Fonctionnement des Ordinateurs

TP2 - Développement embarqué sur plateforme MIPS (3ème partie)

B. Quoitin
Faculté des Sciences
Université de Mons

Résumé

Cette séance a deux objectifs. Le premier est l'introduction d'un nouveau type de périphérique : le *timer*. Ce périphérique permet d'une part de mesurer précisément des durées et d'autre part de déclencher des actions après un certain temps ou à intervalle régulier.

Le second objectif est la mise en oeuvre du mécanisme d'interruptions.

Table des matières

1 Timers

1.1 Principe de fonctionnement

Un *timer* permet de générer précisément des événements périodiques. Il peut servir par exemple pour attendre précisément un temps déterminé ou pour mesurer le temps qui s'est écoulé entre deux événements. Un timer est donc un bon moyen pour remplacer la boucle "qui ne fait rien" utilisée dans les programmes développés lors des séances précédentes. Un timer peut également être utilisé de manière asynchrone en combinaison avec le mécanisme d'interruption, ce qui sera couvert à la Section 2.

Un timer est un périphérique matériel du microcontrôleur. Il comprend typiquement les éléments suivants, illustrés à la Figure 1.

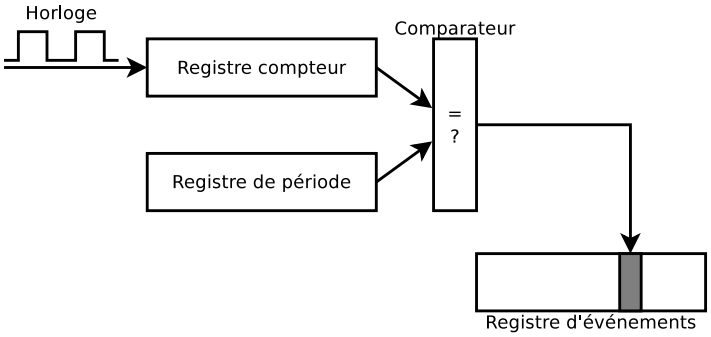


FIGURE 1 – Structure d'un timer

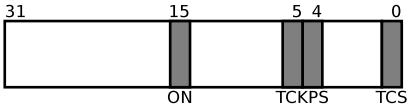
- **Registre de période** : Ce registre détermine la valeur maximum que peut atteindre le registre compteur et donc la « durée » du timer.
- **Registre compteur** : Ce registre est incrémenté de 1 à intervalle régulier, au rythme donné par une horloge. La largeur en bits (p.ex. 16 ou 32 bits) de ce registre détermine la précision du timer.
- **Source d'horloge** : Les transitions du signal d'horloge cadencent l'incrémentation du registre compteur.
- **Comparateur** : Le comparateur détecte si les registres compteur et période sont égaux. Lorsque c'est le cas, le timer a expiré : le registre compteur est remis à 0 et l'expiration du timer est signalée.
- **Registres de configuration** : De la même manière que pour les ports d'entrées/sorties, un timer est configurable via des SFRs accessibles avec les instructions `lw` et `sw`.
- **Flag « événement »** : L'expiration d'un timer peut être testée en vérifiant par programme un bit "flag" dans un registre.

1.2 Timer `T1`

Le microcontrôleur PIC32MX320F128H contient 5 timers nommés `T1` à `T5`. Ces timers ont des caractéristiques différentes (précision, sources d'horloge, etc). Cet exercice repose sur l'utilisation du timer `T1`, un timer dont le registre compteur tient sur 16-bits. La structure de ce timer est illustrée à la Figure 10 en annexe.

Pour utiliser le timer `T1`, il est nécessaire de le configurer. Cette configuration est effectuée au travers de registres SFR spécifiques au timer. En résumé, les opérations suivantes sont nécessaires :

1. Etablir la valeur du registre de période au travers du registre `PR1` (16 bits).
2. Eventuellement pré-charger une valeur dans le registre compteur, via le registre `TMR1` (16 bits). En général, la valeur 0 est chargée dans ce registre.
3. Déterminer la source d'horloge via le registre `T1CON`, à l'aide du bit `TCS` (bit 0). L'oscillateur primaire (à 80MHz) est sélectionné en mettant ce bit à 0.
4. La fréquence d'horloge peut être « pré-divisée » avant d'arriver au timer afin que le registre compteur soit incrémenté moins rapidement. La valeur du pré-diviseur (*prescaler*) est établie via le registre `T1CON`, à l'aide des bits `TCKPS` (bits 4 et 5). Les valeurs `00b`, `01b`, `10b` et `11b` sélectionnent respectivement des divisions par 1, 8, 64 et 256.



5. Remettre à zéro le flag d'expiration via le registre `IFS0`, bit `T1IF` (bit 4).
6. Activer le timer via le registre `T1CON`, bit `ON` (bit 15). Si le bit `ON` vaut 0, le timer est arrêté, sinon il est actif.

1.3 Programmation du timer en C

Les Figures 2 et 3 présentent des fonctions en langage C qui permettent respectivement d'initialiser le timer `T1` et d'attendre son expiration. La fonction `timer_init` se charge d'initialiser le registre `T1CON` de façon à sélectionner l'oscillateur primaire (bit `TCS` = `0`), pré-diviser la fréquence de l'horloge par 8 (bits `TCKPS` = `01b`) et de façon à arrêter le timer (bit `ON` = `0`). La fonction initialise également le registre période (`PR1`) à la valeur 10000.

```
1 void timer1_init()
2 {
3   // ON<15>=0 (timer arrete)
4   // TCKPS<5:4>=01b (1:8),
5   // TCS<1>=0 (oscillateur primaire)
6   T1CON= 0x00000010;
7   PR1= 10000;
8 }
```

FIGURE 2 – Fonction en C permettant d'initialiser le timer `T1`.

La fonction `timer1_wait` permet d'attendre que le timer expire. Cette fonction remet à 0 le registre compteur (`TMR1`). Elle active le timer (bit `ON` du registre `T1CON` = 1). Notez l'utilisation du pseudo-registre `T1CONSET`. Ensuite, la fonction remet à zéro le flag d'expiration `T1IF`. Elle attend ensuite en boucle que ce même flag passe à la valeur 1. Finalement, elle arrête le timer.

Calcul du temps d'expiration. Le temps écoulé entre l'activation du timer et son expiration dépend de deux facteurs. Premièrement, l'écart entre les registres compteur (`TMR1`) et période (`PR1`). Deuxièmement, le rythme auquel les impulsions d'horloge font incrémenter le registre compteur. Ce rythme dépend de la source d'horloge et du pré-diviseur.

Par exemple, supposons que la source interne T_{PBCLK} est utilisée. Cette horloge fonctionne à une fréquence f égale à 80MHz. Supposons également que le pré-diviseur soit configuré de façon à

```
1 void timer1_wait()
2 {
3   TMR1= 0;
4   T1CONSET= 0x00008000; // ON<15>=1
5   IFS0CLR= 0x00000010; // T1IF<4>=0
6   while (!(IFS0 & 0x00000010));
7   T1CONCLR= 0x00008000; // ON<15>=0
8 }
```

FIGURE 3 – Fonction en C permettant d'attendre l'expiration du timer `T1`.

diviser par 64. La valeur initiale du registre compteur est 0. Celle du registre période vaut 5000. Quel est le temps d'expiration ? Ce temps est donné par $64 * (\texttt{PR1} - \texttt{TMR1})/f = 4ms$

1.4 Auto-évaluation

Questions.

— Quelle est le temps attendu par le timer `T1` en supposant qu'il est initialisé par la fonction `timer1_init` et que l'attente soit effectuée avec la fonction `timer1_wait` ?

— Est-il possible d'attendre 1 seconde en utilisant le timer `T1` ?

— Quelle est la durée maximale d'expiration en considérant une fréquence d'horloge égale à 80MHz et les différentes valeurs possibles du *prescaler*.

— Quelle est la plus petite durée non nulle d'expiration possible en considérant une fréquence d'horloge égale à 80MHz et le rôle du *prescaler* ?

— Modifiez le programme qui fait clignoter la LED développé durant la séance précédente de façon à utiliser une attente par timer plutôt qu'une boucle "qui ne fait rien". La LED doit s'allumer durant 100ms, puis rester éteinte durant 900ms. Ce pattern doit se répéter en boucle.

Questions avancées.

— Le langage C permet de faciliter les manipulations de bits ou d'ensemble de bits dans les registres de configuration. Via l'inclusion du fichier d'en-tête `p32xxxx.h`, des variables supplémentaires sont définies. Un exemple particulier est la variable `IFS0bits` qui a un type `struct`. Cette construction permet notamment d'accéder au bit `T1IF` du registre `IFS0` via `IFS0bits.T1IF`. Grâce à cette construction, il n'est pas nécessaire de savoir que `T1IF` est le bit 4. Elle peut également être utilisée pour changer un groupe de bits. Par exemple, `TCKPS` désigne les bits 5 :4 du registre `T1CON`. Ces bits peuvent être accédés directement comme `T1CONbits.TCKPS`, qui apparaît en C comme une variable entière non signée représentée sur 2 bits.
Modifiez votre programme de façon à utiliser cette facilité. De plus, allez consulter le fichier `p32xxxx.h` afin de découvrir la déclaration de `IFS0bits` ou `T1CONbits`.

2 Mécanisme d'interruption

L'utilisation des timers telle que décrite en Section 1 nécessite le test en boucle du flag d'expiration. Pendant l'exécution d'une telle boucle, le processeur ne fait rien d'autre. Cela rend par exemple difficile l'utilisation simultanée de multiples timers alors que le PIC32MX320F128H en possède 5 ! Dans des systèmes « temps-réel », passer un temps trop long dans une boucle peut être problématique et certains événements risquent d'être manqués. Imaginez par exemple que le déclenchement du frein dans une voiture ne soit pris en compte par l'ordinateur de bord qu'après plusieurs centaines de millisecondes !

Une solution alternative au test en boucle réside dans la mise en oeuvre du mécanisme d'interruption. Cette section rappelle quelques éléments du mécanisme d'interruptions de l'architecture MIPS, tout en se plaçant dans le cadre particulier de la plateforme PIC32. La discussion des interruptions est volontairement simplifiée. Un grand nombre de détails techniques pour les étudiants curieux ont été placés en annexe.

2.1 Principe de fonctionnement

Le microcontrôleur PIC32 contient un grand nombre de périphériques. Beaucoup d'entre eux peuvent générer des interruptions matérielles. Or, le coeur MIPS ne peut distinguer que 6 sources d'interruptions différentes. Pour contourner ce problème, les ingénieurs de Microchip ont adjoint au coeur MIPS un contrôleur d'interruptions. Celui-ci se charge d'ordonner les requêtes d'interruption (IRQ — *Interrupt ReQuest*) en provenance des périphériques et de présenter au coeur MIPS une information résumée. L'organisation du système de gestion d'interruptions dans un PIC32 est illustrée à la Figure 5. On y distingue 3 composants principaux : les périphériques générateurs de requêtes d'interruptions, le contrôleur d'interruptions et le coprocesseur `CP0` du coeur MIPS

2.1.1 Coprocesseur `CP0`

Le coprocesseur `CP0` est chargé de la gestion des interruptions dans l'architecture MIPS. Il est situé au sein du coeur MIPS. Pour rappel, le coprocesseur `CP0` contient 3 registres liés à la gestion des exceptions.

— `Cause` (registre 13) : ce registre contient la raison de l'exception. Le champ `ExcCode` contient une valeur numérique indiquant le type de l'exception. Le type 0 correspond à une interruption matérielle.

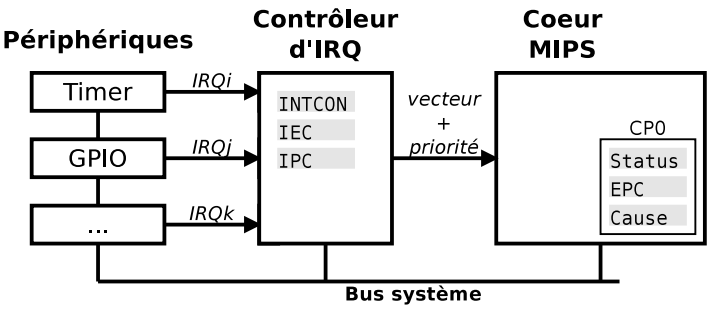


FIGURE 5 – Système de gestion d'interruptions dans un PIC32.

— `Status` (registre 12) : ce registre peut être utilisé pour masquer des exceptions ou pour sauvegarder des exceptions moins prioritaires. Il contient également de l'information sur la source d'une interruption matérielle via les bits `IP2` à `IP7`.

— `EPC` (registre 14) : ce registre contient la sauvegarde du *program counter* (`PC`) en cas d'exception. Ce registre permet de retourner à l'exécution du programme principal une fois l'exception traitée.

2.1.2 Contrôleur d'interruptions

Le contrôleur d'interruptions est chargé de l'arbitrage lorsque plusieurs IRQ sont reçues simultanément. A cet effet, les mécanismes suivants sont mis en oeuvre.

— Un **niveau de priorité** configurable est associé à chaque IRQ. Si plusieurs IRQs sont reçues, celle de plus haute priorité sera traitée en premier. Il y a 7 niveaux de priorité différents.

— Un **numéro de vecteur** est associé à chaque IRQ. Ce numéro de vecteur peut être utilisé pour déterminer l'adresse du gestionnaire d'interruption. Par défaut, dans l'architecture MIPS toutes les IRQs mènent au même gestionnaire d'interruption. Si plusieurs IRQs différentes doivent être distinguées, c'est au logiciel de le faire (voir Figure 4 à gauche). En utilisant le numéro de vecteur d'une IRQ, il est possible d'utiliser plusieurs gestionnaires d'interruptions différents (p.ex. un par IRQ). C'est alors le matériel qui décide vers quel gestionnaire d'interruptions brancher, sur base du numéro de vecteur (voir Figure 4 à droite). PIC32 supporte 64 numéros de vecteur. L'association IRQ / numéro

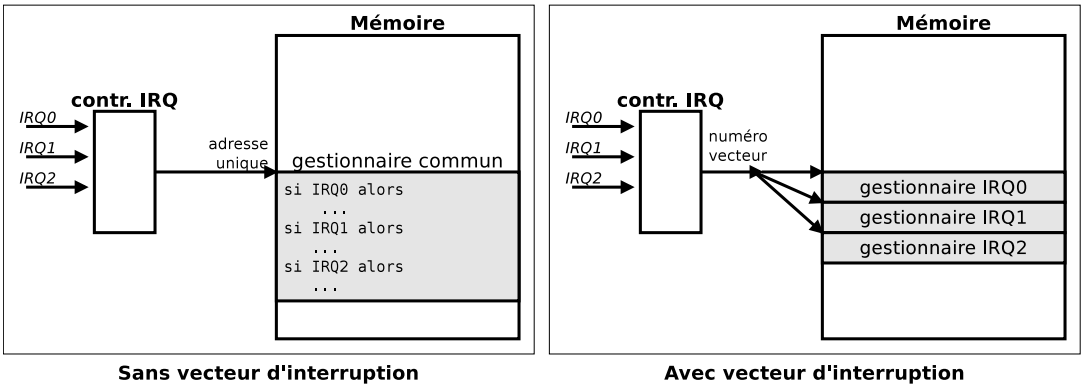


FIGURE 4 – Fonctionnement avec ou sans vecteur d'interruption.

de vecteur n'est pas configurable [1].

Le contrôleur d'interruptions contient plusieurs registres qui permettent de définir la priorité de chaque IRQ (IPC*n* – *interrupt priority control*) et l'activation individuelle de chaque IRQ (IEC*n* – *interrupt enable control*). Ces registres sont décrits de manière plus détaillée en annexe A.3.

2.1.3 Autorisation et configuration des interruptions

La première étape pour la mise en oeuvre des interruptions est l'autorisation globale des interruptions. Le code de la Figure 7 effectue la configuration globale des interruptions. Le code active également le mécanisme des vecteurs d'interruptions que nous souhaitons utiliser. La description détaillée de ce code est fournie à l'Annexe A.2.

```
1  void setup_interrupts()
2  {
3    int val;
4    // Configure les interruptions
5    // (avec vecteurs)
6    asm volatile("mfc0␣␣␣%0,$13" : "=r"(val));
7    val|= 0x00800000;
8    asm volatile("mtc0␣␣␣%0,$13" : "+r"(val));
9    INTCONSET= _INTCON_MVEC_MASK;
10   // Autorise les interruptions
11   asm volatile("ei");
12 }
```

FIGURE 7 – Configuration des interruptions : activation et utilisation des vecteurs d'interruption.

2.2 IRQ du Timer T1

Cette section présente un exemple complet visant à réagir à l'expiration du timer T1 à l'aide du mécanisme d'interruption. Le traitement de l'expiration du timer devient alors asynchrone : le programme ne doit plus vérifier explicitement l'expiration du timer dans une boucle.

Pour rappel, lorsque le timer T1 expire, le flag d'expiration T1IF (*T1 interrupt flag*) est positionné à 1. Si l'interruption de T1 est autorisée, une IRQ est automatiquement générée, le programme principal sera momentanément interrompu, une routine de gestion d'interruption sera exécutée par le processeur, puis l'exécution du programme principal reprendra la où elle avait été interrompue.

1. L'association IRQ / numéro de vecteur est documentée dans la *datasheet* du PIC32.

Le timer T1 génère l'IRQ4 lors de son expiration (flag T1IF vaut 1). Cette IRQ est associée au vecteur numéro 4. Afin d'utiliser cette IRQ, il faut autoriser l'IRQ4, configurer sa priorité et installer un gestionnaire d'interruption pour le vecteur 4. La fonction timer1_init montrée à la Figure 8 est une adaptation de la fonction présentée en Section 1 qui active l'interruption du timer. Une description détaillée de cette fonction est donnée à l'Annexe A.3.

```
1  void timer1_init()
2  {
3    // Initialisation Timer 1
4    // (comme vu auparavant)
5    T1CON= ...;
6    PR1= ...;
7
8    // Configuration de l'interruption T1IF
9    IEC0bits.T1IE= 1;   // Autorisation
10   IPC1CLR= 0x0000001F; // Priorite
11   IPC1SET= 0x0000000C;
12 }
```

FIGURE 8 – Activation des interruptions du timer T1.

La fonction timer1_int_handler présentée à la Figure 6 est le gestionnaire d'interruptions associé au timer T1. Ce qui distingue cette « fonction » d'une fonction classique, ce sont les directives __attribute__((vector(...))) et __attribute__((interrupt(...))) qui la précèdent. Ces directives indiquent notamment qu'il s'agit d'un gestionnaire d'interruption associé au vecteur 4. Une première conséquence de ces directives est que l'adresse de la première instruction de cette fonction sera placée à l'adresse associée au vecteur 4. La seconde conséquence est que l'instruction de retour de la fonction ne sera pas jr $ra mais eret (*return from exception*). ATTENTION, le gestionnaire d'interruption ne doit jamais être appelé par le programme. Les attributs vector et interrupt sont décrits de façon plus détaillée à l'annexe A.4.

Finalement, la Figure 9 illustre comment initialiser interruptions et timer dans la fonction main du programme.

2.3 Auto-évaluation

Questions.

— Ecrivez un programme en langage C qui fait clignoter une LED en utilisant le timer T1 et un gestionnaire d'interruption. Les changements d'état de la LED doivent être effectués à partir du gestionnaire d'interruption. Votre programme doit comporter l'initialisation globale des interruptions, la configuration du timer, y compris l'activation des

```
1  void __attribute__((vector(4))) timer1_int_handler();
2  void __attribute__((interrupt(ipl3soft))) timer1_int_handler();
3  void timer1_int_handler()
4  {
5    LATFINV= 0x00000001;
6    IFS0bits.T1IF= 0; // Remise a zero manuelle flag T1IF
7  }
```

FIGURE 6 – Déclaration d'un gestionnaire d'interruption.

```
1  void main()
2  {
3    setup_interrupts();
4    timer1_init();
5    T1CONbits.ON= 1;
6
7    // Suite du programme ...
8    while (1) {
9      // ...
10   }
11 }
```

FIGURE 9 – Initialisation des interruptions.

interruptions de celui-ci, et un gestionnaire d'interruption.

Questions avancées.

— A l'aide d'`objdump`, observez le code généré pour le gestionnaire d'interruption (`timer1_int_handler`). Pouvez-vous déterminer le rôle des différentes instructions ?

— Toujours à l'aide d'`objdump`, observez la position en mémoire du vecteur d'interruption associé à `T1`. Faites le lien avec la position en mémoire du gestionnaire d'interruption.

A Annexes

A.1 Structure détaillée du Timer `T1`

La Figure 10 présente la structure du Timer `T1` du microcontrôleur PIC32. Il est possible de repérer dans cette figure les différents composants du timer. `TMR1` et `PR1` sont respectivement le registre compteur et le registre de période. Ces registres ont une largeur de 16 bits et, par conséquent, leur valeur maximale est 65535. Le comparateur est situé entre ces derniers. La sortie du comparateur sert notamment à positionner le flag `T1IF` qui indique l'expiration du timer. Le restant du timer sert essentiellement à déterminer la source d'horloge. La source d'horloge peut être l'oscillateur principal du microcontrôleur, désigné par T_{PBCLK} et fonctionnant à 80MHz. Un oscillateur secondaire peut être utilisé, mais il n'est pas implémenté sur la carte ChipKit Uno32. Le signal d'horloge peut être divisé avant d'arriver au compteur de façon à diminuer la fréquence d'incrémentation du compteur. Cette division est effectuée par le bloc nommé *prescaler* (pré-diviseur).

A.2 Configuration globale des interruptions

Cette section montre comment autoriser globalement les interruptions et comment configurer l'utilisation des vecteurs d'interruption. Il s'agit de la première étape pour la mise en oeuvre des interruptions.

1. **Utilisation de vecteurs d'interruption**
   - au niveau du contrôleur d'interruptions : bit `MVEC` (bit 12) du registre SFR `INTCON`.
   - au niveau du coprocesseur `CP0` : bit `IV` (bit 23) du registre `Cause` (registre 13). Lorsque ce bit vaut 1, les vecteurs d'interruption sont interprétés par le coeur MIPS et utilisés pour déterminer l'adresse du gestionnaire d'interruption.
2. **Autorisation globale des interruptions**
   - Les interruptions sont globalement autorisées à l'aide d'une instruction spéciale `ei` (*interrupt enable*).

Le code de la Figure 7 illustre comment les changements décrits ci-dessus peuvent être réalisés dans un programme C. La manipu-

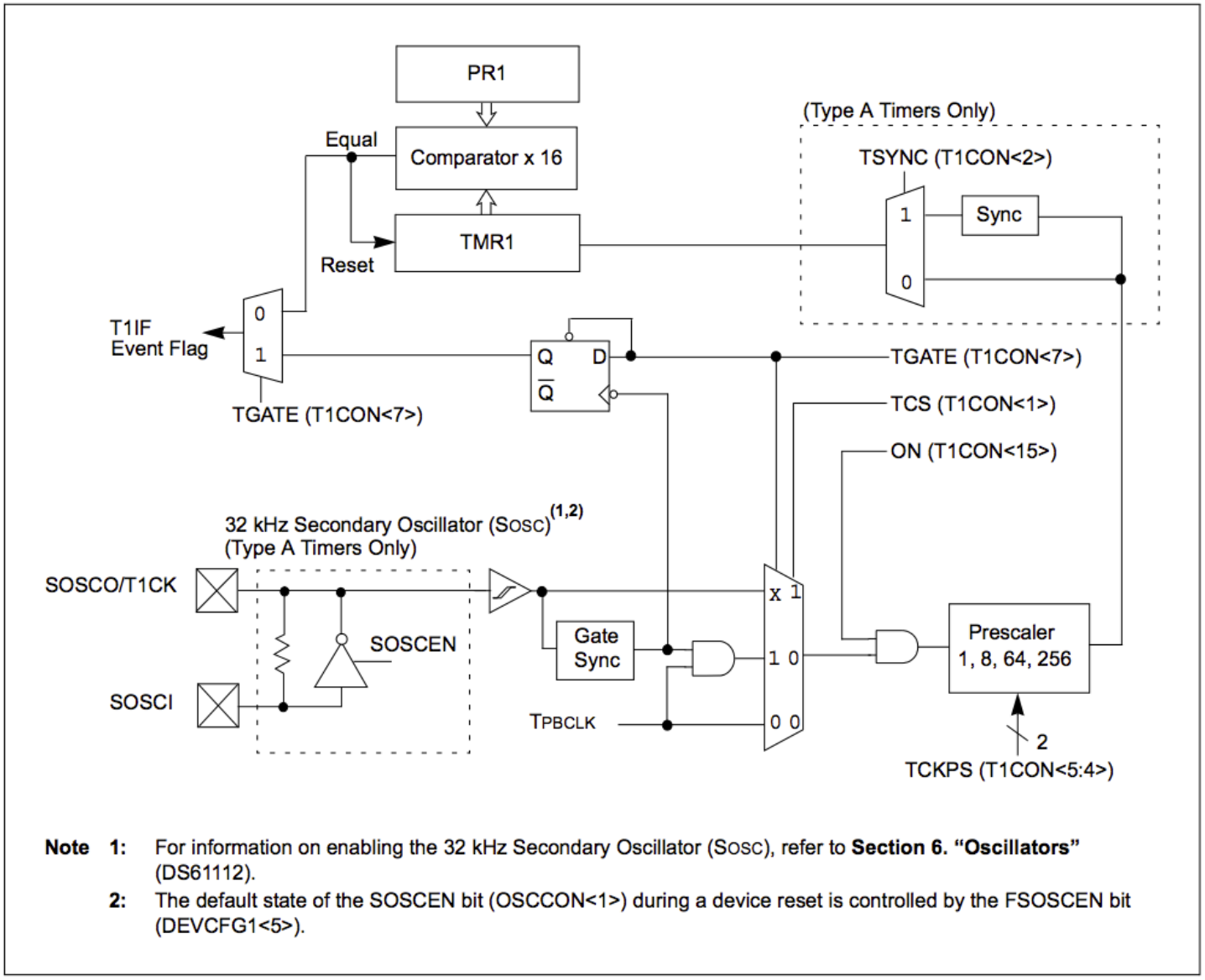


FIGURE 10 – Structure interne du timer `T1` (Source : Microchip Inc).

lation de certains registres (notamment ceux de `CP0`) ne peut être réalisée directement en C. La construction `asm` permet d'insérer du code en langage d'assemblage dans un programme en C. Les lignes 5 à 7 mettent à 1 le bit `IV` du registre `Cause` du coprocesseur `CP0`. La ligne 8 met à 1 le bit `MVEC` du registre `INTCON`. La ligne 10 autorise les interruptions avec l'instruction `ei`.

A.3 Configuration individuelle des interruptions

Chaque IRQ doit être activée individuellement au travers de registres du contrôleur d'interruptions. Chaque IRQ doit également se voir attribuer un niveau de priorité. Les registres du contrôleur d'interruptions impliqués dans cette configuration sont les suivants :

— Des registres `IEC`*n* (*interrupt enable control*) permettent d'autoriser les requêtes d'interruption(IRQ). Par exemple l'IRQ du timer `T1` est autorisée en mettant à 1 le bit `T1IE` (bit 4) du registre `IEC0`.

— Des registres `IPC`*n* (*interrupt priority control*) permettent de configurer la priorité des IRQ. Une priorité égale à 0 désactive l'interruption. Par exemple, la priorité de l'IRQ du timer `T1` est définie par les bits 4 :2 du registre `IPC1`. Une sous-priorité peut également être définie dans les bits 1 :0 du même registre afin de définir quelle interruption est traitée en premier si deux interruptions de même priorité sont déclenchées. Une interruption de sous-priorité inférieure à une autre ne sera pas interrompue par cette dernière.

Un exemple de configuration d'interruption est donné à la Figure 8 pour le Timer `T1`. La fonction `timer1_init` autorise la génération d'une requête d'interruption (IRQ4) lors de l'expiration du timer. A cet effet, à la ligne 8, le bit `T1IE` (*T1 interrupt enable*) du registre `IEC0` est positionné à 1. Cette fonction définit également la priorité associée à l'IRQ4. Les bits 4 :2 du registre `IPC1` sont mis à la valeur de priorité 3. Et les bits 1 :0 (sous-priorité) sont mis à la valeur 0. A cette fin, à la ligne 9, les bits 4 :0 sont mis à zéro en écrivant `0x1F` dans `IPC1CLR`, puis, à la ligne 10, les bits 4 :2 sont mis à la valeur 3 en écrivant `0x0C` dans `IPC1SET`.

☞ ATTENTION, la priorité associée à une IRQ doit correspondre à celle indiquée dans l'attribut `interrupt` du gestionnaire d'interruption associé.

A.4 Gestionnaire d'interruption.

Le gestionnaire d'interruption qui est exécuté lors d'une interruption est une fonction qui fait partie du programme exécuté sur le microcontrôleur. La Figure 6 donne un exemple minimal de gestionnaire d'interruption. Sa déclaration est similaire à une fonction C qui ne prend pas d'argument et qui ne retourne pas de valeur (type `void`). Bien qu'il soit implémenté comme une fonction, un gestionnaire d'interruption ne doit pas être appelé car il utilise notamment une instruction de retour différente (`eret` plutôt que `jr $ra`).

Des attributs spéciaux [2] `vector` et `interrupt` sont assignés à cette fonction à l'aide des directives `__attribute__`. Ces attributs sont utilisés pour positionner la fonction à un endroit précis en mémoire ainsi que pour générer du code indiquant dans quelles conditions ce gestionnaire d'interruption est exécuté.

2. Ces attributs ne font pas à proprement parler du langage C mais sont des extensions propres à la plateforme PIC32.

— `vector` : indique le numéro du vecteur d'interruption. Ce numéro sert au compilateur et au *linker* pour positionner le gestionnaire d'interruption au bon endroit en mémoire.

— `interrupt` : indique au compilateur que la fonction est un gestionnaire d'interruption. Le compilateur génère du code spécifique, comme par exemple l'utilisation de l'instruction `eret`. L'attribut renseigne également sur la façon dont le contexte (les registres) doit être sauvegardé par le gestionnaire d'interruption. Deux approches sont possibles : soit des instructions sont générées pour sauvegarder les registres sur la pile (`soft`), soit des registres spéciaux appelés *shadow registers* sont utilisés (`srs`). Finalement, l'attribut indique la priorité de l'interruption (`ipl`*n*). Attention, cette priorité doit correspondre à la priorité associée à la requête d'interruption.

La Figure 6 contient un exemple complet de gestionnaire d'interruption pour l'IRQ4 du timer `T1`. L'attribut `vector` mentionne le numéro de vecteur 4 associé à l'IRQ4. ATTENTION, le numéro de vecteur n'est pas toujours égal à celui de l'IRQ. L'attribut `interrupt` spécifie la valeur `ipl3soft`, ce qui signifie que l'IRQ a la priorité 3 et qu'une sauvegarde sur la pile doit être générée par le compilateur. Il est possible, en utilisant `xc32-objcopy` de désassembler le code généré par le compilateur C pour le gestionnaire d'interruptions (essayez !).

A.5 Gestion des exceptions

Le mécanisme des interruptions fait partie du mécanisme plus général des exceptions. Le mécanisme des exceptions permet, en plus des interruptions matérielles, le support de la gestion d'erreurs (`faults`) et les appels systèmes (`syscalls` et `traps`).

Cette courte section montre un moyen d'installer un gestionnaire d'exceptions dans un programme pour PIC32 en C. Le compilateur génère de lui-même un gestionnaire d'exceptions « par défaut ». Ce gestionnaire par défaut entre dans une boucle infinie en cas d'exception. Il est possible de remplacer ce gestionnaire d'exception par une fonction C appelée `_general_exception_handler` qui prend deux arguments entiers : `cause` et `status`. Ces arguments sont le contenu des registres de `CP0` qui renseignent sur le type d'exception qui a eu lieu.

Valeur	Type d'exception
0	Interruption matérielle
4	Erreur d'adressage (`load` ou *fetch*)
5	Erreur d'adressage (`store`)
6	Erreur de bus (`fetch`)
7	Erreur de bus (`load` ou `store`)
8	Appel système (`syscall`)
9	Breakpoint
10	Instruction incorrecte ou réservée
12	Dépassement lors d'une opération arithmétique

TABLE 1 – Certaines valeurs du champ `ExcCode` du registre `Cause`.

Note : les bits 6 :2 (`ExcCode`) du registre `cause` peuvent être utilisés pour déterminer le type d'exception qui est survenu. La Table 1 indique certaines des valeurs que peut prendre ce groupe de

bits. Le gestionnaire d'exception pourrait ainsi réagir différemment selon le type d'exception déclenchée.

```
1  void _general_exception_handler(
2    unsigned cause, unsigned status) {
3    LATFINV= 0x00000001;
4    while (1) {
5      delay();
6      LATGINV= 0x00000040;
7      LATFINV= 0x00000001;
8    }
9  }
10
11 int main() {
12   TRISGCLR= 0x00000040;
13   LATGCLR= 0x00000040;
14   TRISFCLR= 0x00000001;
15   LATFCLR= 0x00000001;
16
17   int x= 5;
18   int y= 0;
19   x= x/y;
20
21   while (1);
22 }
```

FIGURE 11 – Exemple de gestionnaire d'exception.

Le programme de la Figure 11 montre comment installer un gestionnaire d'exceptions. Dans l'exemple, ce gestionnaire d'exceptions a le même comportement pour toutes les exceptions : il entre dans une boucle infinie qui fait clignoter alternativement `LD4` et `LD5`. Dans le programme `main`, une exception de type « division par zéro » est déclenchée.

La Figure 12 montre comment générer divers types d'exceptions.

```
1  // Pblm d'alignement lors d'une lecture
2  // l'adresse n'est pas un
3  multiple de 4 */
4  x= *((int *) 1);
5
6  // Pblm d'alignement lors d'une ecriture
7  *((int *) 1)= 5;
8
9  // Trap : division par zero.
10 //   (le compilateur ajoute du code
11 //    pour tester si on divise par 0)
12 int x= 5;
13 int y= 0;
14 x= x/y;
15
16 // Depassement dans op. arithmetique.
17 // Note : le langage C ne peut generer
18 // de telles exceptions car
19 // toutes les operations sont modulo 2^32
20 int y= 0x7FFFFFFF;
21 asm volatile("addi␣%0,␣%0,␣1;" : "+r"(y));
22
23 // Appel systeme (instruction syscall)
24 asm("syscall;");
25
26 // Erreur de bus : pas de memoire
27 // ou de registre a l'adresse 0.
28 // Dans le PIC32MX320F128H, la plage
29 // d'adresses 0x00000000-0x7FFFFFFF
30 // n'est pas implementee.
31 x= *((int *) 0);
32
33 // Instruction invalide
34 asm(".dword␣0xdeadbeef;");
35
36 // Breakpoint
37 asm("break;");
```

FIGURE 12 – Extraits de programmes générant différents types d'exceptions.