POLYTECHNIQUE MONTRÉAL

examen intra

INF3610

Sigle du cours

Identification de l'étudiant(e)		
Nom : Solution	**Prénom :**	
Signature :	**Matricule :**	**Groupe :**

Sigle et titre du cours	*Groupe*	*Trimestre*
INF3610 – Systèmes embarqués	**Tous**	**20243**
Professeur	***Local***	***Téléphone***
Guy Bois	**M-5105**	**5944**

Jour	*Date*	*Durée*	*Heures*
Jeudi	**29 octobre 2024**	**2h15**	**14h45 à 17h00**

Documentation	*Calculatrice*	
☒ Aucune	☐ Aucune	**Les cellulaires, agendas électroniques ou téléavertisseurs sont interdits.**
☐ Toute	☐ Toutes	
☐ Voir directives particulières	☒ Non programmable (AEP)	

Important

Cet examen contient **6** questions sur un total de **14** pages **(exluant cette page)**

- La pondération de cet examen est de **30** **%**
- Répondez directement sur le questionnaire pour les questions 3 et 4a
- Pour le reste, utilisez le cahier de réponse.

L'étudiant doit honorer l'engagement pris lors de la signature du code de conduit

Question 1 (2 points) En vrac, expliquez en quelques lignes la différence entre :

a) *OSIntCtwSw()* et *OSCtxSw()*

OSCtxSw est appelée lors d'une interruption logicielle alors que OSIntCtxSw est appelée

lors d'une interruption matérielle (ISR).

OSCtxSw retire la tâche en cours et la remplace par la nouvelle tâche trouvée par l'ordonnanceur.

OSntCtxSw ne fait que remplacer la tâche trouvée par l'ordonnanceur après l'exécution de ISR (la tâche avant l'ISR ayant été retirée par l'ISR lui-même en début de ISR).

b) *OSIntExit()* et *OSSched()*

OSIntExit() est appelé à la fin de ISR l'interruptions matérielle pour choisir la tâche la plus prioritaire (OSPrioGetHighest) et appelé OSIntCtxSw.

OSSched() est utilisé dans les interruptions logicielles lorsqu'une fonction uC bloque (OSTimeDly(), OSMutexPend(), etc.) et est mis en attente. OSSched() appelle

OSPrioGetHighest() et OSCtxSw.

Question 2 (4.5 points) Synchronisation de tâches et fonctions système

La figure 2.1, présente le code du no 26 des exercices sur laquelle j'ai ajouté 6 étiquettes (0 à 5) d'évènements. Les appels à *OSSemPend(S)* ont la forme *OSSemPend(&S,0, OS_OPT_PEND_BLOCKING, &ts, &err)* alors ceux de OSSemPost(S) ont la forme *OSSemPost(&S, OS_OPT_POST_1, &err).*

```
OS_SEM  S1, S2, S3;
S1 = OSSemCreate(0);    // Ici on simplifie les paramètres, on veut juste montrer que le
                        //sémaphore est initialisé à 0.
S2 = OSSemCreate(0);
S3 = OSSemCreate(0);

// Création des tâches
OSStart(); (0)
.
.
// Pour améliorer la lisibilité, on a simplifier les paramétres des appels de fcts
Void T1()                Void T2()                Void T3()
{                        {                        {
  while(1)                 while(1)                 while(1)
  {                        {                        {
    U(); (3)                 OSSemPend(S1); (1)       OSSemPend(S2); (2)
    OSSemPost(S1); (4)       V(); (5)                 W();
    OSSemPend(S3);           OSSemPost(S2);           OSSemPost(S3);
    X();                     OSSemPend(S1);           OSSemPend(S2);
    OSSemPost(S1);           Y();                     Z();
    OSSemPend(S3);           OSSemPost(S2);           OSSemPost(S3);
  }                        }                        }
}                        }                        }
```

Figure 2.1

a) (1 pt) En faisant abstraction pour le moment de ces 8 étiquettes, expliquez ce que représente cette trace en supposant que U(), V() et W(), X(), Y() et Z() sont 6 fonctions en C.

C'est un problème de synchronisation :

- *3 thread T1(), T2() et T3() : le thread T1() appelle les fonctions funcU() et funcX(), le thread T2() les fonctions funcV() et funcY() et le thread T3() les fonction funcW() et funcZ().*
- *6 fonctions funcU(), funcV(), funcW(), funcX(), funcY(), funcZ() qui doivent toujours être exécutées dans ce l'ordre U, V, W, X, Y, Z peu importe la priorité des threads.*

b) (.5 pt) Proposez une assignation de priorité pour chaque tâche (T1 à T3) selon uC/OS-III afin d'avoir la séquence dans l'ordre de 0 à 7.

P.e. T1 → 3 T2 → 1 T3 → 2

c) (2 pts) Pour chaque évènement et en vous aidant de l'annexe, donnez le détails des transitions 0 à 1, 1 à 2, 2 à 4 et 4 à 5 au niveau du noyau (états des tâches et appels de fonctions systèmes).

-1 pt si juste description et pas de détail et ensuite par bloc de .5 pt (p.e. gestion de la file du sémaphore)

0 à 1 :

- *T1 : passe de DORMANT à READY*
- *T2 : passe de DORMANT à READY*
- *T3 : passe de DORMANT à READY*
- *L'ordonnanceur est appelé et comme T2 a la plus haute priorité cette dernière passe de READY à EXECUTE*

1 à 2 :

- *Puisque S1 est à 0, la T2 passe de RUNNING à PENDING.*
- *La fonction système OS_PendListRemove() est exécutée et elle retire la T1 de OSPrioTbl (mise à 0).*
- *Puis T1 est mis dans la structure d'attente OS_PEND_OBJ de S1 (liste doublement chainée avec TailPtr et HeadPtr) avec l'appel à la fonction système OS_PendListInsertPrio().*
- *L'ordonnanceur OSSched() est appelé et la prochaine tâche la plus propriétaire, T3 passe de READY à RUNNING.*

2 à 4 :

- *Puisque S2 est à 0, la T3 passe de RUNNING à PENDING.*
- *La fonction système OS_PendListRemove() est exécutée et elle retire la T2 de OSPrioTbl (mise à 0).*
- *Puis T2 est mis dans la structure d'attente OS_PEND_OBJ de S2 (liste doublement chainée avec TailPtr et HeadPtr) avec l'appel à la fonction système OS_PendListInsertPrio().*
- *L'ordonnanceur OSSched() est appelé et la prochaine tâche la plus propriétaire, T1 passe de READY à RUNNING.*
- *U() est exécuté*

4 à 5

- *OSSemPost() appel la fonction système OS_PendListRemove() qui sélectionne OS_PEND_OBJ de S1 la tâche la plus prioritaire (pointeur sur HeadPtr) qui est T2.*
- *On exécute alors la fonction système OS_PrioInsert() qui remet T2 actif dans OSPrioTbl[] et passe de PENDING à READY.*

- *L'ordonnanceur OSSched() est appelé et la prochaine tâche la plus propriétaire, T2 passe de READY à RUNNING.*
- *V() est exécutée.*

d) (1 pt) Sachant que l'appel *OSSemPost(S1)* est remplacé par *OSSemPost(&S1, OS_OPT_POST_1 + OS_OPT_POST_NO_SCHED, &err)*, y aura-t-il des changements dans la transition 4 à 5? Justifiez.

OS_OPT_POST_NO_SCHED implique que de 4 à 5 4^{e} point, OSSched() n'est pas appelée, alors on va poursuivre avec l'appel à OSSemPost(S3) et on va bloquer comme de 1 à 2, puis on va revenir au point 5.

Question 3 (2 points) Toujours à partir du code de la figure 2.1 (question 2), on souhaite faire une implémentation mais en utilisant cette fois des Flag (*OSFlagCreate*, *OSFlagPend* et *OSFlagPost*). Complétez les *define* et la boucle *while* de T2 à la figure 3.1. Consultez l'Annexe au besoin.

```
// 3 define à compléter
#define S1  0x01
#define S2  0x02
#define S3  0x04

OS_FLAG_GRP FlagGroup1;
OSFlagCreate(&FlagGroup1, "Flag Group1", (OS_FLAGS)0, &err);

.
.
.
void Task2(void* data)
{
        OS_ERR err;
        CPU_TS ts;
        OS_FLAGS Flags;

        while(1)
        {

OSFlagPend(&FlagGroup1, S1, 0, OS_OPT_PEND_FLAG_SET_ALL +
OS_OPT_PEND_FLAG_CONSUME, &ts, &err);

                xil_printf("J'appelle V()\n");

OSFlagPost(&FlagGroup1, S2, OS_OPT_POST_FLAG_SET, &err);
OSFlagPend(&FlagGroup1, S1, 0, OS_OPT_PEND_FLAG_SET_ALL +
OS_OPT_PEND_FLAG_CONSUME, &ts, &err);

                xil_printf("J'appelle Y()\n");

OSFlagPost(&FlagGroup1, S2, OS_OPT_POST_FLAG_SET, &err);

        }
}
```

Figure 3.1

Question 4 (4 points) Héritage de priorités, ICPP et blocage.

a) Soit la trace de la figure 4.1, entre 4 tâches uC/OS-III de priorités 10, 20, 30 et 40. Sur cette même figure, aucun mécanisme pour minimiser l'inversion de priorité n'est utilisé pour les mutex A et B.

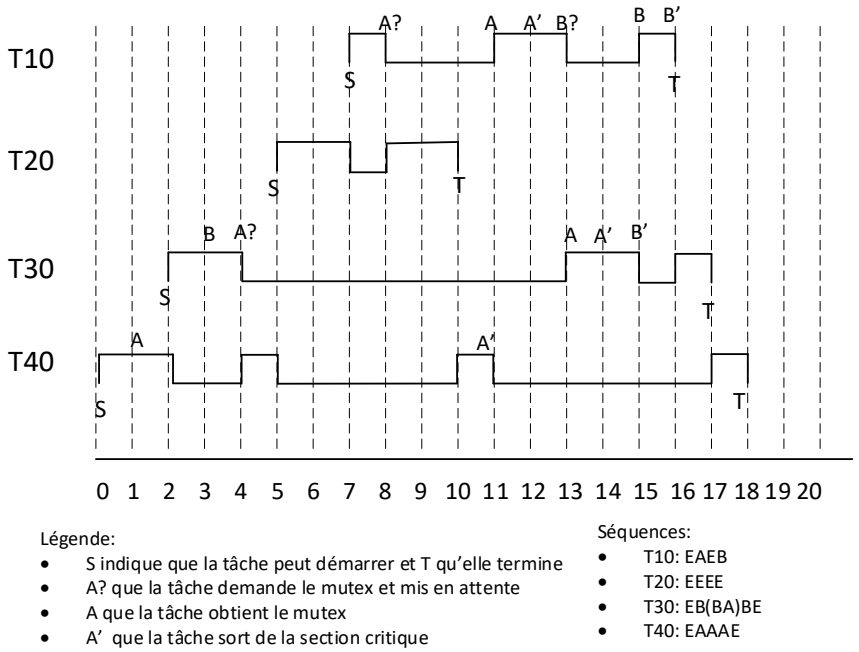


Figure 4.1

a) (3 pts) À partir de la figure 4.1, complétez les traces des figures 4.2 et 4.3 (page suivante) qui considèrent un mécanisme d'héritage de priorité et de ICPP, respectivement.
N.B. Vous pouvez décrochez la page suivante pour mieux suivre avec fig 4.1, mais n'oubliez pas de mettre votre nom et matricule.

b) (1 pt) Donnez les temps de blocage de T20 nommé B_{T20} obtenue sur les fig. 4.2 et 4.3 (surlignez en gras) et comparez avec le B_{T20} théorique.

Sur fig. 4.1 B_{T20} = 3 et sur fig. 4.2 = 3

Sur fig. 4.1 B_{T20} théorique = 3 + 1 + 3 = 7 et sur fig. 4.2 B_{T20} théorique = max(3,1,3) =3

Nom : **Matricule :**

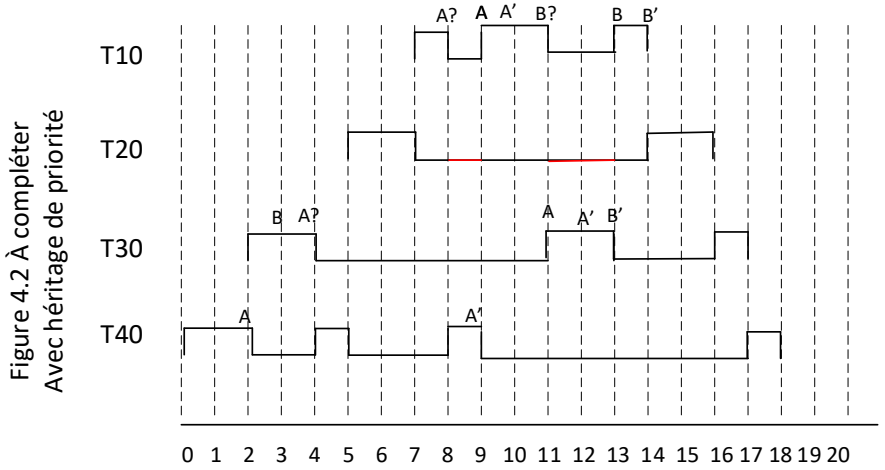


Figure 4.2 À compléter
Avec héritage de priorité

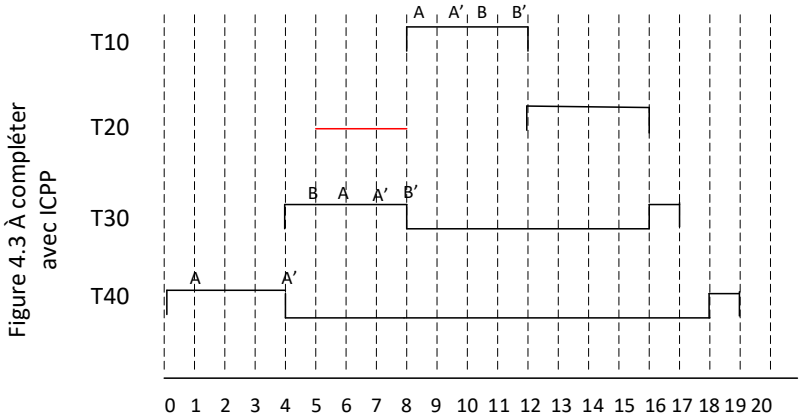


Figure 4.3 À compléter
avec ICPP

Question 5 (2.5 points) Délai et tâche périodique

a) (1 pt) On veut implémenter une tâche i de période Ti = 45 (période) et Ci = 23 (temps d'exécution maximum). On a un tick rate de 1000Hz (1 ms) et **WAITFORTICKS = 23**. Soit le code suivant:

```
while(1) {
  OSTimeDly(45,OS_OPT_TIME_DLY, &err);      //délai absolu
  actualticks = OSTimeGet(&err);
  while(WAITFORTICKS + actualticks > OSTimeGet(&err));
}
```

Va-t-on rencontrer notre spécification qui est celle de pouvoir démarrer une nouvelle période à 0, 45, 90, 135, etc. ticks. Justifiez.

b) (1 pt) On veut implémenter une tâche i de période Ti = 45 (période) et Ci = 23 (temps d'exécution maximum). On a un tick rate de 1000Hz (1 ms) et **WAITFORTICKS = 23**. Soit le code suivant:

```
while(1) {
  OSTimeDly(45,OS_OPT_TIME_PERIODIC, &err);      // délai périodique
  actualticks = OSTimeGet(&err);
  while(WAITFORTICKS + actualticks > OSTimeGet(&err));
}
```

Va-t-on rencontrer notre spécification qui est celle de pouvoir démarrer une nouvelle période à 0, 45, 90, 135, etc. ticks. Justifiez.

c) (.5 pt) Un des 2 appels à OSTimeDly ci-haut mémorise et utilise le champs TickCtrPrev du TCB de la tâche qui fait l'appel. Est-ce pour a) ou pour b). Justifiez.

a) *La réponse est Non*
J'ai accepté 2 réponses vu la confusion avec l'usage de l'option OS_OPT_TIME_DLY et le commentaire délai absolu (qui contredisait l'usage de l'option) :
Délai relatif : accumule les temps d'exécution (ici 23) donc on va faire 0, 68, 136, 204, etc. donc seul le premier fonctionne.
Délai absolu : on reste à 45 donc seul le premier fonctionne.

b) *La réponse est Oui*
Ici uC/OS-III ne tient pas compte des délais d'exécution mais seulement des T_i (45).

c) *A chaque appel de OSTimeDly(45, OS_OPT_TIME_PERIODIC, &err);, on enregistre la Valeur «OSTickCtr» dans OS_TCB.TickCtrPrev* ***juste avant le waiting*** *de OSTimeDly de tâche i (au moment de running à pending).*
La prochaine fois que OSTimeDly () est appelé, 2 possibilités:
   i. *uC/OS-III utilise* ***l'option délai absolu*** *avec un délai de OS_TCB.TickCtrPrev + 45.*
   ii. *uC/OS-III utilise l'option* ***délai relatif*** *avec un délai de 45 – (OStimeGet() - TCB.TickCtrPrev)). P.e. après la 1ere période : 45 – (23 – 0) = 22*

Question 6 (5 points) Laboratoire no 1 partie 1

On a un *Super ARM* sur une nouvelle carte *Super Pynq Z2* qui roule à 3 GHz (au lieu de 650 MHz). Cela permet à uc/OS-III de supporter un tick (OS_CFG_TICK_RATE_HZ) à 30 KHz = 30 000 Hz (donc une période 33.33 us/tick).

On roule sur *Super ARM* un système similaire à celui du lab 1 partie 1, avec les éléments suivants :

- 1 interface de réception connectée à l'Ethernet qui est implémenté avec un ISR (*ISR_input_packet*),
- 1 tâche *TaskQueueing* qui reçoit dans son fifo les paquets du ISR et les prétraite (décodage du paquet, CRC, etc.), puis les transfert à 3 fifos,
- 3 tâches (*TaskComputing*) de priorités différentes assignées chacune à un fifo qui vérifie la validation de la destination et envoie à un port de sortie du routeur via les tâches TaskOutputPort et finalement,
- 64 tâches TaskOutputPort de profondeur 1 qui dirige le paquet vers le bonne interface.

L'acquisition des paquets se fait de manière discontinue par rafale: une rafale dure 500 ms avec un débit d'entrée de 2.5 M (million) paquets/s (en supposant 16 mots de 32 bits par paquets ça fait 512 bps/paquet). Suite à une rafale, on a 500 ms pour vider les fifos, puis on recommence une nouvelle rafale chaque seconde.

Au total 900 instructions sont requises pour le traitement de 1 paquet. La table suivante donne le nombre d'instructions moyen par paquet requis par tâche du système incluant le ISR.

ISR_input_packet	200 instructions
TaskQueueing	200 instructions
TaskComputing	300 instructions
TaskOutputPort	200 instructions

Tableau 6.1

a) (.5 pts) Est-ce réaliste de penser que l'on peut avoir OS_CFG_TICK_RATE_HZ = 30 000 Hz sur un processeur 3 GHz? Justifiez.
b) (1 pt) On suppose qu'on a 4 cœurs sur *Super ARM* mais on se concentre ici sur 1 cœur et chaque cœur exécute 1 instruction par cycle d'horloge à 3GHz. Combien d'instructions peut-on en moyenne exécuter à chaque tick d'horloge sur 1 cœur? Justifiez.
c) (2 pts) Toujours pour 1 cœur, calculez le nombre de ticks et le pourcentage de temps CPU occupé par le ISR, par *TaskQueuing* et *TaskComputing et TaskOutputPort*. Reste-il du temps sur le cœur pour les autres interruptions et les changements de contexte[1]?

[1] N'était pas demandé mais on pouvait le déduire du temps CPU puisque similaire à TaskQueuing

(suite page suivante)

d) (1.5 pts) Déterminez la taille minimum des fifos de *TaskQueueing* et *TaskComputing* pour assurer le bon fonctionnement du système. On suppose que la priorité est la même que celle du lab 1 partie 1 et que la distribution des queues *highQ*, *mediumQ* et *lowQ* est uniforme ; en moyenne de 1/3, 1/3 et 1/3.

Solution

a) Tout à fait, on dit que le rapport fréquence d'horloge du CPU vs quantum (tick) du OS doit être de 100,000 à 1,000,000. Ici on a 3 GHz = 3 000 000 000 / 30 000 Hz = 100,000.

b) C'est en fait la réponse de a) mais en raisonnant avec des périodes. Si on développe :

- *Un processeur à 3 GHz effectue 3 milliards (3 000 000 000) de cycles par secondes donc une période de $0.000\overline{3}$*
- *1 quantum de 30 000 Hz correspond à une période de $33.\overline{3}$ us*
- *Donc $33.\overline{3}$/ $0.000\overline{3}$ = 100,000 instructions*

*c) 1 ISR demande 200 instructions, si je dois faire 2.5 M de ISR ça fait 500 M d'instructions. Or chaque tick fait 100,000 instructions donc **5000 ticks** sont requis pendant ½ seconde. Or j' ai 15000 ticks (30000/2) à ma disposition pour ½ secondes donc OK*

*TaskQueuing demande 200 instructions si je dois faire 2.5 M de paquets ça fait 500 M d'instructions. Or chaque tick fait 100,000 instructions donc **5000 ticks** sont requis pendant 1 seconde. Or en enlevant ceux du ISR, j'ai 25000 ticks à ma disposition donc OK*

*TaskComputing demande 300 instructions si je dois faire 2.5 M de paquets ça fait 750 M d'instructions. Or chaque tick fait 100,000 instructions donc **7500 ticks** sont requis pendant 1 seconde. Or en enlevant ceux de ISR et TaskQueueing, j'ai 20,000 à ma disposition donc OK*

*TaskOutputPort demande 200 instructions si je dois faire 2.5 M de paquets ça fait 500 M d'instructions. Or chaque tick fait 100,000 instructions donc **5000 ticks** sont requis pendant 1 seconde. Si j'enlève ISR, TaskQueueing et TaskComputing, j'ai 12,500 à ma disposition donc OK*

*Il en reste 7500 or 7500/30000 = **25% du CPU on est OK***

N.B. J'ai aussi accepté 1.25M de traitement de paquets pour 1 seconde soit la moitié. On divise donc par 2 les résultats précédents.

d) *Pas besoin pour TaskQueueing car tout traiter durant ½ sec entre les ISRs on the fly.*

Pour taskQueuing, on peut traiter 5000 durant ½ sec entre les ISRs. Pour TaskComputing, on peut pas vraiment tout faire entre ISRs et TaskQueuing (seulement 2500) donc besoin d'un fifo de au moins 5000 x 2 et on peut ensuite diviser le tout par 3.
*Mais, souvenez-vous que dans le lab TaskOutputPort est plus prioritaire que TaskQueueing et TaskComputing. Donc il est possible que TaskOutputport prenne du temps dans la 1ere ½ seconde. Pour cela j'aurais tendance à mettre 7500*2 pour TaskComputing. Mais j'acceptais les 2.*

N.B. Si vous avez pris 1.25 M de paquets en c), on n'a pas besoin de fifo car tout peut se faire en ½ seconde.

Annexe

API uC/OS-III :

```
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).

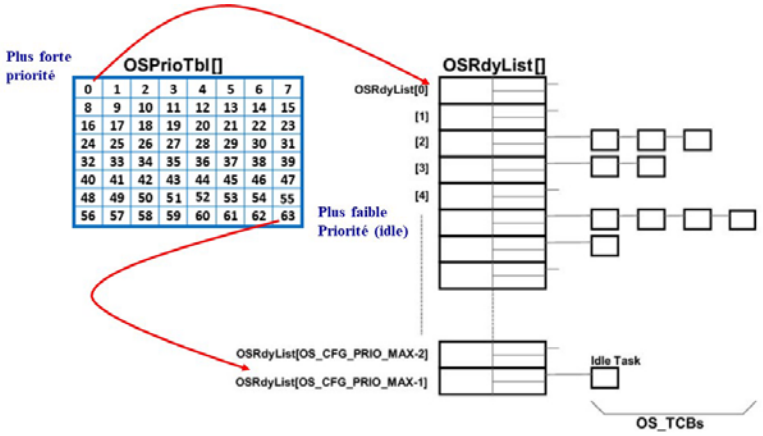
opt
indicates whether the flags are set (OS_OPT_POST_FLAG_SET) or cleared (OS_OPT_POST_FLAG_CLR).
The caller may also "add" OS_OPT_POST_NO_SCHED so that µC/OS-III will not call the scheduler after the post.

p_err
is a pointer to an error code

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

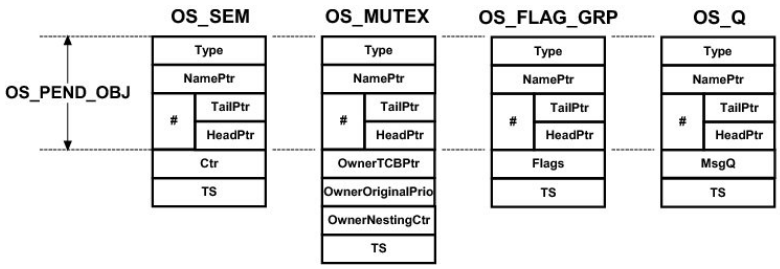
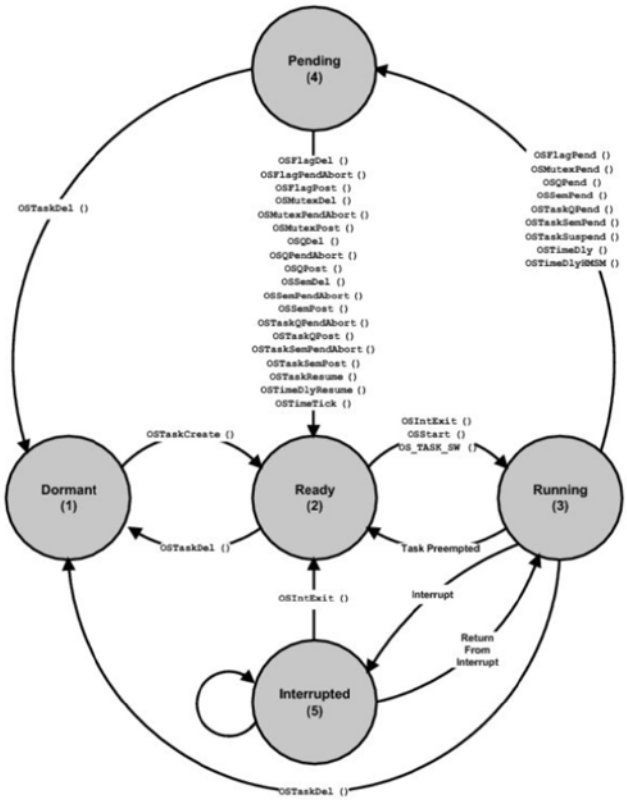


Figure - OS_PEND_OBJ at the beginning of certain kernel objects

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



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*.**

Blocage de ICPP

- **Avec ICPP on a:**

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

Où:

1. **usage est une fonction 0/1: usage(k,1) = 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.**
2. **Lorsque qu'on a plusieurs usage(k,i) = 1 pour un même k, on prend le maximum**
3. **CS(k) le temps maximum requis pour passer au travers la section critique k.**
4. **K étant le nombre total de ressources (mutex)**

- **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.