MSP430 TP4 : Interruptions matérielles

Introduction

L'objectif de ce TP est d'étudier un rouage essentiel dans le fonctionnement des ordinateurs, le mécanisme des interruptions matérielles. En particulier, on va s'intéresser aux notions suivantes : requête d'interruption (IRQ), vecteur d'interruption, masquage d'interruption, routine de traitement d'interruption (ISR), sauvegarde de contexte, acquittement d'interruption.
Tous ces concepts sont situés à la frontière entre le logiciel et le matériel, vous aurez donc à utiliser certaines facultés du langage C propres à la programmation sur machine nue : directives *interrupt*, fonctions intrinsèques, programmation avec «assembleur en ligne».

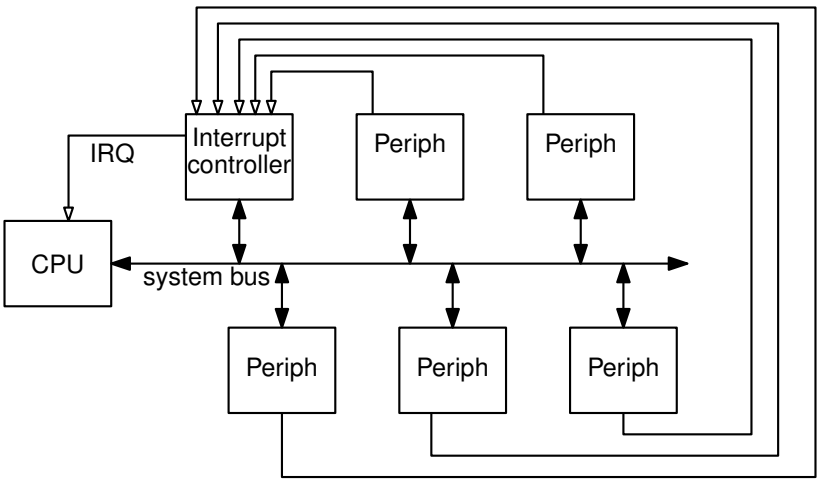
Le prétexte est de programmer un chronomètre numérique, similaire à celui d'une montre bracelet. Votre chronomètre se servira de l'écran LCD pour communiquer avec l'utilisateur, vous aurez donc à réutiliser le code que vous avez écrit pendant les séances précédentes. À la fin du TP, l'utilisateur pourra contrôler le chronomètre (marche/arrêt, remise à zéro) grâce aux deux boutons de la carte mère. Pour la mesure du temps proprement dite, on va faire appel à un nouveau type de périphérique, le *timer*. Ce TP est donc également l'occasion d'illustrer la différence entre les notions de timer, de signal d'horloge, et de générateur d'horloge.

Dans un premier temps, lisez l'encadré ci-dessous, pour vous rafraîchir la mémoire sur les différentes notions mises en jeu.

À savoir : scrutation VS interruptions, requêtes (IRQ), vecteur, routine de traitement (ISR)

La communication entre un périphérique et le processeur se fait en général au travers des registres matériels. Un composant qui veut transmettre une information au programme place cette information dans un de ses registres, et attend que le processeur vienne lire cette valeur. C'est cette technique, appelée *scrutation* (en anglais *polling*) que vous avez utilisée dans un TP précédent pour connaître l'état des boutons. Un inconvénient majeur de cette approche est son incapacité à passer à l'échelle : pour ne pas rater un évènement, le programme doit continuellement aller scruter l'état du matériel, ce qui monopolise le processeur.

L'alternative consiste à mettre en place un mécanisme *d'interruptions* (cf poly page 73). Dans ce cas, le composant qui veut transmettre une information au programme place cette information dans l'un de ses registres, puis envoie au processeur une *requête d'interruption* (en anglais *interrupt request* ou IRQ). Selon les architectures, ces requêtes peuvent transiter sur le bus principal ou sur des fils dédiés appelés des *lignes d'interruptions*. Dans ce cas-là, soit le processeur dispose d'une entrée pour chaque source d'interruptions, soit comme illustré ci-dessous, les lignes d'interruptions sont concentrées par un périphérique dédié, appelé le *contrôleur d'interruption*.



Du côté du processeur, la gestion des interruptions est intégrée au cycle de Von Neumann. Lorsqu'il reçoit une requête, le processeur interrompt automatiquement l'exécution du programme et saute vers une adresse bien connue, à laquelle il s'attend à trouver une *routine de traitement* spécifique (en anglais *interrupt service routine* ISR, ou *interrupt handler*). Chaque ligne d'interruption est ainsi associée à une routine distincte, ce qui permet au programmeur de prévoir un traitement différent pour chaque type d'évènement. La correspondance {requête $n_1 \rightarrow$ routine r_1, requête $n_2 \rightarrow$ routine r_2, etc.} est implémentée par une structure de données appelée la *table des vecteurs d'interruption* (*interrupt vector table* ou IVT). En général il s'agit d'un tableau de pointeurs de fonction, chaque case contenant l'adresse d'une ISR.

Une routine d'interruption est un morceau de code similaire à une fonction, sauf qu'elle est invoquée automatiquement par le processeur, et non pas par un appel explicite. En plus de traiter l'évènement proprement dit en allant lire et/ou écrire dans les registres du périphérique concerné, une ISR devra typiquement *accuser réception* de l'interruption auprès du périphérique et/ou du contrôleur d'interruption. Ensuite, elle peut se terminer et *restaurer le contexte* d'exécution, c'est à dire reprendre l'exécution du programme interrompu, qui ne se sera aperçu de rien.

Exercice 1 Créez un répertoire TP4, et retapez dans un fichier tp4.c le programme suivant. Compilez-le et transférez-le sur la carte.

```
#include "msp430fg4618.h"
#include "lcd.h"
unsigned int cpt;

int main(void)
{
    lcd_init(); // Initialize screen

    P5DIR |= 0x02 ; // Set P5.1 to output direction

    cpt = 0;

    for(;;)
    {
        volatile unsigned int i;

        for(i=0;i<0xFFF0;i++) // software delay
        { // do nothing
        }

        lcd_display_number(cpt);

        cpt++;

        P5OUT = P5OUT ^ 0x02 ; // toggle P5.1 (LED4)
    }
}
```

1 Prise en main du *hardware timer*

Pour mesurer l'écoulement du temps dans un ordinateur, deux solutions sont possibles :

- écrire une boucle logicielle, et espérer que processeur mette « un certain temps » à l'exécuter ;
- utiliser un composant matériel capable de compter le temps, et qui nous préviendra quand la durée voulue est écoulée.

La première solution, si elle convient parfaitement pour ajouter facilement des attentes grossières dans un programme (et c'est ce qu'on a fait jusqu'ici) est difficile à rendre plus précise : il faut regarder finement quelles instructions assembleur sont générées par le compilateur, chercher dans la documentation combien de cycles d'horloge prend l'exécution de chaque instruction, maîtriser les conditions de sortie de boucle, etc. Le travail de mise au point et de maintenance est considérable.

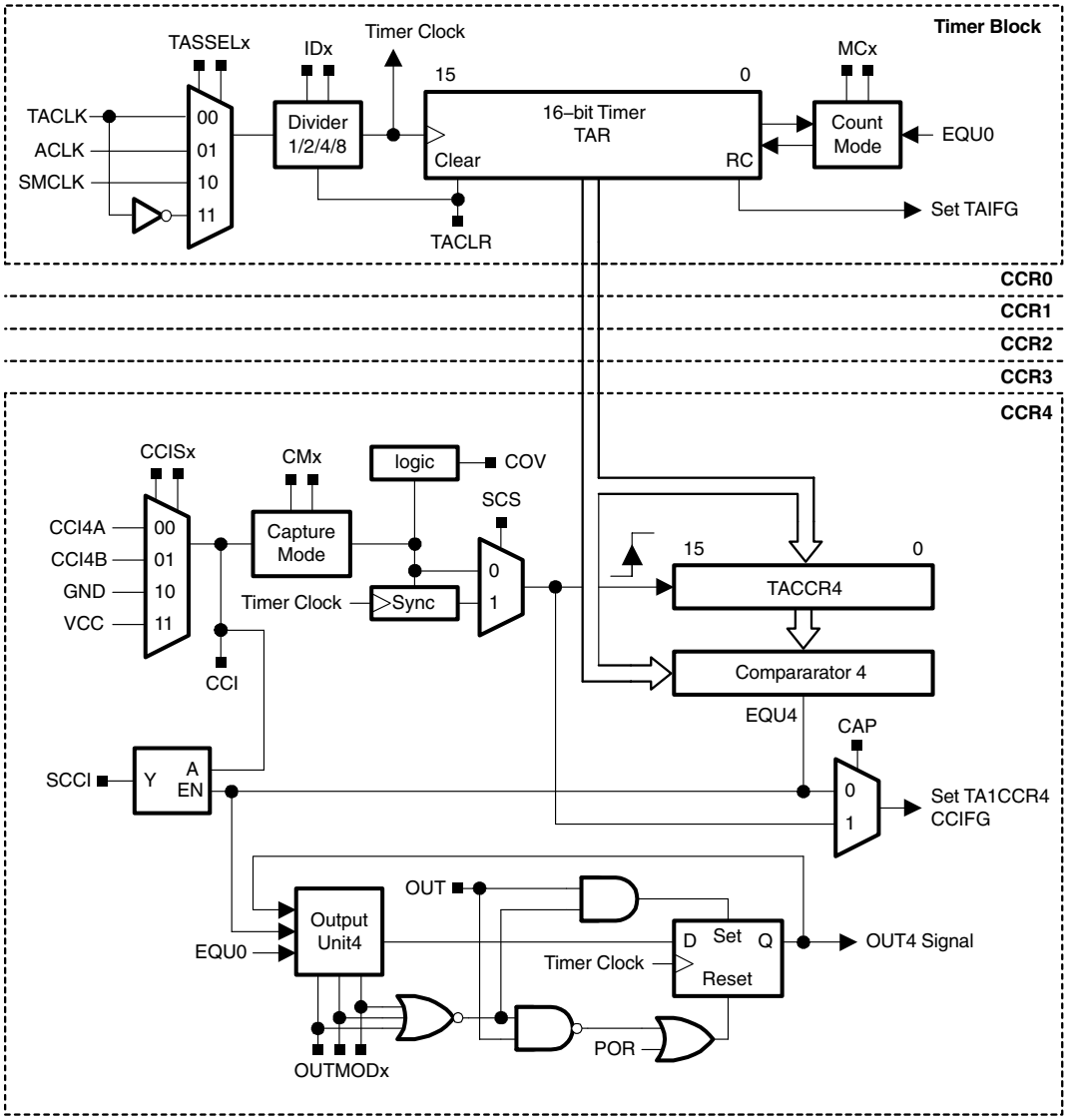
La seconde solution consiste à confier la temporisation proprement dite à un composant matériel, un *timer*. En pratique, un *timer* est essentiellement constitué d'un compteur, c'est à dire un registre capable d'incrémenter sa valeur sur commande. Si on dispose d'un signal oscillant à une fréquence bien connue (c.à.d. un signal d'horloge) alors en branchant ce signal sur la «commande» du compteur, on obtient effectivement une mesure du temps : le programme peut littéralement aller «lire l'heure» quand il en a besoin. Mais, un timer est typiquement également capable de déclencher spontanément certaines actions lorsque le temps imparti est écoulé (c'est à dire lorsque le compteur atteint une valeur prédéfinie), par exemple d'envoyer une interruption au processeur.

Dans cette partie du TP, vous allez d'abord configurer le timer pour qu'il mesure la durée voulue, puis l'interroger par scrutation (*polling*) depuis votre programme. Ensuite, dans la partie suivante, vous mettrez en place un mécanisme d'interruptions. Commencez par lire les deux encadrés ci-dessous qui présentent le fonctionnement du timer.

Extrait de la documentation : msp430x4xx.pdf pages 450 et 451

Timer_A is a 16-bit timer/counter with three or five capture/compare registers. Timer_A can support multiple capture/compares, PWM outputs, and interval timing. Timer_A also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers. Timer_A features include :

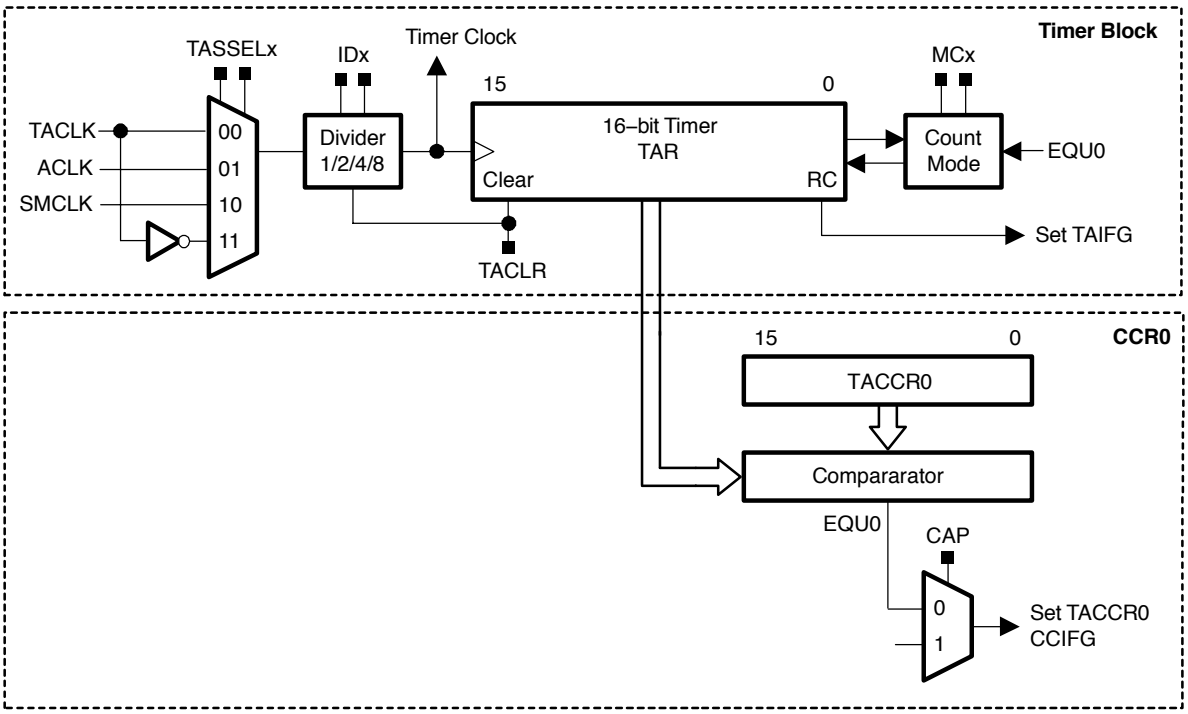
- Asynchronous 16-bit timer/counter with four operating modes
- Selectable and configurable clock source
- Three or five configurable capture/compare registers
- Configurable outputs with PWM capability
- Asynchronous input and output latching
- Interrupt vector register for fast decoding of all Timer_A interrupts



Utile pour le TP : vue simplifiée du Timer_A

Comme vous l'avez vu dans l'encadré page précédente, le timer possède un compteur principal nommé TAR, et cinq blocs identiques dits de «comparaison et capture» nommés CCR0 à CCR4. Chacun de ces blocs CCRx peut travailler soit en mode «capture», soit en mode «comparaison». Le premier mode sert à chronométrer un évènement externe : la valeur de TAR est enregistrée dans TACCRx à l'instant où se produit l'évènement. Le second mode sert à mesurer une durée : la valeur de TAR est en permanence comparée à TACCRx, et le timer génère un évènement lorsque les deux sont identiques (signal EQUx). Le timer est également capable de générer un signal PWM (noté OUTx) mais nous ne nous y intéresserons pas.

Dans ce TP, on ne va utiliser que le canal CCR0, en mode *Compare* (bit CAP à zéro). Vous pouvez donc ignorer la majorité du schéma page précédente pour ne retenir que cette version simplifiée :



Dans ce schéma, chaque petit carré noir est un booléen accessible depuis le logiciel via des registres *memory-mapped* : il s'agit de respectivement de TACTL (*Timer_A Control Register*) et de TACCTL0 (*Capture/Compare Control Register*). Bien sûr, les registres TAR et TACCR0 sont également accessibles depuis le logiciel.

1.1 Choix de la base de temps

Comme expliqué dans l'encadré page suivante, notre timer compte des tops d'horloge, c'est à dire des front montants d'un signal carré. Un multiplexeur nous permet de choisir entre plusieurs sources d'horloge :
TACLK est un signal externe, pris sur une patte d'entrées-sorties de la puce.
INCLK est le complément logique de TACLK.
ACLK est générée par le *Clock Module* de notre puce [1] à une cadence de 32768Hz.
SMCLK est également générée par le *Clock Module* et va 32 fois plus vite que ACLK.

Exercice 2 On souhaite réaliser un chronomètre précis au centième de seconde. Pour chacune des différentes sources d'horloge possibles, combien de cycles faudrait-t-il compter pour mesurer ces 10ms ? Comme on va travailler uniquement en nombres entiers, il nous sera impossible de mesurer exactement la durée voulue. Pour chaque possibilité, calculez la précision relative effectivement obtenue : de combien notre chronomètre va-t-il se tromper à cause de l'erreur d'arrondi ? Exprimez vos différentes réponses en ppm (partie par million) et aussi en secondes par jour.

1. Pour les plus curieux : allez voir la documentation du Clock Module, notamment le schéma page 290

Extrait de la documentation : msp430x4xx.pdf page 452

15.2 Timer_A Operation

The Timer_A module is configured with user software. The setup and operation of Timer_A is discussed in the following sections.

15.2.1 16-Bit Timer Counter

The 16-bit timer/counter register, TAR, increments or decrements (depending on mode of operation) with each rising edge of the clock signal. TAR can be read or written with software. Additionally, the timer can generate an interrupt when it overflows.

TAR may be cleared by setting the TACLR bit. Setting TACLR also clears the clock divider and count direction for up/down mode.

Note: Modifying Timer_A Registers

It is recommended to stop the timer before modifying its operation (with exception of the interrupt enable, interrupt flag, and TACLR) to avoid errant operating conditions.

When the timer clock is asynchronous to the CPU clock, any read from TAR should occur while the timer is not operating or the results may be unpredictable. Alternatively, the timer may be read multiple times while operating, and a majority vote taken in software to determine the correct reading. Any write to TAR takes effect immediately.

Clock Source Select and Divider

The timer clock can be sourced from ACLK, SMCLK, or externally via TACLK or INCLK. The clock source is selected with the TASSELx bits. The selected clock source may be passed directly to the timer or divided by 2, 4, or 8 using the IDx bits. The clock divider is reset when TACLR is set.

1.2 Choix du mode de fonctionnement

Le Timer_A dispose de trois modes opératoires, décrits dans l'encadré page suivante : Up, Down, et Continuous. Chaque mode spécifie une façon différente de compter jusqu'à l'échéance, et aussi quels drapeaux (*flags*) seront levés lorsque le compteur atteint la bonne valeur. Les deux flags qui nous intéressent sont nommés *«TAIFG»* et *«TACCR0 CCIFG»*, et sont également *memory-mapped*, on va donc pouvoir aller les consulter depuis le logiciel. Sous certaines conditions (dépendantes du mode de fonctionnement) le timer lève l'un ou l'autre de ces flags pour indiquer qu'un évènement s'est produit.

Exercice 3 En vous aidant des extraits de documentation reproduits dans les pages suivantes, configurez le timer pour qu'il lève un flag toutes les 10ms.

Exercice 4 Dans votre boucle principale, scrutez l'état du drapeau et n'incrémentez la variable `cpt` qu'à chaque fois que le drapeau est levé. N'oubliez pas de le forcer ensuite à zéro.

Vous devez observer sur l'écran un compteur qui avance à la bonne vitesse. Faites valider par un enseignant.

Extrait de la documentation : msp430x4xx.pdf page 453

15.2.2 Starting the Timer

The timer may be started or restarted in the following ways:

- The timer counts when MCx > 0 and the clock source is active.
- When the timer mode is either up or up/down, the timer may be stopped by writing 0 to TACCR0. The timer may then be restarted by writing a nonzero value to TACCR0. In this scenario, the timer starts incrementing in the up direction from zero.

15.2.3 Timer Mode Control

The timer has four modes of operation as described in Table 15–1: stop, up, continuous, and up/down. The operating mode is selected with the MCx bits.

Table 15–1.Timer Modes

MCx	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TACCR0.
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/down	The timer repeatedly counts from zero up to the value of TACCR0 and back down to zero.

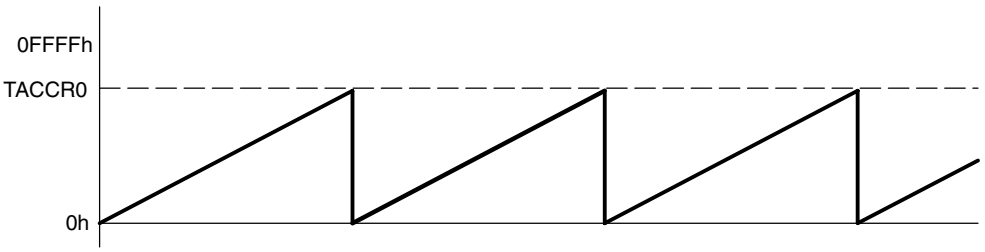
Extrait de la documentation : msp430x4xx.pdf page 467

Register	Short form	Register type	Address	Initial State
Timer_A control	TACTL	Read/write	0160h	Reset with POR
Timer_A counter	TAR	Read/write	0170h	Reset with POR
Timer_A capture/compare control 0	TACCTL0	Read/write	0162h	Reset with POR
Timer_A capture/compare 0	TACCR0	Read/write	0172h	Reset with POR
Timer_A capture/compare control 1	TACCTL1	Read/write	0164h	Reset with POR
Timer_A capture/compare 1	TACCR1	Read/write	0174h	Reset with POR
Timer_A capture/compare control 2	TACCTL2	Read/write	0166h	Reset with POR
Timer_A capture/compare 2	TACCR2	Read/write	0176h	Reset with POR

Up Mode

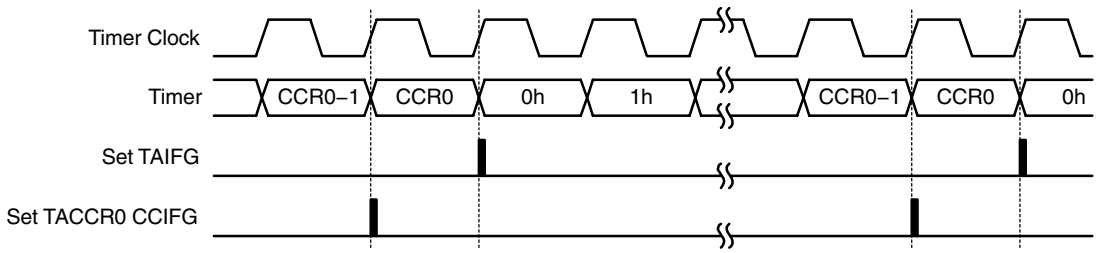
The up mode is used if the timer period must be different from 0FFFFh counts. The timer repeatedly counts up to the value of compare register TACCR0, which defines the period, as shown in Figure 15–2. The number of timer counts in the period is TACCR0+1. When the timer value equals TACCR0 the timer restarts counting from zero. If up mode is selected when the timer value is greater than TACCR0, the timer immediately restarts counting from zero.

Figure 15–2. Up Mode



The TACCR0 CCIFG interrupt flag is set when the timer *counts* to the TACCR0 value. The TAIFG interrupt flag is set when the timer *counts* from TACCR0 to zero. Figure 15–3 shows the flag set cycle.

Figure 15–3. Up Mode Flag Setting



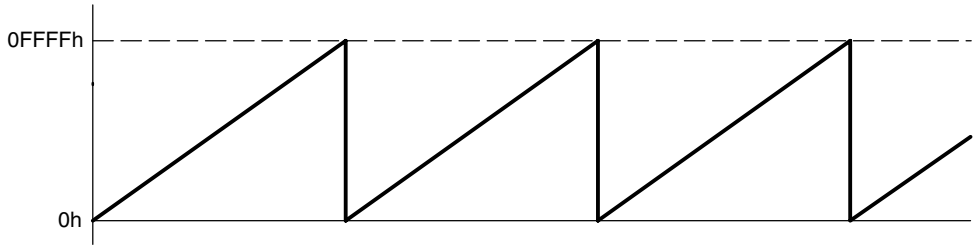
Changing the Period Register TACCR0

When changing TACCR0 while the timer is running, if the new period is greater than or equal to the old period or greater than the current count value, the timer counts up to the new period. If the new period is less than the current count value, the timer rolls to zero. However, one additional count may occur before the counter rolls to zero.

Continuous Mode

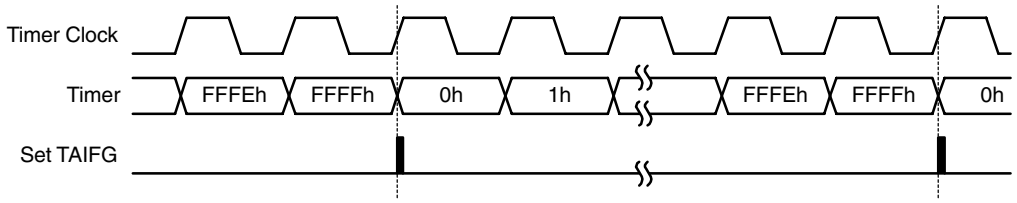
In the continuous mode, the timer repeatedly counts up to 0FFFFh and restarts from zero as shown in Figure 15–4. The capture/compare register TACCR0 works the same way as the other capture/compare registers.

Figure 15–4. Continuous Mode



The TAIFG interrupt flag is set when the timer *counts* from 0FFFFh to zero. Figure 15–5 shows the flag set cycle.

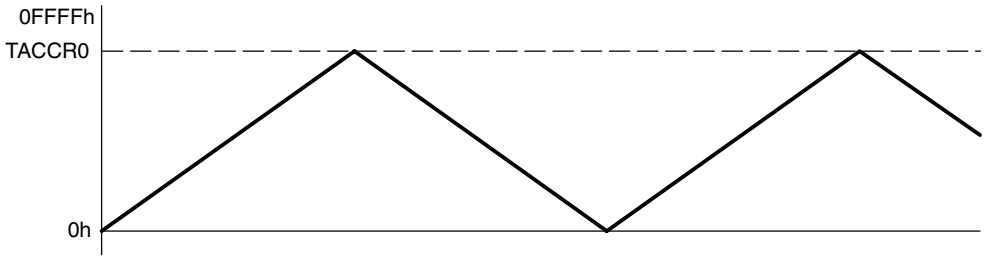
Figure 15–5. Continuous Mode Flag Setting



Up/Down Mode

The up/down mode is used if the timer period must be different from 0FFFFh counts, and if symmetrical pulse generation is needed. The timer repeatedly counts up to the value of compare register TACCR0 and back down to zero, as shown in Figure 15–7. The period is twice the value in TACCR0.

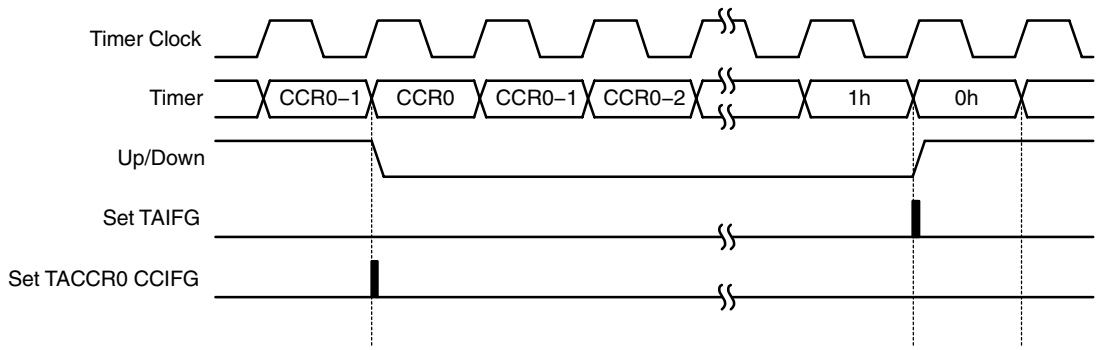
Figure 15–7. Up/Down Mode



The count direction is latched. This allows the timer to be stopped and then restarted in the same direction it was counting before it was stopped. If this is not desired, the TACLR bit must be set to clear the direction. The TACLR bit also clears the TAR value and the clock divider.

In up/down mode, the TACCR0 CCIFG interrupt flag and the TAIFG interrupt flag are set only once during a period, separated by 1/2 the timer period. The TACCR0 CCIFG interrupt flag is set when the timer *counts* from TACCR0 – 1 to TACCR0, and TAIFG is set when the timer completes *counting* down from 0001h to 0000h. Figure 15–8 shows the flag set cycle.

Figure 15–8. Up/Down Mode Flag Setting



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)	w–(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 Inverted TACLK

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

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

2 Interruptions matérielles

Dans la partie précédente, même si c'est un *timer* qui mesure le passage du temps proprement dit, votre programme passe son temps à scruter ce timer. Ce n'est pas très satisfaisant, car le CPU est inutilement monopolisé. Vous aurez remarqué, par exemple, que la diode rouge ne clignote plus, ou plutôt, elle clignote trop vite pour qu'on le perçoive à l'œil nu. En effet, vous avez dû retirer la boucle de temporisation, afin de ne pas rater d'évènements de la part du timer. Dans cette partie du TP, on va donc mettre en place un mécanisme d'interruptions afin de libérer le CPU.

Exercice 5 Commencez par déplacer dans une nouvelle fonction l'incrémentation et l'affichage :

```
// This will get executed 100 times per second
void mon_traitant_interruption_timer(void)
{
  lcd_display_number(cpt);
  cpt++;
}
```

Remettez aussi la boucle de temporisation dans `main()`.

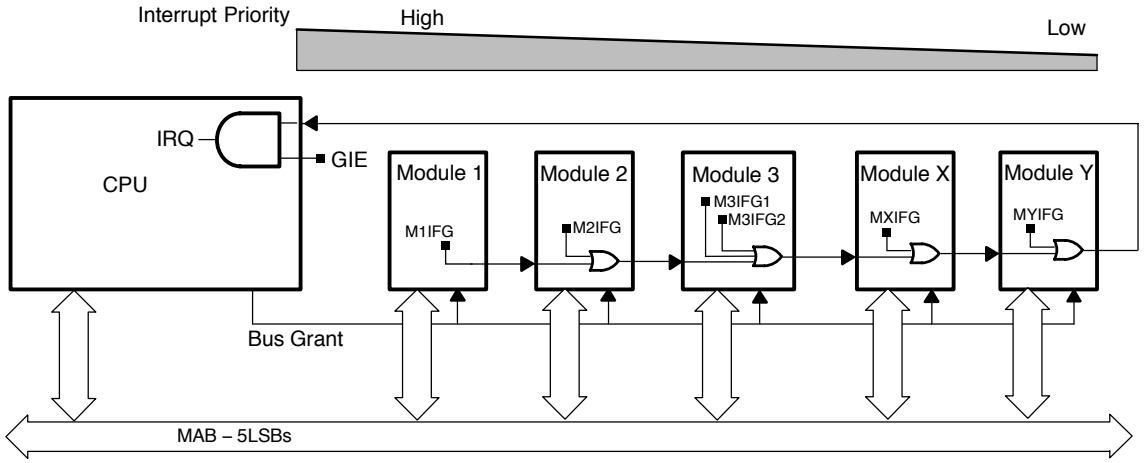
Les exercices qui suivent vous guident dans les manipulations nécessaires pour que cette fonction soit traitée comme un *traitant d'interruption*. Elle sera donc invoquée automatiquement à chaque centième de seconde, de façon transparente pour le programme principal.

Mais dans un premier temps, lisez l'encadré page suivante pour découvrir le fonctionnement des interruptions sur l'architecture MSP430.

Utile pour le TP : les interruptions sur le MSP430

Le fonctionnement des interruptions est détaillé au chapitre 2.2 de la documentation (msp430x4xx.pdf pages 29 et suivantes) et nous en reprenons les grandes lignes ici. Le CPU du MSP430 ne dispose pas d'une ligne d'interruption distincte pour chaque périphérique, mais d'une seule ligne partagée par tous les périphériques. Un composant (par exemple le Module 2) qui veut lever une interruption fait passer son *interrupt flag* à 1 (dans notre exemple, il s'agit donc du bit M2IFG). Certains modules ont plusieurs drapeaux, mais le fonctionnement reste similaire (les petits carrés noirs du schéma représentent des bits accessibles en mémoire dans un registre matériel). Le signal traverse les autres périphériques et atteint le processeur. Celui-ci perçoit donc une requête d'interruption (IRQ) lorsque :

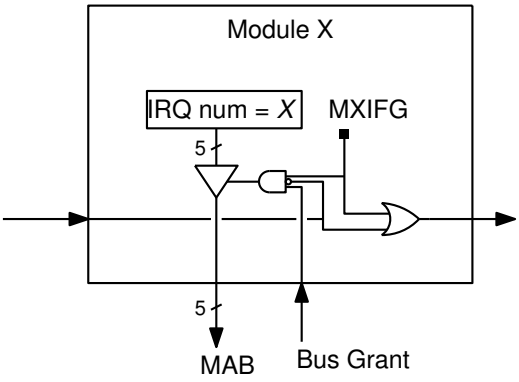
— au moins un des périphériques a un *interrupt flag* levé,
— et le bit GIE du registre SR est vrai (bit *Global Interrupt Enable* du *Status Register*)



Pour savoir de qui vient la requête, le processeur passe alors le signal *Bus Grant* à 1. Mais il se peut que plusieurs périphériques aient des flags levés, et dans ce cas-là on veut les départager par ordre de priorité. Chaque module, grâce à la circuiterie illustrée ci-dessous, émet donc son propre numéro si et seulement si :

— ce module a une interrution en attente (i.e. son *interrupt flag* est levé),
— et aucun module plus prioritaire n'a d'interruption en attente (cf fil de gauche sur le schéma),
— et le signal *Bus Grant* venant du CPU est actif (cf fil venant du bas).

Il y a donc bien un et un seul numéro d'IRQ envoyé au processeur (dans notre exemple, le nombre 2 codé sur cinq bits : 0b00010).



Lorsqu'il reçoit ce numéro, le CPU l'utilise comme indice dans la table des vecteurs, et charge le vecteur dans PC, ce qui revient à sauter à l'adresse de l'ISR. [a] La table est située en mémoire flash à l'adresse 0xFFC0, donc le vecteur numéro x est le mot d'adresse 0xFFC0+2x.

a. pour plus de détails, vous pouvez lire l'encadré page 19

La partie précédente du TP nous a déjà fait bien avancer : les booléens levés à chaque échéance sont

précisément des *interrupt flags*. Il ne nous reste plus qu'à les transformer en IRQ.

Exercice 6 Le timer_A est associé à deux vecteurs distincts (nommés respectivement *«TACCR0 interrupt vector»* et *«TAIV interrupt vector»*). Chacun de ces vecteurs est associé à un ou plusieurs *interrupt flags*. Lisez l'encadré page suivante et choisissez avec quel vecteur vous voulez travailler[2]. Configurez le timer pour qu'il envoie une IRQ au processeur à chaque centième de seconde. (Ne vous occupez pas du côté CPU pour l'instant, c'est l'objet des exercices suivants.)

2. indice : le TP sera plus facile si vous choisissez le premier

15.2.6 Timer_A Interrupts

Two interrupt vectors are associated with the 16-bit Timer_A module:

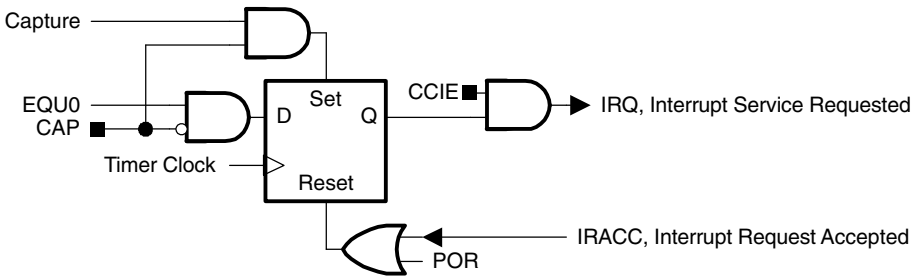
- TACCR0 interrupt vector for TACCR0 CCIFG
- TAIV interrupt vector for all other CCIFG flags and TAIFG

In capture mode any CCIFG flag is set when a timer value is captured in the associated TACCRx register. In compare mode, any CCIFG flag is set if TAR *counts* to the associated TACCRx value. Software may also set or clear any CCIFG flag. All CCIFG flags request an interrupt when their corresponding CCIE bit and the GIE bit are set.

TACCR0 Interrupt

The TACCR0 CCIFG flag has the highest Timer_A interrupt priority and has a dedicated interrupt vector as shown in Figure 15–15. The TACCR0 CCIFG flag is automatically reset when the TACCR0 interrupt request is serviced.

Figure 15–15. Capture/Compare TACCR0 Interrupt Flag



TAIV, Interrupt Vector Generator

The TACCR1 CCIFG, TACCR2 CCIFG, and TAIFG flags are prioritized and combined to source a single interrupt vector. The interrupt vector register TAIV is used to determine which flag requested an interrupt.

The highest priority enabled interrupt generates a number in the TAIV register (see register description). This number can be evaluated or added to the program counter to automatically enter the appropriate software routine. Disabled Timer_A interrupts do not affect the TAIV value.

Any access, read or write, of the TAIV register automatically resets the highest pending interrupt flag. If another interrupt flag is set, another interrupt is immediately generated after servicing the initial interrupt. For example, if the TACCR1 and TACCR2 CCIFG flags are set when the interrupt service routine accesses the TAIV register, TACCR1 CCIFG is reset automatically. After the `RETI` instruction of the interrupt service routine is executed, the TACCR2 CCIFG flag generates another interrupt.

À ce stade, votre timer est correctement configuré, mais il nous reste encore à convaincre le processeur de traiter correctement les requêtes qu'il reçoit. Pour cela, on va avoir besoin de deux choses : faire pointer le bon vecteur d'interruption vers notre routine de traitement, et *activer* les interruptions côté CPU (en passant

à 1 le bit GIE, cf encadré page 13).

Exercice 7 Lisez l'encadré ci-dessous. Quel vecteur d'interruption allez-vous faire pointer sur votre fonction ? À quelle adresse est-t-il situé en mémoire ? Repérez l'emplacement de la table des vecteurs sur la *memory map* du TP2.

Extrait de la documentation : datasheet.pdf page 13

The interrupt vectors and the power-up start address are in the address range 0FFFFh to 0FFC0h. The vector contains the 16-bit address of the appropriate interrupt-handler instruction sequence.

Table 6-3. Interrupt Sources, Flags, and Vectors

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-Up External Reset Watchdog Flash Memory	WDTIFG KEYV (1) (2)	Reset	0FFFEh	31, highest
NMI Oscillator Fault Flash Memory Access Violation	NMIIFG (1) (3) OFIFG(1) (3) ACCVIFG(1) (4)(2)	(Non)maskable (Non)maskable (Non)maskable	0FFFCh	30
Timer_B7	TBCCR0 CCIFG0(4)	Maskable	0FFFAh	29
Timer_B7	TBCCR1 CCIFG1 to TBCCR6 CCIFG6, TBIFG(1)(4)	Maskable	0FFF8h	28
Comparator_A	CAIFG	Maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	Maskable	0FFF4h	26
USCI_A0, USCI_B0 Receive	UCA0RXIFG, UCB0RXIFG(1)	Maskable	0FFF2h	25
USCI_A0, USCI_B0 Transmit	UCA0TXIFG, UCB0TXIFG (1)	Maskable	0FFF0h	24
ADC12	ADC12IFG (1) (4)	Maskable	0FFEEh	23
Timer_A3	TACCR0 CCIFG0(4)	Maskable	0FFECh	22
Timer_A3	TACCR1 CCIFG1 and TACCR2 CCIFG2, TAIFG(1) (4)	Maskable	0FFEAh	21
I/O Port P1 (Eight Flags)	P1IFG.0 to P1IFG.7(1) (4)	Maskable	0FFE8h	20
USART1 Receive	URXIFG1	Maskable	0FFE6h	19
USART1 Transmit	UTXIFG1	Maskable	0FFE4h	18
I/O Port P2 (Eight Flags)	P2IFG.0 to P2IFG.7 (1) (4)	Maskable	0FFE2h	17
Basic Timer 1, RTC	BTIFG	Maskable	0FFE0h	16
DMA	DMA0IFG, DMA1IFG, DMA2IFG(1) (4)	Maskable	0FFDEh	15
DAC12	DAC12.0IFG, DAC12.1IFG(1) (4)	Maskable	0FFDCh	14
Reserved	Reserved(5)		0FFDAh	13
			⋮	⋮
			0FFC0h	0, lowest

(1) Multiple source flags
(2) Access and key violations, KEYV and ACCVIFG, only applicable to FG devices.
(3) A reset is generated if the CPU tries to fetch instructions from within the module register memory address range (0h to 01FFh). (Non)maskable: the individual interrupt-enable bit can disable an interrupt event, but the general-interrupt enable cannot disable it.
(4) Interrupt flags are located in the module.
(5) The interrupt vectors at addresses 0FFDAh to 0FFC0h are not used in this device and can be used for regular program code if necessary.

Exercice 8 Lisez l'encadré page suivante, et modifiez votre code pour faire pointer sur votre routine le vecteur que vous avez choisi.

Utile pour le TP : les routines d'interruptions avec `mspgcc`

Sur le MSP430, la table des vecteurs d'interruptions est implémentée par une zone de mémoire flash, située à une adresse bien connue (0xFFC0). Notre programme ne pourra donc pas écrire dedans ! En effet, le contenu de la mémoire flash est fixé une fois pour toute au moment du transfert du programme vers la carte : en plus du programme proprement dit, l'image mémoire qui est transférée sur la cible contient également toutes les données à écrire en flash, et notamment cette table des vecteurs d'interruption.

Ainsi, lorsqu'on travaille sur MSP430, ce n'est pas le programme, mais la chaîne de compilation (compilateur, assembleur, éditeur de liens, chargeur) qui est responsable de construire la table des vecteurs d'interruption et de la transférer au bon endroit de la mémoire.

À cette fin, le compilateur offre au programmeur des extensions (c.à.d. des formes de syntaxe qui ne font pas partie du langage C standard) afin qu'il puisse directement spécifier les vecteurs d'interruption dans le texte du programme. Pour notre version de mspgcc, la syntaxe est la suivante :

```
void __attribute__ ((interrupt(INTERRUPT_VECTOR_NAME))) routine_name(void)
{
   /* interrupt handling code */
}
```

L'effet de cette décoration est double :

— il place l'adresse de la routine dans le bon vecteur d'interruption,

— et il ordonne au compilateur de traduire cette fonction de façon particulière : le proloque de l'ISR va faire une *sauvegarde* du contexte d'exécution (i.e. le contenu des registres du CPU) et l'épilogue se termine par une *restauration* de ce contexte. De cette façon, notre fonction peut être exécutée à n'importe quel moment, elle n'interférera pas avec le fonctionnement du programme principal.

La mention `INTERRUPT_VECTOR_NAME` doit être remplacée par un code correspondant à l'IRQ souhaitée. Vous trouverez les différentes valeurs possibles tout en bas du fichier `/usr/msp430/include/msp430fg4816.h`

Exercice 9 Compilez votre programme et transférez-le sur la carte. Ouvrez le fichier `.lst` dans un éditeur de texte. Avec mspdebug, examinez le contenu de table des vecteurs (commande `md`) et identifiez où pointe chacun d'eux.

Exercice 10 Il ne vous reste plus qu'à activer les interruptions côté processeur, grâce au bit GIE.
Vous avez plusieurs options pour réaliser cette opération. D'une façon générale, elle n'est pas exprimable directement en C puisque vous devez accéder à un registre du processeur. Il faut donc passer par des instructions assembleur encapsulées dans le code C. Pour cela, vous pouvez utiliser la syntaxe suivante : `__asm("INSTRUCTION");` (vous avez vu la liste des instructions du MSP430 dans le TP1, et elles sont également documentées individuellement en détail dans le manuel msp430x4xx.pdf à partir de la page 61.)
Une autre option est d'utiliser une «fonction intrinsèque», c'est à dire une forme syntaxique similaire à un appel de fonction, qui sera traduit non pas en un `CALL` mais en une séquence d'instructions spécifiques. Les intrinsèques disponibles sont listées dans le fichier d'en-tête `/usr/msp430/include/intrinsics.h`

Exercice 11 En principe c'est gagné. Sinon, faites la mise au point (aka *debugging*) de votre progamme jusqu'à vous assurer que toutes vos réponses sont justes depuis l'exercice 5. Vous devez obtenir *et* le compteur qui avance à 100Hz sur l'écran, *et* la diode qui clignote lentement. Faites valider par un enseignant.

Remarque Attention, si vous avez choisi de travailler avec le second vecteur (TAIFG) alors c'est plus difficile (on vous avait prévenus). Relisez la page 15 jusqu'à comprendre ce que vous devez rajouter à votre programme pour qu'il fonctionne correctement. Vous trouverez aussi des indices dans l'encadré de la page 19.

3 Étude du mécanisme d'interruptions

Exercice 12 Pour cette partie, vous devez modifier votre Makefile (ou équivalent) pour y rajouter parmi les options de compilation [3] l'option `-mcpu=430`. Une invocation du compilateur ressemble donc maintenant à ça :

```
msp430-gcc -mcpu=430 -mmcu=msp430fg4618 -Wall -Werror -O1 -c -o truc.o truc.c
```

Exercice 13 Donnez le code ASM généré par gcc pour `mon_traitant_interruption_timer()`. Ligne par ligne, expliquez ce que fait chaque instruction assembleur. Compilez la même fonction en retirant l'attribut `interrupt` et comparez les deux versions du code en commentant les différences. On va étudier ces différences de plus près dans les quelques exercices ci-dessous.

Exercice 14 En particulier, combien de registres de contexte sont sauvegardés, et pourquoi ? (Justifiez votre réponse en vous appuyant ce que vous avez appris lors des TP précédents et sur la documentation en ligne du compilateur : `http://mspgcc.sourceforge.net/manual/`).

Exercice 15 Lisez les encadrés page suivante et page 20, et expliquez la différence entre les instructions RET et RETI. Pourquoi a-t-on besoin de deux instructions différentes ?

3. pour les plus curieux : il s'agit d'interdire au compilateur d'utiliser le jeu d'instruction étendu «MSP430X». Si ça vous amuse, allez regardez les différences, qui sont expliquées au chapitre 4 de msp430x4xx.pdf

2.2.3 Interrupt Processing

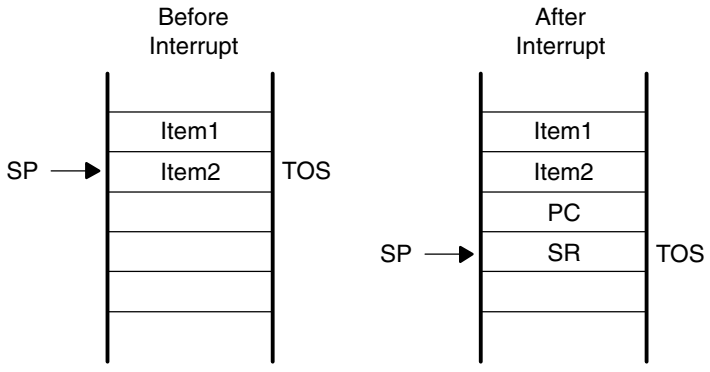
When an interrupt is requested from a peripheral and the peripheral interrupt enable bit and GIE bit are set, the interrupt service routine is requested. Only the individual enable bit must be set for (non)-maskable interrupts to be requested.

Interrupt Acceptance

The interrupt latency is six cycles, starting with the acceptance of an interrupt request and lasting until the start of execution of the first instruction of the interrupt-service routine, as shown in Figure 2–6. The interrupt logic executes the following:

1) Any currently executing instruction is completed.
2) The PC, which points to the next instruction, is pushed onto the stack.
3) The SR is pushed onto the stack.
4) The interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction and are pending for service.
5) The interrupt request flag resets automatically on single-source flags. Multiple source flags remain set for servicing by software.
6) The SR is cleared with the exception of SCG0, which is left unchanged. This terminates any low-power mode. Because the GIE bit is cleared, further interrupts are disabled.
7) The content of the interrupt vector is loaded into the PC: the program continues with the interrupt service routine at that address.

Figure 2–6. Interrupt Processing



Return From Interrupt

The interrupt handling routine terminates with the instruction:

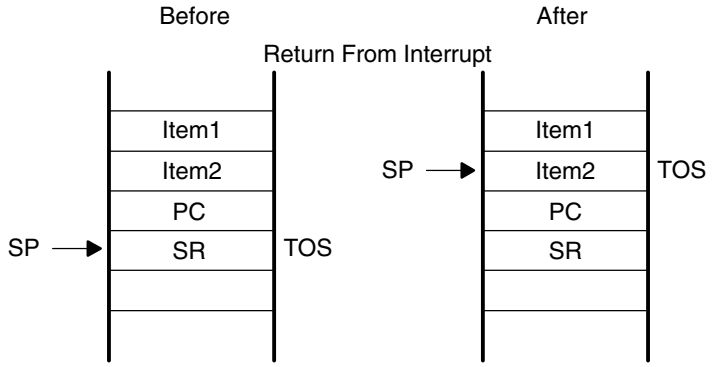
`RETI` (return from an interrupt service routine)

The return from the interrupt takes 5 cycles to execute the following actions and is illustrated in Figure 2–7.

1) The SR with all previous settings pops from the stack. All previous settings of GIE, CPUOFF, etc. are now in effect, regardless of the settings used during the interrupt service routine.

2) The PC pops from the stack and begins execution at the point where it was interrupted.

Figure 2–7. Return From Interrupt



Interrupt nesting is enabled if the GIE bit is set inside an interrupt service routine. When interrupt nesting is enabled, any interrupt occurring during an interrupt service routine will interrupt the routine, regardless of the interrupt priorities.

Exercice 16 Mettez un point d'arrêt (avec `setbreak`) à l'entrée de `lcd_display_number()` et exécutez le programme jusque là. Examinez le contenu de la mémoire (avec `md`) et recopiez dans le tableau ci-dessous le contenu de la pile (toujours un mot par ligne, toujours little-endian). Dans la colonne «divers», indiquez la provenance de chaque valeur : quelle est l'instruction (ou autre) qui a écrit cette valeur ici.

addresse	hexa	(décimal)	divers

4 Interruption sur bouton poussoir

Maintenant qu'on dispose d'un chronomètre précis au centième de seconde, on va lui rajouter une interface utilisateur. Plus précisément, vous allez faire en sorte que les boutons permettent de contrôler le programme. Bien sûr, on pourrait faire ça en scrutation, et d'ailleurs on l'avait fait au TP2. Mais pour ne pas monopoliser le CPU, on va cette fois encore utiliser les interruptions matérielles. Il se trouve que les ports GPIO sur lesquels sont branchés les boutons sont justement capables d'envoyer des IRQ au processeur lorsqu'ils détectent un évènement.

Exercice 17 En vous aidant de l'encadré page suivante, configurez le port P1 pour qu'il génère une interruption lors de l'appui sur un bouton. Écrivez un traitant d'interruption qui réinitialise le compteur.

Exercice 18 Comme le port P1 ne dispose que d'un seul vecteur d'interruption, la pression sur l'un ou sur l'autre des boutons déclenche l'exécution du même ISR. Modifiez votre code pour que SW1 remette le chronomètre à zéro, et que SW2 le mette en pause.

11.2.6 P1 and P2 Interrupts

Each pin in ports P1 and P2 have interrupt capability, configured with the PxIFG, PxIE, and PxIES registers. All P1 pins source a single interrupt vector, and all P2 pins source a different single interrupt vector. The PxIFG register can be tested to determine the source of a P1 or P2 interrupt.

Interrupt Flag Registers P1IFG, P2IFG

Each PxIFGx bit is the interrupt flag for its corresponding I/O pin and is set when the selected input signal edge occurs at the pin. All PxIFGx interrupt flags request an interrupt when their corresponding PxIE bit and the GIE bit are set. Each PxIFG flag must be reset with software. Software can also set each PxIFG flag, providing a way to generate a software-initiated interrupt.

Bit = 0: No interrupt is pending

Bit = 1: An interrupt is pending

Only transitions, not static levels, cause interrupts. If any PxIFGx flag becomes set during a Px interrupt service routine or is set after the `RETI` instruction of a Px interrupt service routine is executed, the set PxIFGx flag generates another interrupt. This ensures that each transition is acknowledged.

Note: PxIFG Flags When Changing PxOUT or PxDIR

Writing to P1OUT, P1DIR, P2OUT, or P2DIR can result in setting the corresponding P1IFG or P2IFG flags.

Note: Length of I/O Pin Interrupt Event

Any external interrupt event should be at least 1.5 times MCLK or longer, to ensure that it is accepted and the corresponding interrupt flag is set.

Interrupt Edge Select Registers P1IES, P2IES

Each PxIES bit selects the interrupt edge for the corresponding I/O pin.

Bit = 0: The PxIFGx flag is set with a low-to-high transition

Bit = 1: The PxIFGx flag is set with a high-to-low transition

Note: Writing to PxIESx

Writing to P1IES or P2IES can result in setting the corresponding interrupt flags.

PxIESx	PxINx	PxIFGx
0 → 1	0	May be set
0 → 1	1	Unchanged
1 → 0	0	Unchanged
1 → 0	1	May be set

Interrupt Enable P1IE, P2IE

Each PxIE bit enables the associated PxIFG interrupt flag.

Bit = 0: The interrupt is disabled

Bit = 1: The interrupt is enabled