



LES

YVES TIEO

Version provisoire. Nous travaillons sur ce document, mais les remarques sont les bienvenues !

MOTIVATION

Un système à microcontrôleur est généralement pourvu d'entrées et de sorties. Le but premier du p

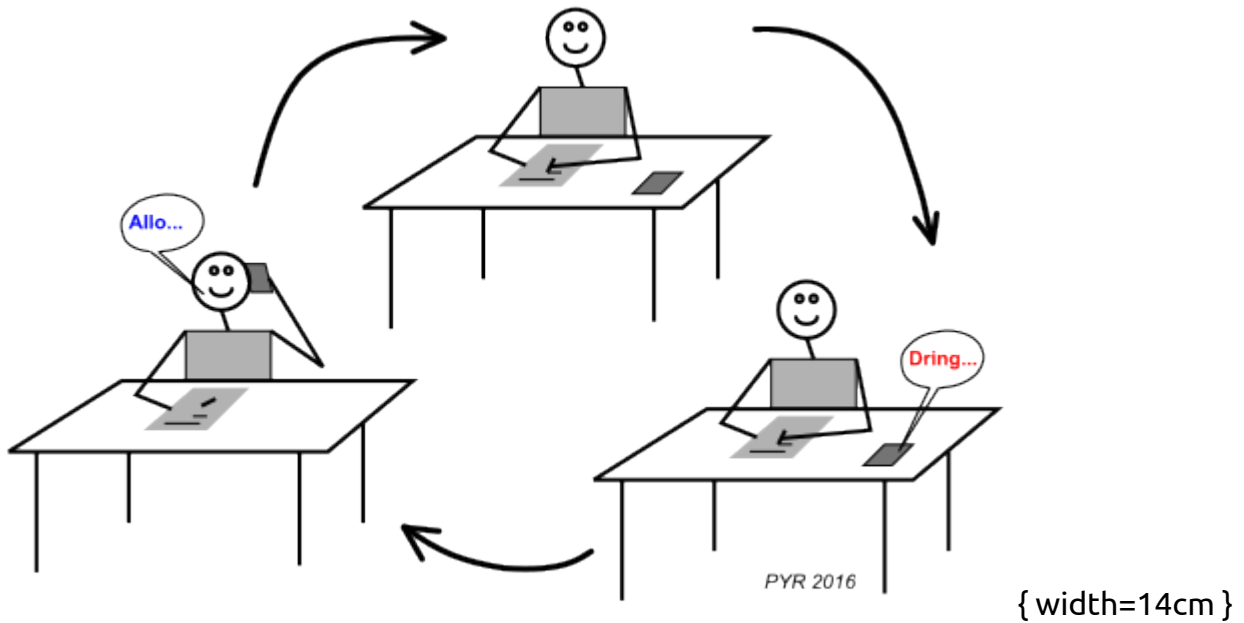
Les enseignes et afficheurs à LED sont plutôt une exception dans ce domaine. Beaucoup d'enseigne

Il existe toutefois des cas où une enseigne ou un afficheur doit réagir à des entrées. Par exemple, u
autre cas où le système doit réagir à un événement est la gestion du temps dans un afficheur multip

DÉFINITION

On appelle **interruption** dans un système informatique l'arrêt temporaire d'un programme au prof
courantes. Prenons un exemple : je suis en train de travailler à mon bureau. Le téléphone sonne. Je v

DRAFT

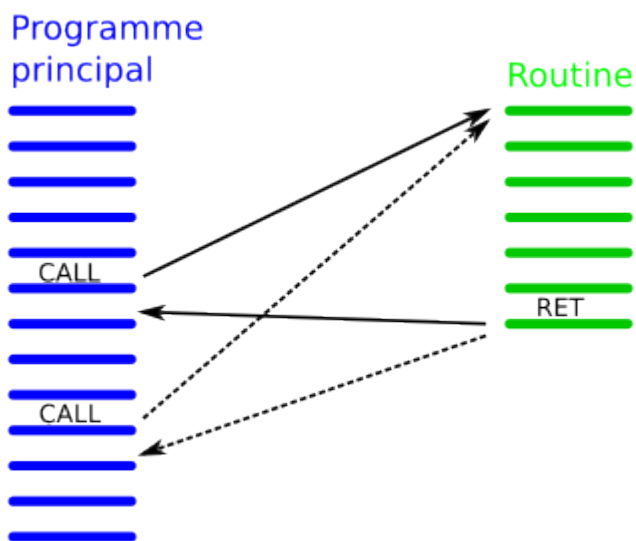


C'est toujours un *événement* qui va produire une interruption. Cet événement a un caractère imprévisible.

IMPLÉMENTATION

Pour utiliser les interruptions, il n'est pas indispensable de comprendre en détail le mécanisme qui les rendent possibles.

Toutefois, nous allons ici faire une petite incursion dans le monde de la programmation en assembleur. Les instructions de saut sont notées en bleu. Dans le cas où une fraction du programme doit s'exécuter plusieurs fois, on a l'habitude d'utiliser des sous-programmes, ce qui correspond aux procédures et aux fonctions dans les langages évolués comme le C.



{ width=7cm }

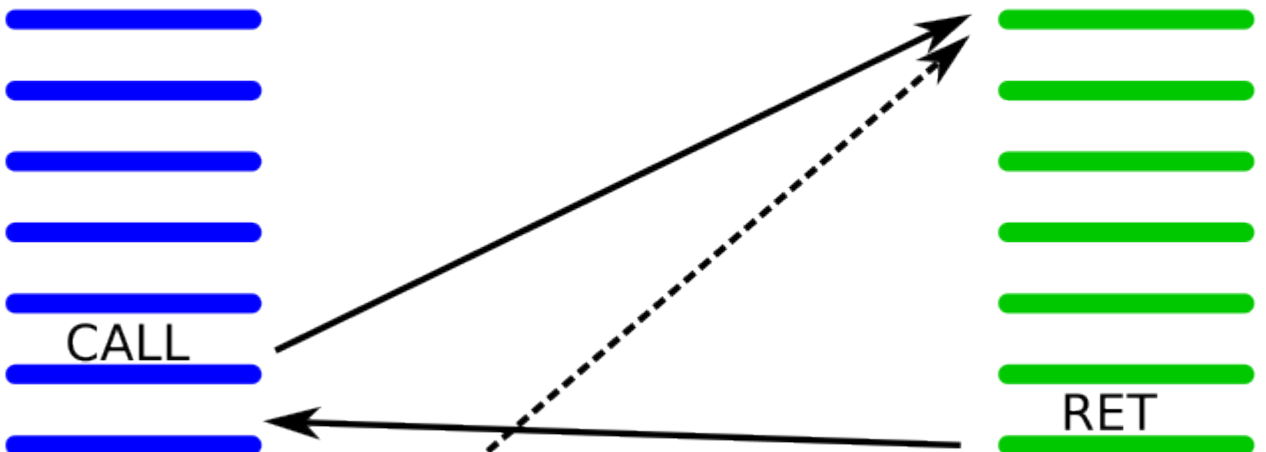
L'appel de la routine se fait par une instruction **Call** dans le programme principal. A la fin de l'exécution, la routine peut être appelée plusieurs fois dans le programme principal.

Notons que l'adresse de retour doit être mémorisée pour que le retour soit possible. C'est une **pile**.

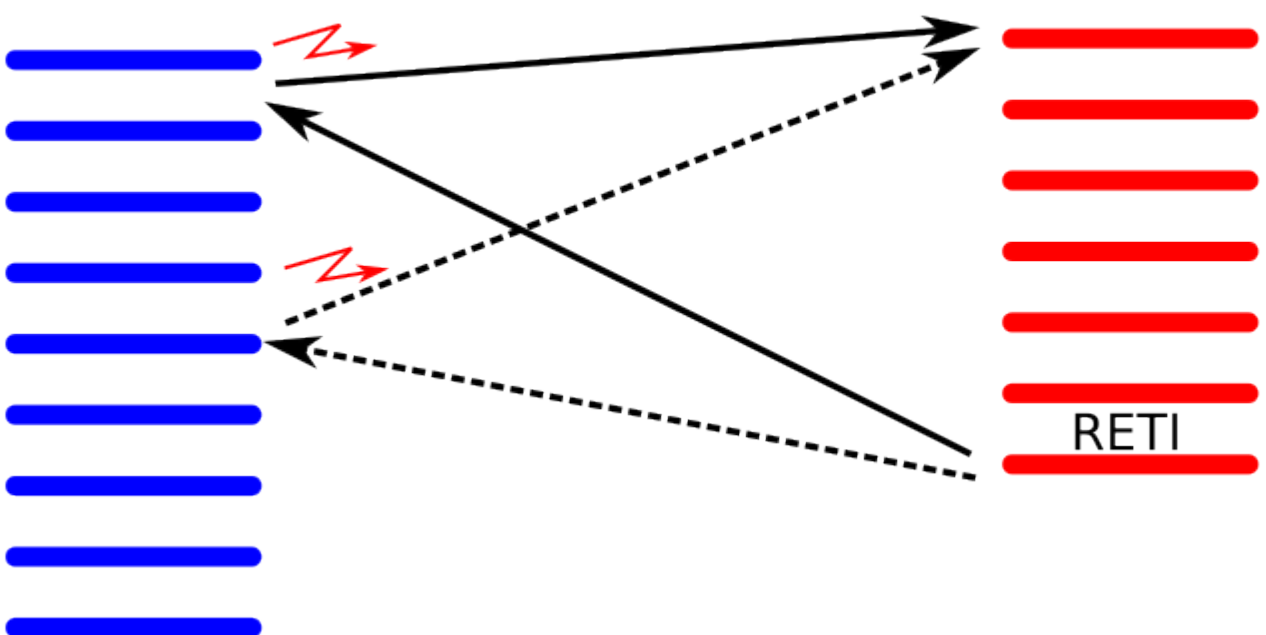
Regardons maintenant la figure suivante. Une nouvelle routine, appelée **Routine d'interruption** est

Programme
principal

Routine



Routine
d'interrup



DRAFT

Rien dans le programme principal ne permet de savoir que cette routine va s'exécuter. C'est un **événement**.

La routine d'interruption se termine aussi par une instruction de retour, appelée **Reti** (*return from interrupt*) avant l'interruption.

NATURE DES ÉVÉNEMENTS

Quels sont ces événements qui vont produire une interruption ? Il en existe principalement deux sortes :

- Les événements **extérieurs** au microcontrôleur. Il s'agit par exemple d'un changement sur une entrée numérique.
- Les événements **intérieurs** au microcontrôleur. Par exemple, beaucoup de microcontrôleurs sont équipés d'un convertisseur analogique-numérique (CAN) et dure un certain temps. Plutôt que d'attendre la fin de la conversion, le programme principal peut interrompre la conversion.

Dans cette catégorie des interruptions intérieures au microcontrôleur, les plus importantes sont celles liées à la gestion de la mémoire et à la gestion des périphériques à LED.

DISCRIMINATION DES SOURCES D'INTERRUPTION

Il existe généralement plusieurs sources d'interruptions sur un microcontrôleur. Lorsqu'une interruption se produit, le programme principal doit consulter les registres pour chaque interruption, pour connaître celle qui a été activée.

Les **vecteurs d'interruption** (*interrupt vectors*) permettent d'être plus efficace : une adresse différente pour chaque interruption.

Souvent ces deux mécanismes vont être utilisés successivement, comme nous le verrons plus bas lors de la programmation.

Voici la table résumée des vecteurs d'interruption pour un MSP430G, y compris l'adresse pour le Reset.

- 0xFFFFE : Reset
- 0xFFFFC : NMI
- 0xFFFFA : Timer1 CCR0
- 0xFFFF8 : Timer1 CCR1, CCR2, TAIFG
- 0xFFFF6 : Comparator_A
- 0xFFFF4 : Watchdog Timer
- 0xFFFF2 : Timer0 CCR0
- 0xFFFF0 : Timer0 CCR1, CCR2, TAIFG
- 0xFFEE : USCI status
- 0xFFEC : USCI receive/transmit
- 0xFFEA : ADC10
- 0xFFE8 : -

- 0xFFE6 : Port P2
- 0xFFE4 : Port P1

Les adresses se trouvent en mémoire flash, ce sont les dernières adresses de l'espace d'adressage d

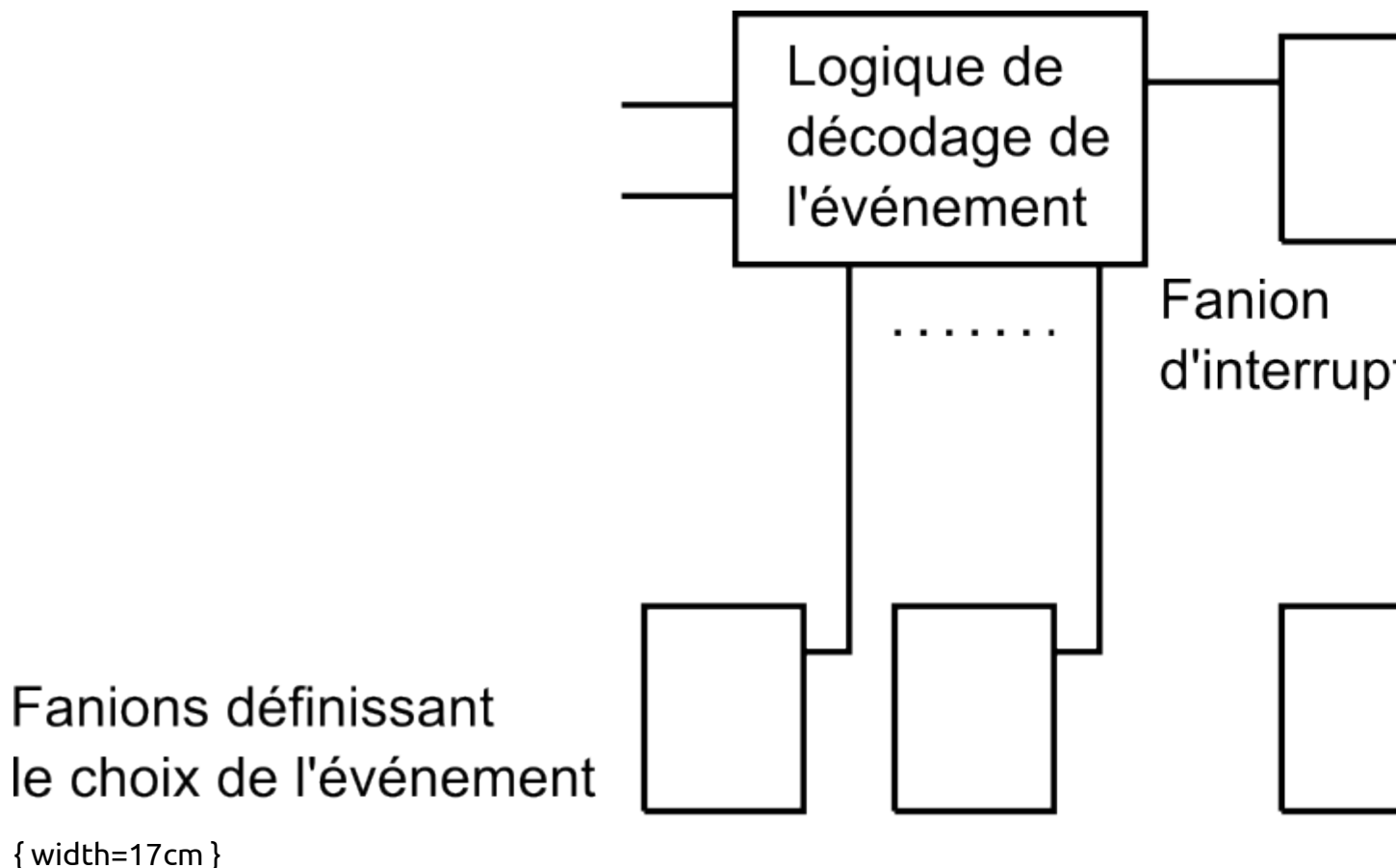
Plusieurs sources d'interruptions nécessitent la scrutation pour déterminer la cause exacte de l'interruption.
des interruptions des Timers : les registre de comparaison 1 et 2, ainsi que l'interruption générale d

MISE EN ŒUVRE D'UNE INTERRUPTION

Plusieurs étapes sont nécessaire pour mettre en œuvre une routine d'interruption :

- 1- Enclencher l'interruption qui nous intéresse. Par exemple une interruption sur une entrée.
- 2- Préciser comment cette interruption doit fonctionner. Par exemple dire sur quel flanc l'interruption doit être déclenchée.
- 3- Enclencher globalement les interruptions. Les microprocesseurs disposent d'un fanion général d'interruption, qui ne doivent pas être interrompues.

Le schéma logique ci-dessous montre la logique qui permet de générer les interruptions et les fanions d'interruption.



On y trouve :

- la logique qui permet de saisir un événement

- les fanions qui règlent la manière dont l'événement est décodé
- le fanion qui enclenche cette interruption particulière
- la porte ET associée à ce fanion pour produire cette interruption
- la porte OU qui permet à toutes les interruptions d'être prises en compte
- le fanion général d'autorisation des interruptions
- la porte ET qui produit finalement les interruptions.

SYNTAXE DES ROUTINES D'INTERRUPTIONS

Le langage C ne définit pas la syntaxe des routines d'interruptions. Plusieurs notations sont utilisées :

```
#pragma vector=NUMERO_DU_VECTEUR
__interrupt void Nom_de_la_routine (void) {
    ...
}
```

La première ligne indique au compilateur à quel vecteur d'interruption la routine sera associée. La seconde ligne, `__interrupt`, permet au compilateur d'utiliser les instructions correspondant à une routine d'interruption.

INTERRUPTION PRODUITE PAR UNE ENTRÉE

Sur les microcontrôleurs MSP430, plusieurs registres permettent de définir la manière dont une broche peut être configurée. Les registres disponibles sont : P1 et P2. On connaît déjà les registres suivants :

- **P1DIR** : détermine le rôle de la broche (entrée ou sortie)
- **P1OUT** : donne la valeur pour les broches de sortie
- **P1IN** : permet de lire la valeur des entrées
- **P1REN** : permet d'enclencher une résistance de tirage (pull-up ou pull-down, selon l'état de bit de P1REN)

Pour mettre en œuvre les interruptions sur des broches du port P1, trois registres supplémentaires sont disponibles :

- **P1IE** : (*Interrupt Enable*) permet l'enclenchement de l'interruption pour chaque bit. L'usage habituel est d'écrire dans ce registre pour choisir quel flanc va causer une interruption, pour chaque bit.
- **P1IES** : (*Interrupt Edge Select*) permet de choisir pour chaque bit le flanc qui va produire l'interruption.
- **P1IFG** : (*Interrupt Flag*) les **fanions d'interruption**. Lorsque qu'une transition telle qu'elle est spécifiée dans P1IES et P1IES se produit, le bit correspondant dans P1IFG est mis à 1.

Voici un programme qui met en œuvre une interruption sur l'entrée P1.3 (le poussoir du Launchpad).

```

1  int main() {
2      WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
3      P1DIR |= (1<<6); // Led verte en sortie
4      P1OUT |= (1<<3); P1REN |= (1<<3); //pull-up sur l'entrée P1.3
5
6      P1IES |= (1<<3); // Mode d'interruption : sur le flanc descendant
7      P1IE |= (1<<3); // Interruption P1 activée sur le bit 3
8      P1IFG &=~(1<<3); // Fanion d'interruption remis à zéro
9      __enable_interrupt(); // General Interrupt Enable
10
11     while(1) { // il n'y a rien à faire dans la boucle principale !
12     }
13 }
14
15 // Routine d'interruption associée au Port P1
16 // Syntaxe spécifique pour les interruptions :
17 #pragma vector=PORT1_VECTOR
18 __interrupt void Port1_ISR(void) {
19     P1IFG &= ~(1<<3); // Fanion d'interruption correspondant au bit 3 remis à 0
20     P1OUT ^= (1<<6); // inverse P1.6 (LED verte)
21 }

```

La première remarque, c'est que la boucle principale `while(1)...` ne fait rien ! En plus des initialisations, il y a une activation d'interruption :

- L'activation d'un bit dans `P1IES` sélectionne le flanc descendant
- L'activation d'un bit dans `P1IE` autorise l'interruption sur l'entrée `P1.3`
- L'appel de la procédure `__enable_interrupt()` autorise globalement les interruptions sélectionnées

La mise à 0 du bit 3 dans le registre des fanions d'interruption `P1IFG` évite qu'un flanc sur l'entrée `P1.3` génère une nouvelle interruption.

La routine d'interruption a été placée à la suite du programme principal, alors que nous avons l'habitude de la placer avant.

SCRUTATION DU BIT CONCERNÉ

Dans notre exemple, seul le bit 3 est concerné par les interruptions. Dans la routine d'interruption, il faut scruter le registre `P1IFG` pour savoir quel bit a généré l'interruption.

Lorsque l'interruption peut provenir de plusieurs entrées, il est alors nécessaire de scruter le registre `P1IFG` pour savoir quel bit a généré l'interruption.

```

1  int main() {
2      ...
3      P1IES &=~((1<<3)|(1<<4)); // interruptions aux flancs montants
4      P1IE |= (1<<3)|(1<<4); // Interruption activée sur 2 entrées
5      P1IFG &=~((1<<3)|(1<<4)); // Fanions d'interruption remis à zéro
6      ...
7
8      #pragma vector=PORT1_VECTOR

```



```

9  __interrupt void Port1_ISR(void) {
10  // scrutation des causes possible de l'interruption :
11  if (P1IFG & (1<<3)) {... ; P1IFG &= ~(1<<3); }
12  if (P1IFG & (1<<4)) {... ; P1IFG &= ~(1<<4); }
13  }

```

INTERRUPTION DE FIN DE CONVERSION

Voici un autre exemple d'interruption, où l'événement est interne au microcontrôleur. L'interruption

```

1  int main() {
2      WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
3      P1DIR |= (1<<6); P1OUT &=~(1<<6); // LED verte en sortie
4      // Activation du convertisseur ADC 10 bits (ADC10) :
5      ADC10CTL0 = ADC10SHT_2 + ADC10ON + ADC10IE; // ADC10ON, interrupt enabled
6      ADC10CTL1 = INCH_1; // Canal 1 = entrée A1 = P1.1
7      ADC10AE0 |= (1<<1); // Enclanchement de l'entrée A1
8      __enable_interrupt(); // General Interrupt Enable
9      ADC10CTL0 |= ENC + ADC10SC; // lance une première conversion
10
11     while(1) { // il n'y a rien à faire dans la boucle principale !
12     }
13 }
14
15 // Routine d'interruption associée à la fin de conversion ADC
16 #pragma vector=ADC10_VECTOR
17 __interrupt void ADC10_ISR(void) {
18     uint16_t val = ADC10MEM; // lit le résultat de la conversion
19     ADC10CTL0 |= ENC + ADC10SC; // lance la conversion suivante
20     if (val > 512) { // Montre sur la LED verte si la valeur dépasse Vcc/2
21         P1OUT |= (1<<6); // LED verte On
22     } else {
23         P1OUT &=~(1<<6); // LED verte Off
24     }
25 }

```