Architecture des Ordinateurs — MSP430 — TP 3

Programmation par interruptions

B3145 I Merlin NIMIER-DAVID & Robin RICARD

Remise en jambe

1. On met le premier bit du port `P1` [motherboard p19] dans la direction de lecture et on lit dedans à chaque cycle pendant l'attente [msp430 11.2.3]. Il correspond à la broche 87 du MSP [datasheet p3].

```
P1DIR = P1DIR & 0xFC; // Configure push buttons as input
for (;;)
{
    // While button is not pressed
    while ( (P1IN & 0x01) != 0x00 );
    // While button has not been released
    while ( (P1IN & 0x01) == 0x00 );

    // ...
}
```

Interruption sur *timer*

Configuration du timer

2. Le `Timer_A` propose les quatre modes de fonctionnement suivants, dont le fonctionnement est détaillé dans [MSP430.pdf I chap 15.2.3 et 15.2.4]

    - **Stop** : Le timer est arrêté, il ne se passe rien.
    - **Up** : Le timer compte de 0 à une valeur au choix, à spécifier dans le champ `TACCR0`. Lorsque la valeur maximale est atteinte, le compte recommence à 9. Remarque : lorsque le timer est passé en mode Up alors que le registre `TAR` a une valeur supérieure à `TACCR0`, il est directement passé à 0. Une interruption `CCIFG` est générée lorsque le compteur atteint `TACCR0`, et une interruption `TAIFG` lorsqu'il repasse à 0 [MSP430.pdf I page 15-6]
    - **Continuous** : Le timer compte de 0 à `FFFFh`, et reprend à 0 lorsque cette valeur maximale est atteinte. L'utlisateur peut configurer différents intervalles de temps

indépendants. Une interruption à la fin de chaque intervalle. La période du prochain intervalle est communiquée au moment de l'interruption [MSP430.pdf I page 15-8].
    - **Up/down** : Le timer compte de 0 à une valeur au choix (à spécifier dans le champ `TACCR0`), puis de cette valeur à 0. Deux interruptions sont générées par période : lorsque la valeur maximale est atteinte, puis lorsque 0 est atteint [MSP430.pdf I page 15-9].

3. D'après [MSP430.pdf I page 15-4], le `Timer_A` peut utiliser les sources d'horloge : `ACLK`, `SMCLK`, ou une horloge externe vie `TACLK` ou `INCLK`. La source est configurée via le registre `TASSEL`. On peut également préciser un diviseur d'horloge (2, 4, ou 8) via le champ `ID` du registre `TACTL`.

4. Les fréquences des horloges sont :

    - `ACLK` : `32.768 kHz` (d'après [Motherboard.pdf I p.7])
    - `SMCLK` : horloge configurable, sourcée par défaut sur l'horloge interne `DCOCLK` avec un multiplicateur de 32. On a donc ici une fréquence de `1.048576 MHz` (d'après [MSP430.pdf I chapitre 5.2]).

5. On souhaite mesurer un intervalle temporel de `10 ms`.

Source	Fréquence	Nombre de cycles pour `10 ms`
`ACLK`	`32.768 kHz`	327.68 cycles
`SMCLK`	`1.048576 MHz`	10485.76 cycles

6. Nous ne pouvons compter qu'un nombre entier de cycles. On aura donc, à chaque intervalle temporel mesuré, une erreur sur le temps mesuré.

Source	Cycles mesurés	Erreur par intervalle	Erreur
`ACLK`	327 cycles	0.68 cycles	20.752 μs
`SMCLK`	10485 cycles	0.76 cycles	0.725 μs

   On pourrait réduire légèrement l'erreur en arrondissant le nombre de cycles mesurés au plus proche :

Source	Cycles mesurés	Erreur par intervalle	Erreur
`ACLK`	328 cycles	0.32 cycles	9.766 μs
`SMCLK`	10486 cycles	0.24 cycles	0.229 μs

   Bien que ces valeurs semblent faibles, l'erreur cumulée pourrait s'avérer gênante. En effet, après 10 secondes avec la source `ACLK`, l'erreur cumulée est de `9 ms`.

7. On choisit la source `ACLK` avec un diviseur de 1. On place le `Timer_A` en mode **Up**. On configure la valeur maximale (`TACCR0`) à 328 (d'après le calcul réalisé à la question précédente). On utilise la référence [MSP430.pdf I p.15-20 et p.15-21].

```
void init_timer (long period)
{
    // Reset the timer (clear any previous configuration)
    TACTL = TACTL | (1 << 2);
    // Set clock source to ACLK
    // Clock divider is 1 by default
    TACTL = TACTL | (1 << 8);

    // Enable Up mode
    TACTL = TACTL | (1 << 4);
    // Set the maximum value for Up mode (328 cycles)
    TACCR0 = 0x148;

    // Enable interruptions
    // Note: timer interrupt vector is TAIV
    TACTL = TACTL | (1 << 1);
}
```

Traitement de l'interruption

8. On utilise le signal d'interruption `TACCR0` comme vu à la question 2. Il correspond à la source Timer_A3 à l'adresse `0FFECh` [datasheet.pdf I p.13].

9. L'instruction de préprocesseur `#pragma opt=value` est équivalente à l'instruction préprocesseur `#define OPT _Pragma("opt=value")` qui définit une option spécifique à la plateforme (*Pragma directive*) (d'après [compiler.pdf I p.255]). Dans notre cas, on utilise la *Pragma Directive* `vector` qui définit quel vecteur d'interruption on va modifier (d'après [compiler.pdf I p.240]).

10. `__interrupt` est un "qualificatif" à appliquer aux fonctions destinées être un *handler* d'interruption. Généralement, on utilise `#pragma` en conjonction avec `__interrupt` [compiler.pdf I p.221]

```
// [msp430fg4618.h l.2284]
#pragma vector=TIMERA0_VECTOR
__interrupt void mon_traitement_interruption_timer(void);
```

11. On n'utilise pas les interruptions `TAIFG` du `Timer_A` mais le mode **capture / compare** du canal `CCR0` . Pour activer les interruptions sur ce canal, d'après [MSP430.pdf I p.15-22], on active le bit `CCIE` du registre `TACCTL0` .

```
TACCTL0 = TACCTL0 | (1 << 4);
```

12. Le bit `GIE` du registre de statut du processeur sert à activer les interruptions masquables [MSP430.pdf | p.3-6].

13. On utilise la "fonction intrinsèque" `__enable_interrupt` fournie par le compilateur, décrite dans [compiler.pdf | p.220].

```
__enable_interrupt();
```

14. Après comparaison de l'évolution du compteur avec un chronomètre, on ajuste le nombre de cycles de timer pour la période. On passe de 328 cycles / période à 326 cycles / période.

Étude du mécanisme d'interruption

15. Code assembleur généré par le compilateur pour le traitement d'interruption :

```
// Sauvegarde du contexte (d'après [compiler.pdf | p.24])
push.w  R13
push.w  R12
push.w  R15
push.w  R14

mov.w   &cpt,R12             // Passage du paramètre cpt
call    #lcd_display_number  // Appel de fonction

inc.w   &cpt                 // Incrémentation de cpt

// Restauration du contexte
pop.w   R14
pop.w   R15
pop.w   R12
pop.w   R13
reti
```

Lorsque l'on supprime la directive `#pragma`, le compilateur supprime la fonction car elle n'est plus appelée. De même, il est impossible de supprimer le qualificatif `__interrupt` en conservant la directive `#pragma` puisque le handler d'interruption doit être défini comme tel.

En supprimant à la fois `#pragma` et `__interrupt`, on obtient l'assembleur suivant :

```
mov.w   &cpt,R12             // Passage du paramètre cpt
call    #lcd_display_number  // Appel de fonction

inc.w   &cpt                 // Incrémentation de cpt
```

On remarque que la sauvegarde du contexte n'a pas été effectuée, ce qui est cohérent.

16. Les registres `R12`, `R13`, `R14` et `R15` sont sauvegardés. D'après [compiler.pdf I p.24], il s'agit des registres utilisés par le handler d'interruption.

17. D'après [CPU.pdf I p.3-57], l'instruction `reti` sert spécifiquement à retourner d'une routine d'interruption, alors que `ret` est l'instruction de retour pour toutes les autres routines. Lorsque l'on utilise `reti`, le contenu du registre de statut du processeur est restauré à la valeur présente avant le saut vers la routine de traitement de l'interruption. En particulier, les bits de statut `N`, `Z`, `C` et `V` sont restaurés. Pour `ret` comme pour `reti`, le `PC` (Program Counter) est restauré à sa valeur précédente, telle que stockée dans la pile (retour à l'instruction), puis incrémenté de 2 pour passer à l'instruction suivante.

18. Les vecteurs contiennent soit `FFFF` (= interruption non traitée) ou une adresse vers laquelle le programme va sauter en cas de déclenchement de l'interruption (= interruption traitée).

    ```
    00FFE0  FFFF FFFF   # Vecteurs non attachés
    00FFE4  FFFF FFFF   # Vecteurs non attachés
    00FFE8  FFFF FFFF   # Vecteurs non attachés
    00FFEC  3242        # Adresse du handler d'interruption
    00FFEE  FFFF FFFF   # Vecteurs non attachés
    00FFF2  FFFF FFFF   # Vecteurs non attachés
    00FFF6  FFFF FFFF   # Vecteurs non attachés
    00FFFA  FFFF FFFF   # Vecteurs non attachés
    00FFFE  3100        # Adresse du `__program_start`
    ```

19. À l'aide de *breakpoints* et des vues Stack et Register, on examine l'évolution de la mémoire lors du traitement d'une interruption.

    - Pendant l'exécution du programme principal, la pile contient l'adresse de retour du `main` (correspondant à un appel à la procédure `exit`). De plus, on a les valeurs notables suivantes :

      ```
      PC = 0x3262
      SP = 0x30FE
      SR = 0x000B
      GIE = 1 // Les interruptions sont activées
      ```

    - Après le saut vers la routine de traitement de l'interruption générée par le timer (mais avant l'exécution de la première instruction) : la pile contient de plus l'adresse de retour de la routine d'interruption (= la valeur de `PC` juste avant l'appel) ainsi que la valeur de `SR` juste avant l'appel. Les registres contiennent :

```
   PC = 0x31E2
   SP = 0x30FA
   SR = 0x0000
   GIE = 0 // Les interruptions sont masquées pendant le traitem
ent

   // Contexte à sauvegarder
   R12 = 0x1103
   R13 = 0x0003
   R14 = 0x0000
   R15 = 0x0014
```

- Après la sauvegarde du contexte mais avant le code utilisateur, la pile contient de plus les valeurs des registres `R12`, `R13`, `R14`, `R15`. Seuls les registres `PC` et `SP` ont été modifiés en conséquence.

- Après l'exécution du code utilisateur mais avant la restauration du contexte, la pile n'a pas été modifiée. Les registres `R12..15` ont été modifiés par le code utilisateur :

  ```
  R12 = 0x0000
  R13 = 0x0000
  R14 = 0x0001
  R15 = 0x000A
  ```

- Après la restauration du contexte mais avant l'exécution de `reti`, la pile est revenue à son état d'avant la sauvegarde du contexte, et de même pour les registres `R12..15`. Les registres `PC` et `SP` ont été modifiés en conséquence.

- Après l'exécution de `reti`, la pile est revenue à sont état initial (avant l'appel de la routine d'interruption). De plus, tous les registres ont été restaurés :

  ```
  PC = 0x3262
  SP = 0x30FE
  SR = 0x000B
  GIE = 1 // Les interruptions sont de nouveau actives
  ```

Interruption sur bouton poussoir

20. D'après [Motherboard.pdf I p.15], les boutons poussoir sont connectés aux ports P1.0 et P1.1. À l'aide de [MSP430.pdf I chap.11.2.6], on configure le bouton poussoir connecté à P1.1 pour générer des interruptions lorsqu'il est pressé (front montant).

```
// Choix de la fonction GPIO (et non périphérique)
P1SEL = 0x0;
P1DIR = P1DIR | 0 << 1; // Direction IN
// Les interruptions seront générées lors
// des transitions 0 -> 1 (bouton pressé)
P1IES = P1IES | 0 << 1;
// Activer les interruptions pour le port P1.1
P1IE = P1IE | 1 << 1;
```

21. Le traitement de l'interruption générée par la pression du bouton réinitialise le compteur. Le vecteur d'interruption correspondant au bouton est `PORT1_VECTOR` (d'après [msp430fg4618.h | l.2282])

```
#pragma vector=PORT1_VECTOR
__interrupt void traitement_pression_bouton(void)
{
    cpt = 0;
    // Acquittement de l'interruption [MSP430.pdf | p.11-5]
    P1IFG = P1IFG & 0 << 1;
}
```

22. Comme à la question 18, on observe la table des vecteurs d'interruptions à l'aide de la vue Mémoire.

```
00FFDE  FFFF FFFF   # Rien + Vecteur non attaché
00FFE2  FFFF FFFF   # Vecteurs non attachés
00FFE6  FF
00FFE7  FF          # Vecteur non attaché
00FFE8  329C        # Adresse du handler d'interruption du bouton
00FFEA  FF
00FFEB  FF          # Vecteur non attaché
00FFEC  3242        # Adresse du handler d'interruption du timer
00FFEE  FFFF FFFF   # Vecteurs non attachés
00FFF2  FFFF FFFF   # Vecteurs non attachés
00FFF6  FFFF FFFF   # Vecteurs non attachés
00FFFA  FFFF FFFF   # Vecteurs non attachés
00FFFE  3100        # Adresse du `__program_start`
```

23. D'après [MSP430.pdf | p.11-5], le registre `P1IFG` contient, au moment de l'appel de la routine de traitement d'interruption, la source de l'interruption. En particulier, on doit distinguer le port P1.0 du port P1.1. On configure les ports P1.0 et P1.1 comme précédemment pour générer une interruption à la pression (front montant). De plus, on modifie les routines de traitement comme suit :

```
#pragma vector=PORT1_VECTOR
__interrupt void traitement_pression_bouton(void)
{
    // Remise à zéro (port P1.1)
    if (P1IFG & (1 << 1) == (1 << 1))
        cpt = 0;
    // Mise en pause (port P1.0)
    else (P1IFG & (1 << 0) == (1 << 0))
        is_paused = 1 - is_paused;

    // Acquittement de l'interruption [MSP430.pdf | p.11-5]
    P1IFG = 0x0;
}

#pragma vector=TIMERA0_VECTOR
__interrupt void mon_traitement_interruption_timer(void)
{
    lcd_display_number(cpt);
    if (is_paused == 0)
        cpt++;
}
```