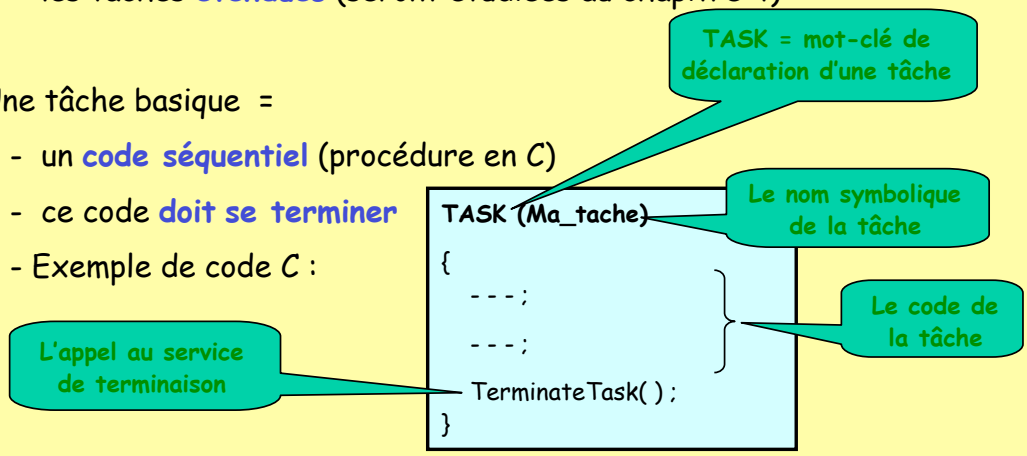
Chapitre 2 -

Services de base de l'exécutif temps réel

☞ La gestion des tâches

- Les tâches sont les **éléments actifs** de l'application
- Deux catégories de tâches dans OSEK/VDX :
  - les tâches **basiques**
  - les tâches **étendues** (seront étudiées au chapitre 4)

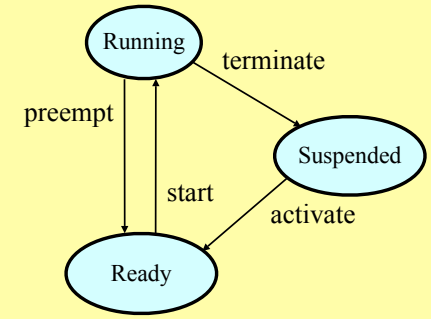
- Une tâche basique =
  - un **code séquentiel** (procédure en C)
  - ce code **doit se terminer**
  - Exemple de code C :



☞ Les états d'une tâche basique

La tâche a le processeur
(elle s'exécute)

Diagramme des états



La tâche est inactive

La tâche est active
(elle n'a pas le processeur)

- les transitions d'état « terminate » et « activate » sont liées à l'exécution des services TerminateTask et ActivateTask
- les transitions d'état « prempt » et « start » sont liées à l'ordonnanceur
- au départ une tâche peut être dans l'état « Suspended » ou « Ready »

☞ Les services pour les tâches basiques

Le service TerminateTask

Forme C : Status **TerminateTask** (void)

le compte-rendu d'exécution n'est pas utilisé

C'est forcément la tâche **en cours d'exécution** qui appelle ce service (on ne peut pas suspendre une autre tâche).
La tâche passe alors dans l'état « suspended »

Exemple :

```
TASK (Tache_1)
{
   - - - ;
   - - - ;
   TerminateTask( ) ;
}
```

☞ Les services pour les tâches basiques

Le service ActivateTask

Forme C : Status **ActivateTask** (Task_Id Nom_Tache)

le paramètre « Nom_Tache » est le nom de la tâche* à activer

le compte-rendu d'exécution n'est pas utilisé

La tâche désignée passe dans l'état « ready ».

Exemple :

```
TASK (Tache_1)
{
  - - - ;
  ActivateTask (Tache_2)
  - - - ;
  TerminateTask( ) ;
}
```

```
TASK (Tache_2)
{
  - - - ;
  - - - ;
  TerminateTask( ) ;
}
```

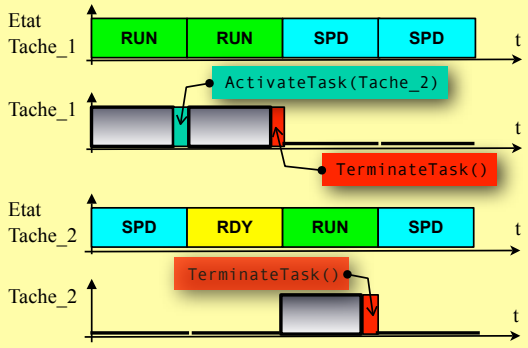
* En réalité les tâches sont connues de l'exécutif via un numéro et non une chaîne de caractères. La correspondance est automatiquement réalisée par le générateur d'application (voir chapitre 3).

☞ Analyse de cet exemple simple

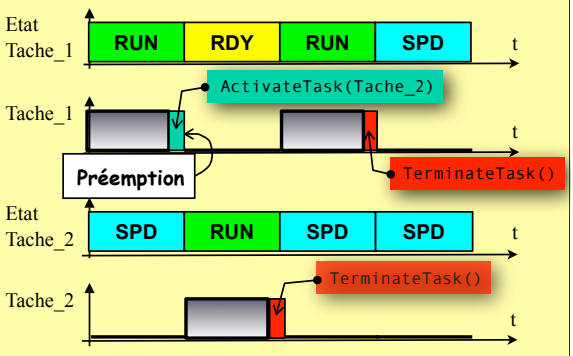
- On suppose que la tâche Tache_1 est activée à l'initialisation (attribut « Autostart » d'une tâche, sera vu au chapitre 3)
- Déroulement temporel :

```
TASK (Tache_1)
{
  - - - ;
  ActivateTask (Tache_2) ;
  - - - ;
TerminateTask( ) ;
}
```

```
TASK (Tache_2)
{
  - - - ;
  - - - ;
TerminateTask( ) ;
}
```



Cas 1 : **priorité** Tache_1 > **priorité** Tache_2



Cas 2 : **priorité** Tache_1 < **priorité** Tache_2

RUN Running RDY Ready SPD Suspended

☞ Remarques

- lorsque les 2 tâches sont terminées il n'y a plus d'activité
  ↳ le procédé contrôlé ne l'est plus !
- donc, en général un système temps réel se compose
  - de tâches périodiques qui, périodiquement, prennent des mesures sur le procédé, font des calculs puis émettent des commandes
  - de tâches non périodiques qui répondent à des stimulis en provenance de l'environnement (ex.: Nmax devient actif)
- Comment activer des tâches dans ces 2 conditions ?
  - **activation périodique** possible :
    - via un timer (et son IT) il faut « faire » un ActivateTask
    - via le mécanisme d'**Alarme** intégré à OSEK/VDX (voir §2.4)
  - **activation non périodique** possible :
    - via une IT quelconque il faut « faire » un ActivateTask

Dans 2 cas on doit « relayer » une IT ⟹ notion d'ISR (voir §2.3)

☞ Remarques

- Que se passe-t-il si une tâche en active une autre qui est toujours active ?
- Dans OSEK/VDX il est possible d'associer à une tâche un compteur (borné) de mémorisation des demandes d'activation.

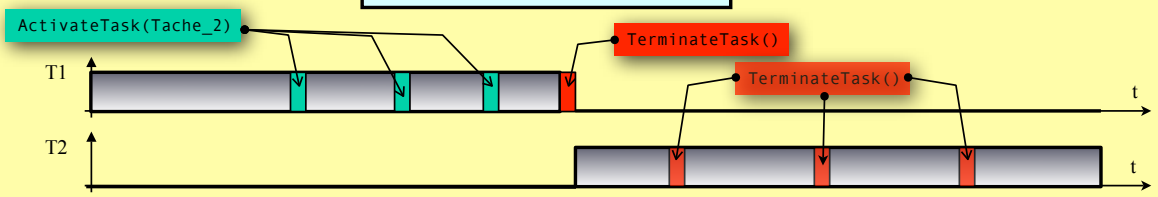
Exemple :

Priorité T1 >

Priorité T2

```
TASK (T1)
{
  ---;
  ActivateTask (T2)
  ActivateTask (T2)
  ActivateTask (T2)
  TerminateTask ( );
}
```

```
TASK (T2)
{
  ---;
  ---;
  TerminateTask ( );
}
```



3 activations successives (sous réserve que le compteur puisse en mémoriser 3)

☞ Les services pour les tâches basiques

Le service ChainTask

Forme C : Status **ChainTask** (Task_Id Nom_Tache)

le paramètre « Nom_Tache » est le nom de la tâche à activer

le compte-rendu d'exécution n'est pas utilisé

Dans un même appel (**action atomique**), la tâche appelante (celle en cours) se termine et la tâche désignée est activée.

Exemple :

```
TASK (Tache_1)
{
  - - - ;
  - - - ;
  ChainTask (Tache_2 ) ;
}
```

```
TASK (Tache_2)
{
  - - - ;
  - - - ;
  TerminateTask ( ) ;
}
```

On pourrait, dans l'exemple, réaliser la « même chose » par l'enchaînement :

```
{
  - - - ;
  ActivateTask (Tache_2);
  TerminateTask ( );
}
```

En fait ces deux opérations **ne pouvant pas être atomiques** on peut être préempté entre les deux appels. Ce n'est donc pas forcément le même comportement.

☞ Ordonnancement

- L'algorithme d'ordonnancement utilise l'attribut « **Priorité** » de chaque tâche.
  - c'est une **valeur entière fixe** (priorité statique) :
    - 0 est la plus faible priorité
    - la valeur maximale dépend de l'implémentation (255)

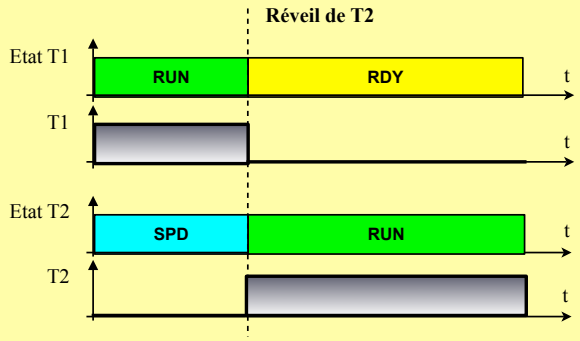
- 3 modes d'ordonnancement (choix à la configuration) existent :
  - le mode « **full preemptive** » (toutes les tâches sont préemptives)
  - le mode « **full non preemptive** » (toutes les tâches sont non-préemptives)
  - le mode « **mixed** » : des tâches sont préemptives, d'autres sont non-préemptives

☞ Mode « Full Preemptive »

- dans ce mode **c'est toujours la tâche la plus prioritaire qui est active**
- Exemple :

T1 active (priorité 5) a le processeur et s'exécute

T2 (priorité 10) est réveillée (une IT active T2 par exemple)



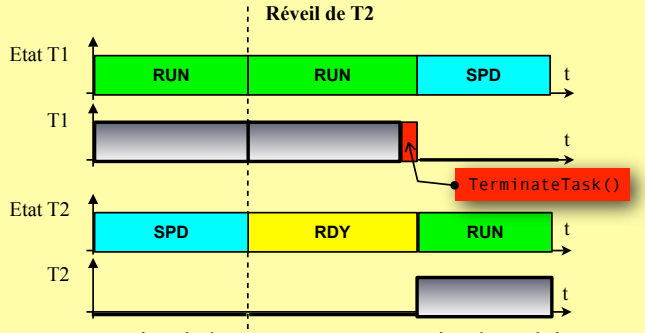
L'ordonnanceur décide, au réveil de T2, de donner le processeur à T2, T1 passe alors dans l'état « Ready ».
Lorsque T2 se terminera, T1 aura à nouveau le processeur pour finir son exécution (C'était la situation du Cas 2 de l'exemple du §2.1).

☞ Mode « Full Non Preemptive »

- dans ce mode **la tâche en cours ne perd le processeur que lorsqu'elle se termine** (lors de son appel à TerminateTask)
- Exemple :

T1 active (priorité 5) a le processeur et s'exécute

T2 (priorité 10) est réveillée (une IT active T2 par exemple)



L'ordonnanceur, au réveil de T2 qui est passée dans l'état « Ready », ne change pas l'état de la tâche en cours et ré-active T1.
Lorsque T1 se termine, l'ordonnanceur alloue le processeur à la tâche la plus prioritaire, T2 dans l'exemple.

☞ Mode « Mixed »

- le mode d'ordonnancement est un attribut des tâches

- Intérêts du mode non-préemptif dans un contexte préemptif :
  - si le temps d'exécution de la tâche est faible, peu différent du temps de commutation de contexte (en cas de préemption)
  - engendre une moindre consommation de mémoire RAM
  - c'est un moyen simple de protéger une ressource critique (si le temps d'utilisation de la ressource est très faible)

☞ Ordonnancement et ressources partagées

Lors de **l'accès en exclusion mutuelle** (via les services ad hoc) aux ressources partagées un protocole spécifique est utilisé :

le protocole PCP (**Priority Ceiling Protocol**, « façon » OSEK/VDX)

Il influe sur les priorités des tâches, donc sur l'ordonnancement (voir le chapitre 4)

☞ Groupe de tâches

- Il existe également une notion de « **groupe de tâches** » dans OSEK/VDX

  Elle n'est pas présentée ici.

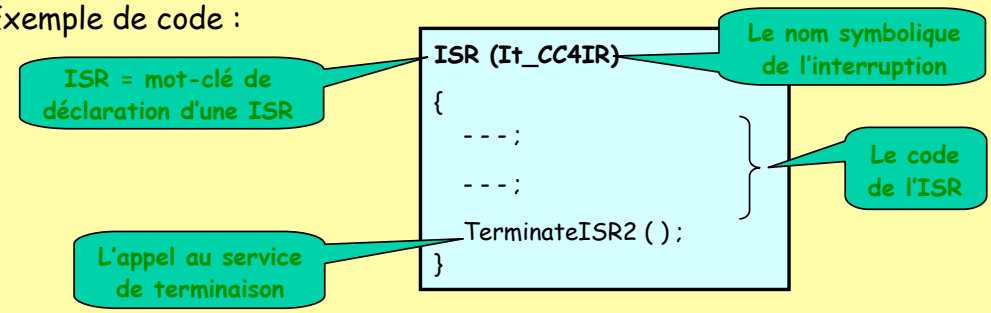
☞ Les ISR : Interrupt Service Routine

Dans l'implémentation Trampoline une ISR est traitée comme une tâche*, notamment elle possède une priorité.

Cependant :

- une ISR est associée à une **interruption matérielle**
- une ISR doit se **terminer** par l'appel au service **TerminateISR2**

Exemple de code :



* Ce choix n'est pas commun. Il est justifié par le fait que OSEK/VDX impose de pouvoir accéder à des données en exclusion mutuelle dans une ISR, ce qui est complexe. Il est bien sûr pénalisant du point de vue des performances temporelles.

☞ Remarques sur les ISR (1)

- Du fait que la requête d'interruption est transformée en une tâche, il n'y a pas de restriction d'utilisation de services de l'OS dans le code de l'ISR (ce qui n'est pas le cas dans l'implémentation classique des ISR)
- La priorité à donner à l'ISR dépend de son importance globale dans l'application, mais toute ISR a une priorité supérieure aux tâches
- L'initialisation de la chaîne matérielle permettant la prise en compte de l'interruption est à la charge de l'utilisateur, **sauf le dernier niveau** :

  dans le cas du C167 il faut :

  - initialiser le registre xxxIC (Interrupt Control) avec les niveaux de hiérarchie souhaités,
  - initialiser le masque local de l'interruption (dans le registre xxxIC)
  - ne pas toucher au registre PSW (ILVL processeur et masque général IEN), c'est le système d'exploitation qui gère cela

  Ces initialisations seront faites dans la routine générale d'initialisation (voir chapitre 3).

☞ Remarques sur les ISR (2)

- Le code de l'ISR traite la requête correspondante. Il peut aussi activer une autre tâche …
- Il est toujours possible de traiter directement une requête d'interruption, en écrivant une routine qui n'appelle aucun service du système d'exploitation

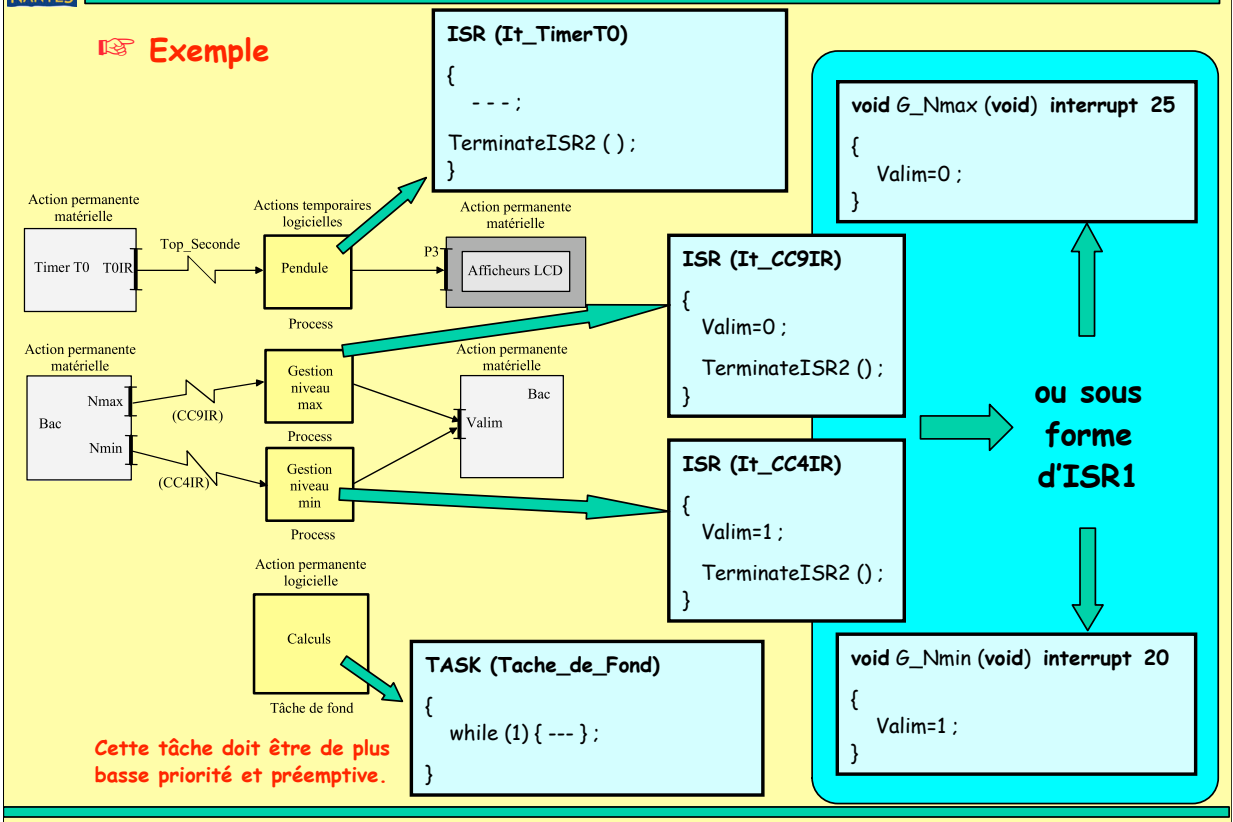
Exemple (avec les règles de la chaîne Keil):

```
sbit  DIV = P7^4;

void Diviseur (void)  interrupt  16
{
   DIV = ~ DIV ;
}
```

- ceci correspond à la notion d'ISR de niveau 1 d'OSEK/VDX (c'est à dire une routine sans appel de services).

☞ Exemple



☞ Les compteurs

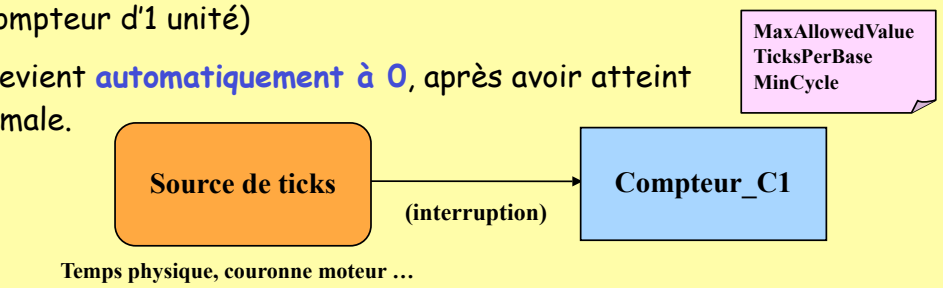
Un « **compteur** » est un dispositif **d'enregistrement** de « ticks » externes, par exemple ceux d'une horloge, ceux du détecteur lié à la couronne moteur qui va permettre de calculer l'avance à l'allumage …

en somme tout dispositif susceptible d'engendrer des « impulsions » récurrentes.

Un compteur est caractérisé principalement par 2 grandeurs :

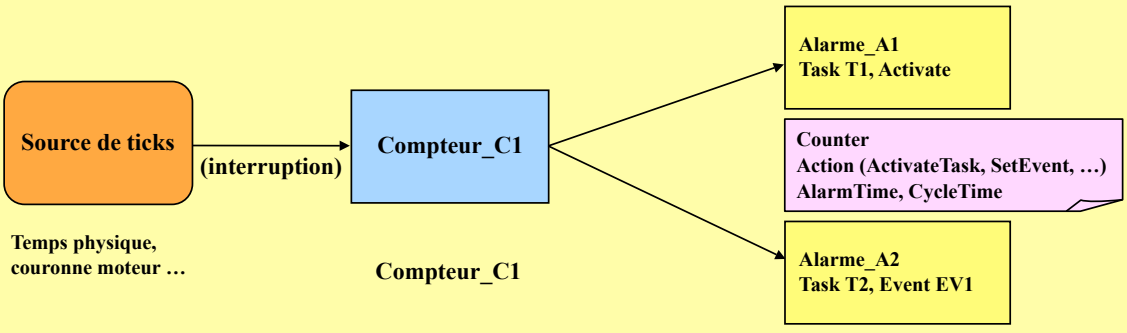
- la **valeur maximale mémorisable** (ex. 255 pour un compteur sur 8 bits)
- un **facteur de pré-division** (par exemple 5 ticks de base pour avancer le compteur d'1 unité)

Le compteur revient **automatiquement à 0**, après avoir atteint sa valeur maximale.



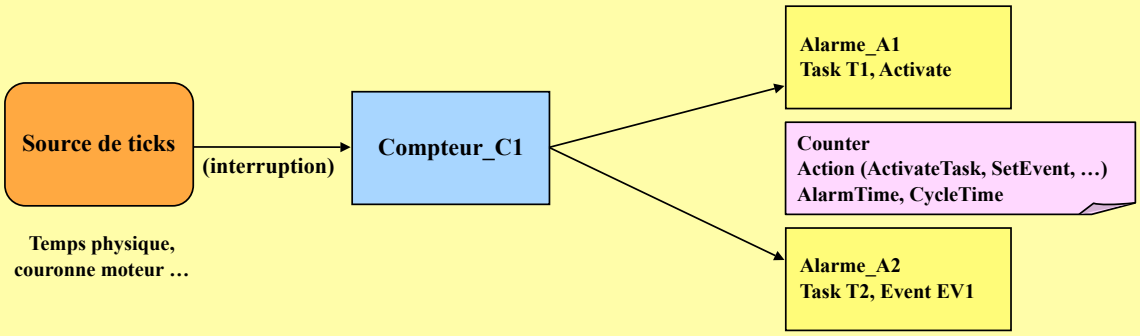
☞ Les alarmes (1)

- Une **alarme** est associée à la configuration à un compteur et une tâche
- l'alarme est déclenchée quand le compteur atteint une **valeur de référence**
- une **action** est alors exécutée, définie à la configuration :
  - **activation** d'une tâche
  - **signalisation** d'une occurrence d'événement (voir chapitre 4)
  - exécution d'une routine de « **callback** » (non détaillé)

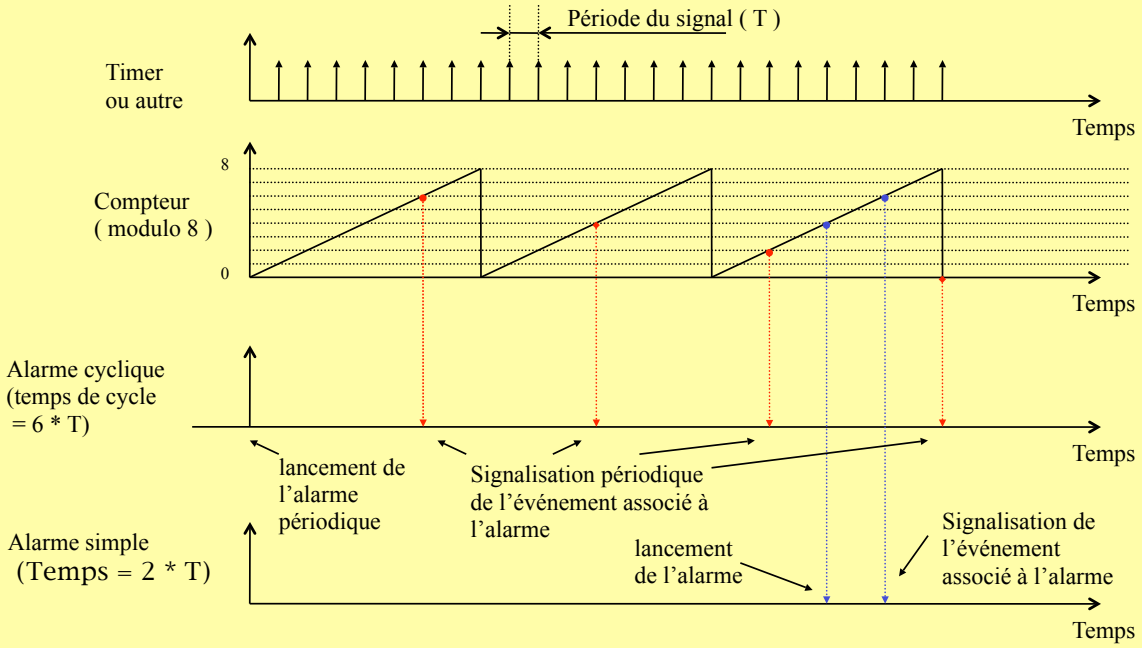


☞ Les alarmes (2)

- une alarme peut être
  - **unique** (valeur de référence relative ou absolue)
  - **cyclique** (ceci est le moyen de créer l'activation des tâches périodiques)
- Paramètres de l'alarme :
  - le compteur associé
  - l'action et ses paramètres
  - la valeur de référence et le temps de cycle si périodique



☞ Exemple avec des actions différentes



☞ Les alarmes (3)

- l'implémentation Trampoline nous offre :
  - une **source de ticks via le timer T6** : il engendre une interruption toutes les **1 ms** (quartz à 40MHz)
  - la possibilité de déclarer compteurs et alarmes
- Exemple de **déclaration OIL** (voir chapitre 3) d'un compteur et d'une alarme :

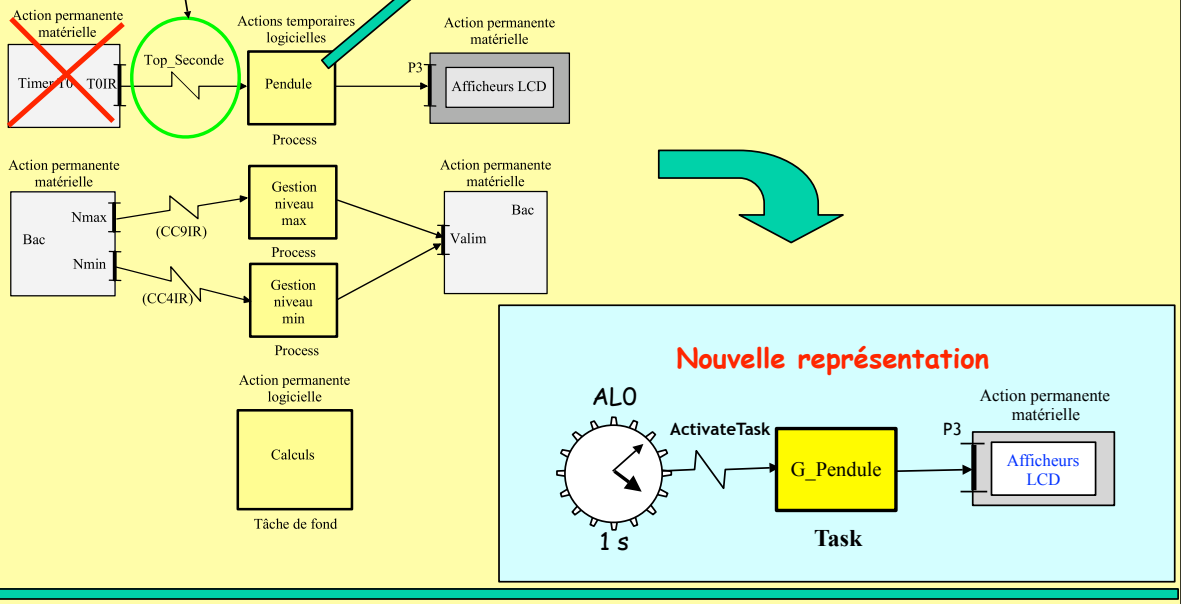
```
COUNTER general_counter {
    TICKSPERBASE = 1;
    MAXALLOWEDVALUE = 2047;
    MINCYCLE = 10;
};
```

```
ALARM Top_Seconde {
    COUNTER = general_counter;
    ACTION = ACTIVATETASK {
        TASK = G_Pendule ;
    };
    AUTOSTART = TRUE {
        ALARMTIME = 10;
        CYCLETIME = 1000;
        APPMODE = std;
    };
};
```

- Toutes les 1 ms le compteur est incrémenté.
- L'alarme Top_Seconde expire pour la première fois à l'instant 10 ;
- La période est de 1000 x 1 ms = 1 s.
- Toutes les 1 s. on active la tâche « G_Pendule ».

☞ Exemple

```
TASK (G_Pendule)
{
  ---;
TerminateISR2 ();
}
```



☞ Les alarmes (4) : services pour les alarmes

- On peut utiliser les services associés aux alarmes pour faire d'autres constructions que la gestion de tâches périodiques
- 2 services pour activer une alarme

SetRelAlarm : arme une alarme dont la valeur de référence est relative à « l'instant présent »

```
Status SetRelAlarm ( AlarmType  Alarm_ID    // le nom de l'alarme
                     TickType   Increment   // le temps relatif au présent
                     TickType   Cycle )     // ≠ 0 si alarme périodique
```

SetAbsAlarm : arme une alarme dont la valeur de référence est une valeur absolue

```
Status SetAbsAlarm ( AlarmType  Alarm_ID    // le nom de l'alarme
                     TickType   Start       // la « date » d'expiration
                     TickType   Cycle )     // ≠ 0 si alarme périodique
```

Les paramètres « Increment » et « Start » désignent l'instant du premier déclenchement, l'alarme est ensuite périodique (si Cycle ≠ 0) ou unique.

☞ Les alarmes (5) : services pour les alarmes

- 2 services pour gérer une alarme

GetAlarm : permet d'obtenir l'état courant de l'alarme

```
Status GetAlarm (  AlarmType  Alarm_ID    // le nom de l'alarme
                   TickType*  Tick        // le nombre de ticks restant avant
                                          // l'expiration de l'alarme
```

CancelAlarm : désarme l'alarme désignée

```
Status CancelAlarm ( AlarmType  Alarm_ID )    // le nom de l'alarme
```

Pour changer les paramètres d'une alarme il faut d'abord l'arrêter (CancelAlarm) puis la ré-activer (SetRelAlarme ou SetAbsAlarm).

Annexe

Rappels sur les interruptions du C167

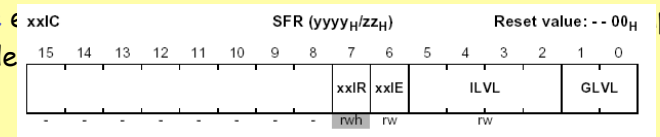
☞ Eléments de base

L'origine des demandes d'interruption est :

- soit externe, par les lignes _NMI et EX0IN (P2.8) à EX7IN (P2.15)(non étudié ici)
- soit interne, par les nombreux éléments intégrés du microcontrôleur
  - entrées de capture, sorties de comparaison
  - liaisons séries, liaisons parallèles
  - PWM, etc…

Il y a **56 sources internes** de demandes d'interruptions dans le C167.

A chaque source e



ple CC0IC pour le canal 0 de

Champs de xxIC : exemple avec CC0IC

CC0IC (FF78h/BCh)			SFR				Reset : 00h
7	6	5	4	3	2	1	0
CC0IR	CC0IE	ILVL				GLVL	
rw	rw	rw				rw	

CC0IR : CC0 Interrupt **Request** : bit (si à 1) indiquant une détection, soit d'une transition active (mode capture), soit d'une comparaison réussie (mode comparaison)

CC0IE : CC0 Interrupt **Enable** : bit jouant le rôle d'un masque local, autorisant (si à 1) la génération d'une requête d'interruption lorsque le bit CC0IR passe à 1.

ILVL : Interupt **Level** : C'est le niveau de priorité de l'interruption. Ceci est très important lorsqu'il y a des requêtes simultanées, car on peut traiter qu'une interruption à la fois.

ILVL est un champ de 4 bits

16 niveaux d'importance possibles pour les 56 IT (0 : le plus faible, 15 : le plus fort)

GLVL : **Group Level**

16 niveaux seulement ➔ notion de groupe pour définir l'ordre interne à un niveau

(0 : niveau le plus faible, 3 : niveau le plus fort)

ILVL + GLVL donnent 64 combinaisons possibles pour 56 demandes (48 sont utilisables)

Toute les interruptions doivent avoir une combinaison (ILVL, GLVL) différente.

Vecteur d'interruption

A chaque interruption est associée une adresse mémoire (en fait 4 octets) dans le segment 0 :

le vecteur d'interruption

A chaque interruption est associé un numéro de « trappe » :

le Trap Number

Ce numéro multiplié par 4 donne l'adresse mémoire du vecteur d'interruption.

Tableau des paramètres pour quelques sources internes

Source	*Drapeau*	*Masque*	*Registre*	*adresse du vecteur*	*N° de trappe*
Canal 0 CAPCOM1	**CC0IR**	CC0IE	CC0IC	**0x00'0040**	**16**
Canal 4 CAPCOM1	CC4IR	CC4IE	CC4IC	0x00'0050	20
Canal 12 CAPCOM1	CC12IR	CC12IE	CC12IC	0x00'0070	28
Canal 13 CAPCOM1	CC13IR	CC13IE	CC13IC	0x00'0074	29
Timer 0	**T0IR**	T0IE	T0IC	**0x00'0080**	**32**
Timer 1	T1IR	T1IE	T1IC	0x00'0084	33

☞ Le registre PSW (Processor Status Word)

PSW
Processor Status Word — SFR ($FF10_H/88_H$) — Reset value: 0000_H

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ILVL				IEN	HLD EN	-	-	-	USR 0	MUL IP	E	Z	V	C	N
rw				rw	rw	-	-	-	rw	rwh	rwh	rwh	rwh	rwh	rwh

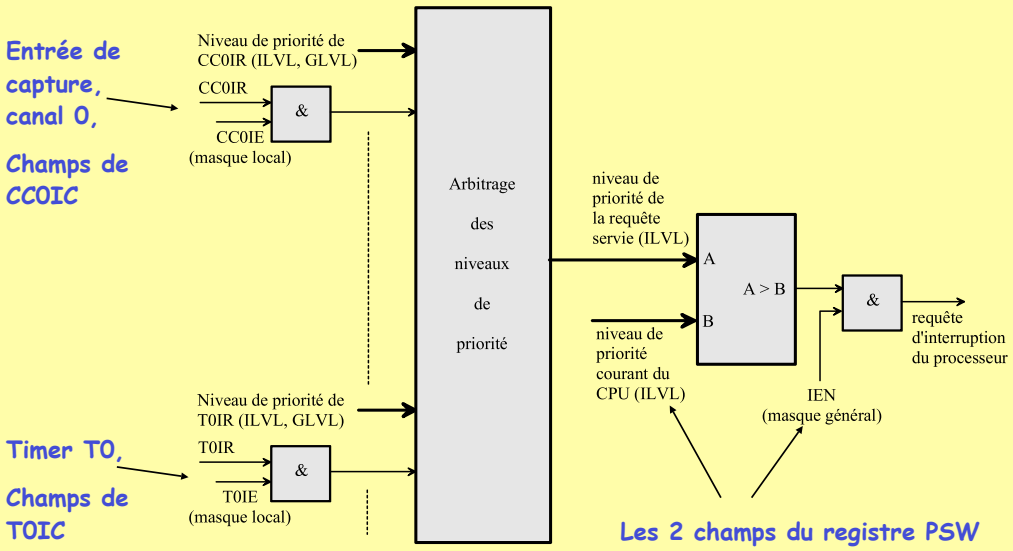
IEN : Interrupt **Enable** (si à 1)

Masque global pour toutes les interruptions. Les autorise (si à 1) ou non.

ILVL : Interrup **Level**

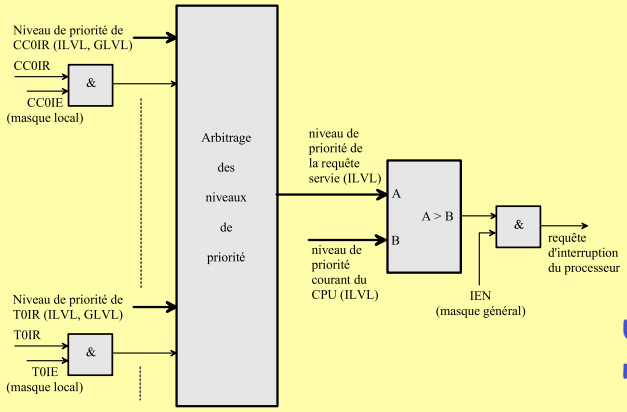
Niveau courant d'interruption pour le processeur

☞ Cheminement d'une requête



Comment une requête progresse-t-elle ?

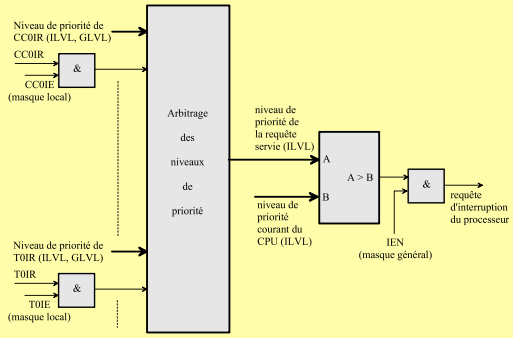
Annexe - Rappels sur les interruptions du C167



Une requête (par exemple CCOIR à 1) ne peut progresser que si :

- elle est autorisée à engendrer une requête (**masque local CCOIE à 1**)
- son niveau d'interruption (**ILVL dans CCOIC**) est strictement supérieur à celui de toute autre requête présente ou bien s'il est égal à celui d'autres requêtes présentes, son niveau de groupe (**GLVL dans CCOIC**) est supérieur à celui des autres requêtes de même niveau
- son niveau d'interruption (**ILVL**) est strictement supérieur à celui du CPU (**champ ILVL de PSW**)
- les interruptions sont autorisées (**masque général IEN à 1**)

Annexe - Rappels sur les interruptions du C167



Remarques :

- Sur **Reset**, PSW est mis à 0 ➔ $(ILVL)_{CPU} = 0$
  - ➔ Une requête avec un niveau de priorité 0 n'est jamais prise en compte
  - ➔ Les interruptions ne sont pas acceptées (IEN à 0)
- Les **niveaux 14 et 15** sont utilisés pour un dispositif spécifique : le PEC (Peripheral Event Controller).
  - ➔ **utiliser** seulement les **niveaux 1 à 13** pour hiérarchiser vos requêtes
- Le programme peut contrôler, à tout moment, les demandes d'interruption qu'il veut accepter en fixant la valeur de ILVL dans PSW

⓪ Le processeur **termine** toujours **l'instruction** en cours (sauf MUL/DIV, voir plus loin)

① **Sauvegarde** sur la pile de PSW, puis de CSP (Code Segment Pointer) si on est en mode segmenté, puis IP (Instruction Pointer). On a ainsi sauvegardé le **contexte minimal** du CPU :

- le registre d'état de la machine (PSW)
- l'adresse de la prochaine instruction à exécuter (IP + CSP)

Ce sont les informations minimales pour pouvoir reprendre le traitement après l'interruption (de programme) ;

② **Mise à jour** de PSW (champ ILVL) par le niveau d'interruption actuellement servi.

➔ seules les requêtes de priorité supérieures à celle en cours pourront interrompre le processeur

③ **Remise à zéro** de la requête servie : le bit xxIR est **automatiquement** remis à 0 ;

④ **Chargement** dans le registre **IP** du vecteur d'interruption (n° de trappe x par 4) ;
Remise à zéro de CSP si on est en mode segmenté (les vecteurs sont dans le segment 0) ;

⑤ A l'adresse du vecteur d'interruption, l'utilisateur met en principe une instruction de saut (jmp cc_UC,…) à la routine de traitement de l'interruption :

➔ **exécution** de ce saut et **le programme se continue dans la routine d'IT**

Annexe - Rappels sur les interruptions du C167

Forme générale valable pour une programmation en assembleur

Point d'entrée:

sauvegarde du contexte	**fait automatiquement par le compilateur C**
traitement de l'IT	**sera le corps de votre fonction**
restitution du contexte	**fait automatiquement par le compilateur C**
reti	**traduit la parenthèse fermante (})**

☞ **Sauvegarde du contexte**

Pourquoi ? Si la routine fait des calculs utilisant les registres, ceux-ci vont être détruits (les valeurs qu'il y avait dans le programme interrompu vont être perdues)

➔ il faut les sauvegarder avant de les utiliser

☞ **Restitution du contexte**

Pourquoi ? il faut rendre au programme interrompu les valeurs qu'il y avait dans les registres avant son interruption

☞ **reti : Return from Interrupt** (instruction assembleur)

Récupération dans la pile des valeurs de PSW, (CSP) et IP pour revenir au point d'interruption dans le programme interrompu : c'est la **restitution du contexte minimal** sauvegardé en acceptant l'interruption.

☞ Format en C d'une routine d'interruption (chaîne KEIL)

Obligatoire car c'est le matériel qui active, donc pas de paramètres

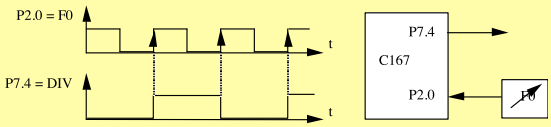
```
void function_name (void)  interrupt  trap_number
```

Obligatoire car je ne peux rien renvoyer

Indique au compilateur que c'est une procédure d'interruption

Le n° de trappe de la requête d'interruption

Exemple pour le diviseur par 2




```
sbit        DIV= P7^4 ;

void Diviseur (void)  interrupt  16
{
        DIV = ~ DIV ;
}
```

Le n° de trappe de CCOIR est 16

A chaque front on bascule la sortie, puis on quitte la procédure

Sera traduit par « reti » pour revenir au programme interrompu

De plus le compilateur a généré l'instruction de saut qui sera placée, lors du chargement, à l'adresse 0x0040 (4 fois 16 en hexa), afin d'activer automatiquement la procédure.

☞ Structure en C du programme principal

```
void main (void)
{
    Initialisation_diverses (ports,timer) ;   ← Ceci est hors interruptions

    Initialisations des interruptions;        ← la validation des interruptions

    Travail de "fond" ;                       ← Ceci peut être interrompu
}
```

Exemple pour le diviseur par 2

```
void Init_ES (void)
{
   DP7 = 0x10 ;
   P7 = 0;
   CCM0 = 1 ;
}
```

```
void Diviseur (void)  interrupt  16
{
   DIV = ~ DIV ;
}
```

ILVL = 4 GLVL = 0

```
void main (void)
{
   Init_ES();
   CC0IC = 0x10 ;  // 0001 0000
   CC0IE = 1 ;     // masque local à CC0
   IEN = 1 ;       // masque général
   while (1)  {
      tache_de_fond ()
   }
}
```

☞ Interruptions, vecteurs, n° de trappes (incomplet)

Source	*Drapeau*	*Masque*	*Registre*	*adresse du vecteur*	*N° de trappe*
Canal 0 CAPCOM1	CC0IR	CC0IE	CC0IC	0x00'0040	16
Canal 1 CAPCOM1	CC1IR	CC1IE	CC1IC	0x00'0044	17
Canal 2 CAPCOM1	CC2IR	CC2IE	CC2IC	0x00'0048	18
Canal 3 CAPCOM1	CC3IR	CC3IE	CC3IC	0x00'004C	19
Canal 4 CAPCOM1	CC4IR	CC4IE	CC4IC	0x00'0050	20
Canal 5 CAPCOM1	CC5IR	CC5IE	CC5IC	0x00'0054	21
Canal 6 CAPCOM1	CC6IR	CC6IE	CC6IC	0x00'0058	22
Canal 7 CAPCOM1	CC7IR	CC7IE	CC7IC	0x00'005C	23
Canal 8 CAPCOM1	CC8IR	CC8IE	CC8IC	0x00'0060	24
Canal 9 CAPCOM1	CC9IR	CC9IE	CC9IC	0x00'0064	25
Canal 10 CAPCOM1	CC10IR	CC10IE	CC10IC	0x00'0068	26
Canal 11 CAPCOM1	CC11IR	CC11IE	CC11IC	0x00'006C	27
Canal 12 CAPCOM1	CC12IR	CC12IE	CC12IC	0x00'0070	28
Canal 13 CAPCOM1	CC13IR	CC13IE	CC13IC	0x00'0074	29
Canal 14 CAPCOM1	CC14IR	CC14IE	CC14IC	0x00'0078	30
Canal 15 CAPCOM1	CC15IR	CC15IE	CC15IC	0x00'007C	31
Timer 0	T0IR	T0IE	T0IC	0x00'0080	32
Timer 1	T1IR	T1IE	T1IC	0x00'0084	33
Timer 7	T7IR	T7IE	T7IC	0x00'00F4	61
Timer 8	T8IR	T8IE	T8IC	0x00'00F8	62
A/D Conv. Complete	ADCIR	ADCIE	ADCINT	00'00A0	40
A/D Overrun Error	ADEIR	ADEIE	ADEINT	00'00A4	41
PWM channel 0..3	PWMIR	PWMIE	PWMINT	00'00FC	63