Capteurs / Info. Embarquée : Programmation en C, interruptions, gestion du Timer

Sebastien.Kramm@univ-rouen.fr

INSA Rouen

2012-2013

(Les n° de page font référence au pdf du PIC24FJ128)

Sommaire

Accès aux registres du microcontroleur en C

- On accède aux registres directement par leur nom (comme en assembleur), via un fichier d'en-tête spécifique à la version utilisée (`p24FJ128GA010.h` dans notre cas).
  - En écriture : `PORTA = 0xf0;`
  - En lecture : `if( PORTA == 0x01 ) ...`
- Problèmes :
  - Comment tester un seul bit parmi 8 (ou 16) ?
  - Comment positionner un bit sans modifier les autres ? (équivalent du `bset`/`bclr` en assembleur)

Manipulation de bits en C

- Le langage C n'offre **pas** d'instruction de manipulation directe de bits.
- On y accède :
  - soit via des opérateurs logiques (AND : `&`, OR : `|`, XOR : `^`)
    Exemples :
    `PORTA = PORTA | 0x0f;` va forçer à 1 les 4 bits de poids faible.
    `PORTA = PORTA & 0xf7;` va forçer à 0 le bit 3.
    `PORTA = PORTA ^ 0x03;` va inverser les 2 bits de poids faible.

Manipulation de bits en C

- Le langage C n'offre **pas** d'instruction de manipulation directe de bits.
- On y accède :
  - soit via des opérateurs logiques (AND : `&`, OR : `|`, XOR : `^`)
    Exemples :
    `PORTA = PORTA | 0x0f;` va forçer à 1 les 4 bits de poids faible.
    `PORTA = PORTA & 0xf7;` va forçer à 0 le bit 3.
    `PORTA = PORTA ^ 0x03;` va inverser les 2 bits de poids faible.

  - soit via des champs de bits (solution recommandée sur Microchip).

Structures de type "champs de bits" prédéfinies

- Le fichier d'en-tête du processeur (`p24FJ128GA010.h`) contient :
  - la déclaration d'une structure C (`struct`) pour chaque registre, dans laquelle les différents bits sont déclarés comme "champ de bit".
    Les champs sont **nommés**, avec un nom lié au registre manipulé.
  - la déclaration d'une variable globale de ce type
    (en `extern`, donc pas l'allocation mémoire).
  - Des définitions de constantes symboliques (`#define ...`)
    permettant d'accéder directement aux broches du port.

Structures de type "champs de bits" prédéfinies

- Le fichier d'en-tête du processeur (`p24FJ128GA010.h`) contient :
  - la déclaration d'une structure C (`struct`) pour chaque registre, dans laquelle les différents bits sont déclarés comme "champ de bit".
    Les champs sont **nommés**, avec un nom lié au registre manipulé.
  - la déclaration d'une variable globale de ce type
    (en `extern`, donc pas l'allocation mémoire).
  - Des définitions de constantes symboliques (`#define ...`)
    permettant d'accéder directement aux broches du port.
- Exemple pour un registre quelconque (de 6 bits ici) :

```
typedef struct tagCNPU2BITS {
  unsigned CN16PUE:1;
  unsigned CN17PUE:1;
  unsigned CN18PUE:1;
  unsigned CN19PUE:1;
  unsigned CN20PUE:1;
  unsigned CN21PUE:1;
} CNPU2BITS;
extern volatile CNPU2BITS CNPU2bits __attribute__((__sfr__));
```

Exemple

- Pour le port A (port générique I/O) :
  - La structure s'appelle `PORTABITS`,
  - Les champs s'appellent `RA0, RA1, ...`,
  - La variable pour y accéder s'appelle : `PORTAbits`.

Exemple

- Pour le port A (port générique I/O) :
  - La structure s'appelle `PORTABITS`,
  - Les champs s'appellent `RA0, RA1, ...`,
  - La variable pour y accéder s'appelle : `PORTAbits`.

- On pourra accéder aux bits via les champs à partir de la variable, avec l'opérateur " ." :

```
PORTAbits.RA0 = 1; // en écriture
if( PORTAbits.RA0 == 0 ) // en lecture
```

Exemple

- Pour le port A (port générique I/O) :
  - La structure s'appelle `PORTABITS`,
  - Les champs s'appellent `RA0, RA1, ...`,
  - La variable pour y accéder s'appelle : `PORTAbits`.

- On pourra accéder aux bits via les champs à partir de la variable, avec l'opérateur " ." :

```
PORTAbits.RA0 = 1; // en écriture
if( PORTAbits.RA0 == 0 ) // en lecture
```

- La constante symbolique pour accèder au bit0 s'appelle `_RA0` :

```
#define _RA0 PORTAbits.RA0
```

- Pour accéder au bit n du port P, on pourra écrire :

```
_RPn = 1;  // en écriture
if( _RPn == 0 )  // en lecture
```

Sommaire

Différence entre prog. "classique" et "embarquée"

- Prog. "classique"

```
int main()
{
//   initialisations
     ...

// traitement  données
    cin >> a;
     ...
    cout << b;

} // fin
```

- Prog. embarquée

```
int main()
{
//   initialisations
     ...

// boucle  infinie
    while( 1 )
    {
        ... // acquisition  évènements
              et traitements
    }

} // fin  jamais  atteinte
```

Différence entre prog. "classique" et "embarquée"

- En réalité, la structure sera souvent de la forme :

```
int main()
{
//   initialisations
    ...

// boucle  infinie
   while( 1 )
      ;  // rien  !
}
```

Différence entre prog. "classique" et "embarquée"

- En réalité, la structure sera souvent de la forme :

```
int main()
{
//   initialisations
    ...

// boucle  infinie
   while( 1 )
      ;  // rien  !
}
```

⇒ Fonctionnement en interruptions (ISR : *Interrupt Service Routine*)

```
void isr_xxx_()
{
// acquittement  interruption
    ...
// traitement  évènement
    ...
}
```

Rappel du principe

- Le fonctionnement en interruptions permet l'exécution automatique de code sur certains évènements, internes ou externes.
- Fonctionnement **asynchrone** par rapport au flot normal du programme.

Rappel du principe

- Le fonctionnement en interruptions permet l'exécution automatique de code sur certains évènements, internes ou externes.
- Fonctionnement **asynchrone** par rapport au flot normal du programme.
- S'oppose à la **scrutation** :

```
while(1)
{
   if(registre == val1 )
      Fonction1();
   if(registre == val2 )
      Fonction2();
    ...
}
```

Rappel du principe

- Le fonctionnement en interruptions permet l'exécution automatique de code sur certains évènements, internes ou externes.
- Fonctionnement **asynchrone** par rapport au flot normal du programme.
- S'oppose à la **scrutation** :

```
while(1)
{
   if(registre == val1 )
      Fonction1();
   if(registre == val2 )
      Fonction2();
    ...
}
```

- Mécanisme implanté sur la puce (*hardware*).
- La liste des sources d'interruptions possibles est précisée dans la table des vecteur d'interruptions.

Table des vecteur d'interruptions (voir p.61)

- Fournit l'information sur :
  - les bits de validation de l'interruption,
  - les flags à réinitialiser dans la routine,
  - les bits gérant la priorité.

TABLE 6-2: IMPLEMENTED INTERRUPT VECTORS

Interrupt Source	Vector Number	IVT Address	AIVT Address	Interrupt Bit Locations		
				Flag	Enable	Priority
ADC1 Conversion Done	13	00002Eh	00012Eh	IFS0<13>	IEC0<13>	IPC3<6:4>
Comparator Event	18	000038h	000138h	IFS1<2>	IEC1<2>	IPC4<10:8>
CRC Generator	67	00009Ah	00019Ah	IFS4<3>	IEC4<3>	IPC16<14:12>
External Interrupt 0	0	000014h	000114h	IFS0<0>	IEC0<0>	IPC0<2:0>
External Interrupt 1	20	00003Ch	00013Ch	IFS1<4>	IEC1<4>	IPC5<2:0>
External Interrupt 2	29	00004Eh	00014Eh	IFS1<13>	IEC1<13>	IPC7<6:4>
External Interrupt 3	53	00007Eh	00017Eh	IFS3<5>	IEC3<5>	IPC13<6:4>
External Interrupt 4	54	000080h	000180h	IFS3<6>	IEC3<6>	IPC13<10:8>
I2C1 Master Event	17	000036h	000136h	IFS1<1>	IEC1<1>	IPC4<6:4>
I2C1 Slave Event	16	000034h	000034h	IFS1<0>	IEC1<0>	IPC4<2:0>
I2C2 Master Event	50	000078h	000178h	IFS3<2>	IEC3<2>	IPC12<10:8>
I2C2 Slave Event	49	000076h	000176h	IFS3<1>	IEC3<1>	IPC12<6:4>
Input Capture 1	1	000016h	000116h	IFS0<1>	IEC0<1>	IPC0<6:4>

En pratique : 3 sections de code à prévoir

1. Initialisations (configuration des interruptions)
   - Validation globale
   - Validation locale
2. Programme principal,
3. Routine(s) d'interruption(s) (*ISR : Interrupt Service Routine*).
   $\Rightarrow$ Doit **acquitter** l'interruption, en "clearant" le flag.
   (Le nom de la routine est prédéfini, voir fichier d'en-tête dédié).

Exemple : détection changement niveau sur broche I/O

- Les ports I/O ont la capacité de générer une demande d'interruption sur un changement de niveau des entrées.
  $\Rightarrow$ fonction *Input Change Notification*, voir p. 102, col. G
- Pour avoir une interruption sur cet évènement, il faut activer le bit de **validation globale** pour les interruptions de ce type
  $\Rightarrow$ voir ligne *Input Change Notification* dans la table.
- Il faut aussi préciser laquelle des entrées CN0 à CN21 va déclencher l'interruption (**validation locale**)
  $\Rightarrow$ voir registres CNEN1 et CNEN2, p.33 & 102.

Sommaire

Présentation

- Objectif d'un *timer* : générer du temps.
- Exemple de contexte d'utilisation, associé aux fonctionnement en interruptions :
  "Execute *telle* fonction dans *tant* de temps".

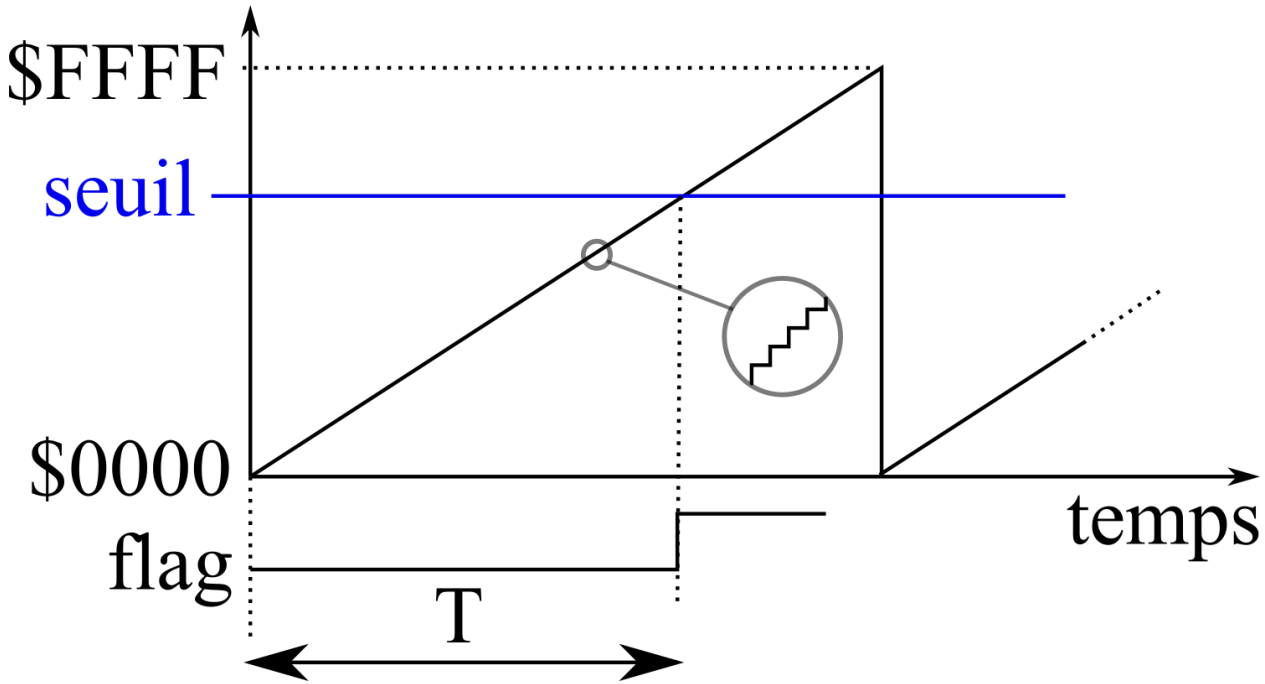
```
int main()
{
   ConfigureTimer( params );
   while( 1 )
      FaireAutreChose();
}

// executée automatiquement au bout dun temps "t"
void _isr_xxx_()
{
    ...
}
```

- Avantage : pas de temps CPU de perdu, le processeur continue d'exécuter le flot normal du programme.

Principe

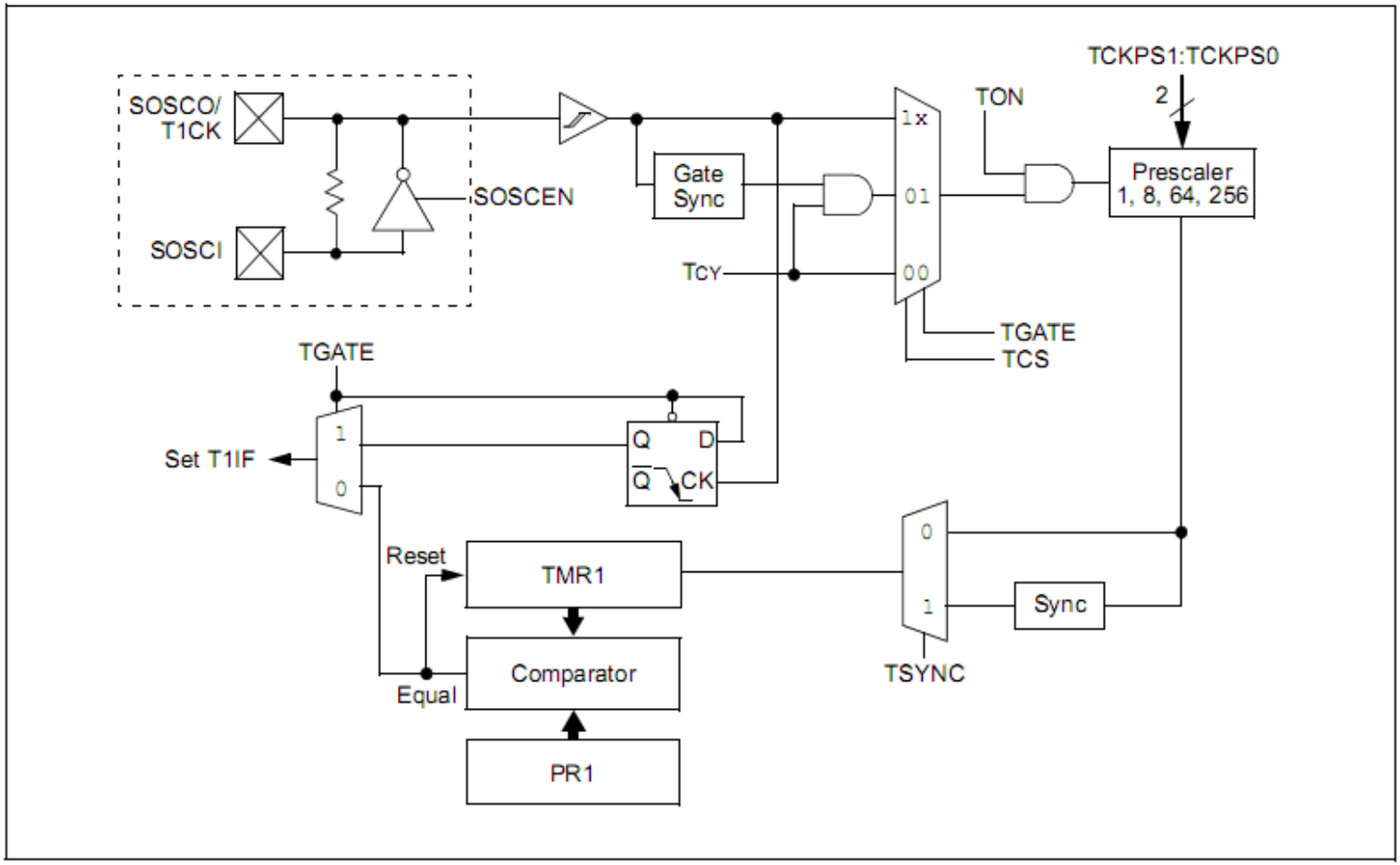
- On associe le temps à une valeur numérique, par le biais d'un compteur binaire piloté par un signal d'horloge.
- Un comparateur binaire active un flag quand il y a égalité entre :
  - la valeur de ce compteur,
  - et la valeur d'un registre (préalablement rempli par le programme utilisateur).
- Ceci est signalé par un flag, qui pourra déclencher une interruption.



Les Timer du PIC24F (p.103)

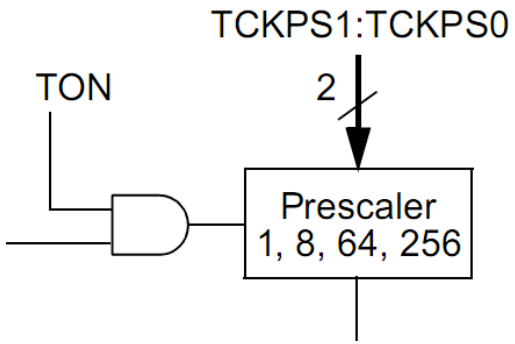
- 5 timers 16 bits, dont certains couplables (32 bits).
- Horloge interne ou externe

FIGURE 10-1: 16-BIT TIMER1 MODULE BLOCK DIAGRAM



En pratique

- On distingue :
  - Le diviseur de fréquence (*prescaler*),
  - Le compteur 16 bits TMR1,
  - Le registre contenant la valeur à comparer (PR1).



- Fonctionnement : Quand TMR1 arrive à la valeur de PR1, ceci :
  - active le flag T1IF,
  - réinitialise TMR1.
- Configuration (phase d'initialisation du prog.) :
  - activer le bit TON ;
  - sélection signal d'horloge ;
  - configuration interruptions sur T1IF.