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**	**202103**
Professeur	***Local***	***Téléphone***
Guy Bois	**M-4115**	**5944**

Jour	*Date*	*Durée*	*Heures*
Lundi	**25 octobre 2021**	**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 **19** pages **(excluant cette page)**

- La pondération de cet examen est de **30** **%**
- Répondez directement sur le questionnaire pour les no 2 et 3
- 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 (3.5 points) En vrac : vrai ou faux avec justification

a) (.6 pt) Lorsqu'on compare le protocole héritage de priorité avec celui de ICPP, on remarque qu'en général ICPP minimise davantage les préemptions (c.-à-d. le fait qu'une tâche puisse être arrêtée par une tâche plus prioritaire qu'elle-même).

Faux, on dit que ICPP minimise davantage le temps de blocage par rapport à l'héritage de priorité mais pas le temps des interférences. Si vous enlevez complètement le temps des blocages il restera toujours le temps des interférences. Notez aussi que ICPP minimise les changements de contexte par rapport à l'héritage de priorité mais il y a une différence entre les changements de contexte et les interférences.

b) (.6 pt) La période d'horloge du processeur est en général une valeur 100,000 à 1 million de fois plus petites que le pas d'ordonnancement (tick), ce qui implique que de manière générale que le tick d'horloge est de l'ordre des *ms*, la période la minuterie est en *us* et la fréquence du processeur est en *ns* ou moins.

Vrai, ça été dit en classe, en général des ticks de l'ordre de la us et moins ont tendance à provoquer trop d'interruptions de la minuterie. Donc pour avoir des ticks dans l'ordre de la ms (1000 Hz) on doit avoir une minuterie qui va plus vite donc dans les MHz. Finalement, on a le processeur qui lui roule en GHz.

c) (.6 pt) Soit les 2 appels de fonctions suivantes :

OSSemPend(&Sem, 0, OS_OPT_PEND_BLOCKING, &ts, &err); **(1)**

OSSemPend(&Sem, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err); **(2)**

Lorsque (1) ne bloque pas lors de son appel (p. ex. sémaphore > 0), il aura alors le même comportement que (2).

Vrai et faux (j'ai accepté les 2 en autant que la justification était appropriée).

Pourquoi vrai : même comportement au sens non bloquant si sem > 0.

Pourquoi faux : si on tient compte du fait qu'il y a une différence avec la gestion d'erreur pour (2)

d) (.6 pt) Lorsqu'une ressource est partagée entre deux sous-routines d'interruption, le mécanisme le plus approprié est de désactiver l'ordonnanceur avant chaque accès à la ressource.

Faux, dans le contexte de 2 ISRs il faut avoir recours à la désactivation des interruptions

e) (1.1 point) Soit une application composée de 4 tâches T4 à T1 (en ordre décroissant de priorité) utilisant trois mutex M1, M2 et M3 :

	M1	M2	M3
T4	5	2	0
T3	0	7	1
T2	6	5	0
T1	7	8	9

Dans le cas de l'utilisation de l'utilisation de ICPP, le temps maximal de blocage pour chaque tâche est : B4 = 8, B3 = 9, B2 = 9, B1 = 0.

Si on se concentre sur M1 pour T4, on a 2 sources de blocage B4 pour T4

-T2 peut bloquer T4 par 6 ticks

-T1 peut bloquer T4 par 7 ticks

-Comme T2 et T1 ne peuvent utiliser le mutex au même moment, on prend le maximum

Conclusion pour M1 : il contribue au blocage de T4 pour un maximum de max(6, 7) = 7 ticks

Si on se concentre sur M2 pour T4, on a 3 sources de blocage B4 pour T4

-T3 peut bloquer T4 par 7 ticks

-T2 peut bloquer T4 par 5 ticks

-T1 peut bloquer T4 par 8 ticks

Conclusion pour M2: il contribue au blocage B4 pour un maximum de max(7, 5 ,8) = 8 ticks

Comme M4 n'utilise pas M3, le blocage de T4 ne vient que de M1 et M2. Or, comme on parle de ICPP on prend le maximum entre les 2 blocages calculés ci-haut:

Conclusion : B4 = max(7,8) = 8 OK

De la même manière que pour B1, calculons B3:

B3 = max(max(5, 8), max(9)) = 9 donc OK

De la même manière que pour B1, calculons B2:

B2 = max(max(7), max(8)) = 8 différent de 9 donc la réponse est fausse

P.S. T1 ne peut jamais bloqué donc T1 = 0 donc OK.

Question 2 (3.5 points) Ordonnancement et héritage de priorité

Soit 6 tâches sous uC/OS-III T1, T2, T3, T4, T5 et T6 ayant respectivement les priorités 24, 20, 16, 8, 4 et 2.

À partir de la spécification de la table 2.1, complétez la trace d'exécution de la figure 2.1 (page suivante) en considérant le protocole héritage de priorité de uC/OS-III. Pour chaque coup d'horloge, indiquez s'il y a lieu le changement de priorité des tâches (en dessous) suite à l'application du protocole. Finalement, indiquez par un trait plus gras le blocage de T4.

Tâche (T_i)	Numéro de la période de départ	Nombre de ticks d'exécution du thread	Séquence d'exécution (les parenthèses indiquent que la tâche possède deux mutex durant un même tick)
T6	6	2	EU
T5	9	2	EE
T4	4	4	V(VQ)(VQ)V
T3	3	3	EQQ
T2	2	3	EEE
T1	0	7	EV(VU)(VU)(VU)(VU)V

Table 2.1

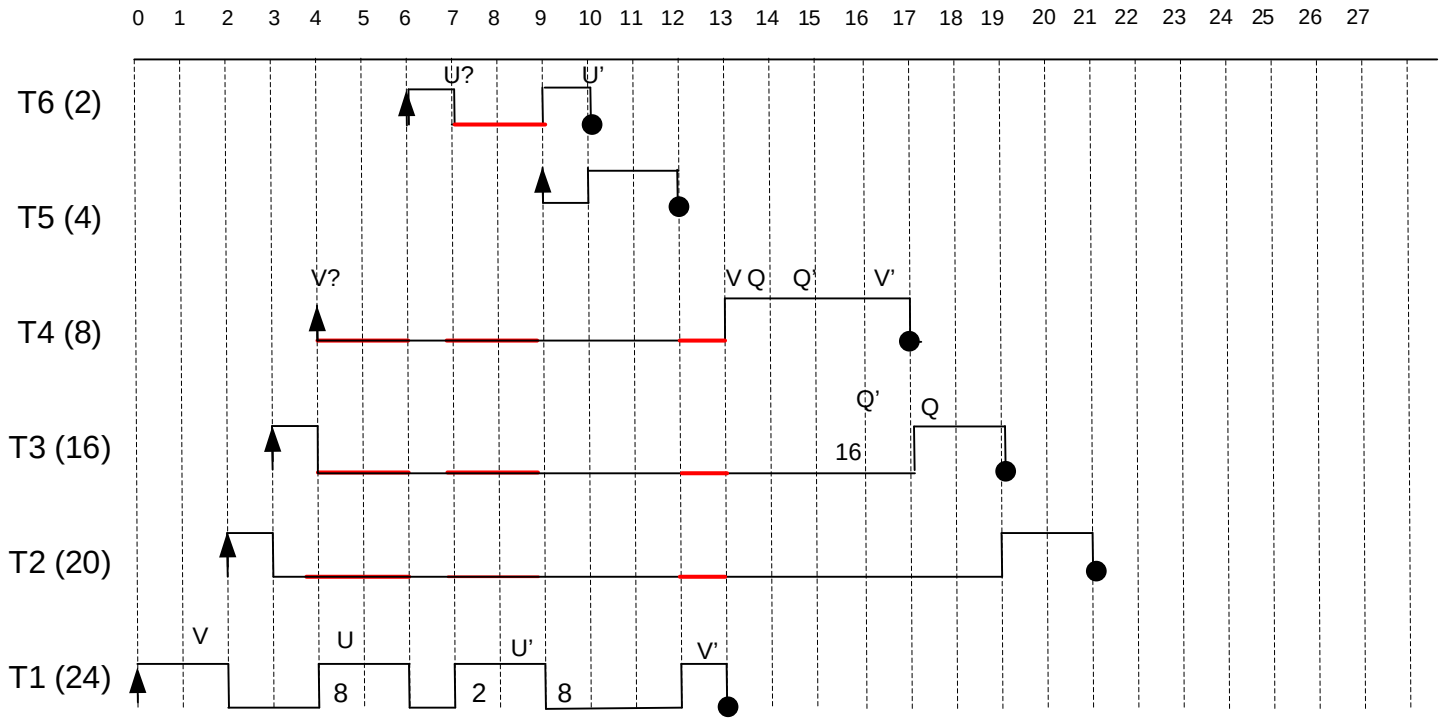

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
T6 (2)
T5 (4)
T4 (8)
T3 (16)
T2 (20)
T1 (24)
U?
U'
V?
V Q
Q'
V'
Q'
Q
16
V
U
U'
V'
8
2
8

Figure 2.1

Question 3 (4 points) Exclusion mutuelle, synchronisation et communication sous uC/OS-III

a) (2 pts) Soit le flôt de données de la figure 3.1. Complétez le code des figures 3.3, 3.4 et 3.5 (étiquettes 1 à 7) pour réaliser le flôt de la Figure 3.1. Complétez directement sur ces figures.

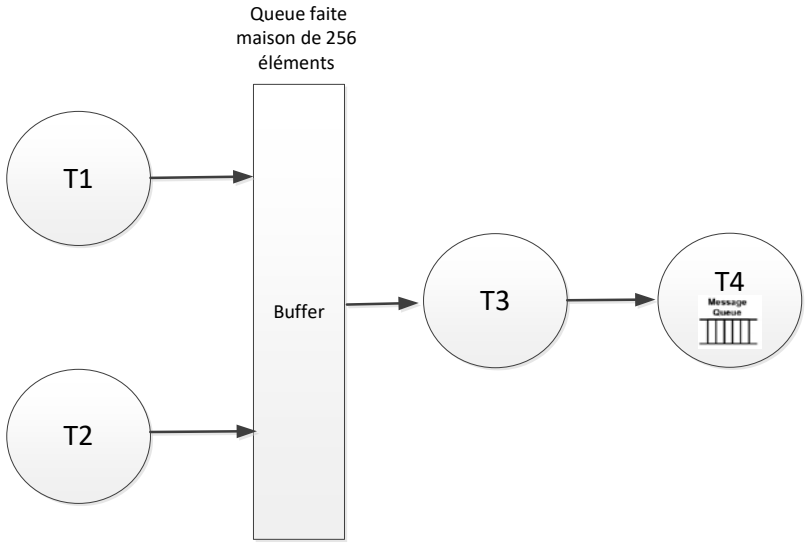


Figure 3.1

a) (2 pts) Soit le flôt de données de la figure 3.2 pour lequel on a remplacé les tâches T1 et T2 par 2 ISR (ISR1 et ISR2). On suppose que T1 et T2 sont deux *fittimers* à une certaine fréquence et à chaque interruption on écrit le numéro de l'interruption et une certaine donnée dans Buffer. Complétez le code des figures 3.6 et 3.7 (étiquettes 8 à 13) pour réaliser le flôt de la Figure 3.2. Complétez directement sur ces figures.

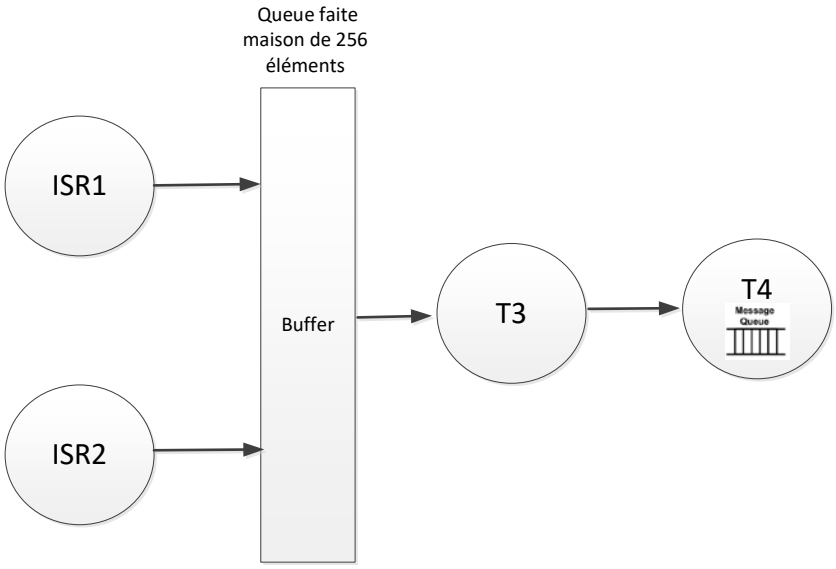


Figure 3.2

```
#define  MSG_QUEUE_SIZE  256   // Size of message queue used in example

typedef struct {
        unsigned short value;
        unsigned char device;
} msg;
.
.
#define  T4Prio  22
#define  T3Prio  19
#define  T2Prio  21
#define  T1Prio  21

.
.

msg buffer[N];
int add = 0, rem = 0;
unsigned short reg;


OSTaskCreate(&T1TCB, "T1", T1, (void*)0, T1Prio, &T1[0u], TASK_STK_SIZE/2, TASK_STK_SIZE,
1, 1, (void*)0, (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &os_err);
OSTaskCreate(&T2TCB, "T2", T2, (void*)0, T2Prio, &T2[0u], TASK_STK_SIZE/2, TASK_STK_SIZE,
1, 1, (void*)0, (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &os_err);
OSTaskCreate(&T3TCB, "T3", T3, (void*)0, T3Prio, &T3[0u], TASK_STK_SIZE/2, TASK_STK_SIZE,
1, 1, (void*)0, (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &os_err);
OSTaskCreate(&T4TCB, "T4", T4, (void*)0, T4Prio, &T4[0u], TASK_STK_SIZE/2, TASK_STK_SIZE,
1024, 1, (void*)0, (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &os_err);
OSTaskCreate(&StartupTaskTCB,"Main Task", StartupTask, (void *) 0, UCOS_START_TASK_PRIO,
&StartupTaskStk[0], 0, UCOS_START_TASK_STACK_SIZE,
0, 0, DEF_NULL, (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &os_err);

OSMutexCreate(&m1, “mutex”, &os_err);

OSSemCreate(&SemFull, "SemFull", 0, &os_err);  // Sémaphore compteur qui assure à une
                                               // tâche que le buffer n’est pas vide

OSSemCreate(&SemEmpty, "SemEmpty", N, &os_err); // Sémaphore compteur qui assure à une
                                                // tâche que le buffer n’est pas plein


.
.
OSStart(&os_err);
return 0;
```

Figure 3.3 (initialisation, rien à compléter ici)

```
void  T1 (void *data)
{
        int i;
        msg dat;
        unsigned short value = 0;
        OS_ERR err;
        CPU_TS ts;
        dat.device = 1;

        while(1){
                for(i=0; i<100; i++) {

1
        OSSemPend(&SemEmpty,0,OS_OPT_NON_BLOCKING, &ts,  &err);
        OSMutexPend(&m1, 0, OS_OPT_BLOCKING, &ts, &err);
                        dat.data = value++;
                        buffer[add++] = dat;
                        if(add>=N)
                                add =0;
2
        OSMutexPost(&m1, 0, OS_OPT_POST_1, &ts, &err);
        OSSemPost(&SemFull, OS_OPT_NONE,  &err);

                        OSTimeDly(1, OS_OPT_TIME_PERIODIC, &err);
                }
                OSTimeDlyHMSM(0, 0, 0, 400, OS_OPT_TIME_HMSM_STRICT, &err);

        }
}

void  T2 (void *data)
{
        int i;
        msg dat;
        unsigned short value = 0;
        OS_ERR err;
        CPU_TS ts;
        dat.device = 2;

        while(1){
                for(i=0; i<100; i++) {
3
        OSSemPend(&SemEmpty,0,OS_OPT_NON_BLOCKING, &ts,  &err);
        OSMutexPend(&m1, 0, OS_OPT_BLOCKING, &ts, &err);
                        dat.data = value++;
                        buffer[add++] = dat;
                        if(add>=N)
                                add =0;
4
        OSMutexPost(&m1, 0, OS_OPT_POST_1, &ts, &err);
        OSSemPost(&SemFull, OS_OPT_NONE,  &err);

                        OSTimeDly(1, OS_OPT_TIME_PERIODIC, &err);
                }
                OSTimeDlyHMSM(0, 0, 0, 400, OS_OPT_TIME_HMSM_STRICT, &err);

        }
}
```

Figure 3.4 (no 2a étiquettes 1 à 4 à compléter)

```
void  T3 (void *data)
{
        msg dat;
        OS_ERR err;
        CPU_TS ts;

        while (1) {


5
```

OSSemPend(&SemFull,0,OS_OPT_BLOCKING, &ts, &err);
OSMutexPend(&m1, 0, OS_OPT_NON_BLOCKING, &ts, &err);

```
                out = buffer[rem++];
                if (rem >= N)
                        rem = 0;


6
```

OSMutexPost(&m1, 0, OS_OPT_POST_1, &ts, &err);
OSSemPost(&SemEmpty, OS_OPT_NONE, &err);
OSTaskQPost(&T4TCB, out, sizeof(out), OS_OPT_POST_FIFO + OS_OPT_POST_NO_SCHED, &err);

```
        }
}

void  T4 (void *data)
{
        OS_ERR err;
        CPU_TS ts;
        OS_MSG_SIZE msg_size;
        msg *in=NULL;

        while(1){


 7
```

msg = OSTaskQPend(0, OS_OPT_PEND_BLOCKING, &msg_size, &ts, &err);

```
         UCOS_Printf("T4 Value = %d Device:  %d\n, msg->value, msg->device);


        }
}
```

Figure 3.5 (no 2a étiquettes 5 à 7 à compléter)

```
void  ISR2 (void *data)
{
       int i;
       msg dat;
       unsigned short value = 0;
       OS_ERR err;
       CPU_TS ts;



8
OS_ENTER_CRITICAL();
OSSemPend(&SemEmpty,0,OS_OPT_NON_BLOCKING, &ts,  &err);
If (err == OS_ERR_NONE) {
                      dat.device = 2;
                      dat.data = value++;
                      buffer[add++] = dat;
                      if(add>=N)
                             add =0;

9


                      OSSemPost(&SemFull, OS_OPT_NONE,  &err);
                          }'
OS_EXIT_CRITICAL();

       }
}

void  ISR1 (void *data)
{
       int i;
       msg dat;
       unsigned short value = 0;
       OS_ERR err;
       CPU_TS ts;

10
OS_ENTER_CRITICAL();
OSSemPend(&SemEmpty,0,OS_OPT_NON_BLOCKING, &ts,  &err);
If (err == OS_ERR_NONE) {

                      dat.device = 2;
                      dat.data = value++;
                      buffer[add++] = dat;
                      if(add>=N)
                             add =0;
11


                      OSSemPost(&SemFull, OS_OPT_NONE,  &err);
                          }'
OS_EXIT_CRITICAL();



       }
}
```

Figure 3.6 (no 2b étiquettes 8 à 11 à compléter)

N.B. À la page précédente, je n'ai pas enlevé de points si la condition If (err == OS_ERR_NONE) { } avait été omise , mais j'ai donné un bonus de .5 à ceux qui l'avaient complétés.

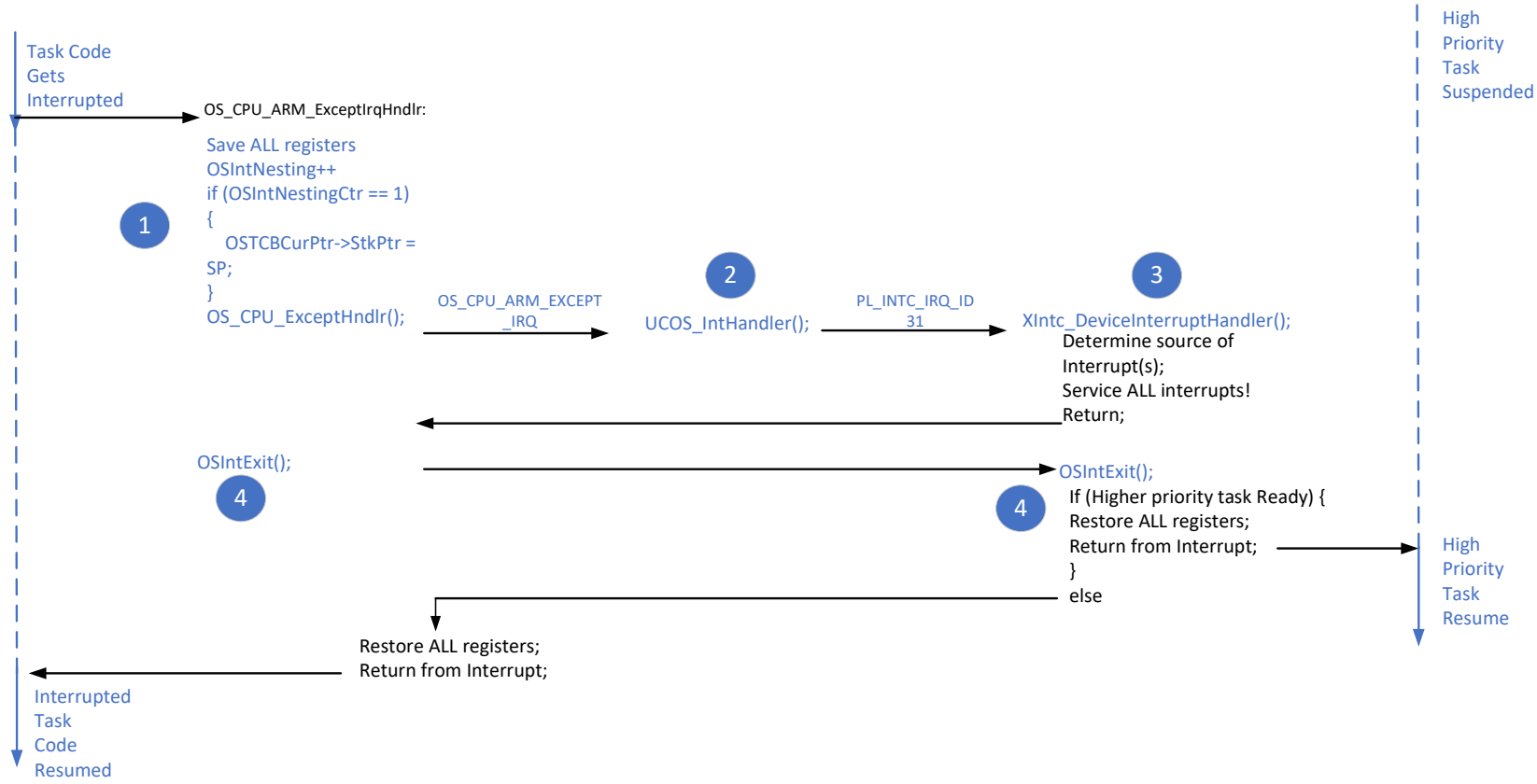
```
void  T3 (void *data)
{
       msg dat;
       OS_ERR err;
       CPU_TS ts;

       while (1) {


12
OS_ENTER_CRITICAL();
                OSSemPend(&SemFull,0,OS_OPT_NON_BLOCKING, &ts,  &err);
If (err == OS_ERR_NONE) {
               out = buffer[rem++];
               if (rem >= N)
                      rem = 0;

13
                OSSemPost(&SemEmpty, OS_OPT_NONE,  &err);
               OSTaskQPost(&T4TCB, out, sizeof(out), OS_OPT_POST_FIFO +
OS_OPT_POST_NO_SCHED, &err);
                              }'
OS_EXIT_CRITICAL();



       }
}
```

Figure 3.7 (no 2b étiquettes 12 à 13 à compléter)

N.B. Le choix de mettre ou pas un If (err == OS_ERR_NONE) { est discutable. On le met pour éviter d'augmenter la latence d'interruption si l'appel à OSSemPend bloque. Mais j'acceptais avec ou sans le test sur err.

Question 4 (3 points) Interruptions sous uC/OS-III

Décrivez le déroulement d'une interruption matérielle (externe) sous uC/OS-III avec une puce Zynq SoC en décrivant le rôle du code aux étiquettes 1 à 4, ainsi que le passage d'une étiquette à l'autre. Pour chaque étiquette, indiquez également où s'exécute le code. Finalement, indiquez dans quelle étiquette (précisez où exactement) vos deux ISRs du lab 1 (*gpio_isr* et *fit_timer_20s_isr*) était appelés.



Étiquette 1 :

Une interruption externe (matérielle) survient. La tâche en cours est interrompue à la fin de son instruction courante. On sauve son contexte sur sa pile d'exécution et on démarre le ISR. On incrémente le compteur d'interruption et s'il s'agit de la première interruption (on peut avoir plusieurs interruptions qui surviennent en cascade) on sauve le contexte de la pile d'exécution car il est possible que le ISR (ou ceux qui vont suivre en cascade) utilise la pile d'exécution (un ISR n'a pas de pile d'exécution). Finalement, via la table des vecteurs du GIC, la tâche principale des ISR est appelée (OS_CPU_ExceptHndlr()) et fait le départage des types d'interruptions (IRQ, FIQ, etc.).

Étiquette 2 :

Puis, comme il s'agit d'un IRQ on appelle la fonction UCOS_IntHandler qui elle détermine si il s'agit d'un IRQ sur ARM (ex. PL_INT_IRQ_ID_29) ou le FPGA (ex. ex. PL_INT_IRQ_ID_31). À partir de là on quitte le GIC et on s'en va sur le FPGA (PIC).

Étiquette 3 :

Sur le FPGA, XIntcDeviceInterruptHandler() appelé par UCOS_IntHandler va passer en revue toutes les entrées du PIC, de la plus prioritaire à la moins prioritaire (e.g. 0 à 31), jusqu'à trouver celle qui a demandé un service. Ce service est alors donné (myISR).

Étiquette 4 :

Finalement une fois le service complétée on appel OSIntExit() qui va dans un premier temps décrémenter le compteur d'interruptions. Si on est à la dernière interruption (d'une cascade) on doit redonner la main à une tâche. La tâche la plus prioritaire sera alors désignée par l'ordonnanceur et un changement de contexte sera réalisé via OSIntCtxSwitch. Si jamais il n'y a pas de tâches plus prioritaires ou que l'ordonnanceur est désactivé, OSIntCtxSwitch va redonner la main à la tâche interrompue.

gpio_isr et fit_timer_20s_isr sont apelés de 3.

Question 5 (3 points) Délais et minuterie

(1 pt) Décrivez la différence entre délai relatif et délai périodique.

(2 pt) Décrivez le comportement du code ci-après (qui se poursuit à la page suivante) pour 2 périodes de la minuterie (watchdog) nommé *Scheduler*. Plus précisément faites la trace avec les xil_printf().

Assumez que la fréquence du système est 1000Hz (OS_CFG_TICK_RATE_HZ) et celle de la minuterie est 10Hz (OS_CFG_TMR_TASK_RATE_HZ). Considérez que la priorité de la fonction de callback de la minuterie est plus prioritaire que celle de TaskPeriodic. Finalement, considérez que OSTimeGet() retourne le nombre de ticks depuis le début de l'exécution.

```
#define      TASK_STK_SIZE            8192

#define      TASK_Periodic_PRIO       15
#define      WAITFOR500TICKS          500


CPU_STK      StartupTaskStk[UCOS_START_TASK_STACK_SIZE];
OS_TCB       StartupTaskTCB;

CPU_STK      TaskPeriodicStk[TASK_STK_SIZE];
OS_TCB       TaskPeriodicTCB;

OS_SEM       Synchro;

OS_TMR       Scheduler;

void         TaskPeriodic(void *data);
void         SchedulerFct(OS_TMR  *p_tmr, void    *p_arg);

void         StartupTask (void *p_arg);


int main (void)
{
    OS_ERR  err;

    UCOS_LowLevelInit();

    CPU_Init();
    Mem_Init();
    OSInit(&err);

    OSSemCreate(&Synchro, "Synchro", 0, &err);

(suite page suivante)
```

```
    OSTmrCreate(&Scheduler,                 /* p_tmr          */
                "Scheduler",                /* p_name         */
                 0,                         /* dly            */
                 10,                         /* period         */
                 OS_OPT_TMR_PERIODIC,       /* opt            */
                 SchedulerFct,              /* p_callback     */
                 0,                         /* p_callback_arg */
                &err);                      /* p_err          */


    OSTaskCreate(&TaskPeriodicTCB,"TaskPeriodic", TaskPeriodic, (void *) 0,
TASK_Periodic_PRIO, &TaskPeriodicStk[0u], TASK_STK_SIZE/2, TASK_STK_SIZE,
20, 0, (void *) 0, (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);
    OSTaskCreate(&StartupTaskTCB,"Main Task", StartupTask, (void *) 0,
UCOS_START_TASK_PRIO, &StartupTaskStk[0], 0, UCOS_START_TASK_STACK_SIZE,
0, 0, DEF_NULL, (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);

    OSStart(&err);
    return 0;
}


void  TaskPeriodic (void *data)
{
    OS_ERR  err;
      CPU_TS ts;
      OS_TICK actualticks = 0;
      OSTmrStart(&Scheduler, &err);
      while(1)
      {
            actualticks = OSTimeGet(&err);
            xil_printf("Attente active de 500 ticks\n");
            while(WAITFOR500TICKS + actualticks > OSTimeGet(&err));
            xil_printf("Fin de l execution - Tick no %d \n",
            OSTimeGet(&err));
            OSSemPend(&Synchro,0, OS_OPT_PEND_BLOCKING, &ts, &err);
            xil_printf("Fin de la periode  - Tick no %d \n",
            OSTimeGet(&err));
      }
}

void  SchedulerFct (OS_TMR  *p_tmr,
                    void    *p_arg)
{
  OS_ERR  err;
  OSSemPost(&Synchro, OS_OPT_POST_1, &err);
}
```

```
Connected to COM6 at 115200
UCOS - uC/OS Init Started.
UCOS - STDIN/STDOUT Device Initialized.
AUCOS - UCOS init done
UCOS - Total configured heap size. t512
UCOS - Total used size after init. 332
Programme initialise -
tente active de 500 ticks
Fin de l execution  - Tick no 500
Fin de la periode  - Tick no 1000
Attente active de 500 ticks
Fin de l execution  - Tick no 1500
Fin de la periode  - Tick no 2000
```

Question 6 (3 points) Gestion des tâches et des évènements dans le lab no 1

Soit le système que vous avez conçu au laboratoire no 1 avec TaskGenerate, TaskComputing, TaskForwarding, TaskOutputPort, TaskStats, TaskReset et TaskStop.

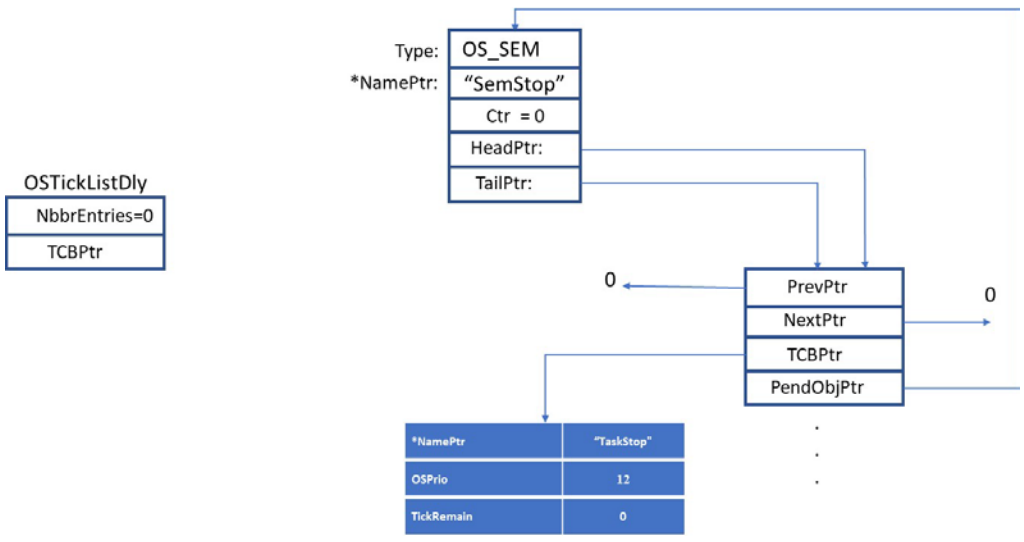
Les priorités des tâches sont les suivantes :

```
#define          UCOS_START_TASK_PRIO          5
#define         TaskGeneratePRIO               15
#define           TaskStatsPRIO                10
#define         TaskComputingPRIO              21
#define         TaskForwardingPRIO             22
#define         TaskOutputPortPRIO             20
#define         TaskResetPRIO                  11
#define         TaskStopPRIO                   12
```

La figure 6.1 ci-après (et qui se poursuit à la page suivante) décrit 5 structures de données du système à un moment donné de son exécution après avoir appuyez sur le bouton de démarrage.

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



(suite page suivante)

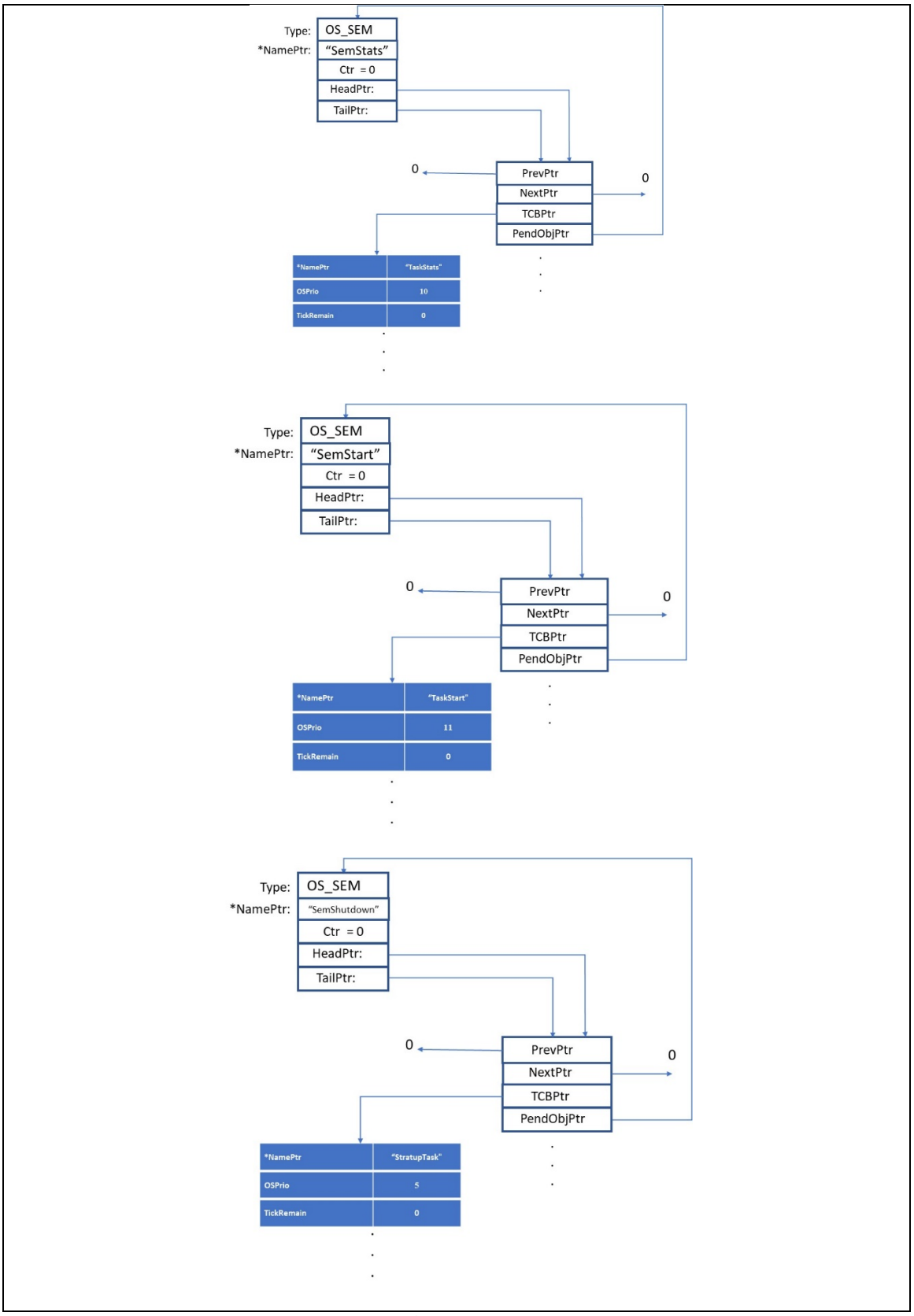

Type:
*NamePtr:
OS_SEM
"SemStats"
Ctr = 0
HeadPtr:
TailPtr:
0
PrevPtr
NextPtr
TCBPtr
PendObjPtr
0
*NamePtr
"TaskStats"
OSPrio
10
TickRemain
0
Type:
*NamePtr:
OS_SEM
"SemStart"
Ctr = 0
HeadPtr:
TailPtr:
0
PrevPtr
NextPtr
TCBPtr
PendObjPtr
0
*NamePtr
"TaskStart"
OSPrio
11
TickRemain
0
Type:
*NamePtr:
OS_SEM
"SemShutdown"
Ctr = 0
HeadPtr:
TailPtr:
0
PrevPtr
NextPtr
TCBPtr
PendObjPtr
0
*NamePtr
"StratupTask"
OSPrio
5
TickRemain
0

Figure 6.1

a) (1 pt) En assumant que vous avez configuré le système et que vous avez ensuite appuyé sur le bouton pressoir pour démarrer le système, en appuyant sur BP0.

   Décrivez alors l'état du système à la Figure 6.1, c'est-à-dire les différentes tâches en cours et leur état. Également selon vous quelle tâche est en exécution.

b) (2 pts) On suppose ensuite qu'une interruption du fittimer arrive et évidemment que les interruptions ne sont pas désactivées.

   En considérant que le nombre de paquets rejetés est inférieur à la limite que vous vous êtes données, indiquez dans un ordre chronologique ce qui se passe lors de cette interruption du fittimer :

   Attention je n'ai pas besoin d'une énumération des appels de fonctions systèmes. Je veux que vous me décriviez dans vos mots ce qui se passe au niveau :

   - Des rendez-vous unilatéraux (e.g. entre ISR et tâche(s))
   - Des changements dans les structures de données décrites ci-haut (ça veut dire indiquez quel(s) bit(s) change(nt) dans OSPrioTbl[] et quel changement s'opère dans les structures de données des sémaphores).
   - Des changements de contexte lorsque OSSched() est appelé (i.e. comment l'ordonnanceur trouve la prochaine tâche la plus prioritaire et entre quoi et quoi se fait le changement de contexte).

a)

On veut l'état du système juste après avoir démarré le système avec BP0 :

TaskGenerate: état Running car c'est la tâche la plus prioritaire parmi les tâches prêtes (Ready)

TaskComputing, TaskForwarding et TaskOutputPort: état Ready i.e. que les bits correspondant sont à 1 mais moins prioritaires.

TaskStats, TaskReset, TaskStop et StartUpTask son ten attentes de leur semaphore respectif.

Remarque: aucune tâche n'est en attente d'un délai.

b)

Le ISR du fittimer (fit_timer_20s_isr) va faire un OSSemPend(&SemStop, ,&err) non bloquant, ce qui va permettre à la tâche TaskStop de démarrer après l'interruption car elle est plus prioritaire que TaskGenerate. Comme le nombre de paquets rejetés est inférieur à la limite, le système va poursuivre son exécution normale mais va avant tout afficher les statistiques avec un appel à OSSemPend(&SemStats,…, &err). Comme SemStats est plus prioritaire que OSSemStop, SemStats va s'executer immédiatement ou après l'exécution de OSsemStop (dépendamment de si on a mis ou non le OS_OPT_PEND_NON_BLOCKING). Dans un cas comme dans l'autre, OSSemStop va se terminer après l'exécution de son corps de boucle, au retour à OSSemPend(&SemStop, 0, OS_OPT_PEND_BLOCKING, &ts, &err). Finalement, TaskGenerate (ou TaskComputing, TaskForwarding et TaskOutputPort) reprendra son exécution.

Annexe

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

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)
```

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

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.

void *OSTaskQPend (OS_TICK timeout,

OS_OPT opt,

OS_MSG_SIZE *p_msg_size,

CPU_TS *p_ts,

OS_ERR *p_err)

opt

determines whether or not the user wants to block if a message is not available in the task's 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 OS_OPT_PEND_NON_BLOCKING is specified, 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.

Returned Value

The message if no error or a NULL pointer upon error. You should examine the error code since it is possible to send NULL pointer messages. In other words, a NULL pointer does not mean an error occurred. *p_err must be examined to determine the reason for the error.

void OSTaskQPost (OS_TCB *p_tcb,

void *p_void,

OS_MSG_SIZE msg_size,

OS_OPT opt,

OS_ERR *p_err)

opt

OS_OPT_POST_FIFO

POST message to task and place at the end of the queue if the task is not waiting for messages.

void OSSchedLock (OS_ERR *p_err)

void OSSchedUnlock (OS_ERR *p_err)

CPU_CRITICAL_ENTER()

CPU_CRITICAL_EXIT()

En résumé

Étapes d'une interruption du lab 1

- On s'intéresse ici à une interruption du PL au PS i.e. entre un périphériqu
  (fit_timer_0, fit_timer_1 ou axi_gpio_0) et le Cortex A9 Core 1