


Systèmes et réseaux temps réel

Chap V: Gestion des interruptions

Systèmes et réseaux temps réel
(youssefrochdi@yahoo.fr)

105



Objectifs

- Montrer comment différer le traitement des événements provoquant des interruptions dans un contexte multitâches.
- Montrer les précautions à prendre lors de l'utilisation des fonctions de l'API dans les routines des traitements d'interruptions.
- Donner les mécanismes de communications entre les routines d'interruptions et les tâches normales.
- Montrer comment on peut prendre en considération des événements asynchrones successifs.

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)

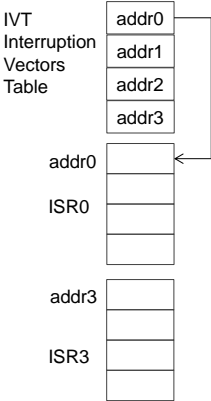
106

1- Interruptions (rappel)

- Une interruption est un mécanisme utilisé par les microprocesseurs ou microcontrôleurs pour:
 - Prendre en considération des événements externes au processeur **asynchrones**: par exemple changement d'état d'une entrée, arrivé d'un octet sur UART, fin de conversion A/N, ...
 - Eviter de bloquer le processeur en attente de la réalisation d'une condition interne, par exemple: fin de conversion A/N, fin de transmission d'un octet sur UART, débordement d'un Timer...
 - Traiter des événements internes exceptions (trap): division par zéro, débordement de la pile, ..
- En général, un microcontrôleur peut avoir **plusieurs sources** d'interruption, qui peuvent être hiérarchisées par des **priorités**.

1- Interruptions (Rappel)

- A chaque interruption est associé:
 - **un vecteur**: zone mémoire fixe contenant l'adresse de la routine de traitement de cette interruption (Interrupt Sub-Routine ISR).
 - Un bit d'activation/désactivation (Enable/Disable bit: **IE**)
 - Un bit d'indication ou drapeau (Flag bit: **IF**)
- L'emplacement où se trouve les vecteurs d'interruptions est fixe et s'appelle Table des vecteurs d'interruptions.

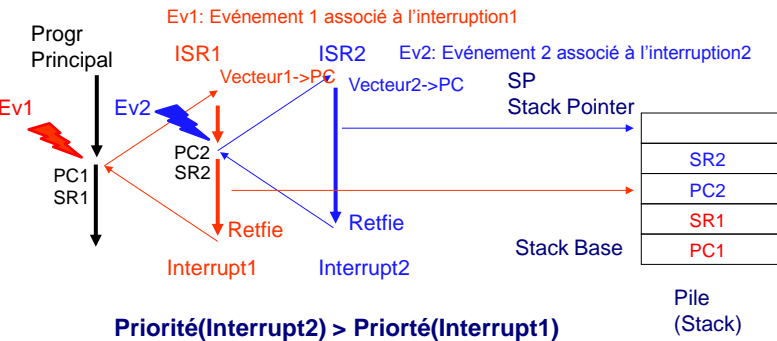


1- Interruptions (Rappel)

- Lorsque l'événement associé à une interruption **activée (IE=1)**, survient, le microC :
 - Termine l'instruction en cours
 - sauvegarde automatiquement dans la pile le PC et le SR (ou une partie du SR) et éventuellement d'autres registres (selon le microC).
 - Charge dans le PC le vecteur correspondant à l'interruption.
- Lorsque l'instruction **RETFIE** (Return from Interruption) est rencontrée dans le ISR (Interrupt Sub-routine), le microC:
 - Restaure le SR et le PC à partir de la pile (et éventuellement les autres registres s'il y en a)
 - Continue le traitement après l'instruction où a été interrompu le programme.

1- Interruptions (Rappel)

- Les interruptions peuvent être imbriquées(Nested):



- Le drapeau (flag) (IF) doit être remis à 0 dans le ISR.

2- Interruptions dans un contexte multi-tâches (1)

- Dans un contexte multi-tâches:
 - Les tâches s'exécutent de manière concurrente (progr principal).
 - Les interruptions interrompent toujours les tâches.
 - Le **scheduler lui-même peut être interrompu** par les interruptions!
- Donc, les sous-programmes d'interruption doivent:
 - éviter d'appeler des fonctions bloquantes
 - Ou faire un traitement long

Au risque de monopoliser le processeur et d'annuler le pseudo-parallélisme, donc le multitâche concurrentiel, et enfin le respect de contraintes temps réel.

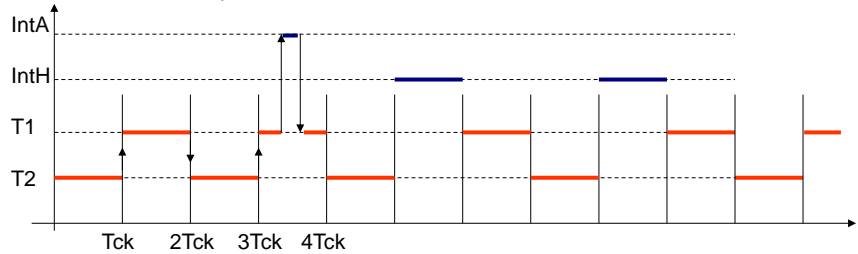
2- Interruptions dans un contexte multi-tâches(2)



Le traitement d'interruption contient des instructions bloquantes et/ou effectue un traitement long et prive les autres tâches du processeur pendant une longue durée
→ Ceci peut avoir de lourdes conséquences sur la réactivité de l'application, et le non respect des contraintes temporelles.

3- Traitement différé des interruptions

Traitement d'interruption en deux: ISR et une tâche Handler



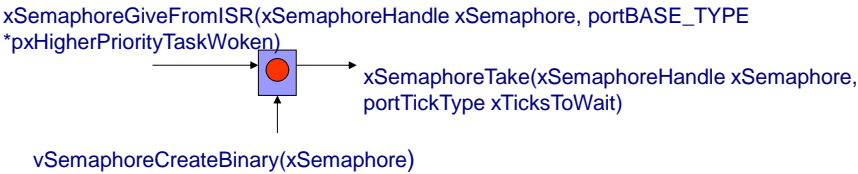
IntA: ISR (traitement court)
IntH: traitement d'Interruption long sous forme **d'une tâche concurrente** avec les autres,
T1: Tâche1, T2: Tâche2, Tck Tick d'interruption du scheduler

L'ISR consiste en un traitement le plus court possible pour prendre des infos sur l'événement déclencheur de l'interruption avant sa disparition, puis débloque/repren une tâche (Handler) pour faire le traitement long conséquent à l'interruption sous forme de tâche concurrente aux autres tâches.

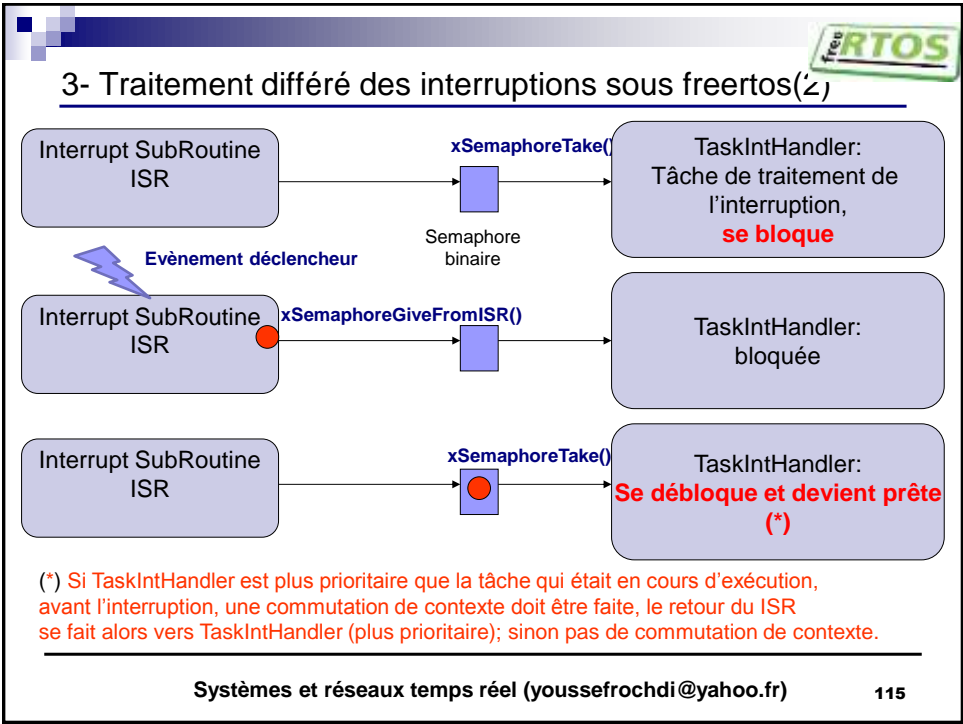
3- Traitement différé des interruptions sous freertos(1)



- FreeRTOS, n'impose aucune stratégie particulière pour traiter les interruptions, mais offre des mécanismes pour réaliser du traitement différé d'une interruption.
- L'un de ces mécanismes est le sémaphore*.
- Un sémaphore **binaire** peut être vu conceptuellement comme une file d'attente pouvant contenir **au plus un seul item**.



*Concept abstrait inventé par Edsger Dijkstra pour l'exclusion mutuelle



4- Application au PIC24FJ128GA010

- Ce MicroC dispose de plusieurs sources d'interruptions (voir [Datasheet Section 8. Interrupt Controller page 63](#)).
 - Les bits d'activation sont dans les registres IECn (n=0,...,4).
 - Les flags de notification sont dans les registres IFSn (n=0,...,4).
 - Les bits de priorité sont dans les registres IPCn (n=0,...,16).
- D'autres registres sont aussi utilisés pour configurer ou renseigner sur les états de traps, INTCON1, INTCON2, INTTREG.
- Des directives pour écrire des ISR:
 - Déclarer les ISR comme des fonctions sans paramètres et retournant un type void (**obligatoire**).
 - Ne pas appeler les ISR à partir de la fonction main ou autre fonction (**obligatoire**).
 - Ne pas appeler des fonctions dans les ISR (**recommandé**).

Systemèmes et réseaux temps réel (youssefrochdi@yahoo.fr) 116

4- Application au PIC24FJ128GA010

- Freertos autorise l'utilisation de certains fonctions de l'API à partir des ISR (dont le nom se termine par FromISR): **ce ne sont pas des fonctions bloquantes.**
- Le compilateur C30 ou xc16 utilise la syntaxe suivante pour définir l'ISR:
`void __attribute__((__interrupt__)) _INT1Interrupt(void);`
Ou en utilisant les macros pour alléger la notation.
`void _ISR _INT1Interrupt(void);`

Le nom n'est pas arbitraire, il est prédéfini par le compilateur; A chaque vecteur d'interruption est associé un nom qui évoque la source d'interruption. Voir le document "Mplab C Compiler 16-bits Devices help.chm" :
[C:/Program%20Files%20\(x86\)/Microchip/xc16/v1.35/docs/vector_docs/PIC24FJ128GA010.html](C:/Program%20Files%20(x86)/Microchip/xc16/v1.35/docs/vector_docs/PIC24FJ128GA010.html)

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)

117

4- Application au PIC24FJ128GA010

TABLE: INTERRUPT VECTORS - PIC24F MCUS

IRQ#	Primary Name	Alternate Name	Vector Function
N/A	_ReservedTrap0	_AltReservedTrap0	Reserved
N/A	_OscillatorFail	_AltOscillatorFail	Oscillator fail trap
N/A	_AddressError	_AltAddressError	Address error trap
N/A	_StackError	_AltStackError	Stack error trap
N/A	_MathError	_AltMathError	Math error trap
N/A	_ReservedTrap5	_AltReservedTrap5	Reserved
N/A	_ReservedTrap6	_AltReservedTrap6	Reserved
N/A	_ReservedTrap7	_AltReservedTrap7	Reserved
0	_INT0Interrupt	_AltINT0Interrupt	INT0 External interrupt 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 Input capture 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 Output compare 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 expired
4	_Interrupt4	_AltInterrupt4	Reserved
5	_IC2Interrupt	_AltIC2Interrupt	IC2 Input capture 2
			OC2 Output compare 2

Extrait de "Mplab C Compiler 16-bits Devices help.chm"

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)

118

freeRTOS

4- Application au PIC24FJ128GA010

■

Exemple: on souhaite rajouter en parallèle des tâches du TP3 la commande d'un moteur pas à pas par microcontrôleur; chaque impulsion sur bouton poussoir provoque alternativement l'avance ou le recul du moteur de 4 pas.

□

Moteurs à 4 phases → RD0-RD3 pour le commander

□

Interruption externe INT1

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)

119

freeRTOS

4- Application au PIC24FJ128GA010

Dans un fichier stepmotor.c

```
#include <p24FJ128GA010.h>
#include "freertos.h"
#include <semphr.h>
#include "task.h"

//the semaphore used to differ the interruption handling
xSemaphoreHandle xSemaphore;

void _ISR_INT1Interrupt(void) {
    signed portBASE_TYPE pxHigherPriorityTaskWoken=pdFALSE;
    //ou void __attribute__((__interrupt__)) _INT0Interrupt(void){
    _INT1IF = 0; //clear INT1 flag
    //give the semaphore to unblock the vTaskINT1Handler
    xSemaphoreGiveFromISR(xSemaphore, &pxHigherPriorityTaskWoken);
    //pxHigherPriorityTaskWoken is set to pdTRUE if giving the semaphore caused
    // a task to unblock, and the unblocked task has a priority higher than
    //the currently running task --> context switch
    if(pxHigherPriorityTaskWoken==pdTRUE) taskYIELD();
}
```

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)

120

4- Application au PIC24FJ128GA010

```
void vTaskINT1Handler(void) {
    unsigned int cmdword; //command word to control stepper Motor
    unsigned char direction = 'f'; //'f' forward moving, 'b' backward moving
    vSemaphoreCreateBinary(xSemaphore); // Create the semaphore
    xSemaphoreTake(xSemaphore, portMAX_DELAY);
    // extern interruption INT1's configuration
    //negative edge, //priority =4; //enable interrupt INTO
    _INT1IEP = 1; _INT1IP = 4; _INT1IF = 0; _INT1IE = 1;
    //RD3-RD0 commands the four StepperMotor's coils
    TRISD = 0xFFFF0; //RD3-RD0 port D as output, the others as inputs
    PORTD = 0X0001; //initial command
    while (1) {
```

Retire le jeton créé par défaut après la création du sémaphore

4- Application au PIC24FJ128GA010

```

while (1) {
    xSemaphoreTake(xSemaphore, portMAX_DELAY); //wait the semaphore (permanently)

    INT1111 = 0; //disable INT1 interruption
    if (direction == 'x') {
        cmdword = 0x1;
        while (cmdword < 8) {
            cmdword = cmdword << 1; //cmdword==0x0002, 0x0004, 0x0008
            PORTID = (PORTID & 0xFFFF0) | cmdword; vTaskDelay(500);
        }
        cmdword = 0x01; PORTID = (PORTID & 0xFFFF0) | cmdword;
        vTaskDelay(500); direction = 'x';
    } else {
        cmdword = 0x08; PORTID = (PORTID & 0xFFFF0) | cmdword; vTaskDelay(500);
        while (cmdword > 1) {
            cmdword = cmdword >> 1; ///cmdword==0x0004, 0x0002, 0x0001
            PORTID = (PORTID & 0xFFFF0) | cmdword; vTaskDelay(500);
        }
        direction = 'x';
    }
}

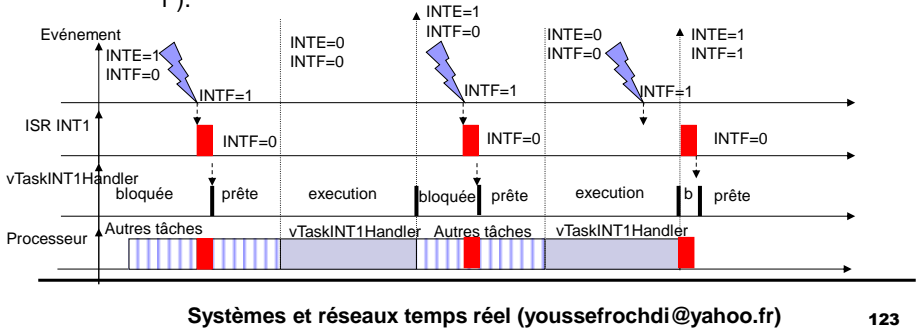
INT1111 = 1; //re-enable INT1
}

```

La tâche vTaskINT1Handler() doit être créée dans le programme principal

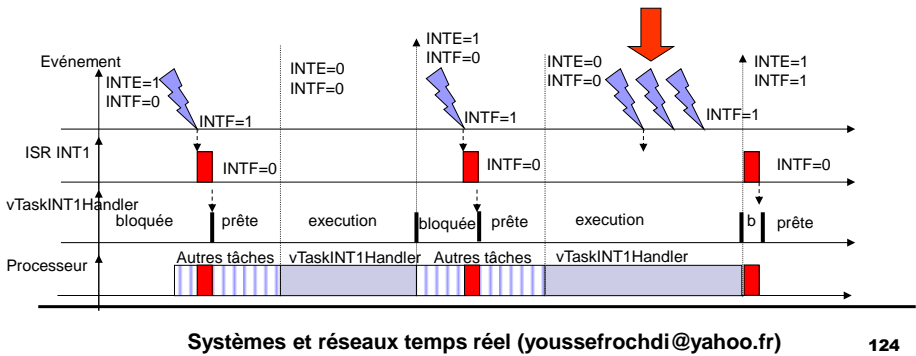
4- Application au PIC24FJ128GA010

- Remarques:
 - Dès que la tâche vTaskINT1Handler() reçoit le sémaphore, elle désactive l'interruption INT1 (`_INT1IE = 0`), jusqu'à ce que le traitement de l'interruption se termine (`_INT1IE = 1`).
 - Lors d'un traitement d'un événement provoquant l'interruption INT1, si l'événement réapparaît une deuxième fois, `_INT1IF` est mis à 1, mais ISR ne sera pas exécutée qu'à la fin du premier traitement (`_INT1IE = 1`).



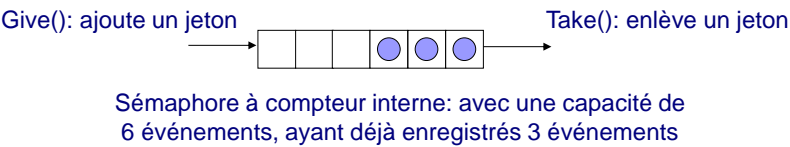
5- Cas d'événements multiples successifs

- Un seul événement, apparaissant lors du traitement d'un autre événement précédent, sera pris en charge.
- Cette technique ne permet pas de prendre en considération les événements asynchrones qui se répètent plusieurs fois de manière rapide et sporadique.
- Ceci est aussi le cas, même si vTaskINTHandler ne désactive pas l'interruption pendant le traitement (Sémaphore binaire: au plus un)



6- Semaphore à compteur interne (1)

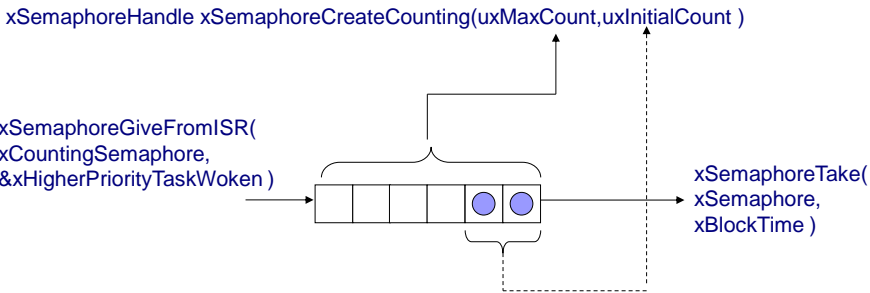
- Pour dépasser le problème évoqué précédemment il faut:
 - vTaskINTHandler() ne désactive pas l'interruption pendant le traitement, pour permettre d'exécuter ISR autant de fois que l'événement apparait.
 - Utiliser un sémaphore capable de mémoriser l'apparition multiple de l'événement → **sémaphore à compteur interne**
- Un sémaphore à compteur interne peut être considéré comme une file d'attente capable de mémoriser N événements.




6- Semaphore à compteur interne(2)



- FreeRTOS implémente le sémaphore à compteur interne:





6- Application au PIC24FJ128GA010 (1)


Dans FreeRTOSConfig.h il faut ajouter
`#define configUSE_COUNTING_SEMAPHORES 1`
Le seul changement dans stepmotor.c

```
void vTaskINT1Handler(void) {
    unsigned int cmdword; //command word to control stepper Motor
    unsigned char direction = 'f'; // 'f' forward moving, 'b' backward moving
    xSemaphore=xSemaphoreCreateCounting(3,0); // Create the semaphore
    // extern interruption INT1's configuration
    //negative edge, //priority =4; //enable interrupt INT0
    _INT1IEP = 1; _INT1IP = 4; _INT1IF = 0; _INT1IE = 1;
    //RD3-RD0 commands the four StepperMotor's coils
    TRISD = 0xFFFF0; //RD3-RD0 port D as output, the others as inputs
    PORTD = 0X0001; //initial command
    while (1) {
        xSemaphoreTake(xSemaphore, portMAX_DELAY); //wait the semaphore (permanently)

        // _INT1IE = 0; //disable INT1 interruption
        if (direction == 'f') {
```

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)

127



7-Utilisation des queues dans les ISR

Les ISR peuvent communiquer avec des tâches en utilisant des Queues, mais ne doivent pas utiliser que les fonctions suivantes de l'API :
`xQueueSendFromISR(), xQueueSendToFrontFromISR(), xQueueSendToBackFromISR(), xQueueReceiveFromISR()`

NB: à l'inverse des autres fonctions de l'API concernant les Queues, il n'y a pas de délai d'attente ...

```
portBASE_TYPE xQueueSendToFrontFromISR( xQueueHandle xQueue,
void *pvItemToQueue, portBASE_TYPE *pxHigherPriorityTaskWoken);
```

`pxHigherPriorityTaskWoken`: est mis à `pdTRUE` s'il y a une tâche bloquée sur cette Queue, de priorité supérieure à celle en cours d'exécution, et dans ce cas une commutation de contexte peut être forcée par `taskYIELD()`.

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)

128

7-Utilisation des queues dans les ISR (Exemple 1/2)

ISR
U1RXInterrupt

xRxdChars

Tâche
Crée la Queue
Traite en boucle les caractères reçus

```
void __attribute__((__interrupt__, auto_psv)) _U1RXInterrupt( void )
{
    char cChar;
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    IFS0bits.U1RXIF = serCLEAR_FLAG;
    while( U1STAbits.URXDA ) //while there are characters in the receiver buffer
    {
        cChar = U1RXREG;
        xQueueSendFromISR( xRxdChars, &cChar, &xHigherPriorityTaskWoken );
    }
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        taskYIELD();
    }
}
```

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)129

7-Utilisation des queues dans les ISR (Exemple 2/2)

ISR
U1RXInterrupt


xRxdChars

Tâche
Crée la Queue
Traite en boucle les caractères reçus

La tâche utilise cette fonction

```
signed portBASE_TYPE xSerialGetChar( xComPortHandle pxPort, signed char *pcRxdChar,
portTickType xBlockTime )
{
    /* Only one port is supported. */
    ( void ) pxPort;
    /* Get the next character from the buffer. Return false if no characters
    are available or arrive before xBlockTime expires. */
    if( xQueueReceive( xRxdChars, pcRxdChar, xBlockTime ) )
    {
        return pdTRUE;
    }
    else{ return pdFALSE; }
}
```

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)130




8- Imbrication des interruptions (1/3)

- Freertos désactive (momentanément) les interruptions lors de traitement de certaines zones critiques. Certaines interruptions peuvent être donc retardées par le noyau freertos(Kernel).
- Plusieurs implémentations récentes de FreeRTOS autorisent l'imbrication des interruptions.
- Certaines implémentations définissent, dans freertosconfig.h, l'une ou les deux macros suivantes:
 - `configKERNEL_INTERRUPT_PRIORITY` : fixe la priorité de l'interruption qui génère les ticks d'hologe pour le noyau (kernel) , par défaut elle est à 1.
 - `configMAX_SYSCALL_INTERRUPT_PRIORITY`: fixe le niveau maximum de priorité, au dessus duquel les fonctions de l'API ne peuvent pas être utilisées.

NB: il s'agit ici des priorités des interruptions et non celles des tâches,
→ ne pas confondre les deux

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)

131



8- Imbrication des interruptions (2/3)


Exemple:

```
#define configKERNEL_INTERRUPT_PRIORITY 1
#define configMAX_SYSCALL_INTERRUPT_PRIORITY 4
```

Les ISR qui n'utilisent aucune fonction API, peuvent utiliser n'importe quel niveau et peuvent aussi être imbriquées.	Priorité 7	Les ISR de ces niveaux ne peuvent pas utiliser les fonctions API, elles ne sont jamais désactivées (donc retardées) par le Kernel.
	Priorité 6	
	Priorité 5	
	Priorité 4	Les ISR de ces niveaux peuvent utiliser les fonctions API, elles sont susceptibles d'être désactivées (donc retardées) par le Kernel, dans les sections critiques; elles peuvent être imbriquées.
	Priorité 3	
	Priorité 2	
	Priorité 1	

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)

132



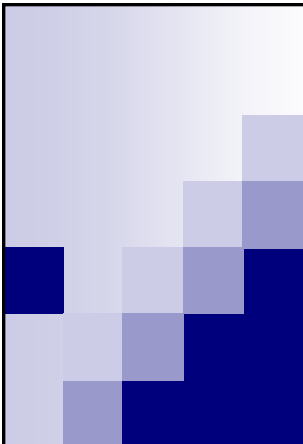
8- Imbrication des interruptions (3/3)

- Les implémentations de freertos qui ne définissent que la macro `configKERNEL_INTERRUPT_PRIORITY`, comme le cas de l'implémentation pour les pic24 (cas du PIC24J128GA010 par exemple), ne permettent aux ISR d'utiliser les fonctions de l'API que s'ils ont un niveau de priorité \leq `configKERNEL_INTERRUPT_PRIORITY`
- Le Kernel de Freertos utilise, pour désactiver les interruptions lors d'un traitement d'une section critique, les deux macros (portmacro.h):

```
#define portDISABLE_INTERRUPTS() SR |= portINTERRUPT_BITS  
#define portENABLE_INTERRUPTS() SR &= ~portINTERRUPT_BITS
```
- Les interruptions qui ne doivent pas être désactivées (momentanément) par le Kernel doivent être définies avec un niveau de priorité **supérieur** à `configMAX_SYSCALL_INTERRUPT_PRIORITY`

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr)

133



Systèmes et réseaux temps réel
Chap VI: Gestion des ressources

Systèmes et réseaux temps réel
(youssefrochdi@yahoo.fr)

134

Objectifs

- Montrer les risques d'avoir des ressources partagées entre plusieurs tâches dans un contexte multitâches.
- Donner des mécanismes pour éviter ce genre de conflits.
- Expliquer l'intérêt des zones critiques et comment les utiliser.
- Expliquer l'intérêt de suspendre le scheduler et comment en tirer profit.

1- Conflits dûs aux ressources partagées (1/4)

- Quand une tâche 1, en exécution, accédant à une ressource (zone mémoire, périphérique, ...) est préemptée par une tâche 2, voulant accéder aussi à la même ressource, elle peut trouver celle-ci dans un état inconsistant...

Exemple 1: Accès au même périphérique

Deux tâches 1 et 2, **utilisant directement** le même LCD pour afficher deux messages : "Température: 45°C " et " Test 1 ok "

1. Tâche 1 s'exécute et affiche "Température", et Tâche 2 est préemptée par tâche2.
2. Tâche 2 s'exécute et affiche " Test 1 ok" , puis elle est préemptée par Tâche1
3. Tâche 1 continue son affichage, l'affichage sur LCD sera:
TempératureTest 1 ok:45°C

1- Conflits dûs aux ressources partagées (2/4)

Exemple 2: Opération non-atomique de type lecture/modification/écriture
L'instruction, en C, suivante:

```
PORTA=PORTA|masque
```

Est traduite par le compilateur C30 code assembleur suivant pour un
PIC24FJ128GA010:

```
MOV PORTA, W1  
MOV 0x1D10, W0  
IOR W1, W0, W0  
MOV W0, PORTA
```

1. Tâche 1 s'exécute et copie PORTA dans W1, puis elle est préemptée par Tâche 2.
2. Tâche 2 s'exécute et mets à jour PORTA avec une valeur V, ensuite elle est préemptée par tâche 1.
3. Tâche 1 continue l'exécution de l'instruction non atomique et écrase la valeur V mise par Tâche 2 dans PORTA

1- Conflits dûs aux ressources partagées (3/4)

Exemple 3: Opération non-atomique sur des variables de taille supérieure à la
taille naturelle des cases mémoires.

En C pour le PIC24FJ128GA010

```
long int Var1, Var2;  
Var2=Var1+0x10;
```

Sera traduit par le compilateur C30 en code assembleur

```
MOV [W14+4], W0  
MOV [W14+6], W1  
MOV #0x10, W2  
MOV #0x0, W3  
ADD W0, W2, [W14++]  
ADDC W1, W3, [W14--]
```

L'instruction se ramène a des opérations sur des mots de 16bits avec
un risque de préemption à tout moment, et un résultat inconsistant.

1- Conflits dûs aux ressources partagées (4/4)

Exemple 4: Utilisation d'une **Fonction non réentrante** par plusieurs tâches.

Une fonction est dite réentrante si elle n'accède à aucune donnée externe à un registre ou à la pile de la tâche qui l'utilise.

→ *Une commutation de contexte due à une préemption n'aura aucun effet sur la consistance des données manipulées par cette fonction.*

<pre>int masque; //variable globale void fct_test(void) //non réentrante { if (PORTA&masque) PORTA=PORTA&(~masque) ; else PORTA=PORTA masque; }</pre>	<pre>void vTask1(void * pv){ masque=0x03; fct_test(); ... } void vTask2(void * pv){ masque=0xF0; fct_test(); ... }</pre>
---	--

2- Exclusion mutuelle: Principe

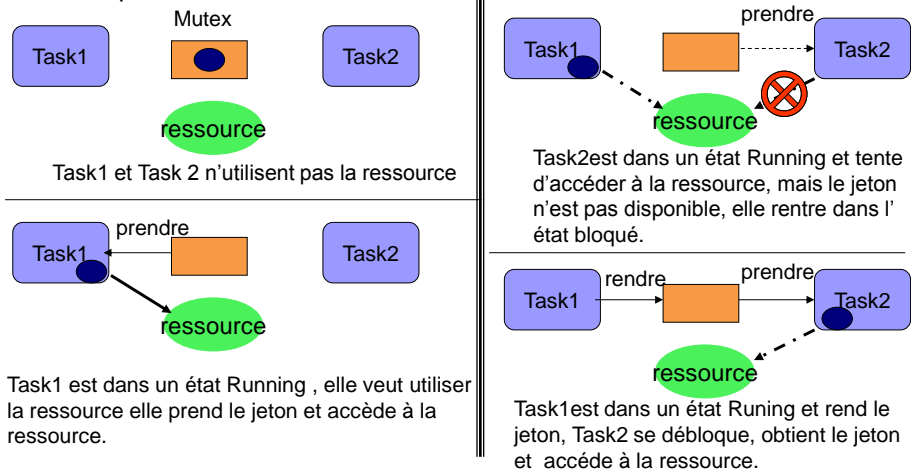
- Exclusion mutuelle, est une technique qui permet de régler les conflits, entre tâches ou entre tâches et ISR, dus aux ressources partagées, et de garantir la consistance des données.
- Principe:
 - Une fois elle a terminé avec une ressource, la tâche doit la libérer pour qu'elle puisse être utilisée par d'autres tâches ou ISR, en la laissant dans un état consistant.
 - Quand une tâche accède à une ressource partagée, elle en garde l'exclusivité de son utilisation.
- Problème: **comment garantir cette exclusivité sachant que les tâches peuvent être préemptées, et que les ISR peuvent interrompre les tâches ?**

3- Exclusion mutuelle: Mutexes

- Mutex(**M**utuel **E**xclusion) est un type de semaphore binaire qui permet de contrôler l'exclusivité de l'accès à une ressource partagée entre deux ou plusieurs tâches.
- Principe:
 - A chaque ressource partagée est affecté (**crée**) un MUTEX(semaphore binaire avec un seul jeton).
 - Pour qu'une tâche puisse accéder à cette ressource elle doit disposer du jeton. (**prendre le jeton**)
 - Quand une tâche accède à une ressource partagée, elle en garde l'exclusivité pour son utilisation.
 - Une fois terminée avec cette ressource la tâche doit la libérer pour les autres tâches (**rendre le jeton**).

4- Exclusion mutuelle: Mutexes

Principe:



5- Exclusion mutuelle: Création de Mutex

```
xSemaphoreHandle xSemaphore = NULL;

// A task that creates a semaphore.
void vATask( void * pvParameters )
{
    // Create the semaphore to guard a shared resource.
    vSemaphoreCreateBinary( xSemaphore );
}
```

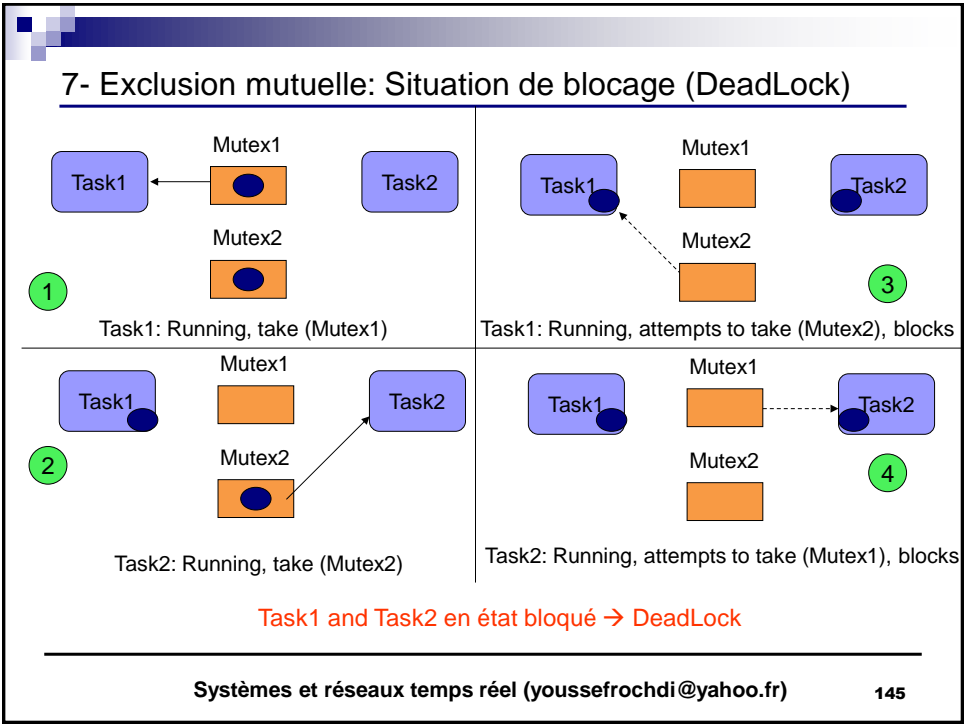
6- Exclusion mutuelle: Utilisation de Mutex

```
// A task that uses the semaphore.
void vAnotherTask( void * pvParameters )
{
    // ... Do other things.

    if( xSemaphore != NULL )
    {
        // See if we can obtain the semaphore. If the semaphore is not available
        // wait 10 ticks to see if it becomes free.
        if( xSemaphoreTake( xSemaphore, ( portTickType ) 10 ) == pdTRUE )
        {
            // We were able to obtain the semaphore and can now access the
            // shared resource.

            // ...

            // We have finished accessing the shared resource. Release the
            // semaphore.
            xSemaphoreGive( xSemaphore );
        }
        else
        {
            // We could not obtain the semaphore and can therefore not access
            // the shared resource safely.
        }
    }
}
```



8- Exclusion mutuelle: Sections critiques

Définition: Une section critique est une partie du code source encadré par des appels aux macros (définies dans portmacro.h) :

`taskENTER_CRITICAL()` et `taskEXIT_CRITICAL()`


Entre ces deux appels :

- La tâche en cours ne peut être préemptée par une autre tâche, (le scheduler étant en effet désactivé).
- La tâche en cours ne peut être interrompue que par des interruptions de niveau de priorité supérieur au niveau `configMAX_SYSCALL_INTERRUPT_PRIORITY` (s'il est défini)

Pour l'implémentation de freertos pour les PIC24F, ces deux macros sont définies comme suit:

```
#define portENTER_CRITICAL()    vPortEnterCritical()
#define portEXIT_CRITICAL()     vPortExitCritical()
```

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr) 146



8- Exclusion mutuelle: Sections critiques

`vPortEnterCritical()` et `vPortExitCritical()` sont définies dans `port.c`


```
void vPortEnterCritical( void )
{
    portDISABLE_INTERRUPTS();
    uxCriticalNesting++;
}

/*-----*/
void vPortExitCritical( void )
{
    uxCriticalNesting--;
    if( uxCriticalNesting == 0 )
    {
        portENABLE_INTERRUPTS();
    }
}

/* Critical section management. */
#define portINTEERRUPT_BITS ((unsigned portSHORT) configKERNEL_INTERRUPT_PRIORITY << (unsigned portSHORT) 5)

#define portDISABLE_INTERRUPTS()    SR |= portINTEERRUPT_BITS
#define portENABLE_INTERRUPTS()     SR &= ~portINTEERRUPT_BITS
```

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr) 147



8- Exclusion mutuelle: Sections critiques

- L'utilisation des sections critiques est un moyen pour réaliser l'exclusion mutuelle et éviter les conflits dûs aux ressources partagées.
- Toutefois les sections critiques ne doivent pas être trop longues au risque de rater bcp de ticks d'horloge et donc d'influencer la précision des délais et aussi d'influencer la réactivité de l'application.
- Pour certaines implémentation de freertos, seule certaines interruptions sont désactivées dans les sections critiques...
- Une autre manière d'implémenter les sections critiques est l'utilisation des fonctions:
`vSuspendAll()` et `xResumeAll()`
Cette méthode arrête le scheduler et empêche donc une commutation de tâche, mais n'empêche pas les ISR de s'exécuter !!!

Systèmes et réseaux temps réel (youssefrochdi@yahoo.fr) 148