TP3, 5TC-BED, ez430-RF2500
Timer et interruption

October 1, 2019

But du TP

Ce TP a pour but de prendre en main les horloges, les timers et le mécanisme fondamental dans l'embarqué des interuptions.

1 Programmation à l'aide des timers

Horloge sur le MSP430

La gestion des horloges est un travail important du programmeur embarqué. Il est important de savoir qu'il existe plusieurs manières de cadencer les différents modules du MSP. Nous référencerons les pages du manuel utilisateur du MSP430F2274 (fichier `MSP430x2xx_Family_Users_Guide.pdf`) disponible sous Moodle. Les horloges sont expliquées en détail au chapitre 5 (Basic Clock module), par manque de temps, nous n'étudieront pas en détail ce module, mais nous utiliserons des configuration déjà proposée dans le pilote `ez430-drivers/src/clock.c`.

En résumé: le `MSP430` possède 4 sources pour ses horloges: deux oscillateurs internes (DCOCLK pour *digitally controlled oscillator* et VLOCLK pour *very low power, low frequency oscillator*) et deux oscillateurs pouvant utiliser une source externe allant jusqu'à 16MHz (LFXT1CLK et XT2CLK). LFXT1CLK qui peut utiliser un cristal (XT1). il n'y a pas de crystal sur la clé EZ430, c'est donc à partir d'oscillateurs interne (habituellement DCO) que les horloges vont être générées (ces oscillateurs sont peu précis et notamment sensibles à la température)

À partir de ces oscilateurs sources, le `MSP430` produit, en divisant plus ou moins les horloges sources, trois horloges utilisées par le système: MCLK (Master clock), SMCLK (Sub-main clock) et ACLK (Auxiliary clock). Les sources et diviseurs de ces différentes horloges peuvent être configurés comme expliqué sur la figure 5-1 de la documentation du `MSP430` reproduit en annexe ici (p. 9). Par défaut MCLK et SMCLK utilisent DCOCLK à 1,1Mhz comme source. Le driver `clock.c` contient un certain nombre de fonction permettant de configurer les horloges à différentes fréquences.

Timer sur le MSP430

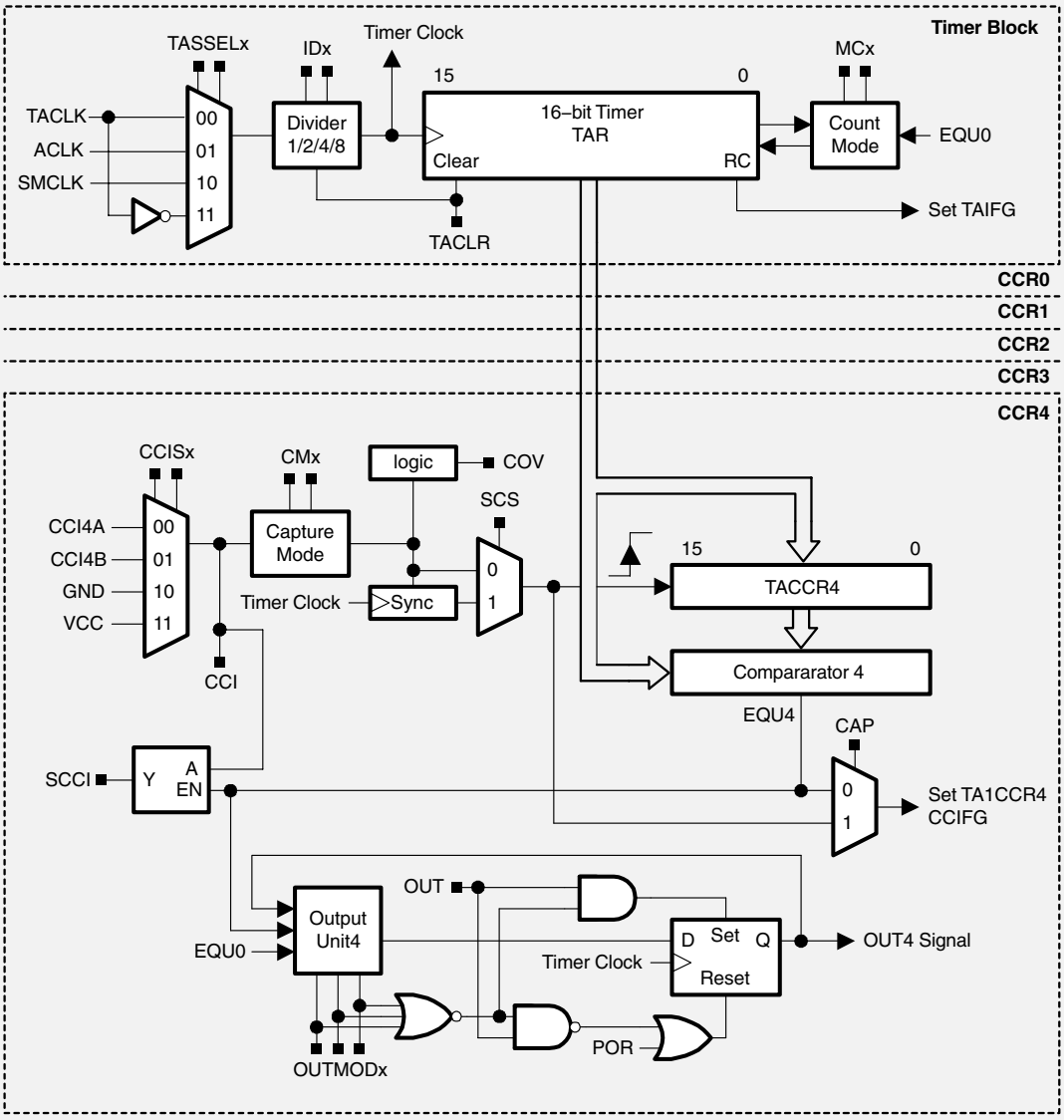
Le MSP430 contient 3 timers (détaillés au chapitre 12 du manuel utilisateur de MSP) qui fonctionnent sur le même principe: il sont initialisés à une certaine valeur, ils incrémentent (ou décrémentent) un registre à chaque top d'horloge et envoient une interruption lorsque ce registre atteint une certaine valeur stockée dans un CCR (capture/compare register). Chaque timer comporte 3 CCR. Le *watchdog timer* envoie systématiquement un reset, il est utilisé pour maintenir les systèmes en vie, si le programme ne ré-initialise pas régulièrement le timer, le MSP est redémarré.

Le timer A est contrôlé par un registre nommé `TACTL`, le registre qui "compte" est nommé `TAR` (`TBCTL` et `TBR` pour le timer B).

Extrait de la documentation: msp430x3xx.pdf pages 394 et 395

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:

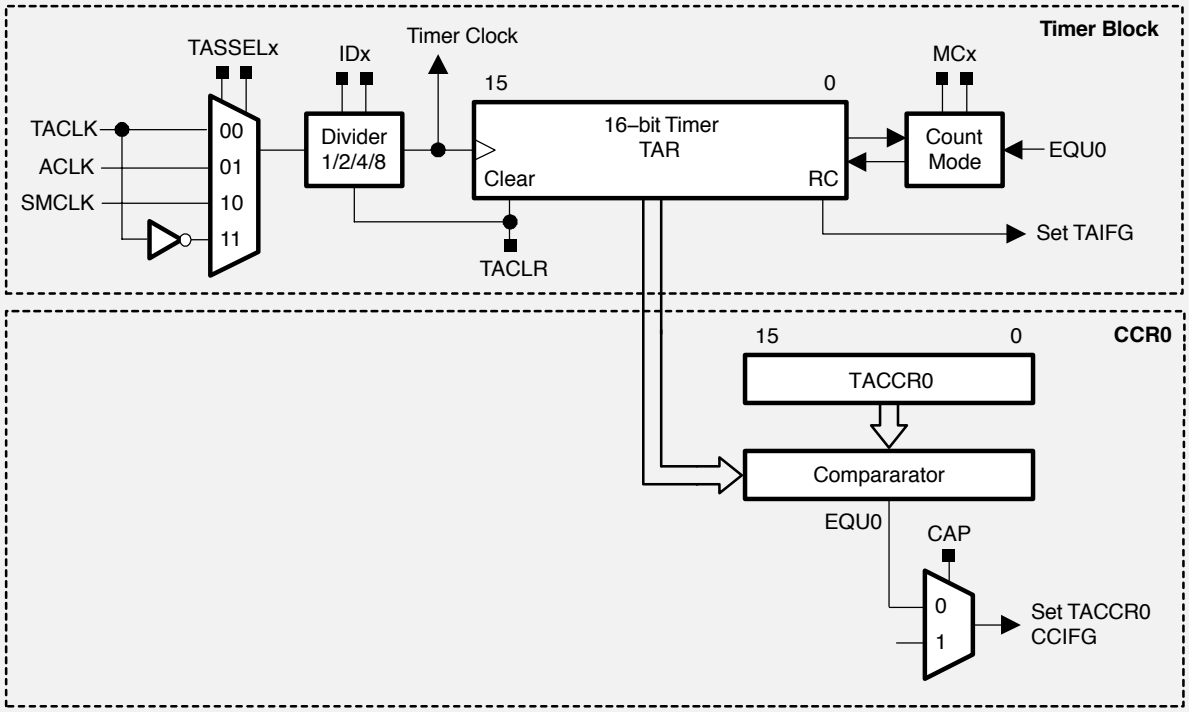
- Asynchronous 16-bit timer/counter with four operating modes
- Selectable and configurable clock source
- Three or five configurable capture/compare registers
- Configurable outputs with PWM capability
- Asynchronous input and output latching
- Interrupt vector register for fast decoding of all Timer_A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

2 Programmer un handler d'interruption sur le `MSP430`

Dans cette partie du TP, on va donc mettre en place un mécanisme d'interruptions afin de libérer le CPU. Attention, comme nous le verrons plus loin la librairie libez430.a que vous avez compilé au début du cours utilise un mécanisme qui ne permet pas de définir directement les interruptions (utilisation de la macro ISR proposée dans `isr_compat.h` par TI), nous allons donc devoir écrire un programme sans les drivers développés dans ez430-driver.

Exercice 1 Commencez par créer le nouveau programme ci-dessous. Lorsque vous le chargez sur la clé, la diode ne doit plus clignoter puisqu'aucune fonction n'a été déclarée comme traitant l'interruption venant du timerA.

```
#include <msp430.h>
#include <legacymsp430.h>
```

```
mon_traitant_interruption_timer(void)
{
      // switch red and green led
      P1OUT ^= BIT0;
      P1OUT ^= BIT1;
}

int main(void)
{
  WDTCTL = WDTPW | WDTHOLD; // watchdog_stop();

  /* leds init */
  P1OUT &= ~(BIT1 | BIT0);
  P1DIR |= (BIT1 | BIT0);
  P1SEL &= ~(BIT1 | BIT0);


  /* timer A initial settings */
  BCSCTL3 |= LFXT1S_2;      // LFXT1 = VLO
  TACCTL0 = 0;           // TCCR0 interrupt disable
  TAR = 0;
  TACCR0 = 20000;           // 20000 ticks ~= 1 seconde
  TACTL = TASSEL_1 + MC_1;  // ACLK, upmode

  /* global enable interrupt */
  eint();


  while (1)
    {
      LPM1;
    }
  return 0;
}
```

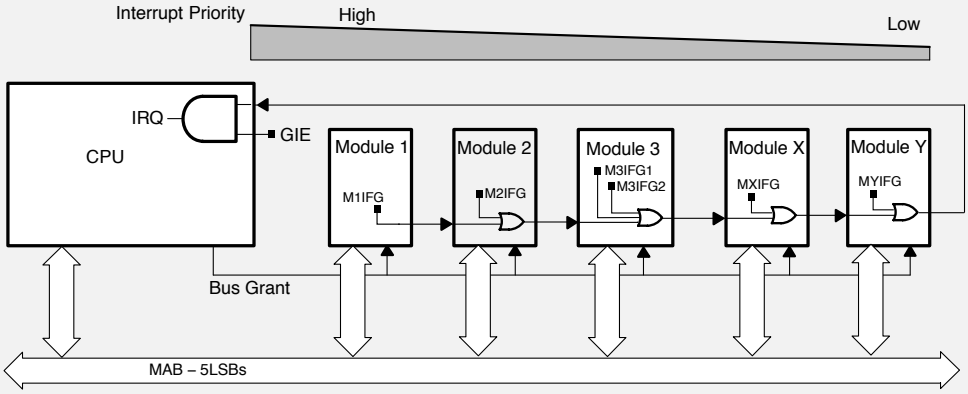
Les exercices qui suivent vous guident dans les manipulations nécessaires pour que le timer envoit une interruption tout les 20000 ticks et que la fonction `mon_traitant_interruption_timer` joue le rôle de *traitant d'interruption*. Elle sera donc invoquée automatiquement à chaque 20000 ticks (environ 1 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 (msp430x2xx.pdf pages 27 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, par exemple TAIFG pour l'interrupt flag du timer A). 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 d'interruption arrive directement au processeur processeur. Celui-ci perçoit donc une requête d'interruption (IRQ) lorsque:

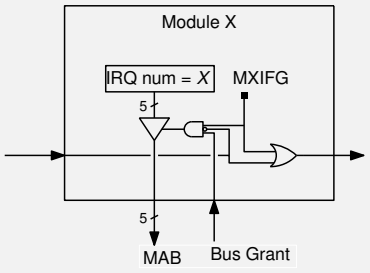
- au moins un des périphériques a un *interrupt flag* levé,
- et le bit GIE du registre SR est vrai (bit *Global Interrupt Enable* du *Status Register*)



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 la routine de traitement de l'interrotion (ISR pour *interrupt service routine*).[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 10

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

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

Exercice 2 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*. Dans ce TP, on va travailler avec TACCR0 qui est un peu plus simple parce que le CPU fait automatiquement l'*acknowledge* une fois l'interruption détectée. la macro à considérer pour activer les interruption est doc TACCTL0 (voir l'extrait de la documentation du Timer A en annexe (p 11)). Modifier l'instruction mettant a jour `TACTCTLO` pour que le timer envoie une IRQ au processeur à chaque de seconde. Ne vous occupez pas du côté CPU pour l'instant, c'est l'objet des exercices suivants mais faites valider par l'enseignant.

Extrait de la documentation: msp430x4xx.pdf page 465

15.2.6 Timer_A Interrupts

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

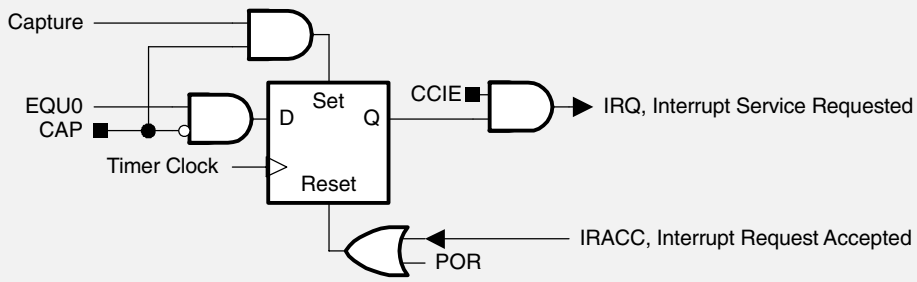
- TACCR0 interrupt vector for TACCR0 CCIFG
- TAIV interrupt vector for all other CCIFG flags and TAIFG

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.

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 (Attention, cette sataxe diffère pour d'autre compilateur, comme IAR par exemple, ou pour d'autre version de mspgcc) :

```
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 3 Après avoir lu l'envadré ci-dessus, et sachant que le vecteur d'interrption pour TACCR0 se nomme `TIMERA0_VECTOR` et modifiez votre code pour que votre fonction `mon_traitant_interruption_timer` soit compilée comme la routine de traitement de l'interuption du TimerA0.

Exercice 4 Dans la solution précédente, c'est la routine d'interruption qui changeait l'état des diode. Faites une autre version du programme dans laquelle, la routine d'interruption fait sortir le MSP de l'état LPM1 et le changement d'état des diodes est fait dans la boucle `while`.

3 Programmation du timer en utilisant le driver timer.c

Le but de cette seconde partie est de prendre en main l'interface de programmation des timer proposée par le driver `timer.c` qui simplifie l'utilisation des timers disponibles sur le MSP430. le fichier `ez430-driver/src/timer.c` utilise un fichier `isr_compat.h` qui définit la macro `ISR` qui permet de définir des handler d'interruption compatible avec d'autre compilateurs.

- Déplacez vous dans le répertoire `$SRC/ez430-drivers/src/`
- Ouvrez le fichier *timer.c*
- Identifiez le traitant d'interruption associé au *timerA*.

Q1— Que fait ce traitant ?

- Identifiez maintenant la fonction *timerA_register_cb*.

Q2— Que fait cette fonction ?

- Continuons à descendre, identifiez maintenant la fonction *timerA_start_ticks*.

Q3— En vous aidant du chapitre consacré aux timers dans la documentation du MSP430 (fichier `MSP430x2xx_Family_User's_Gu` chapitre 12), analysez précisément chaque ligne de cette fonction.

Programmation

Placez vous dans le répertoire SRC/ez430-drivers/example/timer. Ouvrez le fichier *main.c*.

Q4— Que font les lignes :

```
timerB_init();
timerB_register_cb(green_led_cb);
timerB_start();
```

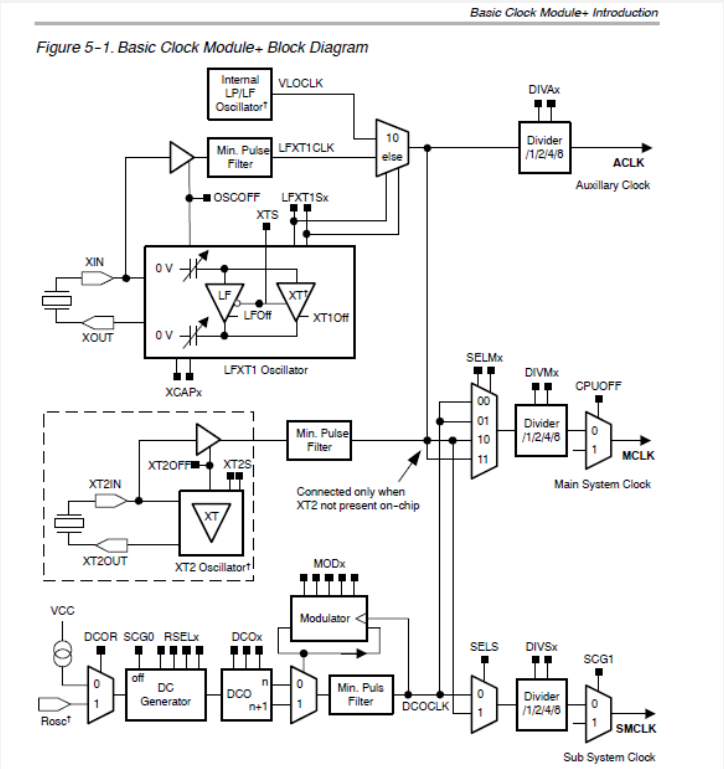
Q5— Complétez le programme implémenté dans *main.c* afin que la led **rouge** s'éclaire deux fois plus souvent que la led verte.

Conclusion

Q6— Dans le premier TP, nous avons programmé une attente de manière différente (exécution de '`nop`', Nous voyons ici une manière plus efficace de de programmer une attente: grâce aux interuptions produite par le timer, nous allons pouvoir, soit endormir le `MSP430`, soit exécuter une autre partie du code pendant l'attente.

A Annexe: Basic Clock Module Diagram

Extrait de la documentation: msp430x2xx.pdf page 5.3 (289)



B Annexe: détails concernant les interruptions

Extrait de la documentation: msp430x4xx.pdf page 34

2.2.3 Interrupt Processing

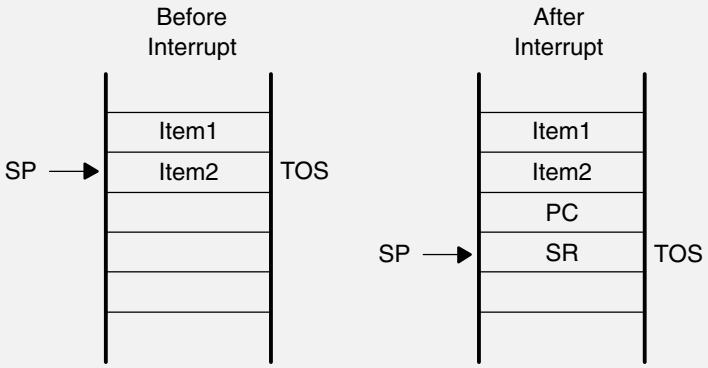
When an interrupt is requested from a peripheral and the peripheral interrupt enable bit and GIE bit are set, the interrupt service routine is requested. Only the individual enable bit must be set for (non)-maskable interrupts to be requested.

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



Extrait de la documentation: msp430x2xx.pdf page 415

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