



# LES TIMERS

PIERRE-YVES ROCHAT, EPFL  
ET YVES TIECOURA, INP-HB

RÉV 2016/02/02

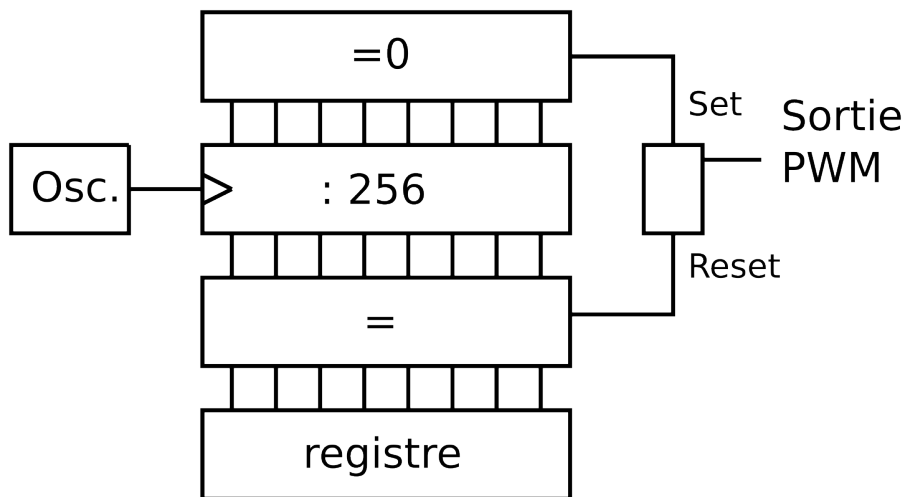
## GESTION PRÉCISE DU TEMPS

Les enseignes et afficheurs à LED, comme beaucoup d'applications des microcontrôleurs, nécessitent souvent une gestion précise du temps qui s'écoule. Les animations doivent être correctement cadencées. Plus difficile encore, la gestion des afficheurs matriciels multiplexés exigent un *timing* précis.

Il est souvent difficile d'assurer correctement cette gestion du temps en utilisant uniquement les instructions du processeur. C'est la raison pour laquelle les microcontrôleurs offrent presque toujours des circuits spécialisés dans le comptage et la gestion du temps, appelés les **timers**.

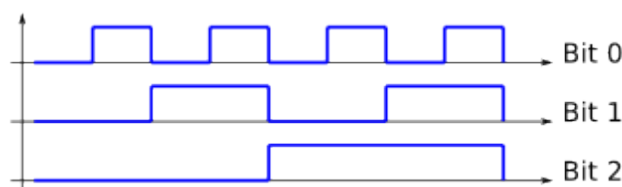
Dans le chapitre sur la Modulation de Largeur d'Impulsion (PWM), nous avons suggéré le montage suivant, pour faciliter la génération de signaux PWM :

DRAFT



Compteur générant du PWM

Ce montage est basé sur un compteur binaire, qu'on appelle aussi un diviseur de fréquence. Rappelons qu'à chaque flanc montant de l'horloge, le compteur passe à la valeur binaire suivante. On peut observer que, lorsqu'un signal de fréquence fixe  $F_0$  est placé sur l'entrée, les sorties successives prennent des fréquences sous-multiples : la fréquence est divisée par 2, par 4, par 8, etc.



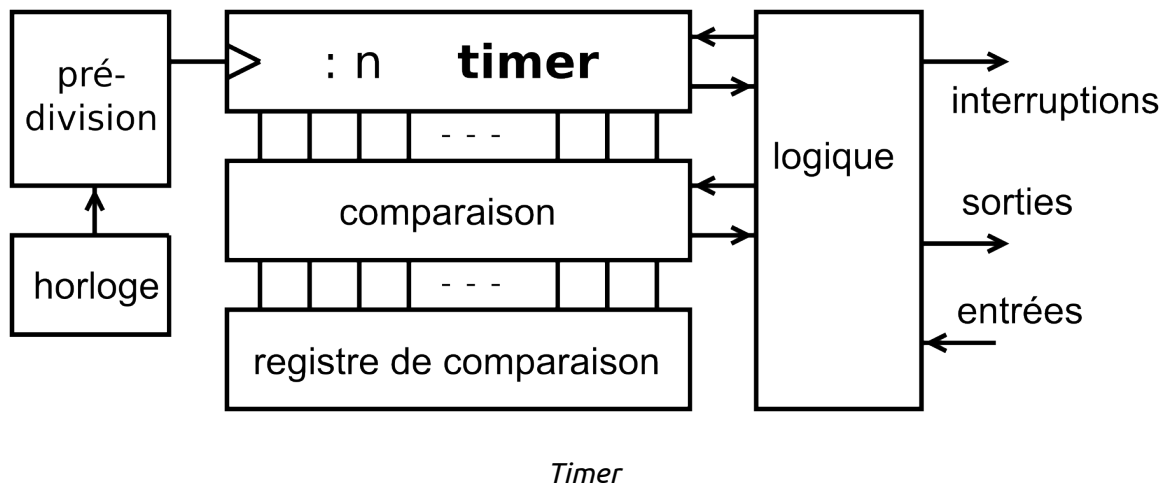
Chronogramme d'un compteur binaire

Le terme anglais *timer* désigne le compteur binaire, mais aussi souvent l'ensemble du montage. Les traductions françaises, *minuterie* ou *temporisateur*, ne sont que rarement utilisées. C'est la raison pour laquelle nous utiliserons ici plutôt l'anglicisme *Timer\**, comme s'il s'agissait d'un nom propre. Que les puristes nous pardonnent !

Le PWM n'est pas la seule application des Timers. Beaucoup de tâches liées le plus souvent à la gestion du temps ou au comptage d'événements peuvent lui être confiées.

## LES TIMERS

La figure ci-dessous généralise ce concept :

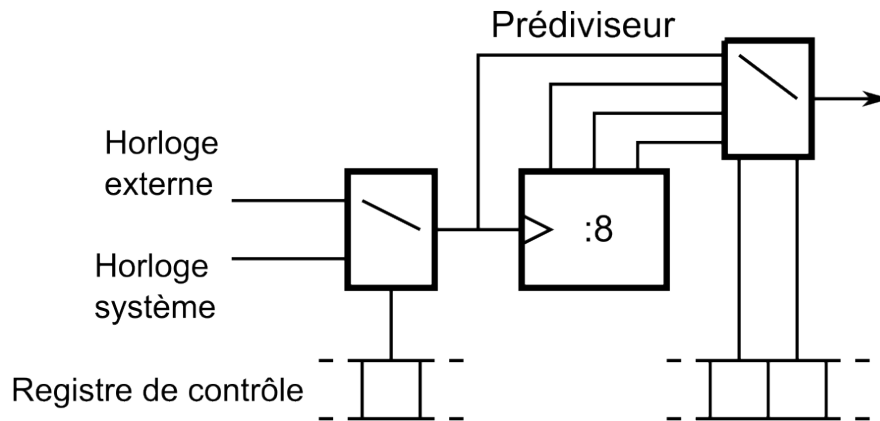


On y trouve :

- un **compteur binaire**. Il peut être de 8 bits, 16 bits, parfois même de 32 bits. C'est le Timer proprement dit.
- un **horloge**, c'est à dire un oscillateur (Osc). Il s'agit généralement de l'horloge également utilisée pour le processeur.
- un système de **choix d'horloge et de pré-division**, qui permet de choisir une fréquence d'horloge bien adaptée au problème à résoudre.
- un logique de **comparaison** (par exemple l'égalité)
- un **registre de comparaison**, associé à la logique de comparaison. Plusieurs registres de comparaison sont souvent présents.
- une logique de gestion, permettant de faire interagir des **entrées** et des **sorties** avec le Timer, ainsi qu'à générer des **interruptions** dans certaines conditions.

## PRÉDIVISION

Voici comment peut se présenter le choix de l'horloge et de pré-division :



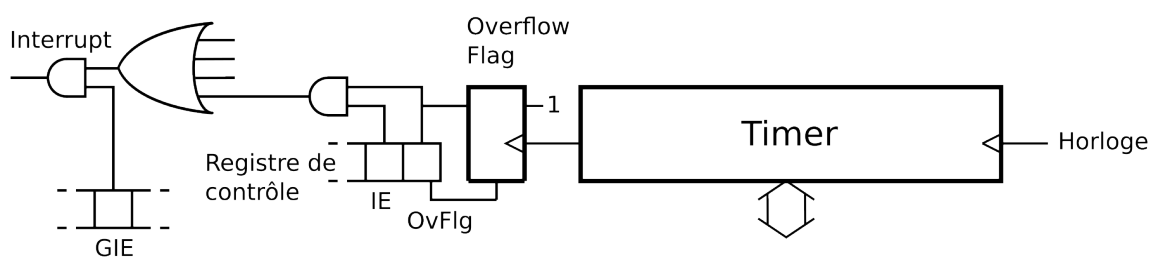
*Exemple de système de choix de l'horloge*

Un premier multiplexeur permet de choisir entre une horloge extérieure et une horloge externe. Un compteur binaire, utilisé en diviseur de fréquence, fournit des signaux à des fréquences sous-multiples. Un second multiplexeur permet de choisir la fréquence qui commande le Timer.

Les deux multiplexeurs sont commandés par des bits d'un registre de contrôle, dont le rôle est de fixer le mode de fonctionnement du Timer.

## LOGIQUE DE GESTION

Une logique permet de mettre en oeuvre le Timer. Elle diffère beaucoup d'un microcontrôleur à l'autre. En voici un exemple très simple :



*Exemple de logique de gestion d'un timer*

On y trouve une bascule qui détecte le dépassement de capacité du Timer. C'est le moment où le compteur binaire repasse à la valeur zéro. La bascule est mise à 1 à ce moment.

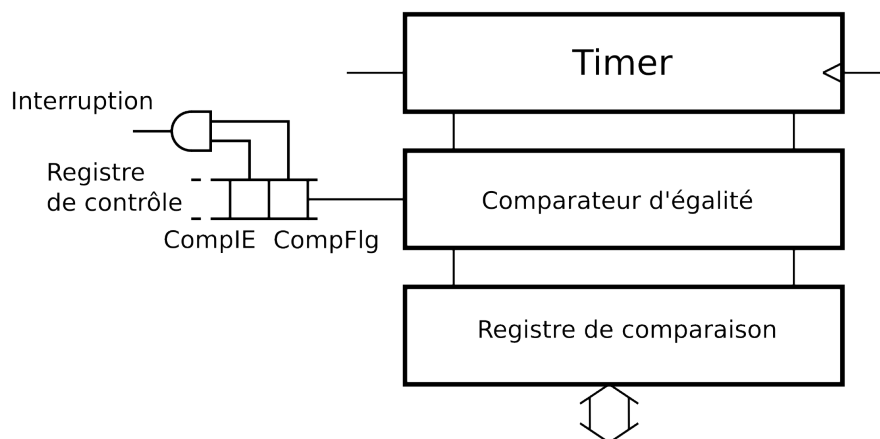
Elle fait généralement partie d'un registre de contrôle et peut donc être lue à tout moment.

Il faut pouvoir remettre ce fanion à zéro lors qu'il a été pris en compte. Parfois, il faut écrire un 0 dans le bit correspondant du registre. Mais sur certains microcontrôleurs, c'est l'écriture de la valeur "1" qui met ce fanion à 0. C'est le cas des Timer des AVR.

La génération d'interruptions est très importante dans l'utilisation des Timers. Ici, on voit un Fanion **IE** *Interrupt Enable* qui permet de générer une interruption. En effet, la porte **ET** nécessite que **IE** soit à 1 pour que l'interruption soit transmise. Elle ne sera effective que si l'autorisation générale des interruptions est activée (**GIE** : *General Interrupt Enable*), comme toutes les autres interruptions.

## REGISTRES DE COMPARAISON

La présence d'un ou de plusieurs registres de comparaison associé à un Timer le rend beaucoup plus intéressant. En voici un exemple simple :



*Exemple de registre de comparaison*

Un comparateur d'égalité est placé entre le Timer et un registre, dont il est possible à tout moment de modifier la valeur. Chaque fois que le Timer a la même valeur que le registre de comparaison, le fanion pass à 1. Le nouveau, il est possible de générer une interruption, avec un mécanisme similaire à celui du dépassement de capacité.

## LES TIMERS DES MICROCONTRÔLEURS

Quelques années après les premiers microprocesseurs, des circuits spécialisés sont apparus sur le marché avec des Timers. C'est le cas du très célèbre **8253** d'Intel, datant de 1981, dont on trouve encore des descendants dans les PC modernes.

Les microcontrôleurs ont eux aussi très vite été complétés par des Timers, comme le célèbre **PIC16x84**, qui incluait déjà un unique compteur 8 bits très simple, mais très utile.

Les microcontrôleurs **ARM** ont tous plusieurs Timers. Le **ATmega328**, connu pour équiper les Arduino, a trois Timers, le Timer0 à 8 bits, le Timer 1 à 16 bits et le Timer2 à 8 bits, mais différent du Timer0. Ces Timers ont une fonctionnalité riche, permettant de nombreuses applications.

Les microcontrôleurs plus modernes ont souvent de Timers très complexe. Dans les familles de microcontrôleurs **ARM**, les Timers diffèrent d'un fabricant à l'autre : cette partie du microcontrôleur est propriétaire, elle n'est pas développée par la société ARM.

Nous étudierons ici les Timers utilisés dans les microcontrôleurs **MSP430G** de Texas Instrument, qui se trouvent sur la carte Launchpad.

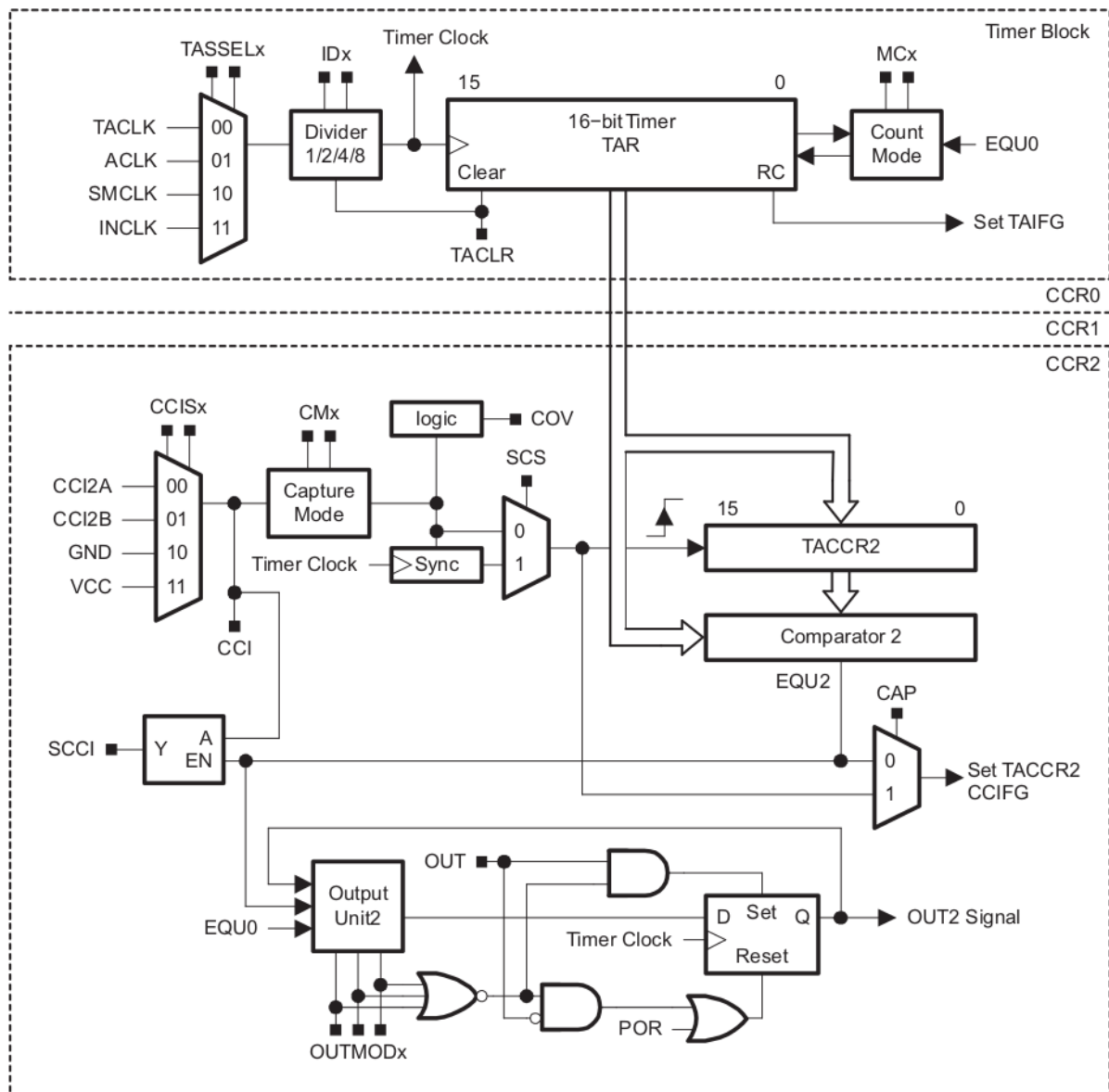
## TIMER A DU MSP430

Les MPS430 de la série G disposent de Timers de 16 bits, en nombre est en configurations variables selon les modèles. Le MSP430G2231 (boîtier 14 pattes) en a un seul, disposant de deux registres de comparaison. Le MSP430G2553 en a deux, chacun disposant de trois registres de comparaisons.

Le fonctionnement des ces registres est très bien documenté : 20 pages de documentation, bien évidemment en anglais. Voici les références du document : MSP430x2xx Family User's Guide, literature Number: SLAU144H. On le trouve facilement sur Internet.

Afin de se familiariser avec la lecture de la documentation, nous allons partir des documents fournis par Texas Instrument pour tenter de comprendre le minimum vital pour mettre en œuvre un de ces Timers. Nous allons aussi respecter la syntaxe proposée pour l'accès aux registres.

La figure ci-dessous donne la vue d'ensemble du Timer A :



**Figure 12-1. Timer\_A Block Diagram**

## Timer A du MSP430

Ce schéma n'est pas simple, mais il est clair et complet. On y trouve un compteur 16 bits appelé **TAR**. Il est possible à tout moment de lire sa valeur. Il est aussi possible d'écrire une nouvelle valeur, mais nous n'utiliserons pas cette possibilité ici.

Ce compteur reçoit une horloge qu'il est possible de sélectionner parmi plusieurs sources. Un pré-diviseur peut être utilisé, qui donne le choix entre la fréquence d'origine et des divisions par 2, 4 ou 8. Le compteur peut compter selon plusieurs modes de comptage.

Un registre de contrôle de 16 bits est associé à chaque Timer, appelé **TACTL**. Il peut aussi apparaître sous le nom *TA0CTL*, pour les microcontrôleurs qui ont plusieurs Timers A (le

deuxième s'appelant alors TA1CTL). Il n'apparaît pas explicitement dans le schéma, mais c'est de lui que proviennent plusieurs signaux (TASSELx, IDx, TACLR, etc). Ce sont les différents bits de ce registre qui vont permettre de choisir l'horloge, les pré-diviseurs, le mode de comptage, etc.

Voici comment la documentation le décrit ce registre TACTL :

### 12.3.1 TACTL, Timer\_A Control Register

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
Unused	Bits 15-10	Unused					
TASSELx	Bits 9-8	Timer_A clock source select					
		00 TACLK					
		01 ACLK					
		10 SMCLK					
		11 INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)					
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock.					
		00 /1					
		01 /2					
		10 /4					
		11 /8					
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power.					
		00 Stop mode: the timer is halted.					
		01 Up mode: the timer counts up to TACCR0.					
		10 Continuous mode: the timer counts up to 0FFFFh.					
		11 Up/down mode: the timer counts up to TACCR0 then down to 0000h.					
Unused	Bit 3	Unused					
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.					
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request.					
		0 Interrupt disabled					
		1 Interrupt enabled					
TAIFG	Bit 0	Timer_A interrupt flag					
		0 No interrupt pending					
		1 Interrupt pending					

#### Registre TACTL

Parcourons quelques bits de ce registre de contrôle pour choisir les valeurs pour notre premier exemple :

- TASSELx permet de choisir l'horloge. Utilisons l'horloge du processeur : SMCLK. Les deux bits correspondants doivent prendre la valeur binaire 10. Texas Instrument utilise la syntaxe suivante : TASSEL\_2 (valeur 2 pour les bits TASSEL).
- IDx permet de choisir la pré-division. Choisissons une division par 8. La valeur est ID\_3.
- MCx permet de choisir le mode de comptage. Choisissons le mode continu. La valeur est MC\_2.



L'instruction d'initialisation de notre timer sera donc : `TACTL = TASSEL_2 + ID_3 + MC_2;`

## PREMIER PROGRAMME AVEC LE TIMER A

Voilà un premier programme... qui va faire clignoter une LED !

Il débute comme toujours par l'instruction de mise hors-service du compteur *Watchdog*, mais aussi par deux instructions permettant de choisir une des fréquences calibrées d'usine, ici 1 MHz :

```
int main() {
    WDTCTL = WDTPW + WDTHOLD;
    BCCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ; // Fréquence CPU
    P1DIR |= (1<<0); // Led en P1.0
    TACTL0 = TASSEL_2 + ID_3 + MC_2;
    while (1) { // boucle infinie
        if (TACTL0 & TAIFG) {
            TACTL0 &=~TAIFG;
            P1OUT ^= (1<<0); // Toggle Led
        }
    }
}
```

Comment fonctionne la boucle principale ? Chaque fois que le fanion TAIFG passe à 1, la LED est inversée. Le fanion TAIF (qui se trouve aussi dans le registre TACTL) signale un dépassement de capacité, c'est-à-dire le retour à zéro du compteur. Il doit être remis à zéro en vue du prochain cycle. Calculons la période de clignotement : l'horloge de 1 MHz est divisée par 8 par le pré-diviseur. Le Timer reçoit donc 125 kHz (période : 8  $\mu$ s). Le Timer a 16 bits, il va donc faire un cycle complet en 65'536 coups d'horloge, soit 524 ms.

## LES REGISTRES DE COMPARAISON

L'intérêt principal des Timers réside dans les registres de comparaison qui leur sont associés. Dans le schéma de la page 1, on voit qu'il y a trois registres de comparaison, notés 0, 1 et 2. Le détail est donné pour le groupe 2.

Ces trois registres de comparaison se nomment CCR0, CCR1 et CCR2. Ces registres permettent de mémoriser une valeur qui va être en permanence comparée avec la valeur du Timer TAR.

A chacun de ces registres de comparaison est associé un registre de contrôle, appelés TACCLT0, TACCLT1 et TACCTL2.

La figure suivante donne la description de ce registre. Elle n'est pas simple :

Timer\_A Registers

www.ti.com

#### 12.3.4 TACCTLx, Capture/Compare Control Register

15	14	13	12	11	10	9	8
CMx		CCISx		SCS	SCCI	Unused	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	r0	rw-(0)
7	6	5	4	3	2	1	0
OUTMODx		CCIE		CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)
<b>CMx</b>	Bit 15-14	Capture mode					
		00 No capture					
		01 Capture on rising edge					
		10 Capture on falling edge					
		11 Capture on both rising and falling edges					
<b>CCISx</b>	Bit 13-12	Capture/compare input select. These bits select the TACCRx input signal. See the device-specific data sheet for specific signal connections.					
		00 CCIxA					
		01 CCIxB					
		10 GND					
		11 V <sub>CC</sub>					
<b>SCS</b>	Bit 11	Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock.					
		0 Asynchronous capture					
		1 Synchronous capture					
<b>SCCI</b>	Bit 10	Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit					
<b>Unused</b>	Bit 9	Unused. Read only. Always read as 0.					
<b>CAP</b>	Bit 8	Capture mode					
		0 Compare mode					
		1 Capture mode					
<b>OUTMODx</b>	Bits 7-5	Output mode. Modes 2, 3, 6, and 7 are not useful for TACCR0, because EQUx = EQU0.					
		000 OUT bit value					
		001 Set					
		010 Toggle/reset					
		011 Set/reset					
		100 Toggle					
		101 Reset					
		110 Toggle/set					
		111 Reset/set					
<b>CCIE</b>	Bit 4	Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.					
		0 Interrupt disabled					
		1 Interrupt enabled					
<b>CCI</b>	Bit 3	Capture/compare input. The selected input signal can be read by this bit.					
<b>OUT</b>	Bit 2	Output. For output mode 0, this bit directly controls the state of the output.					
		0 Output low					
		1 Output high					
<b>COV</b>	Bit 1	Capture overflow. This bit indicates a capture overflow occurred. COV must be reset with software.					
		0 No capture overflow occurred					
		1 Capture overflow occurred					
<b>CCIFG</b>	Bit 0	Capture/compare interrupt flag					
		0 No interrupt pending					
		1 Interrupt pending					

Registre TACCRx

Modifions notre programme de la manière suivante :

```
int main() {
    ...
    TACCR0 = 62500; // * 8us = 500ms
    while (1) { // boucle infinie
        if (TACCTL0 & CCIFG) {
            TACCTL0 &=~CCIFG;
            TACCR0 += 62500;
            P1OUT ^= (1<<0); // Toggle Led
        }
    }
}
```

Au début du programme, le registre de comparaison a été initialisé à 62500, une valeur qui correspond à une demi-seconde dans notre cas :  $62500 * 8 \mu s = 500 \text{ ms}$ . Une fois cette valeur atteinte, il faut ajouter 62500 à la valeur courant du registre de comparaison. On va dépasser la capacité du registre, qui a 16 bits. On obtiendra :  $(62500 + 62500) \text{ modulo } 65536 = 59464$  ! Mais comme le Timer augmente toujours et qu'il a lui aussi 16 bits, cette valeur est bien la bonne valeur pour la prochaine comparaison.

Si vous avez des doutes, imaginez qu'il est 9h50 et que vous voulez faire sonner votre réveil dans 30 minutes. Vous devez le régler sur 10h20. En ne tenant compte que des minutes, on a bien :  $(50 + 30) \text{ modulo } 60 = 20$

## LES INTERRUPTION ASSOCIÉES AUX TIMERS

L'intérêt principal des Timers est de les associer à des interruptions. Modifions le programme de la manière suivante :

```
int main() {
    ...
    TACTL |= TAIE; // interruption de l'overflow
    _BIS_SR(GIE); // autorisation générale des interruptions

    while (1) { // boucle infinie
    }
}

// Timer_A1 Interrupt Vector (TAIV) handler
#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer_A1(void) {
    switch(TAIV) {
        case 2: // CCR1 : not used
```

```

        break;
    case 4: // CCR2 : not used
        break;
    case 10: // Overflow
        Led1Toggle;
        break;
}
}

```

Notez le nom de la routine d'interruption. Elle ne concerne par le Timer1 ! Elle est la seconde routine d'interruption du Timer0, la première étant présentée dans le prochain exemple.

L'interruption associée au Timer lui-même correspond à un *overflow* (dépassement de capacité, c'est le passage de la plus grande valeur à zéro). La syntaxe de la routine d'interruption est un peu compliquée. Il faut la copier et non pas chercher à la comprendre ! Notez qu'elle varie selon les compilateurs : il ne s'agit pas d'une norme du C. Dans ce cas, trois sources différentes d'interruption (overflow, comparaison 1 et comparaison2) sont regroupées dans une même routine d'interruption. Un registre appelé TAIV permet de connaître dans chaque cas la cause de l'interruption). Les valeurs 2 4 et 10 sont le choix arbitraire du fabricant : il faut respecter scrupuleusement la syntaxe du switch TAIV... case... Il n'a pas été nécessaire de remettre à zéro le fanion TAIFG : c'est la gestion matérielle des interruptions qui le fait automatiquement au moment de l'appel de la routine d'interruption.

## INTERRUPTION DE COMPARAISON

De même, une interruption peut être associée à chaque registre de comparaison. Cette fois, c'est dans le registre TACCTLx (x valant 0, 1 ou 2) qu'il faut activer le fanion d'interruption.

```

int main() {
    ...
    TACCTL0 |= CCIE; // interruption de la comparaison
    _BIS_SR(GIE); // autorisation générale des interruptions

    while (1) { // boucle infinie
    }
}

#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A0(void) {
    CCR0 += 62500;
}

```

```
P1OUT ^= (1<<0); // Toggle Led  
}
```

Les Timers offrent de très nombreuses possibilités. L'étude détaillée de la documentation peut prendre du temps. De nombreux exemples sont fournis par les fabricants, pour en illustrer divers usages.