Initiation à la programmation sur

Digital Signal Controller

CREMMEL Marcel

Lycée Louis Couffignal

STRASBOURG

Initiation à la programmation sur dsPIC

Objectifs :
- ➢ Apprentissage de l'environnement de développement MPLAB
- ➢ Introduction à la programmation de microcontroleurs en langage C
- ➢ Analyse de petits programmes en "C"

1. Généralités sur l'élaboration d'un programme

On ne confie pas l'élaboration d'un programme complet à un électronicien. Mais il doit assimiler quelques notions pour être capable de :

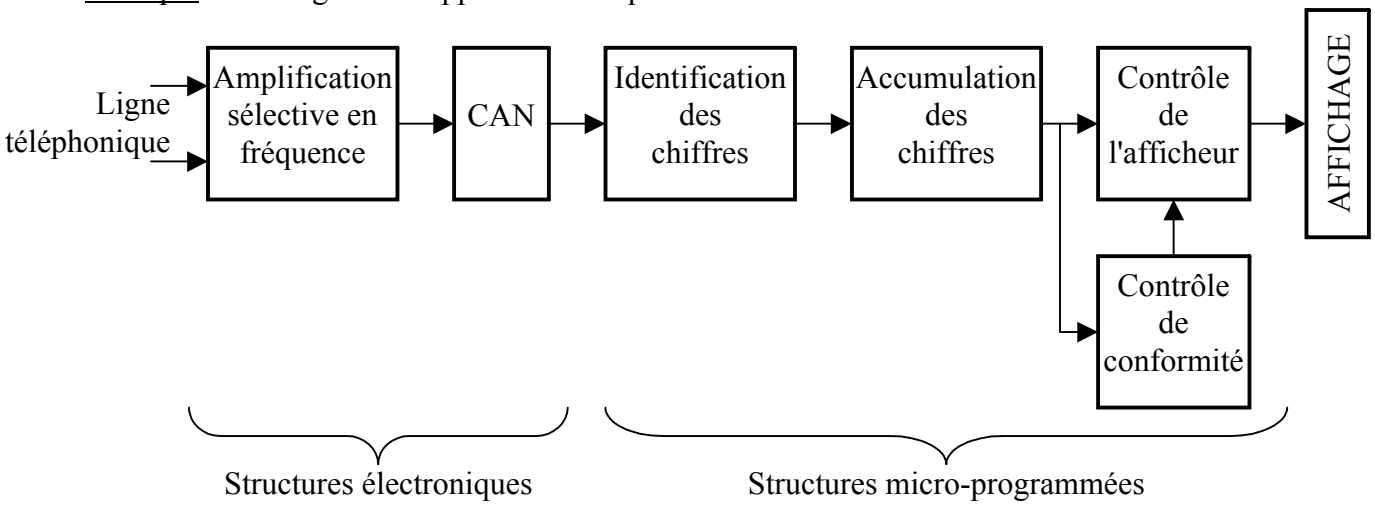
- lire et comprendre des segments de programme liés étroitement aux structures matérielles intégrées ou associées au microcontroleur (entrées-sorties binaires, gestion de capteurs et d'actionneurs, CAN et CNA, claviers, afficheurs, écrans, …),
- modifier ou écrire ce type de segment de programme,
- et surtout de tester et valider ces segments isolément et au sein du programme principal.

L'élaboration d'un programme est divisée en plusieurs étapes :

1. **Analyse des fonctions principales à réaliser :**
   Le travail consiste à décomposer chacune des fonctions principales en fonctions secondaires suffisamment simples. Les descriptions de ces F.S. sont détaillées pour permettre leurs élaborations sans maîtriser l'amont et l'aval et ainsi travailler efficacement en équipes.
   On décide aussi dans cette phase de la répartition des fonctions :
   - réalisées par le "hard",
   - réalisées par le "soft",
   - réalisées de façon mixte (hard + soft)

   Du point de vue du traitement du signal par une fonction, il y a une distinction importante entre les réalisations "électronique" (notamment analogique) et micro-programmées :
   - Dans une structure électronique, toutes les fonctions sont opérationnelles simultanément (dans un amplificateur par exemple).
   - Une structure micro-programmée ne comporte qu'un seul processeur et celui-ci ne peut traiter qu'une seule fonction à la fois. Cela peut poser problème si plusieurs fonctions sont chargées de réagir "instantanément" à des événements extérieurs : il faut imaginer par exemple toutes les fonctions réalisées par l'UC embarquée dans une voiture qui doivent réagir aux ordres de l'automobiliste, gérer le fonctionnement du moteur, s'occuper de l'ABS etc.

   Les schémas fonctionnels des parties micro-programmées sont en fait des diagrammes temporels où les fonctions traitent le signal ou l'information dans un ordre déterminé.
   Exemple : affichage du n° appelant en téléphonie :



La structure électronique fournit un flux numérique continu représentatif du signal analogique prélevé sur la ligne téléphonique.
La fonction d'identification des chiffres réalise une démodulation de ce signal et fournit, s'il existe, le code du chiffre ou de la lettre identifié.
Ce caractère est accumulé dans une mémoire par la fonction suivante pour reconstituer le n° appelant.
Quand le dernier chiffre a été accumulé, le numéro de téléphone est transmis aux 2 fonctions "Contrôle de conformité" et "Contrôle de l'afficheur". Cette dernière activera l'affichage si aucune erreur est détectée.
On constate que l'information est traitée en étapes temporelles à partir de la fonction "identification des chiffres" car le numéro appelant est transmis séquentiellement. Ces fonctions peuvent donc être réalisées par une structure programmée.

Note : la fonction "Identification des chiffres" doit être opérationnelle à tout moment (un nouveau chiffre peut se présenter alors que le précédent est encore traité par la fonction d'accumulation par exemple). Comment, dans ce cas, le processeur peut-il s'occuper des autres fonctions ? En fait, il lui reste du "temps libre" entre 2 échantillons successifs fournis par le CNA. Ce temps est utilisé pour réaliser les autres fonctions. Ce genre de traitement est facilement mis en œuvre par l'utilisation des interruptions.

2. Choix de la structure matérielle du système micro-programmé :

Suivant les applications, 3 choix sont possibles :

- Microcontroleur avec CPU, ROM, RAM et périphériques adaptés à l'application intégrés sur la même puce. Les constructeurs proposent un choix très vaste : on trouve des microcontroleurs à 8 broches pour les applications les plus simples (thermostat par exemple) jusqu'à plus de 100 broches s'il l'application exige un LCD graphique.
  Le choix dépend de l'application, de la vitesse de traitement, de la consommation, du prix et des habitudes de l'entreprise.
- DSP = Digital Signal Processor. On choisit ces processeurs quand une forte puissance de calcul est nécessaire (ex : codeurs/décodeurs MP3 ou DVD). Ils sont coûteux et gourmands en énergie.
  Le dsPIC de Microchip est un mixage de µC et de DSP moyennement performant. Il est bien adapté aux applications "audio" et de commandes de moteurs asservis.
- Ordinateur type PC : c'est une plate-forme universelle, capable de réaliser toutes les fonctions imaginables, mais pas toujours adaptée aux applications. Il existe des PC "industriels" robustes, conçus pour fonctionner dans un environnement agressif.

3. Choix du langage et de l'outil de développement :

Au final, le µP exécute toujours une suite d'instructions "machines" (ou "assembleur") placé dans la mémoire programme à des adresses consécutives.
A l'époque des pionniers, on affectait presque manuellement chaque case de la mémoire par les instructions du programme ! La mise au point était très fastidieuse au point d'abandonner souvent en cours de route.
Pour limiter la fuite des clients découragés, les fabricants ont rapidement proposé des outils de développement permettant de raccourcir le temps de mise au point et aussi de faciliter l'élaboration des "gros" programmes en équipes (ex : Windows XP sur Pentium).

Ces outils permettent, avec un ordinateur de bureau :

- de **rédiger** le programme dans un éditeur de texte avec toutes ses facilités (copier/coller – fichiers inclus – etc …). On utilise un des langages décrits ci-dessous.
- de **compiler** ce fichier "texte" avec le compilateur pour produire un fichier "binaire". Ce dernier regroupe l'ensemble des instructions machine du programme telles qu'elles seront placées dans la mémoire du microcontroleur. Le transfert dans cette mémoire est réalisé par un programmateur.
- de **simuler** le programme à l'aide du simulateur avant son transfert effectif dans la mémoire du microcontroleur.
  Cette étape est obligatoire, car la première écriture d'un programme comporte toujours des erreurs (bug=cafard), même quand il est très simple.
- de **debugger** le programme implanté dans l'environnement réel ("in circuit") avec le "debugger".

Les langages proposés sont (pour les applications à base de µC) :

– l'**assembleur** (ou langage machine) : on est au plus près du µP et on utilise les mnémoniques de ses instructions que l'assembleur va traduire en code "machine".
Exemple sur dsPIC :

Mnémonique des instructions	Travail de l'assembleur	Code machine (placé en mémoire programme organisée en mots de 24 bits)	Adresse (16 bits)
mov #10,W0	→	0x2000A0	0x1E00
mov W0,0x0802	→	0x884010	0x1E02

On constate qu'une instruction est codée sur un seul mot de 24 bits dans la mémoire programme. Ce type de codage permet d'augmenter la vitesse de traitement en réduisant le nombre d'accès mémoire par instruction.

On utilise le langage assembleur quand les temps d'exécution et la taille du programme sont critiques.

– **le C :** c'est le langage le plus utilisé (Windows XP est écrit en C).
L'ingénieur ou le technicien programmeur n'a pas besoin de connaître le jeu d'instruction du µP utilisé pour réaliser un programme performant (rapide et compact). Il utilise des constantes, des variables, des opérateurs, des structures de programmations et d'E/S standardisés. Le compilateur C, généralement fourni par le fabricant du µP, se charge de traduire le programme en langage "assembleur" puis en code machine.
Ainsi, un programme écrit en C pour un certain µP peut être facilement transposé à un autre µP, même d'un autre fabricant.
Les problèmes subsistants sont généralement liés aux structures des périphériques spécifiques à chaque µC. Mais le temps de conversion est beaucoup plus court que s'il fallait réécrire tout le programme avec de nouvelles instructions.

La difficulté pour nous, électroniciens, est la maîtrise des finesses du langage. Mais c'est le travail des informaticiens ! Les compétences d'un électronicien se limitent aux quelques notions nécessaires à l'**analyse et à la modification de fonctions microprogrammées simples** et liées à des structures matérielles (E/S, interruptions, CAN, CNA, liaisons séries, …).

Exemple de compilation C sur dsPIC :

Lignes de programme en C	Travail du compi-lateur C	Traduction en assembleur	Travail de l'assem-bleur	Code machine (24 bits)	Adr. (16 bits)
`VAR2=VAR2+VAR1;`	→	`mov.w 0x0806,0x0002`	→	`0x804031`	`0x0182`
		`mov.w 0x0804,0x0000`		`0x804020`	`0x0184`
		`add.w 0x0002,0x0000,0x0000`		`0x408000`	`0x0186`
		`mov.w 0x0000,0x0806`		`0x884030`	`0x0188`
`VAR_LOC =VAR2<<2;`		`mov.w 0x0806,0x0000`		`0x804030`	`0x018A`
		`sl 0x0000,#2,0x0004`		`0xDD0142`	`0x018C`

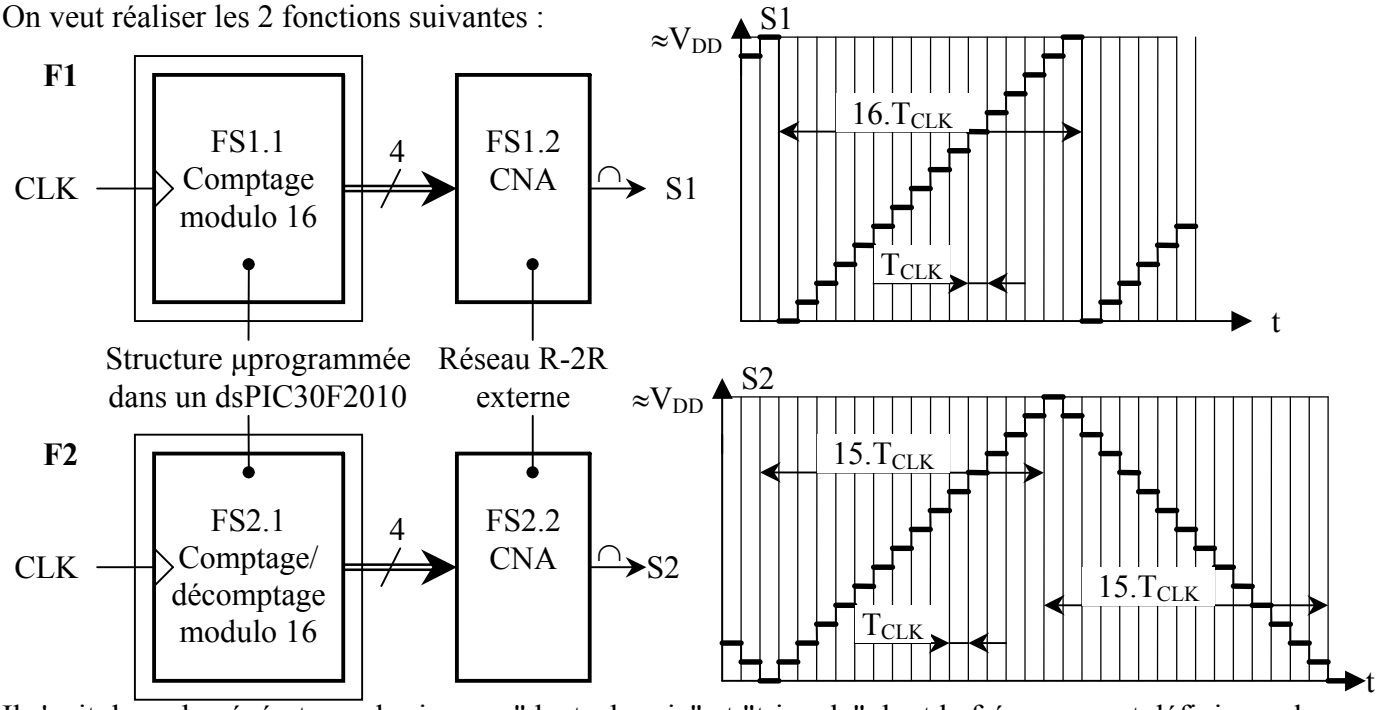
Notes : dans cet exemple :

- VAR1 et VAR2 sont les identificateurs de 2 variables de taille 16 bits placées en RAM aux adresses respectives 0804h et 0806h
- VAR_LOC est une variable 16 bits placée dans le registre W4 (adresse 0x0004) du CPU (le dsPIC comporte 16 registres de travail, nommés W0 à W15).
  En fait, le programmeur en C n'a le plus souvent pas à se soucier de ces détails techniques (sauf la taille des variables). Le compilateur fait généralement les meilleurs choix d'affectation en mémoire des variables nécessaires au programme (dans l'exemple donné, la variable VAR_LOC est placée dans un registre CPU ce qui réduit les temps d'accès et augmente par conséquent la vitesse de traitement).

- **les autres langages (PASCAL, BASIC, …)** sont très peu utilisés pour développer des applications à base de microcontroleur.

2. Programmes simples en C

On veut réaliser les 2 fonctions suivantes :



Il s'agit donc de générateurs de signaux "dents de scie" et "triangle" dont la fréquence est définie par le signal CLK externe.

2.1 Compteur_soft_C.c

Il s'agit d'une première version de la fonction F1 dépourvue d'entrée CLK externe. Cette fonction n'a pas de sens car elle n'a pas d'entrée ! C'est une introduction à la réalisation de la vraie fonction au §2.3.

2.1.1 Création du projet "Compteurs" avec l'environnement de développement MPLAB :

→ *Suivre la procédure décrite dans le document ressources "Aide mémoire MPLAB C30"*

→ *Créer et ajouter au projet le fichier "`Compteur_soft_C.c`"*

→ *Écrire le programme du §2.1.2*

2.1.2 Analyse du "source"

```
#include  <p30f2010.h>

int main(void)
{
 TRISE=0xFFF0;
 while (1)
 {
  LATE = LATE + 1;
 }
}
```

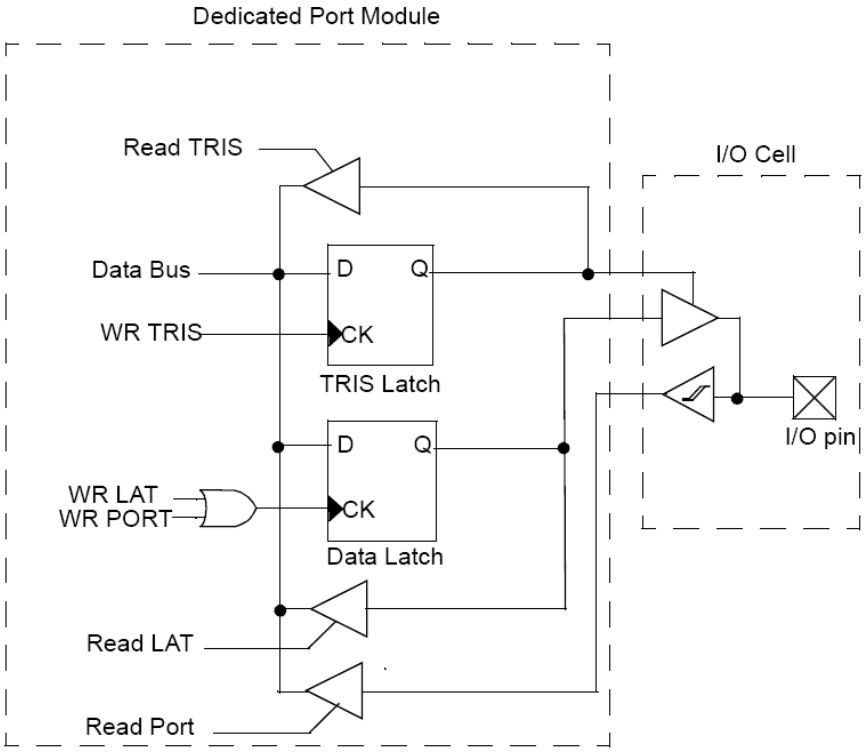
- La première ligne indique au compilateur d'inclure le fichier "`p30f2010.h` ". Ce fichier texte respecte la syntaxe "C" et définit toutes les constantes liées au µC utilisé : entre autres les adresses des registres de configuration et des ports d'E/S. Par exemple :
  - TRISE représente le contenu (16 bits) de l'adresse `0x02D8` pour un dsPIC30F2010
  - LATE représente le contenu (16 bits) de l'adresse `0x02DC` pour un dsPIC30F2010
- La structure

```
void main(void)
{
}
```

définit le bloc du programme principal (= main) exécuté au "reset". Le mot "main" ne peut pas être remplacé par un autre. Si cette fonction se termine (pas de boucle sans fin) → "reset".
Note : les accolades `{ }` sont toujours utilisées par paire pour définir le début et la fin d'un bloc de lignes de programme.

- `TRISE=0xFFF0;` Il s'agit d'une affectation : la case mémoire TRISE d'adresse 0x02D8 est affectée avec la valeur 0xFFF0. Noter le ";" qui est le terminateur de la ligne de programme.
- Structure :
```
while (1)
      {
      }
```
Le µP exécute en boucle (indéfiniment) les lignes du bloc de programme délimité par les 2 accolades car le test (1) est toujours vrai (toute valeur différente de 0 est synonyme de "vrai").
- Le bloc de la structure "`while`" ne comporte qu'une ligne de programme :
```
LATE = LATE + 1;
```
D'un point de vue mathématique : elle est fausse ! En fait, il ne s'agit pas d'une égalité, mais d'une affectation : le registre `LATE` (16 bits) est affecté avec la valeur actuelle de `LATE` augmentée de 1.
En résumé : à droite de "=" : valeurs actuelles
à gauche de "=" : valeur future après calculs

Pour bien comprendre le programme, il faut décrire ce que représentent TRISB et LATB dans le µC :



Chaque broche d'E/S du µC comporte la structure dessinée ci-contre.
Les signaux de contrôle "Read TRIS", "WR TRIS", "WR_LAT", "WR PORT", "Read LAT" et "Read Port" sont communs pour toutes les broches (jusqu'à 16) d'un port (A, B, C …). Ils sont produits par une structure de décodage d'adresse qui reconnaît les adresses correspondantes.

Exemple : si "I/O pin" correspond à la broche RE3 (bit 3 du port E) :
- Data Bus = D3 interne
- WR TRIS est activé par une écriture à l'adresse TRISE=0x2D8
- WR LAT est activé par une écriture à l'adresse LATB=0x2DC
- Read LAT est activé par une lecture à l'adresse LATE=0x2DC
- Read Port est activé par une lecture à l'adresse PORTE=0x2DA

Les sorties des ports du µC sont de type "3 états" :
- états logiques "0" ou "1" quand le buffer ▷ est actif (TRISEbits.x = "0")
- état haute impédance ("HiZ") quand le buffer est inactif (TRISEbits.x = "1")

L'indice "x" représente le bit concerné du port (0 à 15, mais tous sont rarement utilisés).
Les états logiques en mode "sortie" sont ceux des bascules "Data Latch"

A noter que les états des broches sont lus via un buffer à trigger de Schmitt.

→ *Comment est configuré le port E après l'exécution de la ligne "`TRISE=0xFFF0;`" ?*
→ *Comment évoluent les sorties du port E à l'exécution de la ligne "`LATE=LATE+1;`".*

2.1.3 Simulation du programme

→ *Vérifier que le "debugger" de l'environnement de développement MPLAB est en mode "simulation"*
→ *Compiler le programme et corriger les éventuelles erreurs*
→ *Ouvrir une fenêtre "watch" pour observer les états des registres TRISE, LATE et PORTE en binaire.*
→ *Placer un point d'arrêt sur la 1° ligne de la fonction "main" et lancer la simulation.*
→ *Vérifier la fonction du programme par une simulation en mode "pas à pas" et "run".*
→ *Commentez les valeurs max des registres LATE et PORTE*

2.1.4 Version "assembleur" du programme

Le µP exécute en fait une suite d'instructions placées dans la mémoire flash. Cette suite d'instruction est le résultat de la compilation.
La version du programme traduite en "assembleur" peut être visualisée en ouvrant une fenêtre "Disassembly Listing" (menu "view").

→ *Compléter alors le tableau suivant :*

C	Traduction "ASM" des instructions "C"
`TRISE = 0xFFF0;`	

`while (1)`
`{`
`LATE=LATE+1;`
`}`	

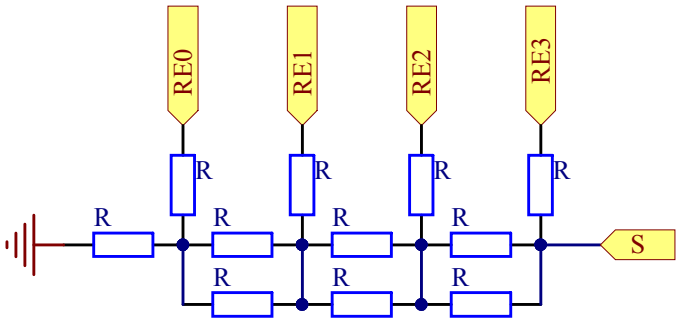
→ *Décrire les opérations réalisées par chaque instruction en détails (sauf l'instruction "lnk")*
→ *Donner ci-dessous le contenu de la mémoire "flash" affectée avec ce programme*

Adresse (16 bits)	**Contenu (24 bits)**	**Instruction**
`0x0180`	`0xFA0000`	`lnk #0x0`

2.1.5 Test du programme sur la plaquette

→ *Alimenter sous 5V la plaquette équipée d'un dsPIC30F2010 et d'un MAX202*
→ *Lancer le "bootloader" Ingenia et établir la communication avec de dsPIC*
→ *Programmer le µC avec le fichier "Compteurs.hex", résultat de la compilation*
→ *L'application est lancée quelques secondes après un nouveau "reset" (temps d'attente du "bootloader")*
→ *Vérifier alors la réalisation de la fonction*
→ *Justifier la fréquence mesurée sur RE0*

→ *Câbler le réseau R-2R suivant :*



On prendra R = 39kΩ ou une valeur proche.

Si les résistances sont toutes égales :

$$V_S = \frac{V_{DD}}{2^4} \times \left(RE3 \times 2^3 + RE2 \times 2^2 + RE1 \times 2^1 + RE0 \times 2^0 \right)$$ avec REx = 0 ou 1

→ *Observer l'évolution de Vs.*
Mesurer la durée d'un palier. Sont-ils tous de durée égale ?
Mesurer la fréquence de la "dent de scie"

2.2 Compteur_soft_bis_C.c

On met en œuvre ici une autre structure algorithmique pour réaliser la même fonction :

```
#include  <p30f2010.h>
int i;
int main(void)
{
  int i;
  TRISE=0xFFF0;
  while (1)
  {
   for (i=0;i!=16;i++)
    {
      LATB=i;
    }
  }
}
```

→ *Créer et éditer le fichier "`Compteur_soft_bis_C.c`"*
→ *Affecter ce fichier au projet et compiler le projet*

2.2.1 Variable "i"

Le programme utilise une variable identifiée "i" (on peut changer le nom). Elle est déclarée par la ligne: `int i;` "int" indique au compilateur que la variable est un entier signé de taille 16 bits (pour une taille de 8 bits on utilise "char").
Cette variable doit être mémorisée en RAM. Le compilateur C connaît le plan mémoire du µC cible (grâce au fichier p30f2010.gld) et place cette variable à une adresse convenable,

→ *A quelle adresse est mémorisée la variable i ? Est-ce une adresse correcte ?*
→ *Constater sur la maquette que les vitesses d'exécution des 2 versions sont sensiblement différentes.*

2.2.2 Boucle itérative : for (exp1;exp2;exp3);

Cette structure permet de faire exécuter un certain nombre de fois le bloc du programme qui suit délimité par les 2 accolades.

→ *Expliciter le rôle des 3 expressions (exp1, exp2 et exp3) par la simulation.*

2.2.3 Modification du programme

→ *Modifier le programme pour réaliser la fonction F2 (voir le début du §2)*
→ *Vérifier le fonctionnement sur la maquette et mesurer les caractéristiques du signal S produit.*
→ *Un palier est un peu plus long que les autres. Expliquer pourquoi.*

2.3 Compteur_C.c

Il s'agit de réaliser la fonction F1 avec une horloge CLK externe appliquée sur l'entrée RD0. Deux solutions sont étudiées pour détecter un flanc montant et réagir à cet événement :

- par scrutation
- par interruption

2.3.1 Solution par scrutation : "Compteur_C_V1.c"

```
#include  <p30f2010.h>
int main(void)
{
 TRISE=0xFFF0;
 while (1)
 {
  while (!PORTDbits.RD0) {} // Les 2 accolades peuvent être supprimées si le
                            // bloc est vide (il faut alors un ";")
  LATE++;
  ________________________  // A compléter
 }
}
```

→ *Compléter le programme*

→ *Ouvrir une fenêtre "Stimulus Controller" et placer 2 lignes pour affecter l'entrée RD0 (à "0" et "1")*

→ *Vérifier la réalisation de la fonction F1 par simulation*

→ *Programmer le dsPIC et vérifier la fonction sur la maquette. L'horloge est réalisée par un générateur de fonction : veiller aux bonnes valeurs de niveaux "H" et "L".*

→ *Évaluer le temps de réaction (observer le "jitter") et la fréquence maximum de fonctionnement.*

Dans cette étude le dsPIC ne réalise que cette fonction. Dans les applications réelles le µC est chargé de dizaines ou de centaines de fonctions différentes en plus de cette fonction de comptage.
Dans un programme **structuré**, **chaque fonction est réalisée par une** "***function***". Comme pour les fonctions à structure matérielle, une *function* en C comporte des entrées, un traitement sur ces entrées et des sorties.
Dans la nouvelle version du programme donnée ci-dessous, la fonction F1 est réalisée par la *function* "Fonction_F1" :

```
#include  <p30f2010.h>

void Fonction_F1(void)
{
 while (!PORTDbits.RD0) {}
 LATE++;
 ____________________  // A compléter
}

int main(void)
{
 TRISE=0xFFF0;
 while (1)
 {
  Fonction_F1();
  //On place ici les autres appels de fonctions que le programme doit traiter
 }
}
```

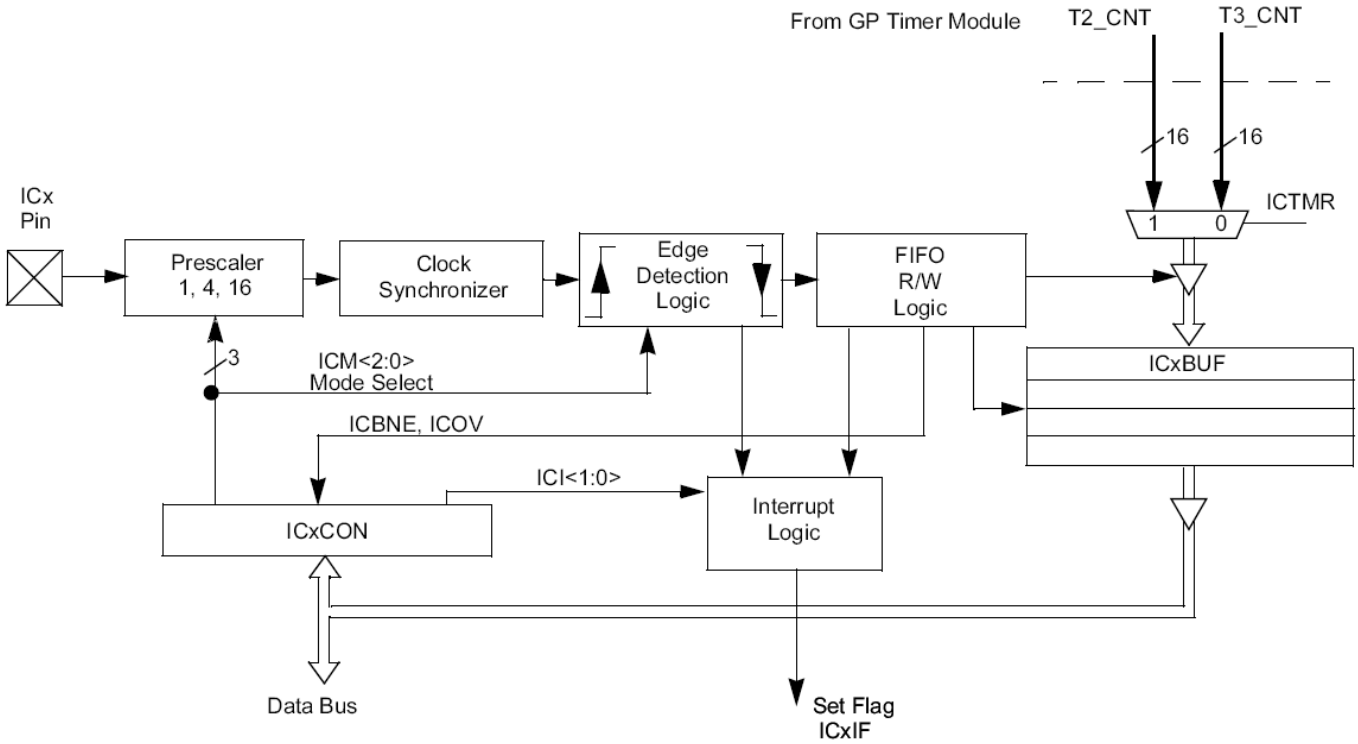
Les 2 mots "void" (vide) qui encadrent le nom de la fonction indiquent l'absence de paramètres en entrée et en sortie. En effet, les E/S de cette fonction sont "matérielles" (broches du µC) et non informatiques.
La fonction est "appelée" en utilisant son nom, les paramètres éventuels placés entre les 2 parenthèses.

→ *Simuler le programme en observant **l'appel** de la fonction (utiliser {}) et son **retour** vers le programme appelant.*

Ce petit exemple démontre le défaut majeur de cette version : si le signal d'horloge appliqué sur RD0 est stable (pas de changement d'état), **la fonction F1 ne se termine jamais** et le dsPIC ne peut réaliser les autres fonctions du programme.

2.3.2 Solution par interruption : "Compteur_C_V2.c"

Cette version est beaucoup plus souple et efficace grâce à l'architecture des dsPIC qui gère les interruptions de façon efficace. Chaque périphérique du µC peut provoquer des interruptions.
Dans cet exemple on exploite une structure matérielle intégrée dans le dsPIC : la fonction "Input Capture" (le dsPIC en comporte 4). Correctement configurée, elle provoque une demande d'interruption sur un flanc montant ou descendant du signal d'entrée IC1 (entrée partagée avec RD0).



La structure "Input Capture 1" (associée à l'entrée IC1) est configurée par les états quelques bits du registre IC1CON :

Register 13-1: ICxCON: Input Capture x Control Register

Upper Byte:							
U-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
—	—	ICSIDL	—	—	—	—	—
bit 15							bit 8

Lower Byte:							
R/W-0	R/W-0	R/W-0	R-0, HC	R-0, HC	R/W-0	R/W-0	R/W-0
ICTMR	ICI<1:0>		ICOV	ICBNE	ICM<2:0>		
bit 7							bit 0

Les effets de chaque bit sont décrits dans le manuel de référence du dsPIC.

→ *Déterminer le contenu du registre IC1CON pour que l'indicateur IC1IF soit positionné à "1" en réaction à un flanc montant sur IC1.*

Source du programme :

```
#include  <p30f2010.h>

void _ISR _IC1Interrupt (void) //ISR = Interrupt Service Routine
{
 LATE++;
 IFS0bits.IC1IF=0;// Raz indicateur interruption
}

int main(void)
{
 U1MODE=0;          // Inhiber l'UART à cause du bootloader (qui utilise IC1)
 TRISE=0xFFF0;
 IC1CON =          ;// Interruption à chaque flanc montant sur RD0=IC1
 IEC0bits.IC1IE=1; // Validation interruption IC1 (TEST)
 while (1)
 {
 }
}
```

Cette version du programme ressemble beaucoup à la version précédente : la *function* "Fonction_F1" semble simplement remplacée par la *function* d'interruption IC1Interrupt.
Mais en regardant de plus près, on constate :

- qu'il n'y a **plus de boucle d'attente dans le traitement de la fonction F1** : le temps CPU n'est pas gâché;
- qu'il n'y a **plus aucun appel à cette fonction dans la boucle sans fin du programme principal !**

La *function* _IC1Interrupt (nom réservé) est en fait une **fonction d'interruption** appelée par la mise à "1" de l'indicateur IC1IF (IC1 Interrupt Flag) alors que le µC exécute son programme normalement. Celui-ci est **interrompu** pour exécuter cette fonction d'interruption. A la fin de son traitement, le programme interrompu reprend comme s'il ne s'était rien passé.
La seule condition est l'ouverture de la porte de validation associée à l'indicateur IC1IF par le bit IC1IE du registre IEC0.

La macro "_ISR" indique au compilateur de placer le vecteur d'interruption de "IC1Interrupt" à la bonne adresse dans la mémoire programme (voir le document "Description des dsPIC").

La fonction F1 est réalisée en partie par le "hardware" et en partie par le "software" :

- détection du flanc montant par la structure "hardware" de "Input Capture 1"
- interruption du programme principal et appel de la fonction "IC1Interrupt"
- traitement par "software" de la fonction d'interruption "IC1Interrupt" (incrémentation du compteur)
- retour au programme principal à la fin du traitement

→ *Compléter le programme*
→ *Simuler le programme et constater la mise à "1" de l'indicateur IC1IF et l'appel de la fonction d'interruption "IC1Interrupt" en réaction au ↑ sur RD0=IC1*
→ *Vérifier la réalisation de la fonction sur la maquette*
→ *Évaluer le temps de réaction (observer le "jitter") et la fréquence maximum de fonctionnement*
→ *Supprimer la ligne "*`IFS0bits.IC1IF=0;`*" et simuler le programme.*
Expliquer le comportement observé.
→ *Résumer les intérêts de cette version*