existe.

Une variable locale n'existe que durant l'exécution de sa fonction. Si on veut conserver une valeur d'un appel de la fonction à l'autre, on peut utiliser une variable globale (déconseillé) ou une variable locale statique, qui existe toujours (comme une variable globale) mais qui n'est accessible que par sa fonction. Un exemple de variable statique se trouve à la section ***Echange de données avec une ISR***.

Le type booléen

Il n'existe pas de type booléen, les instructions conditionnelles (if, while, switch) ont comme paramètre un entier (ou une expression dont le résultat est un entier), qui est considéré comme faux si il vaut 0 et vrai sinon.

De même, les opérateurs logiques (!, && et ||) et relationnels (==, !=, <, >, <= et >=) ont comme résultat 0 ou 1 pour *vrai* ou *faux*.

C n'est pas un langage orienté objet

Les classes n'existent donc pas en C, de même que tous les concepts associés (polymorphisme, héritage, méthode, encapsulation).

On peut toutefois définir un type de donnée appelé *structure* pour grouper plusieurs données, comme les champs d'une classe. Par exemple, la variable *PORTBbits* (décrite dans *Programmation d'une carte à µC*) est une structure.

Les fonctions

En C, toutes les fonctions sont globales (pas de fonction définie au sein d'une autre fonction). Cela implique que leur nom doit être unique (les noms sont sensibles à la casse).

La notion de surcharge de fonctions n'existe pas.

Modularité

Un programme C peut être découpé en plusieurs fichiers. Cette découpe peut être choisie arbitrairement par le programmeur.

Lors de la compilation du projet, chaque fichier est compilé séparément. Il arrive fréquemment que les variables et fonctions définies dans un fichier soient utilisées dans un autre fichier. Par exemple, prenons deux fichiers *main.c* et *myMath.c* contenant chacun une fonction.

main.c

```
#include "myMath.h"

int main()
{
    int a = 2 ;
    int b = 3 ;

    c = pow(a,b) ;
        .
        .
        .
}
```

myMath.c

```
int c;

int pow(int base, int exp)
{
    long result = 1;

    while (exp > 0) {
        result *= base;
        exp--
    }
    return(result);
}
```

Pour que le compilateur puisse compiler le fichier main.c, il doit avoir connaitre la fonction pow(). Toutefois, il n'a pas besoin de la définition complète de la fonction, il lui suffit de connaître ce qu'on appelle le prototype ou la déclaration de la fonction :

```
long pow(int base, int exp);
```

Ce dernier contient toutes les informations dont il a besoin, à savoir le nombre de paramètres et leur type, ainsi que le type retourné par la fonction.

Le même mécanisme s'applique aussi aux variables globales partagées par plusieurs fichiers.

Ces déclarations sont rassemblées dans un fichier "d'en-tête" (Header file en anglais, d'où l'extension .h). Ce fichier doit être inclus dans les fichiers utilisant les fonctions qui y sont déclarées ; dans notre exemple, la première ligne de *main.c* : elle inclut le fichier *myMath.h*, jouant un rôle similaire à l'instruction *import* de Python et JAVA.

Le fichier *myMath.h* de notre exemple sera donc :

```
// variable globale définie dans math.c et utilisée dans main.c
extern int c;

// Déclaration de la fonction pow, qui calcule base^exp
int pow(int base, int exp);
```

```
// définit la constante pi
#define PI   3.1416
```

Pour différencier la déclaration et la définition d'une variable, on ajoute le mot-clé *extern* devant la déclaration, pour préciser que cette variable est définie en dehors du fichier où le fichier d'en-tête est inclus. Nous avons décrit cette possibilité de partager des variables globales entre plusieurs fichiers sources, mais il est en général déconseillé de l'utiliser (c'est souvent une indication que le problème a été mal découpé).

La dernière ligne de *myMath.h* est une macro de pré-compilation (comme *#include*). Elle permet ici de définir la constante π de manière symbolique, sans utiliser d'espace mémoire.

Une macro de pré-compilation est une commande que le compilateur exécute avant la compilation proprement dite. Il s'agit en fait de commandes d'édition du code :

- La macro *#define* indique au compilateur qu'il doit rechercher son 1er argument (PI dans notre exemple) dans le code et le remplacer par son 2ème argument (3,14), exactement comme la commande *remplacer* d'un éditeur de texte.
- De même, la macro *#include* indique qu'il faut la remplacer par le contenu du fichier en paramètre.

Il existe d'autres macros de pré-compilation, mais nous ne les aborderons pas dans ce document.

Interruptions

Les interruptions sont souvent utilisées dans les systèmes embarqués, elles permettent de réagir rapidement aux évènements urgents :

- Dans le cas du *polling*, le délai entre l'apparition de l'évènement et sa détection par le programme varie entre 0 et le temps d'exécution de la boucle principale (on appelle ce délai la latence).
- Dans le cas d'une interruption, la latence est constante, c'est le temps nécessaire à la sauvegarde du contexte, qui est relativement court.

L'interruption garantit donc une latence fixe pour un évènement, ce qui est particulièrement utile si la boucle principale est longue.

Si la boucle principale est courte, le polling peut être plus rapide qu'une interruption. En particulier, il n'y a, en général, pas d'intérêt à avoir une boucle principale vide et une ISR qui gère un évènement.

Echange de données avec une ISR

Les ISR n'acceptent pas de paramètres et ne renvoient aucune valeur. L'échange de données entre la boucle principale et une ISR doit donc passer par des variables globales.

Dans ce cas, il faut prendre quelques précautions liées au caractère asynchrone des ISR. Imaginons l'exemple suivant :

- Notre système commande un processus dont la régulation doit être faite toutes les 10ms.
- Comme on a choisi une régulation prédictive non-linéaire, performante mais gourmande en calcul, le code qui l'implémente s'exécute en 8ms.
- La consigne du régulateur est fournie au système par une liaison UART à 57600 bauds. Il s'agit d'un *long*, il est donc envoyé en 4 trames UART, en commençant par l'octet de poids faible. On n'a aucune information sur le timing de l'envoi de la consigne.

La régulation étant une tâche qui dure longtemps, on décide de la traiter dans la boucle principale, en utilisant le Timer2 en polling comme base de temps.

Une trame UART de 11 bits à 57600 bauds dure 191µs, il est donc impossible de gérer la communication UART dans la boucle principale car celle-ci dure beaucoup trop longtemps, on décide donc de la gérer par interruption. On obtient le code suivant pour la boucle principale et l'ISR :

```
long consigne = 0;    // par défaut, la consigne vaut 0


int main(void)
{
    regulatorInit();
    uartInit();
    timer2Init();    // initialise la base de temps à 10ms

    while (1) {
        if (IFS0bits.T2IF) {
            IFS0bits.T2IF = 0;
            regulatorExecute(consigne);
        }
    }
}


void _ISR _U1RXInterrupt(void)
{
    static int byte = 0;    // indique l'octet de la consigne attendu (0 -> 3)
    static long data = 0;   // variable temporaire pour reconstruire la donnée

    IFS0bits.U1RXIF = 0;
    switch(byte) {
        case 0:
            byte = 1;
            data = (long)U1RXREG;
            break;
        case 1:
            byte = 2;
            data |= ((long)U1RXREG) << 8;
            break;
        case 2:
            byte = 3;
            data |= ((long)U1RXREG) << 16;
```

```
            break;
        case 3:
            byte = 0;
            consigne = data | ((long)U1RXREG) << 24;
            break;
    }
}
```

Ce code présente 2 problèmes potentiels liés à l'usage partagé de *consigne* par la boucle principale et l'ISR :

- Certains compilateurs, en voulant optimiser le code, peuvent décider qu'il n'est pas nécessaire d'aller relire *consigne* à chaque appel de *regulatorExecute()* , puisqu'elle n'est pas modifiée dans la boucle principale.

  On peut facilement résoudre ce problème en ajoutant le qualificatif *volatile* au début de la définition de la variable. Cela indique au compilateur que la valeur de la variable peut changer n'importe quand.

- Le second problème se produit lorsque la boucle principale est interrompue pendant qu'elle lit *consigne*. En effet, lire un *long* n'est pas une opération atomique, elle est divisée en 2 lectures de *int* (puisque le dsPIC est un µC 16 bits). Il peut donc se produire le scénario suivant :
  - La boucle principale commence à exécuter la ligne :

```
regulatorExecute(consigne);
```

  - Pour cela, elle commence par lire la valeur de *consigne* en mémoire, en lisant le mot de poids faible.
  - Durant cette instruction, l'interruption de l'UART est activée par la réception du dernier octet d'une nouvelle consigne.
  - Le µC termine donc la lecture en cours.
  - Ensuite, l'ISR est exécutée, elle modifie *consigne.*
  - Le µC reprend l'exécution de la boucle principale, l'instruction suivante va lire le mot de poids fort de *consigne.*
  - La valeur que la boucle principale a lue est donc composée du mot de poids faible de l'ancienne valeur de *consigne* et du mot de poids fort de sa nouvelle valeur.

  La valeur qui sera passée à *regulatorexecute()* peut donc être complètement fausse. Pour éviter cela, on peut imaginer 2 solutions :

  - Utiliser une section critique
  - Utiliser un *ping-pong buffer*

Modification de la boucle principale

Pour éviter ces problèmes, nous allons utiliser la règle de bonne pratique qui consiste à ne pas accéder directement à la variable globale dans la boucle principale et à utiliser une fonction pour la lire :

```
long getConsigne(void);    // prototype de la fonction de lecture de la consigne

int main(void)
{
    regulatorInit();
    uartInit();
```

```
    timer2Init();    // initialise la base de temps à 10ms


    while (1) {
        if (IFS0bits.T2IF) {
            IFS0bits.T2IF = 0;
            regulatorExecute(getConsigne());
        }
    }
}
```

Cette fonction, l'ISR et la variable globale seront définies ensemble pour éviter les problèmes décrits ci-dessus.

Section critique

Le principe de la section critique est de désactiver l'interruption pendant la lecture, puis de la réactiver. Cela assure que la variable ne sera pas modifiée par l'ISR pendant la lecture.

C'est la méthode la plus souvent utilisée.

Néanmoins, avec cette technique, l'ISR n'est pas active tout le temps et on ne peut donc plus garantir que la latence de l'ISR est constante. Dans certaines applications, cela peut être un problème.

Si on l'applique à notre exemple, on obtient le code ci-dessous.

```
volatile long consigne = 0;    // par défaut, la consigne vaut 0


long getConsigne(void)
{
     long data;


    IEC0bits.U1RXIE = 0;    // Désactivation de l'interruption
    data = consigne;        // Lecture de la variable partagée
    IEC0bits.U1RXIE = 1;    // Réactivation de l'interruption
    return (consigne);
}
```

Dans ce cas, l'ISR ne change pas.

Ping-pong buffer

Cette technique consiste à utiliser 2 variables partagées, un pouvant être lue par la boucle principale et l'autre dans laquelle l'ISR peut écrire. Lorsque l'ISR a reçu une nouvelle valeur et a fini de l'écrire, on échange les rôles des 2 variables :

```
volatile long consigne0 = 0;    // par défaut, la consigne vaut 0
volatile long consigne1 = 0;    // par défaut, la consigne vaut 0
volatile int bufferToRead = 0; // flag indiquant quel buffer peut être lu


long getConsigne(void)
{
    if (bufferToRead == 0) {
        return (consigne0);
    } else {
        return (consigne1);
    }
}


void _ISR _U1RXInterrupt(void)
{
    static int byte = 0;    // indique l'octet de la consigne attendu (0 -> 3)
    static long data = 0;   // variable temporaire pour reconstruire la donnée

    IFS0bits.U1RXIF = 0;
    switch(byte) {
        case 0:
            byte = 1;
            data = (long)U1RXREG;
            break;
        case 1:
            byte = 2;
            data |= ((long)U1RXREG) << 8;
            break;
        case 2:
            byte = 3;
            data |= ((long)U1RXREG) << 16;
            break;
        case 3:
            byte = 0;
```

```
            if (bufferToRead == 0) {
                consigne1 = data | ((long)U1RXREG) << 24;
                bufferToRead = 1;
            } else {
                Consigne0 = data | ((long)U1RXREG) << 24;
                bufferToRead = 0;
            }
            break;
    }
}
```