## Question 4 (4 points) Synchronisation et communications entre tâches sous uC/OS-III

a) (2 pts) Le code suivant implante une pile partagée de profondeur *Size*. Les fonctions *push()* et *pop()* peuvent être appelées simultanément par plusieurs tâches. La fonction *push()* bloque jusqu'à ce qu'un espace soit libre dans la pile. De son côté, la fonction *pop()* bloque jusqu'à ce qu'une donnée soit disponible sur la pile. Finalement, les tâches doivent s'exclure entre elles. Utilisez les fonctions µCOS-III appropriées pour implanter ce comportement. Pour chaque étiquette (1 à 10), donnez le code. Il ne peut y avoir qu'un seul appel de fonctions par étiquette.

```
int stackPointer;
int stack[Size];

OS_Sem Empty, Full;
OS_MUTEX Mutex;
…

void push(int a)
{
   …
  1.  OSSemPend(&SemEmpty, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
  2.  OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err);

    stack[stackPointer] = a;
    stackPointer++;

  3.  OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);
  4.  OSSemPost(&SemFull, OS_OPT_POST_1, &err);
  …
}
int pop()
{
  …
  5.  OSSemPend(&SemFull, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
  6.  OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
    int tmp = stack[stackPointer];
    stackPointer--;
  7.  OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);
  8.  OSSemPost(&SemEmpty, OS_OPT_POST_1, &err);
    …
      return tmp;
}
main()
{
…
  9.  OSSemCreate(&SemFull, "SemFull", 0, &err);
  10. OSSemCreate(&SemEmpty, "SemEmpty", 0, &err);
  11. OSMutexCreate(&Mutex, "mutPrint", &err);
…
}
```

b) (2 pts) Le morceau de code suivant implante la lecture d'une donnée sur un périphérique fonctionnant par interruption. *globaltmp* est une variable globale partagée par plusieurs tâches et ISRs. La fonction *readdata* sert à lire la donnée sur le périphérique. Utilisez les fonctions µCOS-III pour implanter ce comportement. Pour chaque étiquette (1 à 7), donnez le code. Il ne peut y avoir qu'un seul appel de fonctions par étiquette. Ne vous souciez pas de la création des tâches ou des ressources.

```
OS_SEM  sem;

int globaltmp;
void main(void)
{
    …
    OSSemCreate(&Sem, "Sem", 0, &err);
    …
}

void Tache(void* p)
{
    int tmp;
    …

    while(1)
    {
     …
    1. OSSemPend(&Sem, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
    2. CPU_CRITICAL_ENTER()
          tmp = globaltmp;
    3. CPU_CRITICAL_EXIT()
    …

    }
}
```

```
void ISR(void)
{
        …
    4. OSSemPend(&Sem, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err);
    5. CPU_CRITICAL_ENTER()

       readdata(&globaltmp);

    6. CPU_CRITICAL_EXIT()
    7. OSSemPost(&Sem, OS_OPT_POST_1, &err);
        …
}
```

## Question 5 (6 points) Ordonnancement, blocage et inversion de priorité

Considérant l'ordonnancement basé sur RMA :

a) (2 pts) En utilisant la condition d'ordonnancement de *Liu* et *Layland* et la méthode d'analyse du pire cas d'exécution $R_i$ présentée en classe, que peut-on dire des tâches de la Table 5.1 du point de vue ordonnancement? N.B. ici on utilise aucun mutex.

| Tâches | Période | Temps d'exécution |
|---|---|---|
| T1 | 50 | 10 |
| T2 | 40 | 10 |
| T3 | 30 | 15 |

Table 5.1

On assigne selon RMA

$T_3 \rightarrow 30$ (+ prioritaire) $T_2 \rightarrow 40$ $T_1 \rightarrow 50$ (- prioritaire)

et on fait d'abord le test de Liu and Layland :

15/30 + 10/40 + 10/50 = .95 > .78 donc on ne peut rien dire

Passons au 2e test qui est celui de l'analyse du pire cas :

Toujours en assignat selon RMA

$T_3 \rightarrow 30$ (+ prioritaire) $T_2 \rightarrow 40$ $T_1 \rightarrow 50$ (- prioritaire)

L'équation de R dont on a besoin ici est la suivante :

$$W_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil C_j$$

$W_{3,0} = C_3 = 15 < 30 \rightarrow$ ok

$W_{2,0} = C_2 = 10$

$w2,1 = 10 + \left\lceil \frac{10}{30} \right\rceil * (15) = 25$

$w2,2 = 10 + \left\lceil \frac{25}{30} \right\rceil * (15) = 25 \leq T2 = 40 \rightarrow OK$

$W_{1,0} = C_1 = 10$

$$w1,1 = 10 + \left\lceil \frac{10}{30} \right\rceil * (15) + \left\lceil \frac{10}{40} \right\rceil * (10) = 35$$

$$w1,2 = 10 + \left\lceil \frac{35}{30} \right\rceil * (15) + \left\lceil \frac{35}{40} \right\rceil * (10) = 50$$

$$w1,3 = 10 + \left\lceil \frac{50}{30} \right\rceil * (15) + \left\lceil \frac{50}{40} \right\rceil * (10) = 60$$

$$w1,4 = 10 + \left\lceil \frac{60}{30} \right\rceil * (15) + \left\lceil \frac{60}{40} \right\rceil * (10) = 60 > T1 = 50 \rightarrow donc\ pas\ OK$$

Conclusion : si T1, T2 et T3 possèdent chacun une contrainte dure, on ne peut accepter cette assignation du RMA.

b) (2 pt) Un programme consiste en 5 tâches A, B, C, D et E (en ordre décroissant de priorité, A étant la plus prioritaire) et 6 ressources R1 à R6 qui sont protégées par des mutex réalisant le protocole héritage de priorité. Le temps d'accès maximal aux ressources a été calculé grâce à *CPU_TS_Get64()* donné dans le tableau suivant :

| R1 | R2 | R3 | R4 | R5 | R6 |
|---|---|---|---|---|---|
| 5 us | 150 us | 75 us | 300 us | 250 us | 175 us |

Table 5.2

Les ressources sont utilisées par les usagers de la façon suivante :

| Tâche | Ressource |
|---|---|
| A | R3 |
| B | R1, R2 |
| C | R3,R4,R5 |
| D | R1,R5,R6 |
| E | R2, R6 |

Table 5.3

Calculez le blocage maximal des tâches A à E.

$B_E = 0$ // Par définition aucun tache de priorité // inférieure donc 0

$B_D = \max(150) + \max(175) = 325$ us // doit tenir compte de R2 et R6 resp.

$B_C = \max(5) + \max(150) + \max(250) = 405$ us // doit tenir compte de R1, R2 et R5 resp.

$B_B = \max(5) + \max(150) + \max(75) = 230$ us // doit tenir compte de R1, R2 et R3 resp.

$B_A = \max(75) = 75$ us // doit tenir compte de R3

c) (2 pt) Considérez les 6 tâches suivantes T8 à T24 sous µC et la séquence d'exécution suivante:

| Tâche | Priorité | Nombre de *ticks* en attente avant de démarrer. | Nombre de *ticks* à exécuter | Séquence d'exécution |
|---|---|---|---|---|
| T8 | 8 | 11 | 5 | EAAAE |
| T10 | 10 | 8 | 4 | ECCE |
| T13 | 13 | 5 | 5 | EEAAA |
| T15 | 15 | 3 | 4 | EBBE |
| T18 | 18 | 2 | 3 | EEE |
| T24 | 24 | 0 | 10 | EBBBBBAAAE |

Table 5.4

À partir des spécifications de la Table 5.4, complétez la trace d'exécution de la Figure 5.1 de la page suivante en considérant le **mécanisme héritage de priorité** pour prévenir l'inversion de priorité. Indiquez les priorités de T13 et T24 directement sur le schéma de la fig. 5.2. Finalement indiquez la valeur du blocage de T18 i.e. $B_{T18}$ (vous pouvez aussi l'indiquez sur la Fig. 5.1).

N.B. Utilisez les symboles suivants pour compléter la figure 5.1 :

- *A? (or B? or C?) veut dire qu'une requête a été demandée (via OSMutexPend) pour accéder à la section critique, mais la section critique était occupée;*
- *A (or B or C) veut dire qu'une requête pour accéder à la section critique a été acceptée; et*
- *A' (or B' or C') indique que la tâche a terminé son exécution dans la section critique.*

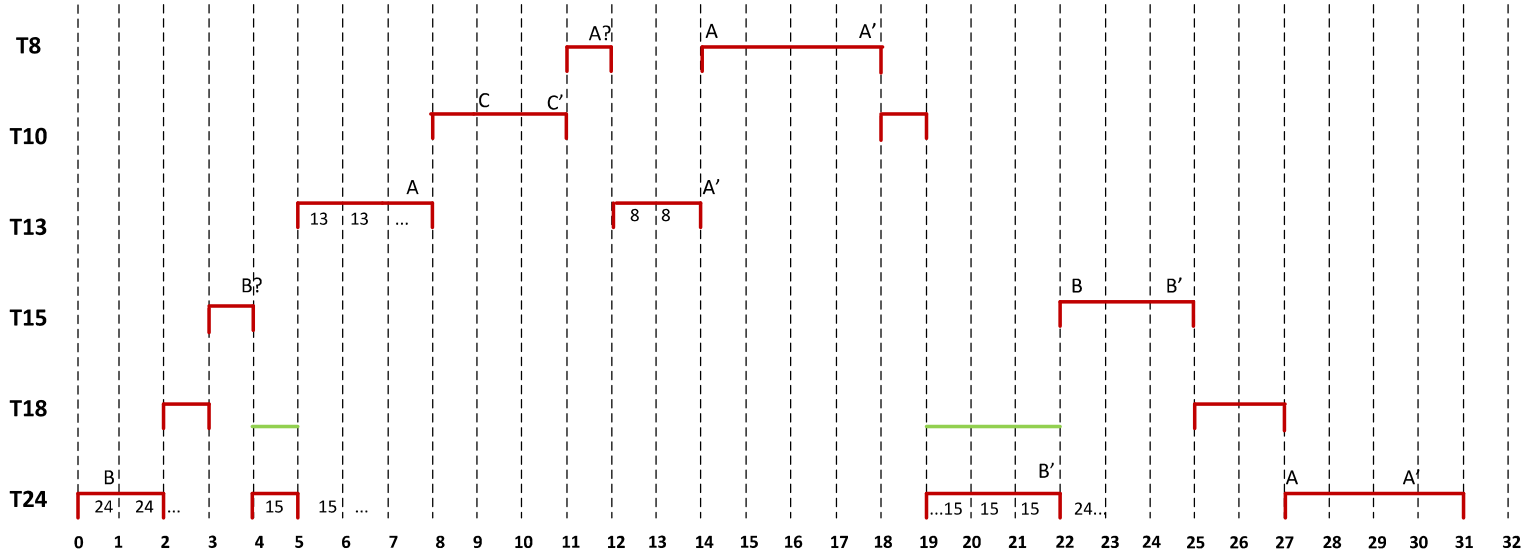



**Figure 5.2**

$B_{T18} = 4$ (voir lignes en verts)

n.b. ceci est le vrai blocage après exécution, mais on avait voulu le blocage pire cas sans exécution on aurait dit max(5) + max(3) = 8

# Annexe

## API uC/OS-III :

```
void  OSMutexCreate (OS_MUTEX  *p_mutex,
                     CPU_CHAR  *p_name,
                     OS_ERR    *p_err)
```

p_mutex
is a pointer to a mutex control block that must be allocated in the application. The user will need to declare a "global" variable as follows, and pass a pointer to this variable to OSMutexCreate():
OS_MUTEX MyMutex;

p_name
is a pointer to an ASCII string used to assign a name to the mutual exclusion semaphore. The name may be displayed by debuggers or µC/Probe.

p_err
is a pointer to a variable that is used to hold an error code

```
void  OSMutexPend (OS_MUTEX  *p_mutex,
                   OS_TICK   timeout,
                   OS_OPT    opt,
                   CPU_TS    *p_ts,
                   OS_ERR    *p_err)
```

timeout
specifies a timeout value (in clock ticks) and is used to allow the task to resume execution if the mutex is not signaled (i.e., posted to) within the specified timeout. A timeout value of 0 indicates that the task wants to wait forever for the mutex. The timeout value is not synchronized with the clock tick. The timeout count is decremented on the next clock tick, which could potentially occur immediately.

opt
determines whether the user wants to block if the mutex is not available or not. This argument must be set to either:
OS_OPT_PEND_BLOCKING, or
OS_OPT_PEND_NON_BLOCKING

```
void  OSMutexPost (OS_MUTEX  *p_mutex,
                   OS_OPT    opt,
                   OS_ERR    *p_err);
```

opt
OS_OPT_POST_NONE
No special option selected.
OS_OPT_POST_NO_SCHED
Do not call the scheduler after the post, therefore the caller is resumed even if the mutex was posted and tasks of higher priority are waiting for the mutex.
Use this option if the task calling OSMutexPost() will be doing additional posts, if the user does not want to reschedule until all is complete, and multiple posts should take effect simultaneously.

p_err
is a pointer to an error code

```
void  OSSemCreate (OS_SEM     *p_sem,
                   CPU_CHAR   *p_name,
                   OS_SEM_CTR  cnt,
                   OS_ERR     *p_err)
```

p_sem
: is a pointer to the semaphore control block. It is assumed that storage for the semaphore will be allocated in the application. In other words, you need to declare a “global” variable as follows, and pass a pointer to this variable to OSSemCreate():
OS_SEM MySem;

p_name
: is a pointer to an ASCII string used to assign a name to the semaphore. The name can be displayed by debuggers or µC/Probe.

cnt
: specifies the initial value of the semaphore.
If the semaphore is used for resource sharing, you would set the initial value of the semaphore to the number of identical resources guarded by the semaphore. If there is only one resource, the value should be set to 1 (this is called a binary semaphore). For multiple resources, set the value to the number of resources (this is called a counting semaphore).
If using a semaphore as a signaling mechanism, you should set the initial value to 0.

p_err
: is a pointer to a variable used to hold an error code

```
void  OSSemPend (OS_SEM   *p_sem,
                 OS_TICK   timeout,
                 OS_OPT    opt,
                 CPU_TS   *p_ts,
                 OS_ERR   *p_err)
```

timeout
: allows the task to resume execution if a semaphore is not posted within the specified number of clock ticks. A timeout value of 0 indicates that the task waits forever for the semaphore. The timeout value is not synchronized with the clock tick. The timeout count begins decrementing on the next clock tick, which could potentially occur immediately.

opt
: specifies whether the call is to block if the semaphore is not available, or not block.
OS_OPT_PEND_BLOCKING
: to block the caller until the semaphore is available or a timeout occurs.

OS_OPT_PEND_NON_BLOCKING
: If the semaphore is not available, OSSemPend() will not block but return to the caller with an appropriate error code.

p_err
: is a pointer to a variable used to hold an error code:
OS_ERR_NONE If the semaphore is available.
OS_ERR_PEND_WOULD_BLOCK if this function is called as specified OS_OPT_PEND_NON_BLOCKING, and the semaphore was not available.
OS_ERR_TIMEOUT If the semaphore is not signaled within the specified timeout.

```
OS_SEM_CTR  OSSemPost (OS_SEM  *p_sem,
                       OS_OPT   opt,
                       OS_ERR  *p_err)
```

opt
: determines the type of post performed.
OS_OPT_POST_1
Post and ready only the highest-priority task waiting on the semaphore.
OS_OPT_POST_ALL
Post to all tasks waiting on the semaphore. You should only use this option if the semaphore is used as a signaling mechanism and never when the semaphore is used to guard a shared resource. It does not make sense to tell all tasks that are sharing a resource that they can all access the resource.
OS_OPT_POST_NO_SCHED
This option indicates that the caller does not want the scheduler to be called after the post. This option can be used in combination with one of the two previous options.
You should use this option if the task (or ISR) calling OSSemPost() will be doing additional posting and, the user does not want to reschedule until all done, and multiple posts are to take effect simultaneously.

p_err
: is a pointer to an error code

```
void  OSFlagCreate (OS_FLAG_GRP  *p_grp,
                    CPU_CHAR     *p_name,
                    OS_FLAGS      flags,
                    OS_ERR       *p_err)
```

p_grp
: This is a pointer to an event flag group that must be allocated in the application. The user will need to declare a "global" variable as shown, and pass a pointer to this variable to OSFlagCreate():
OS_FLAG_GRP MyEventFlag;

p_name
: This is a pointer to an ASCII string used for the name of the event flag group. The name can be displayed by debuggers or by µC/Probe.

flags
: This contains the initial value of the flags to store in the event flag group. Typically, you would set all flags to 0 events correspond to set bits and all 1s if events correspond to cleared bits.

p_err
: is a pointer to an error code

```
OS_FLAGS  OSFlagPend (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS      flags,
                      OS_TICK       timeout,
                      OS_OPT        opt,
                      CPU_TS       *p_ts,
                      OS_ERR       *p_err)
```

p_grp
: This is a pointer to an event flag group that must be allocated in the application. The user will need to declare a "global" variable as shown, and pass a pointer to this variable to OSFlagCreate():
OS_FLAG_GRP MyEventFlag;

p_name
: This is a pointer to an ASCII string used for the name of the event flag group. The name can be displayed by debuggers or by µC/Probe.

flags
: This contains the initial value of the flags to store in the event flag group. Typically, you would set all flags to 0 events correspond to set bits and all 1s if events correspond to cleared bits.

p_grp

is a pointer to the event flag group.

flags
is a bit pattern indicating which bit(s) (i.e., flags) to check. The bits wanted are specified by setting the corresponding bits in flags. If the application wants to wait for bits 0 and 1 to be set, specify 0x03. The same applies if you'd want to wait for the same 2 bits to be cleared (you'd still specify which bits by passing 0x03).

timeout
allows the task to resume execution if the desired flag(s) is (are) not received from the event flag group within the specified number of clock ticks. A timeout value of 0 indicates that the task wants to wait forever for the flag(s). The timeout value is not synchronized with the clock tick. The timeout count begins decrementing on the next clock tick, which could potentially occur immediately.

opt
specifies whether all bits are to be set/cleared or any of the bits are to be set/cleared. Here are the options:
OS_OPT_PEND_FLAG_CLR_ALL check all bits in flags to be clear (0)
OS_OPT_PEND_FLAG_CLR_ANY check any bit in flags to be clear (0)
OS_OPT_PEND_FLAG_SET_ALL Check all bits in flags to be set (1)
OS_OPT_PEND_FLAG_SET_ANY Check any bit in flags to be set (1)
The caller may also specify whether the flags are consumed by "adding" OS_OPT_PEND_FLAG_CONSUME to the opt argument. For example, to wait for any flag in a group and then clear the flags that satisfy the condition, you would set opt to:
OS_OPT_PEND_FLAG_SET_ANY + OS_OPT_PEND_FLAG_CONSUME

p_err
is a pointer to an error code

```
OS_FLAGS  OSFlagPost (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS      flags,
                      OS_OPT        opt,
                      OS_ERR       *p_err)
```

p_grp
is a pointer to the event flag group.

flags
specifies which bits to be set or cleared. If opt is OS_OPT_POST_FLAG_SET, each bit that is set in flags will set the corresponding bit in the event flag group. For example to set bits 0, 4, and 5, you would set flags to 0x31 (note that bit 0 is the least significant bit). If opt is OS_OPT_POST_FLAG_CLR, each bit that is set in flags will clear the corresponding bit in the event flag group. For example to clear bits 0, 4, and 5, you would specify flags as 0x31 (again, bit 0 is the least significant bit).

opt
indicates whether the flags are set (OS_OPT_POST_FLAG_SET) or cleared (OS_OPT_POST_FLAG_CLR).
The caller may also "add" OS_OPT_POST_NO_SCHED so that µC/OS-III will not call the scheduler after the post.

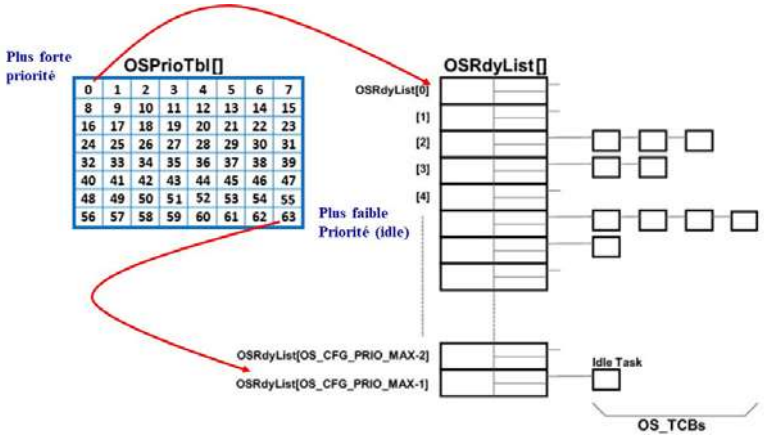
p_err
is a pointer to an error code

Finalement, la désactivation et la réactivation des interruptions se fait respectivement avec les macros *CPU_CRITICAL_ENTER()* et *CPU_CRITICAL_EXIT()*.

Bloc 4 :

```
OS_PRIO  OS_PrioGetHighest (void)
         {
             CPU_DATA  *p_tbl;
             OS_PRIO    prio;


             prio  = (OS_PRIO)0;
             p_tbl = &OSPrioTbl[0];
             while (*p_tbl == (CPU_DATA)0) {                (1)
                 prio += DEF_INT_CPU_NBR_BITS;              (2)
                 p_tbl++;
             }
             prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);     (3)
             return (prio);
         }
```




| Function | Description |
|---|---|
| `OS_PrioGetHighest()` | Find the highest priority level |
| `OS_PrioInsert()` | Set bit corresponding to priority level in the bitmap table |
| `OS_PrioRemove()` | Clear bit corresponding to priority level in the bitmap table |

**Table - Priority Level access functions**

| Function | Description |
|---|---|
| `OS_RdyListInit()` | Initialize the ready list to "empty" (see the figure below) |
| `OS_RdyListInsert()` | Insert a TCB into the ready list |
| `OS_RdyListInsertHead()` | Insert a TCB at the head of the list |
| `OS_RdyListInsertTail()` | Insert a TCB at the tail of the list |
| `OS_RdyListMoveHeadToTail()` | Move a TCB from the head to the tail of the list |
| `OS_RdyListRemove()` | Remove a TCB from the ready list |

**Table - Ready List access functions**

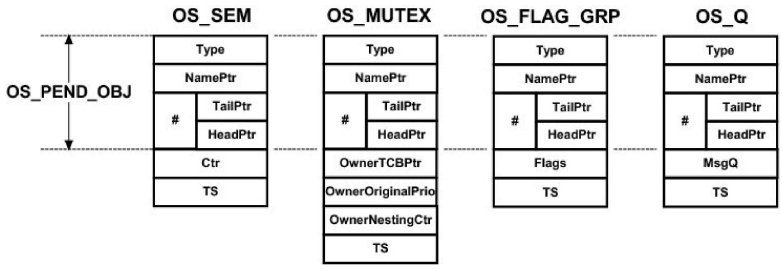



**Figure - OS_PEND_OBJ at the beginning of certain kernel objects**

| Function | Description |
|---|---|
| `OS_PendListChangePrio()` | Change the priority of a task in a pend list |
| `OS_PendListInit()` | Initialize a pend list |
| `OS_PendListInsertPrio()` | Insert a task in priority order in the pend list |
| `OS_PendListRemove()` | Remove a task from a pend list |

**Table - Pend List access functions**

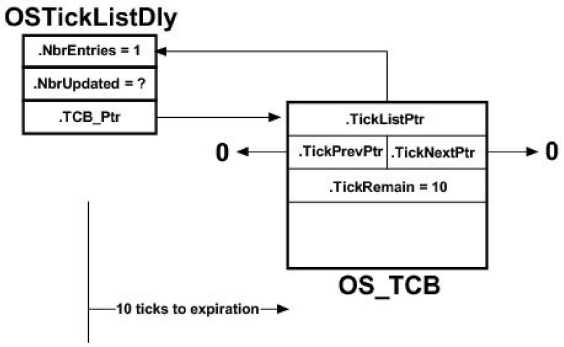
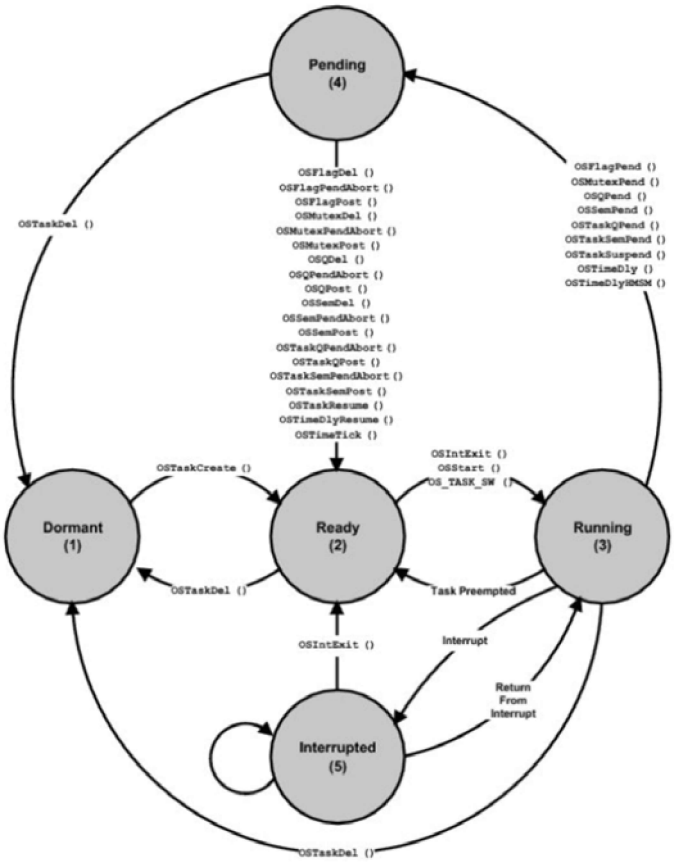



**Figure - Inserting a task in the delayed tasks tick list**




**États d'une tâche sous uC/OS-III**

Bloc3 :

## Héritage de priorité

- **Avec l'héritage de priorité on a:**

$$B_i = \sum_{k=1}^{K} usage(k,i)CS(k)$$

**Où:**

- ***usage*** **est une fonction 0/1:** ***usage(k,i)*** **= 1 si la ressource** ***k*** **est utilisé par au moins une tâche de priorité inférieure à** ***i*****, et au moins une tâche de priorité supérieure ou égale à** ***i*****, sinon 0.**
- **CS(*k*) le temps requis pour passer au travers la section critique *k*.**

- **Nomenclature qu'on va utiliser et qu'on retrouve dans la littérature:**

N = Nombre de tâches i dans le système (i varie donc de 1 à N)
$P_i$ = Priorité d'une tâche i (0 étant la plus prioritaire et 63 la moins prioritaire)
$T_i$ = Période d'execution d'une tâche i (ex. 1 tâche doit s'executer à toutes les 4 ticks d'horloges)
$I_i$ = Temps maximum pour lequel une tâche i est préemptée (ou interférée) par une (ou des) tâche(s) de plus grande priorité durant Ti.
$B_i$ = Temps maximum de blocage d'une tâche i durant sa période $T_i$ (e.g., tâche qui ne peut plus s'executer à cause qu'une (ou des) tâche(s) de moins grande priorité occupe une section critique requise par $T_i$ ou par une tâhe de priorité plus grande que $T_i$)
$C_i$ = Temps d'exécution maximum sur le CPU de la tâche i
U = Utilisation du CPU par les différentes tâches $T_i$ d'un système = $\sum_{i=1}^{N} C_i/Ti$
CS = Le temps maximum de changement de contexte d'une tâche i
$R_i$ = Temps d'exécution réel d'une tâche i durant sa période $T_i$
$R_i = C_i + I_i + B_i + CS$

Important: pour démontrer qu'une assignation est valide on devra avoir $R_i \leq T_i$ pour toutes les tâches i du système.

## Interférence

- **La formule de l'interférence nommé ω est donnée par:**

**représente la source d'interférences (préemptions) possibles qu'on nomme $I_i$**

$$w_i^0 = C_i$$

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

**où**
***hp*(*i*) est l'ensemble des tâches *j* plus prioritaire que *i* et qui font subir à *i* une préemption. On vient donc ajouter à $C_i$ la somme de toute les interférences (préemptions) possibles.**

$$\sum_{i=1}^{N} \left( \frac{C_i}{T_i} \right) < N(2^{1/N} - 1)$$

| N | Utilization Bound |
|---|---|
| 1 | 100.0% |
| 2 | 82.8% |
| 3 | 78.0% |
| 4 | 75.7% |
| 5 | 74.3% |
| 10 | 71.8% |