Les interruptions

Définition

Une interruption est un signal demandant au processeur de suspendre temporairement l'exécution du programme courant afin d'effectuer des opérations particulières.

Ce mécanisme permet d'implémenter une réaction à une sollicitation en respectant les exigences suivantes:

- ⇨ offrir un délai de réponse très bref,
- ⇨ programmation indépendante du code en cours d'exécution.

Le système d'interruption est le dispositif incorporé au séquenceur qui détecte les signaux d'interruption.

Ces signaux arrivent de façon asynchrone, à n'importe quel moment, mais ils ne sont pris en compte qu'à la fin de l'opération en cours.

Origine des interruptions

Les interruptions peuvent être déclenchées :

soit par un composant extérieur au processeur

- ⇨ requête d'interruption (Interrupt ReQuest, IRQ), signalée via une ou plusieurs
- ⇨ broches configurables ou non,

soit par un périphérique interne au composant (cas des microcontrôleurs ou des DSP)

- ⇨ Interruption initiée par un timer, convertisseur A/N, watchdog, …

soit la CPU

- ⇨ exception d'exécution ou dépassement lors d'un calcul arithmétique,
- ⇨ interruption logicielle
- ⇨ …

Mécanisme d'interruption

Lorsqu'une interruption survient , les opérations suivantes sont effectuées par le processeur.

1. L'instruction en cours termine son exécution.
2. L'adresse de la prochaine instruction à exécuter est sauvegardée sur la pile.
3. L'adresse de la routine d'exécution appropriée est obtenue en chargeant le contenu d'un vecteur d'interruption.
4. La routine d'interruption est exécutée.
5. A la fin de la routine d'interruption, le processeur reprend l'exécution du programme suspendu. À l'adresse sauvegardée sur la pile.

Types

- Il y a 3 types d'interruptions:

1. Reset du système
2. Interrupt non-masquable (*NMI – Non Maskable Interrupt*)
3. Interrupt masquable

Interrupts non masquables

- Ne sont pas masqués par le General Interrupt Enable bit (GIE) mais activés par des bits individuels (NMIIE, ACCVIE, OFIE).

- Lorsqu'un interrupt NMI est accepté, tous les bits d'activation individuels sont réinitialisés automatiquement, l'exécution du programme commence à l'adresse stockée dans le vecteur d'interrupts non masquables (0FFFCh).
- Le programme utilisateur doit mettre les bits NMI en *enable* explicitement pour les réenclencher.

- Un NMI peut être généré par une des 3 sources :
  - Un pic (edge) sur la pin RST/NMI (lorsque configurée en mode NMI)
  - Une faute d'oscillateur.
  - Une violation d'accès dans la mémoire flash.

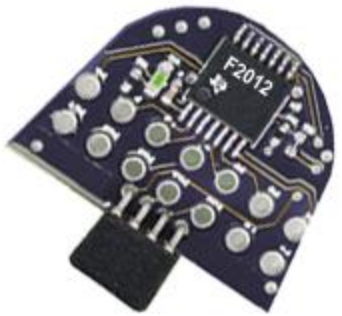
Interrupts masquables

- Les interrupts masquables sont causés par des périphériques ayant des capacités d'interruption.

- Chaque source d'interrupt masquable peut être **activée/désactivée individuellement** par un bit individuel et/ou elles peuvent être tous activées/désactivées par le bit GIE dans le registres de statut (SR).

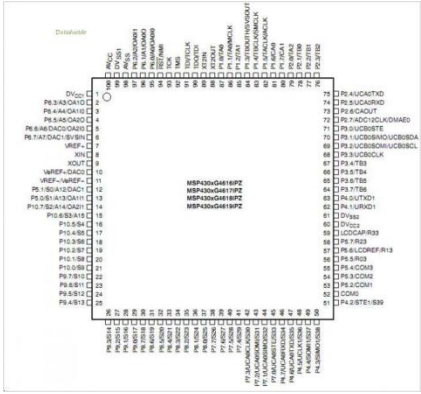
Sources d'interruption possible pour le MSP430F201x

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-up External reset Watchdog Timer+ Flash key violation PC out-of-range (see Note 1)	PORIFG RSTIFG WDTIFG KEYV (see Note 2)	Reset	0FFFEh	31, highest
NMI Oscillator fault Flash memory access violation	NMIIFG OFIFG ACCVIFG (see Notes 2 and 4)	(non)-maskable, (non)-maskable, (non)-maskable	0FFFCh	30
			0FFFAh	29
			0FFF8h	28
Comparator_A+ (MSP430x20x1 only)	CAIFG (see Note 3)	maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer_A2	TACCR0 CCIFG (see Note 3)	maskable	0FFF2h	25
Timer_A2	TACCR1 CCIFG. TAIFG (see Notes 2 and 3)	maskable	0FFF0h	24
			0FFEEh	23
			0FFECh	22
ADC10 (MSP430x20x2 only)	ADC10IFG (see Note 3)	maskable	0FFEAh	21
SD16_A (MSP430x20x3 only)	SD16CCTL0 SD16OVIFG, SD16CCTL0 SD16IFG (see Notes 2 and 3)	maskable		
USI (MSP430x20x2, MSP430x20x3 only)	USIIFG, USISTTIFG (see Notes 2 and 3)	maskable	0FFE8h	20
I/O Port P2 (two flags)	P2IFG.6 to P2IFG.7 (see Notes 2 and 3)	maskable	0FFE6h	19
I/O Port P1 (eight flags)	P1IFG.0 to P1IFG.7 (see Notes 2 and 3)	maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	16
(see Note 5)			0FFDEh ... 0FFC0h	15 ... 0, lowest

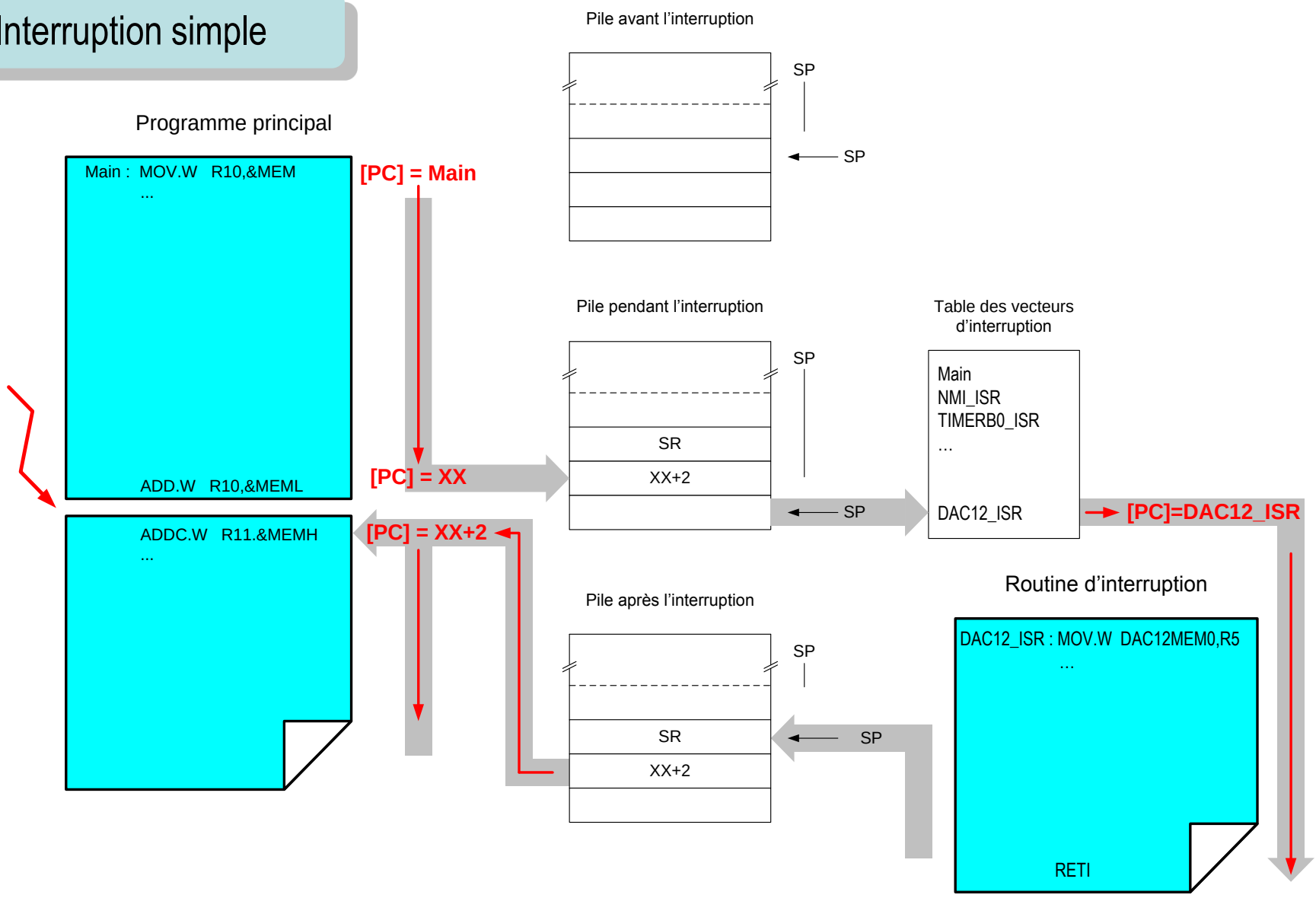


Sources d'interruption possible pour le MSP430FG4617

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-Up External Reset Watchdog Flash Memory	WDTIFG KEYV (see Note 1 and 5)	Reset	0FFFEh	31, highest
NMI Oscillator Fault Flash Memory Access Violation	NMIIFG (see Notes 1 and 3) OFIFG (see Notes 1 and 3) ACCVIFG (see Notes 1, 2, and 5)	(Non)maskable (Non)maskable (Non)maskable	0FFFCh	30
Timer_B7	TBCCR0 CCIFG0 (see Note 2)	Maskable	0FFFAh	29
Timer_B7	TBCCR1 CCIFG1 ... TBCCR6 CCIFG6, TBIFG (see Notes 1 and 2)	Maskable	0FFF8h	28
Comparator_A	CAIFG	Maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	Maskable	0FFF4h	26
USCI_A0/USCI_B0 Receive	UCA0RXIFG, UCB0RXIFG (see Note 1)	Maskable	0FFF2h	25
USCI_A0/USCI_B0 Transmit	UCA0TXIFG, UCB0TXIFG (see Note 1)	Maskable	0FFF0h	24
ADC12	ADC12IFG (see Notes 1 and 2)	Maskable	0FFEEh	23
Timer_A3	TACCR0 CCIFG0 (see Note 2)	Maskable	0FFECh	22
Timer_A3	TACCR1 CCIFG1 and TACCR2 CCIFG2, TAIFG (see Notes 1 and 2)	Maskable	0FFEAh	21
I/O Port P1 (Eight Flags)	P1IFG.0 to P1IFG.7 (see Notes 1 and 2)	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 (see Notes 1 and 2)	Maskable	0FFE2h	17
Basic Timer1/RTC	BTIFG	Maskable	0FFE0h	16
DMA	DMA0IFG, DMA1IFG, DMA2IFG (see Notes 1 and 2)	Maskable	0FFDEh	15
DAC12	DAC12.0IFG, DAC12.1IFG (see Notes 1 and 2)	Maskable	0FFDCh	14
Reserved	Reserved (see Note 4)		0FFDAh	13
			...	...
			0FFC0h	0, lowest

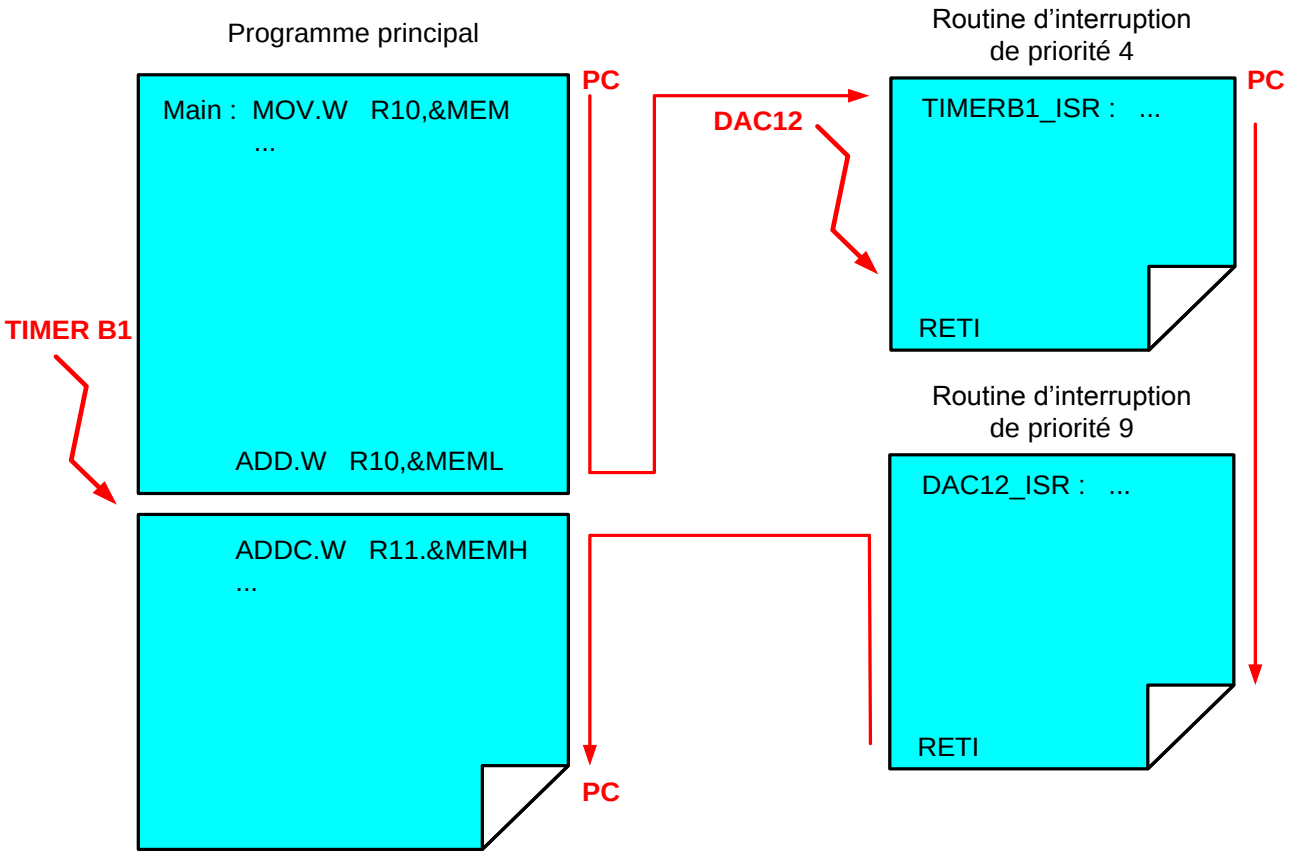


Interruption simple



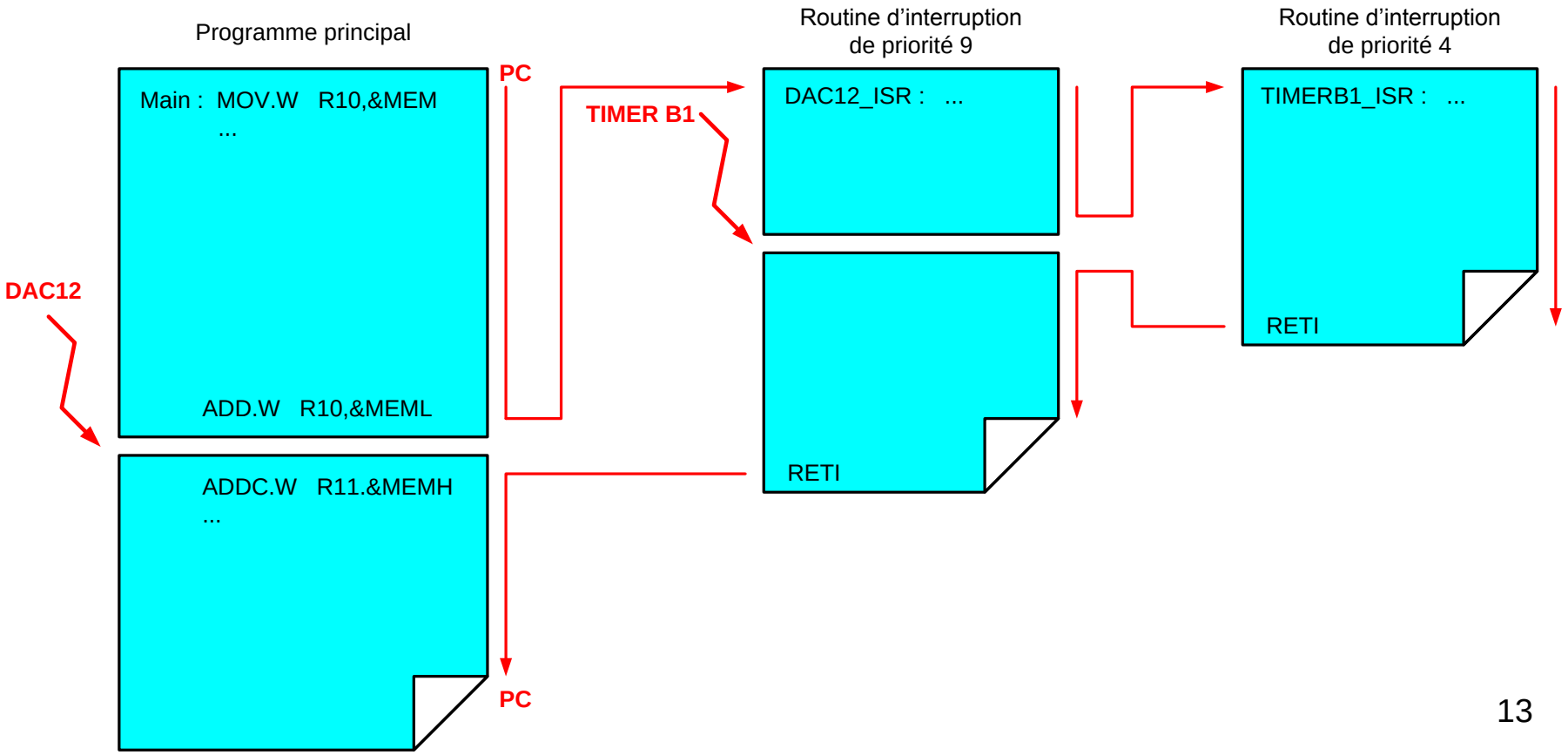
Interruptions multiples

Durant l'exécution d'une routine d'interruption une autre interruption **moins** prioritaire survient.



Interruptions multiples

Durant l'exécution d'une routine d'interruption une autre interruption **plus** prioritaire survient.



Sauvetage du contexte

Le bon fonctionnement d'un programme ne doit pas être altéré par les interruptions pouvant survenir pendant son exécution.

Il est donc indispensable que les routine d'interruption préservent l'état du processeur (valeur des bits d'états (flags), la configuration des périphériques, …).

Pour cette raison, les routines d'interruption commencent par des opérations de **sauvegarde** du contexte (sur la pile ou dans un espace de stockage spécifique), et se terminent par une **restauration** de celle-ci.

Remarques :

Certains processeurs effectuent automatiquement une sauvegarde de tout ou d'une partie du contexte lors d'une interruption.

La sauvegarde et la restauration du contexte peuvent parfois être facilitées par l'utilisation d'instruction spécifiques

Sur les processeurs désactivant automatiquement les interruptions lors d'un interruption, la réactivation de celles-ci à lieu lors de la restauration du contexte.

Gestion des interruptions

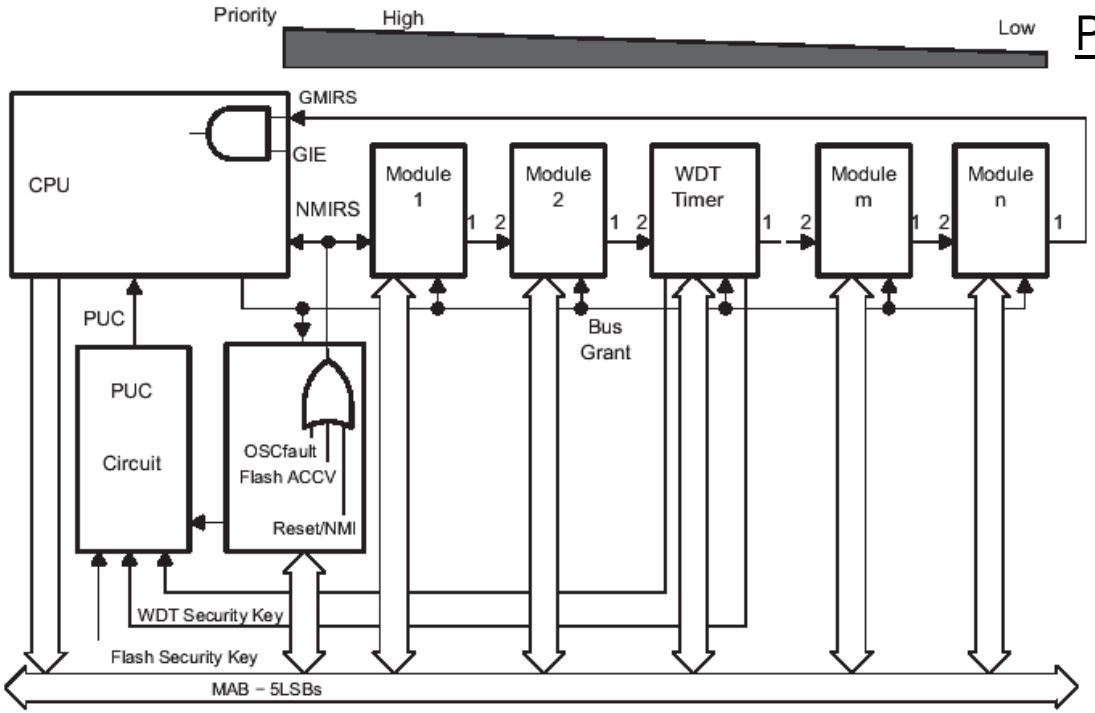
Selon l'application du microcontrôleur, certaines opérations critiques ne peuvent pas être interrompues.
Il est donc nécessaire de disposer d'une **hiérarchie d'activation et de désactivation** des interruptions.

Lorsque plusieurs sources d'interruption sont disponibles, des priorités différentes peuvent leur être attribuées. On peut alors activer que les priorités supérieures à un seuil donné.
Les priorités permettent également de départager des conditions d'interruption survenant au même moment

L'activation et la désactivation des interruptions s'effectue grâce à des instructions spécifiques, ou bien en modifiant la valeur d'un registre particulier

Priorité

Les priorités par défaut des interruptions sont fixées et définies par la chaine de connexion comme montré dans la figure suivante:



Priorité des interrupts

- Plus près du CPU un module est, plus haute sera sa priorité.
- Les priorités d'interruptions déterminent quel interrupt est choisi lorsqu'il y en a plusieurs simultanés.

Contrôle des interruptions

A la mise sous tension, les interruptions masquables sont par défaut désactivées afin de permettre l'initialisation correcte du programme.

Lorsqu'une interruption est déclenchée, certains processeurs **désactivent automatiquement** les interruption de priorité inférieur ou égale. C'est au programmateur de les réactiver si la routine d'interruption doit elle-même pouvoir être interrompue.

Dans ces cas, si une condition d'interruption survient alors que les interruptions sont désactivées, le processeur la mémorise dans un registre dédicacé afin de déclencher l'interruption dès qu'elle redeviendra possible.

Certaines architectures possèdent un type d'interruption qui ne peut pas être désactivé (*Non Maskable Interrupt, NMI*). Leur usage programmé se limite à des situations exceptionnelles.

Programmation des interruptions

Les compilateurs C fournissent des mécanismes permettant de programmer les interruptions sans recourir au langage assembleur.

Certaines fonctions peuvent être désignées comme étant des routines d'interruption (par exemple le mot-clé **interrupt** ou la directive **#pragma interrupt** en C)

En général, ce mécanisme **réalise automatiquement le sauvetage** du contexte et la mise à jour des vecteurs d'interruptions

L'activation et la désactivation des interruptions s'effectue via des macros ou des directives de compilation spécifiques (par exemple **enable()** / **disable()**, mot clé **critical** en C)

Il est parfois nécessaire de signaler au compilateur que la valeur d'une variable peut être modifiée par une routine d'interruption, afin d'éviter des optimisations incorrectes (par exemple par le mot-clé **volatile** en C).

Exemple : MSP430FG4617 : Définition de la table d'interruption

```
/************************************************************
* Interrupt Vectors (offset from 0xFFC0)
************************************************************/
#define DAC12_VECTOR          (14 * 2u) /* 0xFFDC DAC 12 */
#define DMA_VECTOR            (15 * 2u) /* 0xFFDE DMA */
#define BASICTIMER_VECTOR     (16 * 2u) /* 0xFFE0 Basic Timer / RTC */
#define PORT2_VECTOR          (17 * 2u) /* 0xFFE2 Port 2 */
#define USART1TX_VECTOR       (18 * 2u) /* 0xFFE4 USART 1 Transmit */
#define USART1RX_VECTOR       (19 * 2u) /* 0xFFE6 USART 1 Receive */
#define PORT1_VECTOR          (20 * 2u) /* 0xFFE8 Port 1 */
#define TIMERA1_VECTOR        (21 * 2u) /* 0xFFEA Timer A CC1-2, TA */
#define TIMERA0_VECTOR        (22 * 2u) /* 0xFFEC Timer A CC0 */
#define ADC12_VECTOR          (23 * 2u) /* 0xFFEE ADC */
#define USCIAB0TX_VECTOR      (24 * 2u) /* 0xFFF0 USCI A0/B0 Transmit */
#define USCIAB0RX_VECTOR      (25 * 2u) /* 0xFFF2 USCI A0/B0 Receive */
#define WDT_VECTOR            (26 * 2u) /* 0xFFF4 Watchdog Timer */
#define COMPARATORA_VECTOR    (27 * 2u) /* 0xFFF6 Comparator A */
#define TIMERB1_VECTOR        (28 * 2u) /* 0xFFF8 Timer B CC1-2, TB */
#define TIMERB0_VECTOR        (29 * 2u) /* 0xFFFA Timer B CC0 */
#define NMI_VECTOR            (30 * 2u) /* 0xFFFC Non-maskable */
#define RESET_VECTOR          (31 * 2u) /* 0xFFFE Reset [Highest Priority] */
```

Déclaration d'une interruption en C

Programme principal

```
void main(void)
{
   WDTCTL = WDTPW + WDTHOLD;     // Stop WDT

   P1IE  = 0x01;   // enable interrupt sur P1.0
   P1IFG = 0x00;
  __enable_interrupt();…
```

Routine déclarée comme routine d'interruption

```
   …
}
//routine d'interruption qui survient lorsqu'un des p1.x a un flanc
#pragma vector=PORT1_VECTOR
__interrupt void Port1_ISR (void)
{
  if ((P1IFG & 0x01) == 1)   …;
  P1IFG = 0x00;
}
```

#define PORT1_VECTOR
(20 * 2u) /* 0xFFE8 Port 1 */

Routine d'interruption correspondante.

Directives #pragma

La directive **#pragma** est définie par le standard ISO/ANSI C.
Elle permet de garder une certaine portabilité du software en contrôlant de manière particulière les extensions spécifiques à un type de processeur ou microcontrôleur.

Le compilateur fournit un set de directives **pragma** prédéfinies, lesquelles peuvent être utilisées pour contrôler le comportement du compilateur, par exemple sur la manière d'allouer de la mémoire, autoriser des mots clés supplémentaires ou générer des message d'avertissement (warning).

La plupart des directives de pragma sont prétraitées par le compilateur, ce qui signifie que des macros sont substitués à une directive pragma.

Insertion d'un vecteur dans la table d'interruption

Syntaxe : #pragma vector=vector_address

Description : Directive pragma permettant de spécifier l'adresse de la table d'interruption dans laquelle doit se trouver l'adresse de la routine d'interruption déclarée à la ligne suivante.

Exemple :

```
#pragma vector=TIMERA0_VECTOR
__interrupt void TIMERA0_ISR (void)
{
    …
    …
}
```

0xFFEC: adresse définie pour le MSP430FG4617, peut différer selon les modèles

Nom de la routine d'interruption

Interruption en C: échanges de données

Le fait qu'une interruption peut être déclenchée à tout moment **complique** les échanges de données entre les divers routines d'interruption et le programme principal.

Exemple: un contrôle de la précision d'une température est effectué par une routine d'interruption en mesurant deux fois à la suite. Une alarme est déclenchée si les deux températures diffèrent. Si une autre interruption a été exécutée juste entre les deux mesures ce contrôle sera faussé.

Si une autre interruption a lieu ici, la comparaison sera fausse

```
static volatile int temp[2];
void contrôle(void)
{   int temp0, temp1;
    for( ; ; )
       {
        temp0=temp[0];
        temp1=temp[1];
        if (temp0 !=temp1) alarm_on()
       }
}
#pragma vector = ADC12_VECTOR
__interrupt void mesure(void)
{
  temp[0] = ADC12MEM0;          // résultat de la première mesure
  temp[1] = ADC12MEM1;          // résultat de la deuxième mesure
}
```

Interruption en C : échanges de données

Condenser la comparaison des deux mesures en une seule instruction C ne résout pas le problème.

```
…
void contrôle(void)
{
    int temp0, temp1;
    for( ; ; )
       {
         if (temp[0] !=temp[1]) alarm_on()
        }
}
```

En effet cette comparaison est **traduite en plusieurs instructions machine** par le compilateur

Ce type d'erreur peut être très difficile à déceler et à reproduire

Interruption en C: échanges de données

Le problème provient du fait que la lecture des mesures effectuées par le programme principal devrait constitue une section critique qui ne peut être interrompue:

```
static volatile int temp[2];
void contrôle(void)
{
   int temp0, temp1;
   for( ; ; )
     {
      __disable_interrupt();
      temp0=temp[0];
      temp1=temp[1];
      __enable_interrupt();
      if (temp0 !=temp1) alarm_on();
     }
}
#pragma vector = ADC12_VECTOR
__interrupt void mesure(void)
{
  temp[0] = ADC12MEM0;          // résultat de la première mesure
  temp[1] = ADC12MEM1;          // résultat de la deuxième mesure
}
```

Masquage des interruptions

Inconvénient :
Introduction d'un temps de latence

Temps de réponse à une interruption (latence)

La durée pouvant s'écouler entre une requête d'interruption et l'exécution des opérations appropriées par la routine d'interruption qui lui est associée est appelée **temps de réponse** ou **temps de latence**.

Cette durée est influencée par 4 paramètres :

1. Le **plus long intervalle** pendant lequel une interruption de priorité **égale ou supérieure** a celle de **Int** peut être désactivé.
2. Le temps requis par l'exécution des routines d'interruption **plus prioritaire** que la routine en question.
3. Le **délai maximum** entre le déclenchement d'une interruption et l'appel de la routine correspondante.
4. Le temps écoulé entre **l'appel de la routine d'interruption** et l'exécution des opérations appropriées.

Temps de réponse à une interruption (latence)

On a donc intérêt à :

- ⇨ désactiver les interruptions le moins de temps possible
- ⇨ écrire des routines d'interruption brèves et efficaces

Exemple 1: **interruption sur le port P1**

La préparation dans le `main()` sera du type :

```
#include "io430.h«
#include "intrinsics.h"

int main( void )
{
  WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer

  P1IES &= 0x00;   // interrupt P1.x au flancs montants
  P1IE  |= 0xF0;   // enable interrupt P1.4 à P1.7
  P1IFG &= ~0xF0;  // on s'assure que les flags sont bien à zéro
  __enable_interrupt();

  while(1)
  { . . . }
}
```

La routine d'interruption est écrite comme :

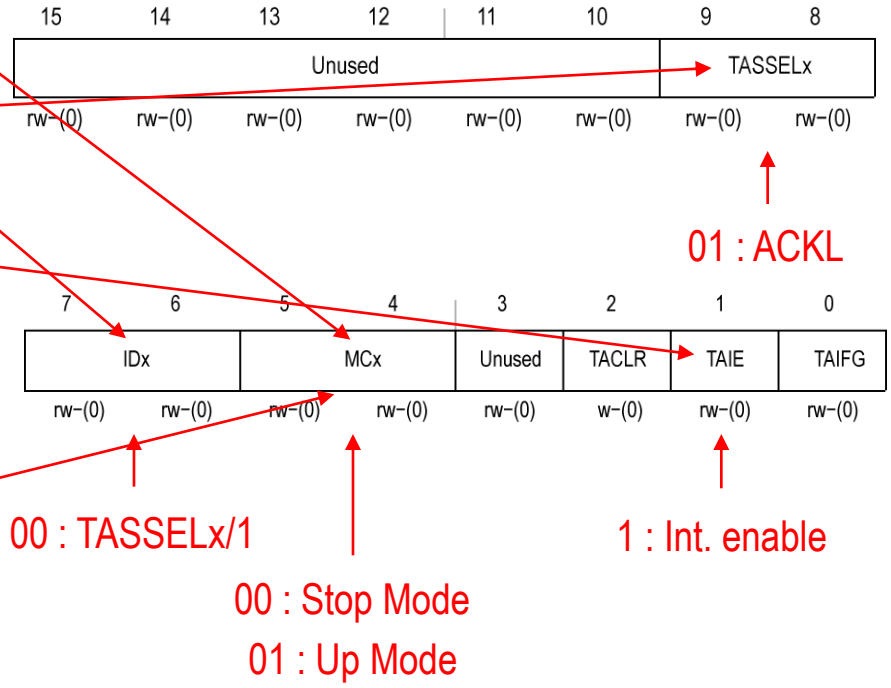
```
// routine d'interruption qui survient lorsqu'un des P1.x a un flanc
#pragma vector=PORT1_VECTOR
__interrupt void Port1_ISR (void) // PORT1ISR (void)
{
  if (P1IFG & 0x10) // par exemple, test si l'interruption est causé par P1.4
   ...faites «toggler» une LED ...

  P1IFG = 0;        // il FAUT remettre le flag à zéro !
}
```

Exemple 2: **initialisation d'une interruption du Timer A**

```
void timerA_init()              //initialisation timer A
{
  TACTL_bit.TAMC = 0;           // Stop Timer A
  TACTL_bit.TASSEL = 1;         // CLK source : ACLK
  TACTL_bit.TAID = 0;           // ACLK / 1
  TACTL_bit.TAIE = 1;           // Interrupt enable
  TACCTL0_bit.CCIE = 1;         // Interrupt compare
  TACCR0 = 1000;                // Valeur max du compteur
  TACTL_bit.TAMC = 1;           // Compteur en up mode
}
```

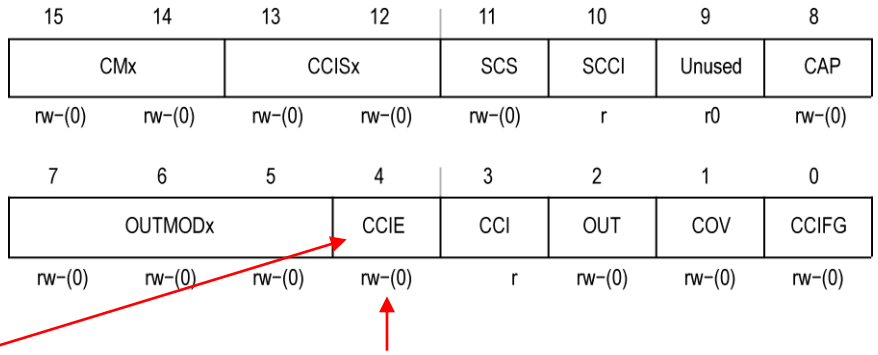
Registre TACTL



Exemple 2: initialisation d'une interruption du Timer A

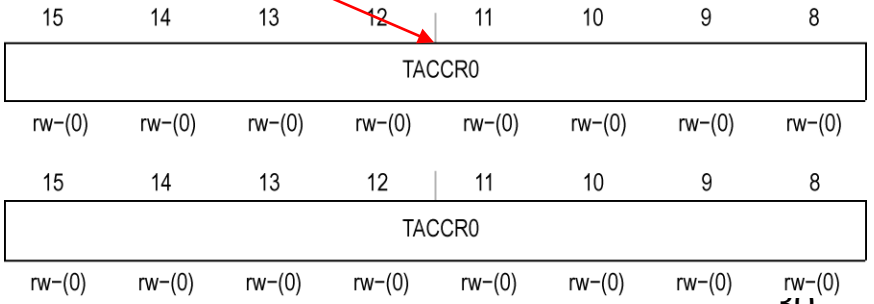
```
void timerA_init()              //initialisation timer A
{
   TACTL_bit.TAMC = 0;          // Stop Timer A
   TACTL_bit.TASSEL = 1;        // CLK source : ACLK
   TACTL_bit.TAID = 0;          // ACLK / 1
   TACTL_bit.TAIE = 1;          // Interrupt enable
   TACCTL0_bit.CCIE = 1;        // Interrupt compare
   TACCR0 = 1000;               // Valeur max du compteur
   TACTL_bit.TAMC = 1;          // Compteur en up mode
}
```

Registre TACCTL0



1 : Int. sur comparaison

Registre TACCR0



Exemple 2: routine interruption du Timer A

```
// ---------------------------------------------------- //
//  Interrupt on Timer A
// ---------------------------------------------------- //
#pragma vector = TIMERA0_VECTOR
__interrupt void int_timerA (void)
{
  __disable_interrupt();
  … ;
  … ;
__enable_interrupt();
 }
```

Adresse de la table d'interruption dédiée au interruption du Timer A CC0
#define TIMERA0_VECTOR (22 * 2u) /* 0xFFEC Timer A CC0 */

Définition de la routine d'interruption