ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

LES TIMERS SUR LE MSP430

Pierre-Yves Rochat

rév 2020/03/27

Introduction

Comme pour les interruptions, la manière d'implémenter les timers varie considérablement d'une famille de microcontrôleur à une autre. Nous étudierons ici les timers utilisés dans les microcontrôleurs MSP430G de Texas Instruments, 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 et en configurations variables selon les modèles. Par exemple Le MSP430G2231 avec un boîtier de 14 pattes en a un seul, disposant de deux registres de comparaison. Le MSP430G2553 en a deux, disposant chacun de trois registres de comparaison. Le MSP430F5529 a 3 Timers : TA0 avec 5 registres de comparaison, TA1 et TA2 avec chacun 3 registres de comparaison. Il dispose aussi d'un timer du type B, qui ne sera pas décrit ici.

Le fonctionnement de ces registres est très bien documenté : 20 pages, 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 nous familiariser avec la lecture de la documentation, nous allons nous baser sur les documents fournis par Texas Instruments pour comprendre le minimum nécessaire à la mise en œuvre d'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 :

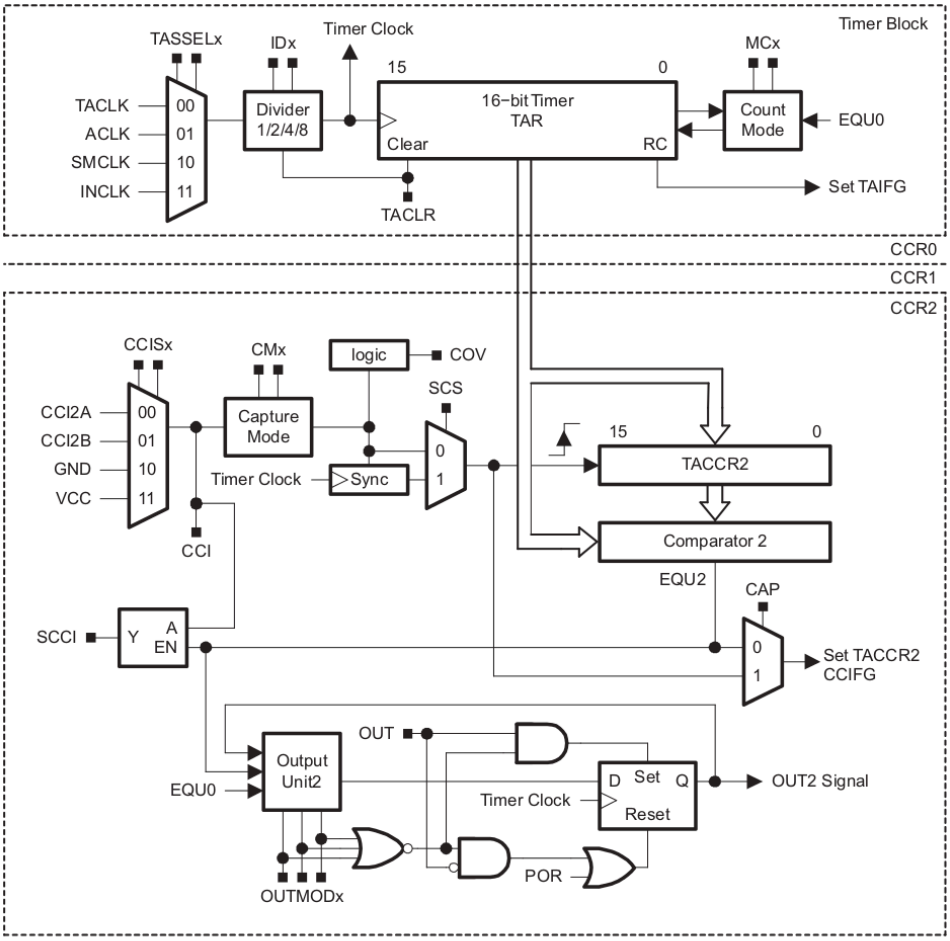

Figure 12-1. Timer_A Block Diagram

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 un signal d'horloge qu'il est possible de sélectionner parmi plusieurs sources. Un prédiviseur 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.

Un registre de contrôle de 16 bits appelé TACTL est associé à chaque timer. Il peut aussi apparaître sous le nom TA0CTL, pour les microcontrôleurs qui ont plusieurs TIMER A (le deuxième s'appelant alors TA1CTL). Il n'apparaît pas explicitement dans le schéma, mais c'est de lui que proviennent plusieurs signaux (TASSETx, IDx, TACLR, etc.) Ce sont les différents bits de ce registre qui vont permettre de choisir l'horloge, les prédiviseurs, 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 TACLT

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 Instruments utilise la syntaxe suivante : TASSEL_2 (valeur 2 pour les bits TASSEL).
- IDx permet de choisit 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 commence 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; // Watchdog hors service
  BCSCTL1 = CALBC1_1MHZ;
  DCOCTL = CALDCO_1MHZ;     // Fréquence CPU
  P1DIR |= (1<<0); // P1.0 en sortie pour la LED
  TACTL0 = TASSEL_2 + ID_3 + MC_2;
  while (1) { // Boucle infinie
    if (TACTL0 & TAIFG) {
      TACTL0 &= ~TAIFG;
      P1OUT ^= (1<<0); // Inversion LED
    }
  }
}
```

Comment fonctionne la boucle principale ? Chaque fois que le fanion TAIFG passe à *1*, l'alimentation de 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 prédiviseur. Le timer est donc commandé à une fréquence de 125 kHz ce qui correspond à une période de 8 µs. Le timer a 16 bits, il va donc faire un cycle complet en 65'536 coups d'horloge, soit environ 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.

À chacun de ces registres de comparaison est associé un registre de contrôle, appelé respectivement TACCLT0, TACCLT1 et TACCTL2.

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

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)

CMx	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
CCISx	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_{CC}
SCS	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
SCCI	Bit 10	Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit
Unused	Bit 9	Unused. Read only. Always read as 0.
CAP	Bit 8	Capture mode
	0	Compare mode
	1	Capture mode
OUTMODx	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
CCIE	Bit 4	Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.
	0	Interrupt disabled
	1	Interrupt enabled
CCI	Bit 3	Capture/compare input. The selected input signal can be read by this bit.
OUT	Bit 2	Output. For output mode 0, this bit directly controls the state of the output.
	0	Output low
	1	Output high
COV	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
CCIFG	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; // 62500 * 8 us = 500 ms
  while (1) {     // Boucle infinie
    if (TACCTL0 & CCIFG) {
      TACCTL0 &= ~CCIFG;
      TACCR0 += 62500;
      P1OUT ^= (1<<0); // Inversion LED
    }
  }
}
```

Au début du programme, le registre de comparaison a été initialisé à 62’500, une valeur qui cor-

respond à une demi-seconde dans notre cas : *62'500 × 8 µs = 500 ms*. Une fois cette valeur atteinte, il faut ajouter 62'500 à la valeur courant du registre de comparaison. On va dépasser la capacité du registre, qui a 16 bits. On obtiendra : *(62'500 + 62'500) % 65'536 = 59'464* où le signe *%* représente l'opération de modulo, c'est-à-dire le reste de la division entière. Mais comme le timer augmente toujours et qu'il a lui aussi 16 bits, cette valeur est effectivement la bonne pour la prochaine comparaison.

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

Les interruptions 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 vide
  }
}

// Timer_A1 Interrupt Vector (TAIV) handler
#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer_A1 (void) {
  switch (TAIV) {    // discrimination des sources d'interruption
  case  2:           // TACCR1 : not used
    break;
  case  4:           // TACCR2 : not used
    break;
  case 10:           // Overflow
    P1OUT ^= (1<<0); // Inversion LED
    break;
  }
}
```

Notez le nom de la routine d'interruption. Elle ne concerne pas le TIMER 1 ! Elle est la seconde routine d'interruption du TIMER 0, 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 *0*). 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'interrup-

tion (OVERFLOW, COMPARAISON 1 et COMPARAISON 2) 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 des instructions *switch TAIV... case...* Il n'a pas été nécessaire de remettre à zéro le fanion TAIFG, car 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 vide
  }
}

#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A0 (void) {
  TACCR0 += 62500;
  P1OUT ^= (1<<0); // Inversion LED
}
```

PWM par interruption

En combinant les interruptions du dépassement de capacité et de la comparaison, on peut produire un signal PWM sur n'importe quelle broche du microcontrôleur :

```
int main() {
  ...
  TACTL |= TAIE;   // Interruption de l'overflow
  TACCTL0 |= CCIE; // Interruption de la comparaison
  _BIS_SR (GIE);   // Autorisation générale des interruptions
  while (1) {      // Boucle infinie vide
  }
}

#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer_A1 (void) {
  switch (TAIV) {    // discrimination des sources d'interruption
  case  2:           // TACCR1 : not used
```

```
    break;
  case  4:           // TACCR2 : not used
    break;
  case 10:           // Overflow
    P1OUT |= (1<<0); // Activer le signal au début du cycle
    break;
  }
}

#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A0 (void) {
  P1OUT &=~(1<<0); // Désactiver le signal au moment donné par le registre de comparaison
}
```

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 les divers usages.