POLYTECHNIQUE MONTRÉAL

examen intra

INF3610

Sigle du cours

Identification de l'étudiant(e)		
Nom :	**Prénom :**	
Signature :	**Matricule :**	**Groupe : 1**

Sigle et titre du cours		*Groupe*	*Trimestre*
INF3610 – Systèmes embarqués		**Tous**	**202203**
Professeur		*Local*	*Téléphone*
Guy Bois		**M-5105**	**5944**
Jour	*Date*	*Durée*	*Heures*
Lundi	**24 octobre**	**2h30**	**15h00 à 17h30**

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 **15** pages

- La pondération de cet examen est de **30** **%**
- Répondez directement sur le questionnaire pour le no 2. Pour le reste répondre dans le cahier de réponse.
- Remettre le cahier de réponse et le questionnaire.
- Ne pas écrire en rouge

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

Question 1 (4 points) En vrac : vrai ou faux avec <u>justification</u>

a) (.5 pt) Dans les centrales nucléaires, le temps de réaction à un évènement (qui peut se traduire par une latence d'interruption) est généralement de l'ordre des millisecondes. Linux peut donc être utilisé car il ne s'agit pas d'un système temps réel.

Faux car on a besoin de déterminisme et contrainte dure

b) (.5 pt) Soit un convertisseur analogique-numérique qui fonctionne en mode périodique de manière très précise : un résultat de conversion est disponible à toutes les 1 ms exactement. L'utilisation d'un *watchdog* en uC/OS-III plutôt qu'une interruption externe est alors tout aussi efficace et ce peu importe la charge du système (c.-à-d. le nombre de tâches).

Faux car le watchdog de uC/OS-III utilise une tache lors du déclenchement de la fonction de call back. Or la priorité d'une tâches est toujours inférieure à celle d'un ISR.

c) (.5 pt) On put utiliser un mutex uC/OS-III pour faire une synchronisation pourvu que la valeur initiale demandée du sémaphore soit de 1.

Faux on ne peut séparer OSMutexPend() et OSMutexPost, i.e. fonctionne par pair

d) (.5 pt) Il est préférable de demander à une tâche de se supprimer elle-même plutôt que de la supprimer directement.

Vrai ça permet d'éviter les situation de deadlock si la tâche est en plein milieu d'un mutex, elle sort de la section critique et se termine ensuite.

e) (.5 pt) Une *pile d'exécution* contient des données d'une tâche alors que le TCB (Task Control Block) sert à enregistrer des informations au sujet des tâches actives dans un programme.

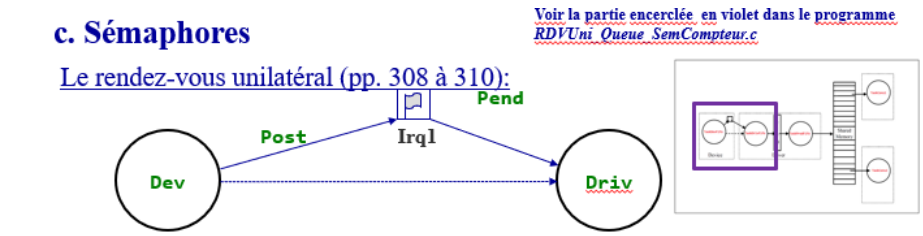
Faux c'est l'inverse

f) (.5 pt) Un des désavantages de uC/OS-III est d'être limité à 64 tâches.

Faux, les structure de données de uC/OS-III (e.g. OSPrioTbl) sont extensibles à 128, 256 et même 512 tâches.

(suite page suivante)

g) (1 pt) Dans le code de la figure suivante vue en classe, si au lieu d'utiliser un sémaphore pour réaliser le rendez-vous entre *TaskDevFifo* et *TaskDrivFifo* on utilisait le flag de uC/OS-III, il ne serait alors plus nécessaire d'avoir une fonction dont le rôle est d'empêcher *TaskDrivFifo* de lire plusieurs fois une même valeur (c'est-à-dire équivalent à *OSSemPend(&Irq1, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err)*.




```
void TaskDevFifo (void *data){
   int i;
   unsigned short value = 0;
   OS_ERR err;
   CPU_TS ts;
   data = data;

   while(1){
      xil_printf("\Debut du stream !\n");
      for(i=0; i<255; i++) {
         OSTimeDly(1, OS_OPT_TIME_PERIODIC, &err);
         OSSemPend(&Irq1, 0,
         OS_OPT_PEND_NON_BLOCKING, &ts, &err);
         reg1 = value++;
         OSSemPost(&Irq1, OS_OPT_POST_NO_SCHED, &err);
      }
      xil_printf("\nFin du stream !\n");
      OSTimeDlyHMSM(0, 0, 0, 800,
OS_OPT_TIME_HMSM_STRICT, &err);  // Delay for 800 millisecond
   }
}
```

```
void  TaskDrivFifo (void *data){
   OS_ERR err;
   CPU_TS ts;
   data = data;
   while (1) {
      OSSemPend(&Irq1, 0,
      OS_OPT_PEND_BLOCKING, &ts, &err);
      Msg *msg = malloc(sizeof(msg));
      msg->value = reg;
      msg->device = 1;
      OSQPost(&Fifo, msg, sizeof(msg),
    OS_OPT_POST_FIFO, &err);
      xil_printf("\nValeur produite: %d\n",
reg);
      if (err == OS_ERR_Q_MAX || err ==
OS_ERR_MSG_POOL_EMPTY) {
         xil_printf("\nQueue pleine !\n");
       }
   }
}
```

Figure 1.1 pour la question 1g.

Vrai car par définition si dans le OSFlagPost de TaskDevFifo on fait un OS_OPT_POST_FLAG_SET on met le bit à 1 et si on le refait plus tard une 2e fois (sans qu'il y est eu consommation de TaskDrivFifo avec OSFlagPend) le bit reste 1. Autrement dit c'est comme si on avait mod 1. N.B. Il ne faut pas oublier de faire un consume dans le OSFlagPend…

En résumé:

- *Paramètre du côté du producteur TaskDevFifo :*

OS_OPT_POST_FLAG_SET_ALL

- *Paramètre du côté du consommateur TaskDrivFifo :*

OS_OPT_PEND_FLAG_SET_ALL + OS_OPT_PEND_BLOCKING + OS_OPT_PEND_FLAG_CONSUME

Question 2 (4 points) Trace d'exécution avec mécanismes pour contrer l'inversion de priorité

Considérez les 6 tâches suivantes T1 à T6 sous µC/OS-III (Figure 2.1). La séquence d'exécution peut être interprétée comme suit:

Tâche	Priorité	Nombre de *ticks* en attente avant de démarrer.	Nombre de *ticks* à exécuter	Séquence d'exécution
T6	8	11	5	EAAAE
T5	10	8	4	ECCE
T4	13	5	5	EEBBB
T3	15	3	4	EBBE
T2	18	2	3	EEE
T1	24	0	10	EAAAAABBBE

Figure 2.1

À partir des spécifications de la Figure 2.1 :

a) (2 pts) Complétez la trace d'exécution de la Figure 2.2a (page 4) en considérant le mécanisme héritage de priorité pour prévenir l'inversion de priorité :

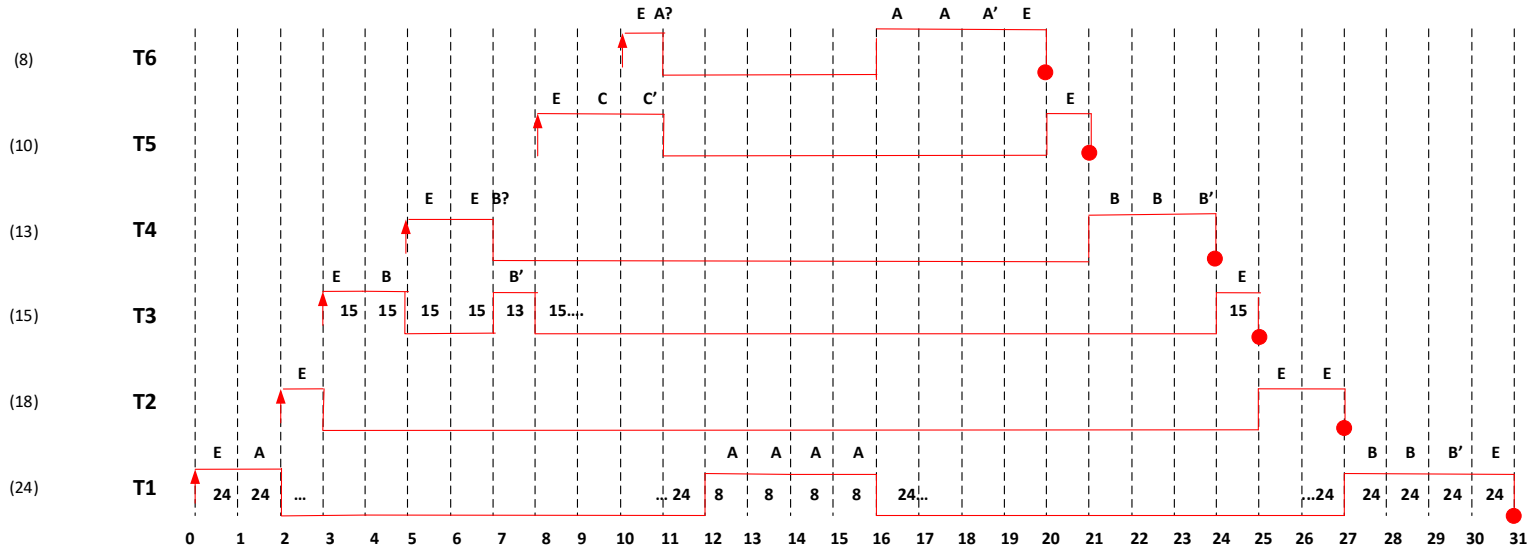
   i. Indiquez également la priorité de T1 et T3 (juste en dessous de T1 et T3 sur la fig. 2.2a) pour chaque tick d'horloge d'exécution.
   iii. Remplissez la table OSPrioTbl (Fig 2.2b) juste après le tick 12 en indiquant l'état de chaque tâche.

b) (2 pts) Complétez la trace d'exécution de la Figure 2.3a (page 4) en considérant le mécanisme ICPP pour prévenir l'inversion de priorité :

   i. Indiquez également la priorité de T1 (juste en dessous de T1 sur la fig. 2.2b) pour chaque tick d'horloge d'exécution.
   iii. Remplissez la table OSPrioTbl (Fig 2.3b) juste après le tick 12 en indiquant l'état de chaque tâche.

N.B. Utilisez les symboles suivants pour compléter les figures 2.2 et 2.3 :

- *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.*

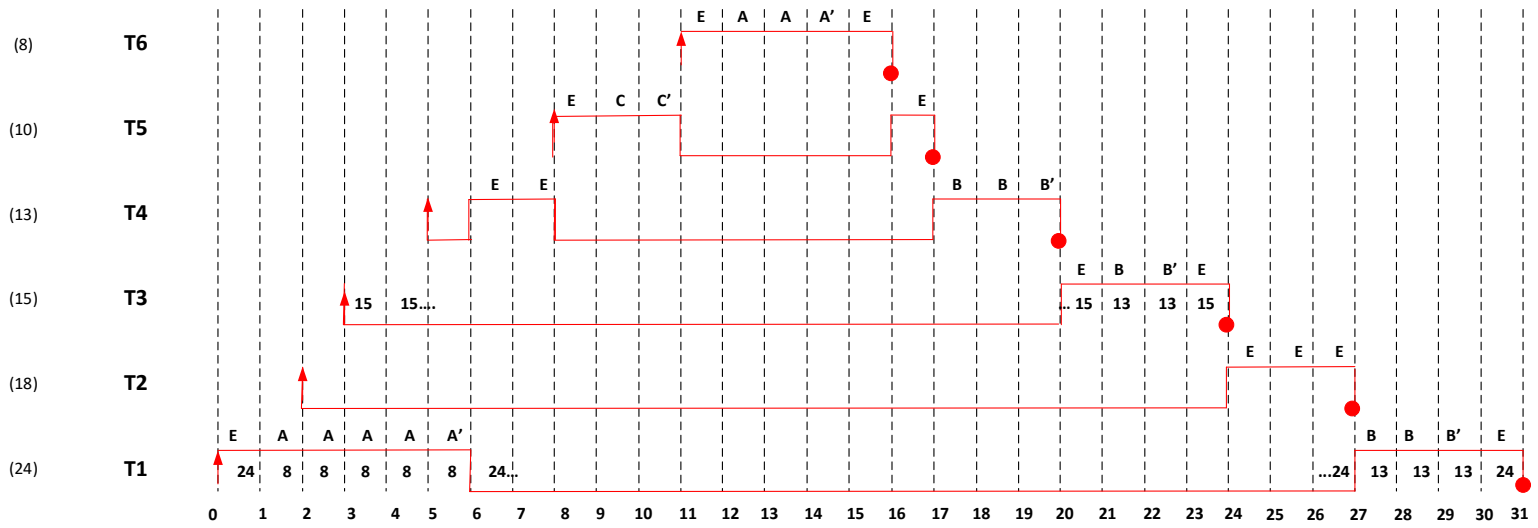


a)

	0	1	2	3	4	5	6	7
0								
1	1		1			1		1
2			1					
3								
4								
5								
6								
7								1

b)

Figure 2.2



a)

	0	1	2	3	4	5	6	7
0								
1	1		1			1		1
2			1					
3	1							
4								
5								
6								
7								1

b)

Figure 2.3

Question 3 (2 points) Temps de blocage

Sachant que nous avons maintenant deux protocoles pour contrer l'inversion de priorité avec uC/OS-III : ICPP et héritage de priorité. Soit 4 tâches T1, T2, T3 et T4 partageant 5 ressources critiques A, B, C, D et E chacune gérée par 5 mutex. Le tableau 3.1 indique pour chaque ressource, le temps d'exécution maximal demandé par chaque tâche (un 0 indique que la ressource n'est pas utilisée par la tâche).

	Priorité	A	B	C	D	E
T1	1	12	5	9	8	0
T2	2	10	0	7	0	6
T3	3	0	3	0	7	13
T4	4	10	0	8	0	5

Tableau 3.1

a) (1 pt) En considérant le protocole héritage de priorité, estimez une borne supérieure sur le temps de blocage maximum de la tâche T1. Expliquez clairement vos calculs.

max(10, 10) + max(3) +max(7,8) + max(7) = 28

b) (1 pt) En considérant le protocole ICPP, estimez une borne supérieure sur le temps de blocage maximum de la tâche T1. Expliquez clairement vos calculs.

max(max(10, 10), 3, max(7,8), 7) = 10

Question 4 (4 points) Gestion des évènements

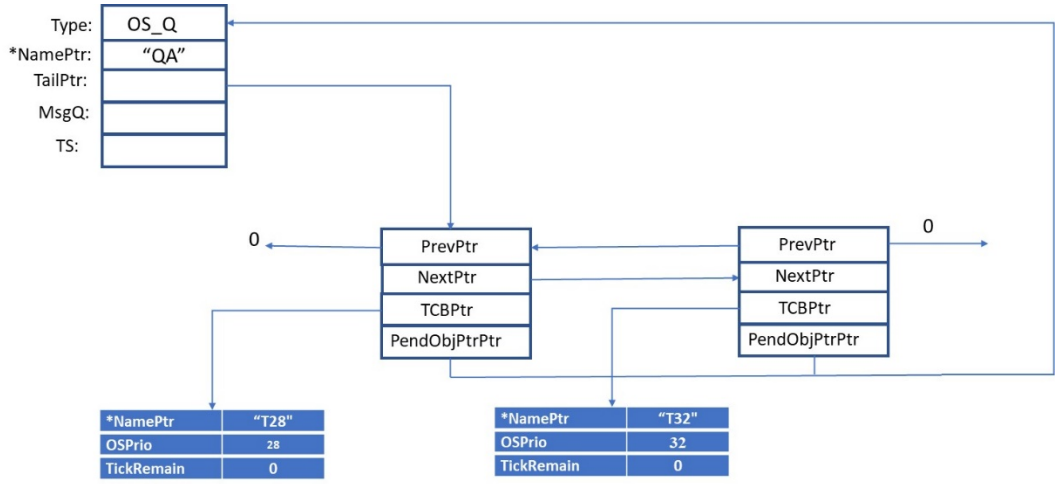
Soit l'état d'un système sous uC/OS-III à la figure 4.1 au tick no 5.

a) (**.5 pt**) Décrivez l'état du système à la Figure 4.1, c'est-à-dire les différentes tâches en cours (sous forme Ti) et leur état.

b) (**.5 pt**) Expliquez comment la tâche la plus prioritaire à être exécutée est déterminée. Expliquer votre démarche.

c) (**1.5 pt**) On suppose que la tâche trouvée en b) fait appel à *OSQPend(&QA, 1, OS_OPT_PEND_BLOCKING, &msg_size, &ts, &err)* au tick no 6. Indiquez ce qui s'est passée sur la figure 4.1 suite à cet appel.

d) (**1.5 pt**) Nous sommes au début du tick no 7, la tâche la plus prioritaire trouvée en c) est-elle toujours la plus prioritaire? Indiquez ce qui s'est passé dans un ordre chronologique depuis l'évènement survenu en c).



	0	1	2	3	4	5	6	7
[0]	0	0	0	0	0	0	0	0
[1]	0	0	0	1	0	0	0	0
[2]	0	0	0	0	0	0	0	0
[3]	0	0	0	1	0	1	0	0
[4]	0	0	0	0	0	0	0	0
[5]	0	0	0	0	0	0	0	0
[6]	0	0	0	0	0	0	0	0
[7]	0	0	0	0	0	0	0	1

OSPrioTbl

Figure 4.1 État du système au tick no 5 (les TCB en gris sont évidemment incomplets)

a) *T28 et T32 sont en attente d'un mot du fifo Qa. T11 est la tache qui s'exécute alors que T27, T29, et T63 sont prêtes.*

b) *La fonction OSPrioGetHighest() détermine la tâche ayant la plus grande priorité (plus petite valeur). Pour cela on parcours OSPrioTbl ligne par ligne, en débutant à la ligne 0, et on s'arrête à la première ligne non nul (avec au moins 1un 1). Considérant chaque ligne comme 8 taches on a dans prio un multiple de 8, ici prio = 8.*

   Puis avec la fonction CPU_CntLeadZeros on trouve la position du premier bit à 0 à partir de la gauche et en débutant à 0 auquel on additionne prio de l'étape précédente. Ici on aura prio = 8 + 3 = 11

c) *Ici OSQPend veut consommer une donnée mais déjà 2 tâches sont en attentes, donc on peut supposer que la queue est vide et que T11 sera mis en attente. Mais d'abord, la tâche T11 passe donc de RUNNING à PENDING, et le bit de OSPrioTbl est mis à 0. Puis T11 est mis en attente juste avant T28 car plus prioritaire. On aura donc dans l'ordre T11, T28 et T32 en attente. Notez aussi le délai de 1 pour la durée d'attente de T11 qui sera alors aussi mis dans le OSTickListDelay.*
   L'ordonnanceur est appelé via OSSched() et OS_PrioGetHighest(), puis T27 est désignée puisque c'est la tâche la plus prioritaire qui est READY. Donc T27 passe de READY à RUNNING puis on appel OSCtxSw pour la démarrer.

d) *Au tick 7, comme rien d'autres ne s'est produit depuis c), T11 sera libéré de OSTickListDelay lors du ISR de l'horloge (OSTimeTick) et donc quittera la liste d'attende QA (QA revient comme il était avant c). Le bit de OSPrioTbl de T11 est mis à 1 et T11 passe de PENDING à READY.*
   L'ordonnanceur est appelé via OSIntExit() et OS_PrioGetHighest(), puis T11 est désignée puisque c'est la tâche la plus prioritaire qui est READY. Donc T11 passe de READY à RUNNING puis on appel OSIntCtxSw pour la démarrer. Notez que T27 passe de RUNNING à READY.

Question 5 (3 points) À propos de OSSched()

Soit les extraits de code de uC/OS-III de la figure 5.1:

a) (.5 pt) Dans quel contexte est utilisée la fonction *OSSched()* avec uC/OS-III?

b) (2 pts) Décrivez le fonctionnement de *OSSched()* (lignes 10 à 21 de la figure 5.1). Notez que *CPU_INT_DIS()* et *CPU_INT_EN()* sont des fonctions qui appellent respectivement les instructions assembleur qui active/désactive les interruptions.

c) (.5 pt) Quelle différence existe-t-il entre *OSctxSw* et *OSIntCtxSw*? Justifiez.

```
1
2   // Extrait de  os_cpu.h
3    #define  OS_TASK_SW()                OSCtxSw()
4
5
6
7   //  Extrait de os_core.c
8
9   void  OSSched (void)
10  {
11      CPU_INT_DIS();
12      OSPrioHighRdy   = OS_PrioGetHighest();
13
14      OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;
15      if (OSTCBHighRdyPtr == OSTCBCurPtr) {
16          CPU_INT_EN();
17          return;
18      }
19
20      OS_TASK_SW();
21      CPU_INT_EN();
22
23  }
24
25  //Extrait de os_prio.c
26
27  OS_PRIO  OS_PrioGetHighest (void)
28  {
29      CPU_DATA  *p_tbl;
30      OS_PRIO    prio;
31
32
33      prio  = 0u;
34      p_tbl = &OSPrioTbl[0];
35      while (*p_tbl == 0u) {
36          prio = (OS_PRIO)(prio + (CPU_CFG_DATA_SIZE * 8u));
37          p_tbl++;
38      }
39      prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);
40
41      return (prio);
42
43  }
```

Figure 5.1

a) Interruption logicielle, i.e. lorsqu'une tâche bloque (état pending).

b)

i. On désactive les interruptions

ii. On appel OSPrioGestHighest pour trouver la tâche la plus prioritaire (voir réponse à la question 4b).

iii. On va chercher le pointeur (TCB) de la tâche correspondant à la priorité en ii) On suppose ici qu'il y en a une seule.

iv. On regarde si la tâche trouvée en iii est la même que celle qui s'exécute. Si c'est le cas on fait un return (pas besoin de changer la contexte) sinon on fait un changement de contexte avec OSCtxSw.

v. On réactive les interruptions.

c) OSCtxSw est appelée d'une interruption logicielle alors que OSIntCtxSw est appelée lors d'une interruption matérielle (ISR). OSntCtxSw ne fait que changer le contexte de la tâche désignée la plus prioritaire après le ISR alors que OSCtxSw retire aussi la tâche en cours.

Question 6 (3 points) Gestion du temps

Un concepteur souhaite mettre en place une tâche périodique de 14 ms, plus précisément il souhaite que la tâche se réveille, exécute un code (dont le temps est évidemment inférieur à 14 ms) et se rendorme pour 14 ms et ains de suite. Pour cela il utilise le code générique suivant :

```
while(1){

OSTimeDlyOSTimeDlyHMSM(0, 0, 0, 14, OS_OPT_TIME_PERIODIC,
&err);

Code périodique à exécuter

}
```

Notez que *OS_OPT_TIME_PERIODIC* joue le même rôle que lorsqu'il est utilisé dans *OSTimeDly* i.e. lorsqu'il demande un mode périodique.

a) (2 points) Sachant que la minuterie principale (timer) du système est réglée à 3 ms (333.33 Hz), quelle sera la précision de ce délai? Hint : il y a 3 sources possibles d'incertitude, expliquez à l'aide d'un schéma.

Il fallait ici m'expliquer les 3 sources du 31 des exercices.

b) (1 point) Aurait-on les mêmes délais en terme périodique si on remplace *OS_OPT_TIME_PERIODIC* par *OS_OPT_TIME_DLY* (mode relatif)? Expliquez.

Non nous n'aurons pas les mêmes délais puisqu'avec le mode OS_OPT_TIME_DLY, car le temps d'exécution de la tâche s'ajoute au délai d'attente. Autrement dit le cas no 3) en a) c'est-à-dire qu'on assure que la tâche repartira au maximum 1 tick après que le délai de la période est rencontrée, ne tient plus.

Annexe

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

Arguments

opt

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.

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

Arguments

opt

OS_OPT_POST_1
: Post and ready only the highest-priority task waiting on the semaphore.

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.

```
void  OSTimeDlyHMSM (CPU_INT16U   hours,
                     CPU_INT16U   minutes,
                     CPU_INT16U   seconds,
                     CPU_INT32U   milli,
                     OS_OPT       opt,
                     OS_ERR      *p_err)
```

Arguments

hours

is the number of hours the task is delayed. Depending on the opt value, the valid range is 0..99 (OS_OPT_TIME_HMSM_STRICT), or 0..999 (OS_OPT_TIME_HMSM_NON_STRICT). Please note that it not recommended to delay a task for many hours because feedback from the task will not be available for such a long period of time.

minutes

is the number of minutes the task is delayed. The valid range of values is 0 to 59 (OS_OPT_TIME_HMSM_STRICT), or 0..9,999 (OS_OPT_TIME_HMSM_NON_STRICT). Please note that it not recommended to delay a task for tens to hundreds of minutes because feedback from the task will not be available for such a long period of time.

seconds

is the number of seconds the task is delayed. The valid range of values is 0 to 59 (OS_OPT_TIME_HMSM_STRICT), or 0..65,535 (OS_OPT_TIME_HMSM_NON_STRICT).

milli

is the number of milliseconds the task is delayed. The valid range of values is 0 to 999 (OS_OPT_TIME_HMSM_STRICT), or 0..4,294,967,295 (OS_OPT_TIME_HMSM_NON_STRICT). Note that the resolution of this argument is in multiples of the tick rate. For instance, if the tick rate is set to 100Hz, a delay of 4 ms results in no delay because the delay is rounded to the nearest tick. Thus, a delay of 15 ms actually results in a delay of 20 ms.

opt

is the desired mode and can be either:

OS_OPT_TIME_HMSM_STRICT

(see above)

OS_OPT_TIME_HMSM_NON_STRICT

(see above)

OS_OPT_TIME_DLY

Specifies a relative delay.

OS_OPT_TIME_TIMEOUT

Same as OS_OPT_TIME_DLY.

OS_OPT_TIME_PERIODIC

Specifies periodic mode.

OS_OPT_TIME_MATCH

Specifies that the task will wake up when OSTickCtr reaches the value specified by hours, minutes, seconds and milli.

```
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)
```

Arguments

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

If OS_CFG_FLAG_MODE_CLR_EN is set to DEF_ENABLED in os_cfg.h, check all bits in flags to be clear (0)

OS_OPT_PEND_FLAG_CLR_ANY

If OS_CFG_FLAG_MODE_CLR_EN is set to DEF_ENABLED in os_cfg.h, 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

Finally, you can specify whether you want the caller to block if the flag(s) are available or not. You would then "add" the following options:

OS_OPT_PEND_BLOCKING

OS_OPT_PEND_NON_BLOCKING

Note that the timeout argument should be set to 0 when specifying OS_OPT_PEND_NON_BLOCKING, since the timeout value is irrelevant using this option. Having a non-zero value could simply confuse the reader of your code.

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

Arguments

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.

```
void  *OSQPend (OS_Q        *p_q,
                OS_TICK     timeout,
                OS_OPT      opt,
                OS_MSG_SIZE *p_msg_size,
                CPU_TS      *p_ts,
                OS_ERR      *p_err)
```

Arguments
p_q

is a pointer to the queue from which the messages are received.

timeout

allows the task to resume execution if a message is not received from the message queue within the specified number of clock ticks. A timeout value of 0 indicates that the task is willing to wait forever for a message. The timeout value is not synchronized with the clock tick. The timeout count starts decrementing on the next clock tick, which could potentially occur immediately.

opt
determines whether or not to block if a message is not available in the queue. This argument must be set to either:

OS_OPT_PEND_BLOCKING, or
OS_OPT_PEND_NON_BLOCKING

Note that the timeout argument should be set to 0 when specifying OS_OPT_PEND_NON_BLOCKING, since the timeout value is irrelevant using this option.

p_msg_size

is a pointer to a variable that will receive the size of the message (in number of bytes).

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

OS_ERR_TIMEOUT
If a message is not received within the specified timeout.

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*.**

- **Avec ICPP on a:**

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

Où:

- ***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.**
- **CS(*k*) le temps requis pour passer au travers la section critique *k*.**

```
OS_PRIO  OS_PrioGetHighest (void)
{
    CPU_DATA  *p_tbl;
    OS_PRIO    prio;
    prio  = 0u;
    p_tbl = &OSPrioTbl[0];
    while (*p_tbl == 0u) {
        prio = (OS_PRIO)(prio + (CPU_CFG_DATA_SIZE * 8u));
        p_tbl++;
    }
    prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);

    return (prio);
}
```