

Polytechnique de Montréal - DGIGL

Laboratoire #1: Routeur sur puce FPGA

INF3610 – Hiver 2024

Laboratoire no 1, séance no 1 : Routeur simple s'exécutant sur une
puce SoC Zynq 7020 sous μ C/OS-III

Guy Bois

1. Objectif général

L'objectif de ce laboratoire est de concevoir une application temps réel pour un système embarqué en ayant recours au RTOS μ C/OS-III, pour en faire une implémentation sur FPGA SoC avec la carte PYNQ-Z2.

Les objectifs spécifiques du laboratoire no 1 (séances no 1 à 3) sont :

- Faire l'apprentissage de μ C/OS-III et de certaines notions de temps réel
- Utiliser et comprendre le mécanisme de drivers (pilotes) et d'interruption.
- Développer un petit driver (pilote) sous μ C/OS-III
- S'initier aux environnements de développement embarqués, tels Vivado et Xilinx SDK pour créer respectivement une plate-forme matérielle du Zynq SoC et son BSP (Board Support Package).
- Étudier les spécificités de la programmation SoC sur une carte PYNQ-Z2.
- Réaliser une programmation de type AMP sur 2 processeurs en utilisant une mémoire partagée.

N.B. μ C/OS-III, l'architecture de la puce Zynq SoC 7020 et la carte PYNQ-Z2 sera présentée en classe.

2. Laboratoire en 3 séances

Ce laboratoire sera divisé en 3 séances. En gros les thématiques seront les suivantes :

Séance no 1 (2 et 9 février)

- Validation fonctionnelle d'un routeur sur puce
- Ajout de timestamp (en français horodatage) pour analyse de performance
- Gestion de mémoire

Séance no 2 :

- Watchdog
- Interruption et écriture de drivers

Séance no 3 :

- Mécanismes de partage de tâches du routeur sur 2 processeurs en mode AMP

3. Mise en contexte

Application

La **figure 1** illustre un routeur de paquets permettant l'acheminement de paquets d'une source à une destination. Dépendamment de la destination, les paquets transitent à travers un ou plusieurs routeurs.

Par exemple, pour aller de la **source 0** à la **destination 1**, les paquets vont passer par le routeur 1, le routeur 2 et le routeur 4 alors que pour aller de la **source 0** à la **destination 2**, les paquets vont passer par le routeur 1, le routeur 3 et le routeur 5. Le routeur 1 devra donc regarder l'adresse de destination de chaque paquet pour décider si ce dernier doit transiger vers le routeur 2 ou le routeur 3. La fonction principale d'un routeur est donc de prendre un paquet et de le renvoyer au bon endroit en fonction de la destination finale. Finalement, un routeur peut aussi supporter une qualité de service (**QoS**) en triant et priorisant les paquets selon qu'il s'agit par exemple d'un paquet audio, vidéo ou encore contenant des données quelconques.

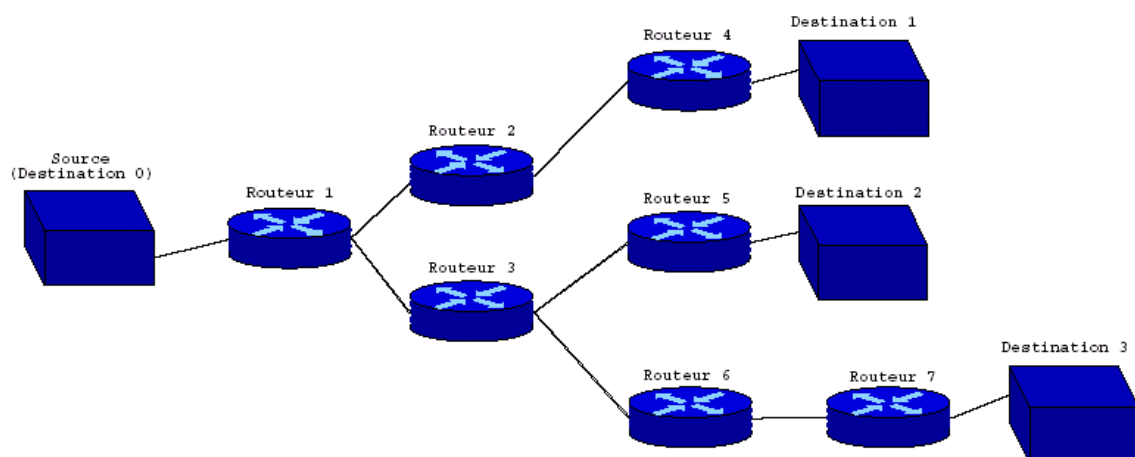


Figure 1 Exemple d'un routeur de paquets simplifié

Évidemment, un routeur peut supporter bien d'autres fonctions. Toutefois, dans ce laboratoire, nous nous concentrerons sur les trois énumérées au paragraphe précédent. Le format des paquets sur le réseau est le suivant :

4	4	4	4	8	40
Octets	Octets	Octets	Octets	Octets	Octets
Source	Dest	Type	CRC	Timestamp	DATA

Ces paquets sont de taille fixe (**64 octets**) et possèdent six champs, soit une **source** indiquant la provenance du paquet, une **destination**, un **type** pour la qualité de service, un contrôle de redondance cyclique ou CRC (Cyclic Redundancy Check), un **timestamp** indiquant un moment précis et finalement les **données** transportées. La définition en langage C de cette structure vous sera fournie.

Plateforme matérielle ciblée

La flexibilité des puces multiprocesseurs configurables Zynq utilisées en laboratoire nous permet de faire différents partitionnements matériels/logiciels. Pour cette première partie, nous utiliserons comme cible sur laquelle s'exécutera le routeur, la plate-forme de la figure 2 tel que vous l'avez construite dans le tutoriel de Vivado.

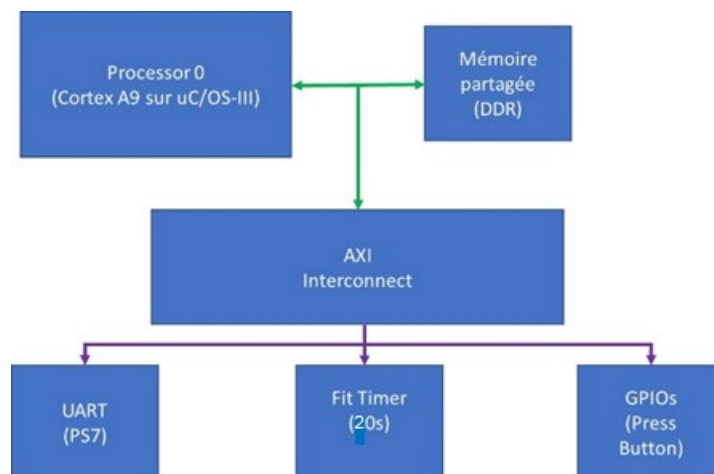


Figure 2 Architecture ciblée

4. Fonctionnement général et description des tâches

La **figure 3** illustre le flot des paquets à l'intérieur du routeur. Il y a plusieurs manipulations utilisant les fonctions uC/OS-III tel que synchronisation, mutex, minuterie et communication par queue de messages (fifo). D'autres fonctionnalités seront ajoutées aux séances 2 et 3.

Pour la séance 1, un code de départ vous est déjà donné et vous devrez ajouter 2 fonctionnalités.

Dans ce qui suit, on vous donne une brève description des différentes tâches de ce code de départ. Plus précisément, la fig. 3 illustre les tâches, et leur TCB correspondant, ainsi que les différents fifo externe ou interne qui relient les tâches:

Interface d'entrée/génération des paquets (TaskGenerate)

Cette tâche s'occupe de générer des paquets aléatoires qui seront traités par le routeur. Cette tâche fait partie du banc de test (testbench) puisqu'elle pourrait être par la suite remplacée par la vraie interface d'entrée. *TaskGenerate* **alterne** entre **deux modes**, soit le **mode génération** (rafale) et le **mode attente**. Chaque mode s'exécute en alternance pour une durée allant jusqu'à 1 sec¹ (`OSTimeDlyHMSM(0, 0, 01, 10, OS_OPT_TIME_HMSM_STRICT, &err);`). Durant le mode génération un nombre de paquets aléatoire entre 1 et 255 est généré, avec également pour chaque paquet une source, une destination, un type (priorité), un CRC-16 et des données (payload) aléatoires. ~~Chaque génération de paquet est entrecoupée d'un délai de 1 ms, c'est-à-dire `OSTimeDlyHMSM(0, 0, 0, 1, OS_OPT_TIME_HMSM_STRICT, &err);`, afin de laisser les tâches à plus basse priorité de s'exécuter et ainsi commencer à vider les fifos. Ces valeurs (1s vs 1 ms) sont des exemples, c'est-à-dire qu'on peut ensuite faire varier ces valeurs pour valider la performance du système.~~

Finalement, notez que *TaskGenerate* écrit directement dans le fifo du TCB de la première tâche du routeur, c'est-à-dire *TaskComputing*².

Calcul (TaskComputing)

Elle doit répondre à plusieurs critères :

- Dans un premier temps, elle doit valider la provenance des paquets. Ainsi, tous les paquets provenant d'une plage d'adresse à rejeter doivent être rejetés. Ces plages vous sont fournies (les define qui débutent par REJECT_). Plutôt que de détruire le paquet rejeté immédiatement, celui-ci est écrit dans le fifo de TaskStats pour statistiques puis détruit une

¹ Valeur suggérée, mais à valider à la section 5.3.3

² Il s'agit d'un cas particulier d'usage de fifo beaucoup plus compact que l'usage du QPend. J'en glisserai un mot en classe. Pour l'instant, vous pouvez consulter le user manual p. 394 à 400.

fois que la tâche statistique a été exécutée (celle-ci s'exécute périodiquement toutes les 20 secondes).

- Une fois l'adresse vérifiée, on vérifie que le paquet n'a pas été corrompu. ~~Vous devrez implémenter plus loin la fonction CRC, pour l'instant on émule un appel à CRC par une attente active de 5 ticks d'horloge.~~
- Ensuite, les paquets doivent être envoyés dans différentes files selon le type de paquets (vidéo, audio et autres, respectivement 0, 1 et 2). Si jamais cette file est pleine, les paquets sont détruits immédiatement (c.-à-d. ils ne vont pas dans le fifo de TaskStats), mais le nombre de paquets rejeté pour chaque fifo est mémorisé dans une variable globale et celui-ci sera affiché par la tâche statistique (à toutes les 30 secondes).

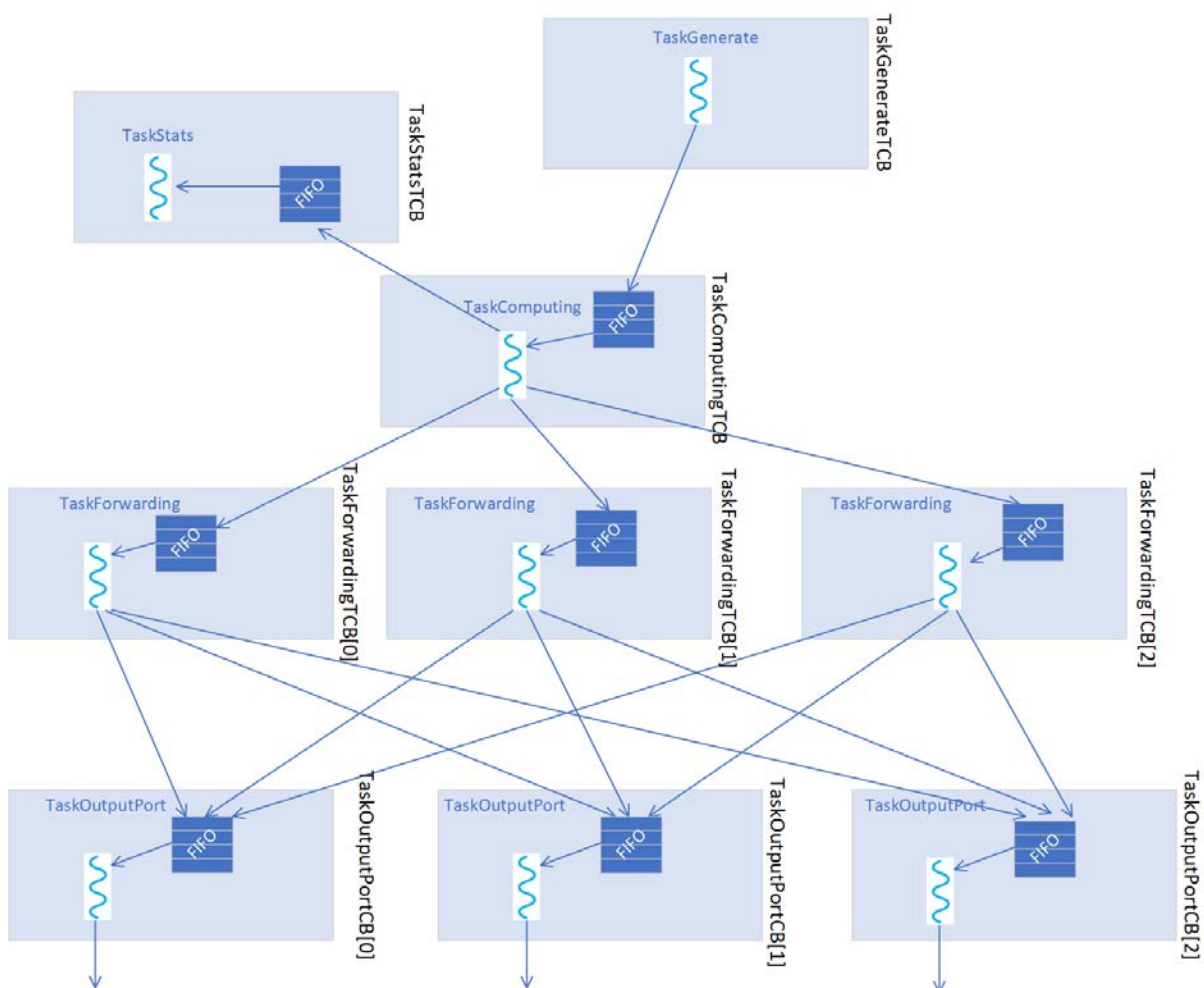


Figure 3 Flots de données dans le routeur

Forwarding (TaskFIFOForwarding)

- Cette tâche va lire dans les trois queues en respectant la priorité définie par cette qualité de service : les paquets vidéo sont plus prioritaires que les paquets audios qui sont eux-mêmes plus prioritaires que les autres paquets. Essayez ici de comprendre comment cette priorité est implémentée via un tableau de tâches `&TaskFIFOForwardingTCB[i]` (en lien avec la question no 2).
- La tâche va ensuite lire l'adresse de destination du paquet dans une table de routage. Dans les routeurs complexes, cette table est souvent réalisée en matériel. Ici, nous simplifions cette étape : le premier quart des adresses correspond à l'interface 1, le second quart correspond à l'interface 2, le troisième quart correspond à l'interface 3 et finalement le dernier quart à une diffusion (broadcast) sur les 3 interfaces.

- 0 `<= Destination < 1073741823` -> interface 1
- 1073741824 `<= Destination < 2147483647` -> interface 2
- 2147483648 `<= Destination < 3221225472` -> interface 3
- 3221225473 `<= Destination < 4294967295` -> BROADCAST

Ce qui en hexadécimal correspond à (lignes 46 à 53 de routeur.h):

```
#define INT1_LOW      0x00000000
#define INT1_HIGH     0x0FFFFFFF
#define INT2_LOW      0x10000000
#define INT2_HIGH     0x1FFFFFFF
#define INT3_LOW      0x20000000
#define INT3_HIGH     0x2FFFFFFF
#define INT_BC_LOW    0xC0000000
#define INT_BC_HIGH    0xFFFFFFFF
```

Attention: dans le cas des paquets broadcastés, vous remarquerez qu'on alloue de l'espace pour les paquets nouvellement créés.

- La tâche va finalement écrire dans la bonne interface. Ici les interfaces sont représentées par un tableau de tâches `&TaskOutputPortTCB[i]` où chaque tâche exécute un code identique `TaskOutputPort`. C'est dans le fifo de `&TaskOutputPortTCB[i]` (ici de taille 1) que `TaskFIFOForwarding` va écrire le paquet (similaire à l'écriture de `TaskGenerate`). Notez bien, si jamais la file est pleine, les paquets sont détruits immédiatement (c.-à-d. ils ne vont pas dans le fifo de `TaskStats`), mais le nombre de paquets rejeté pour chaque fifo est mémorisé dans une variable globale et celui-ci sera affiché par la tâche statistique (à toutes les ~~32~~0 secondes).

Interface de sortie (TaskOutputPort)

Cette tâche représente le périphérique d'arrivée ou un autre routeur. Ici on va simplement lire et imprimer les paquets. Ce qui nous sera utile pour valider le fonctionnement du routeur. Comme mentionné ci-haut on a un tableau de tâches *&TaskOutputPortTCB[i]* où chaque tâche exécute le code de *TaskOutputPort*. Chaque instance de tâche *TaskOutputPort* devra recevoir son id à la création qui servira à manipuler le bon mail box. Pour un exemple de tableau de tâches et de passage de paramètres à une tâche lors de sa création, inspirez-vous du programme *creation_de_taches.c*

Statistiques (TaskStats)

Cette tâche est réveillée toutes les ~~20~~30 secondes et procure un certain nombre d'informations depuis le dernier affichage ou encore depuis le début de l'exécution. Par exemple, *TaskStats* peut énumérer les paquets rejetés depuis les ~~20~~30 dernières secondes (*nbPacketSourceRejete*), mais aussi le nombre total de paquets rejetés depuis le début de l'exécution (*nbPacketSourceRejeteTotal*). De l'information sur l'utilisation des fifos est également donnée. Prenez le temps d'examiner.

En ce qui concerne l'assignation des priorités aux tâches

Il faut s'assurer de perdre le moins de paquets possible dans le fifo d'entrée et de minimiser l'utilisation des différents fifo (taille de la fifo). Nous avons fourni au départ une assignation que vous devrez justifier à la question no 3.

5. Travail à effectuer – séance no 1 :

5.1 On assume les tutoriels suivants ont été complétés de Vivado

Vivado : <https://moodle.polymtl.ca/course/view.php?id=2652#section-3>

SDK : <https://moodle.polymtl.ca/course/view.php?id=2652#section-3>

5.2 Routeur sur puce

Cette partie contient 3 étapes allant de 5.3.1 à 5.3.3.

5.3.1 Bien comprendre le code de départ

5.3.1.1 Importation et configurations du code de départ

Le code de départ vous est donné sur Moodle (section laboratoire no 1)

Pour importer les 2 fichiers dans SDK faire les 3 étapes suivantes :

- Tout comme pour l'application de Hello World sous SDK, créez une nouvelle application nommée lab1_part1. Cette application contiendra un fichier app.c que vous détruisez (delete). Puis cliquez sur le répertoire src de lab1_part1 pour le sélectionner (afin que les fichiers que vous allez importer à l'étape 3 soient automatiquement mis dans src).
- Tapez sous l'onglet File -> Import puis cliquez sur File System et Next (Figure 4).

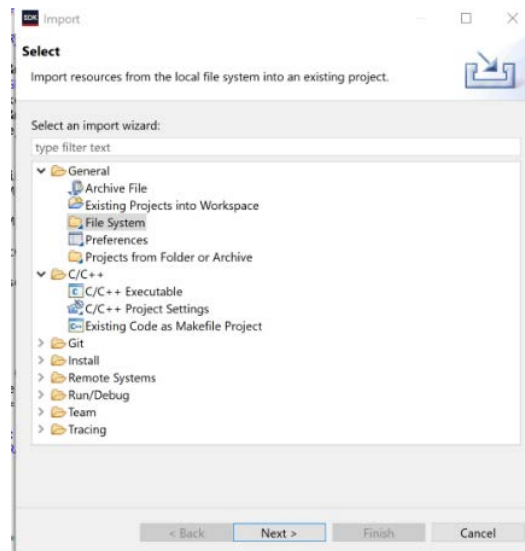


Figure 4

- Cliquez sur routeur.c et routeur.h et sur finish. Les 2 fichiers devraient apparaître dans src de lab1_part1 (Figure 5).

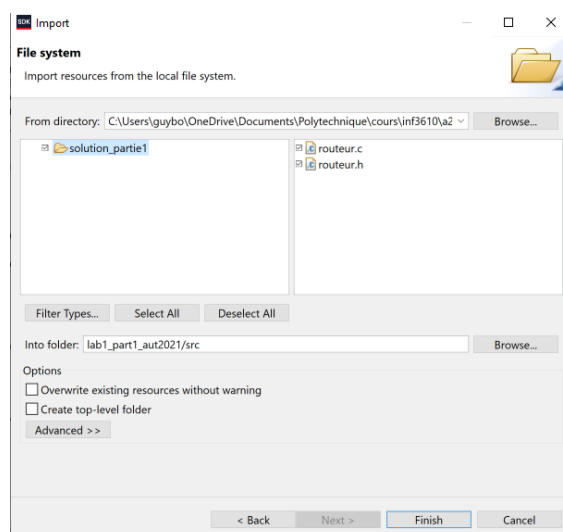


Figure 5

Une fois le code importé, nous allons faire 2 étapes de configuration. Nous allons d'abord nous assurer d'avoir les bons paramètres (variables globales) de uC/OS-III. Pour cela, faites ce qui suit :

- Dans l'onglet de gauche cliquez avec le bouton de droite sur *ucos_bsp_0* -> *Board Support Package Setting* (Figure 6), puis cliquer sur la fenêtre qui apparaîtra (Figure 7) cliquez sur *ucos_osiii* en ouvrant l'onglet 13. Assurez-vous que les variables globales *OS_CFG_MSG_POOL_SIZE* (le nombre d'éléments maximum quand on somme toutes les fifos) et *OS_CFG_TICK_RATE_HZ* (tick ou quantum) sont bien à 6000 et 1000 respectivement.

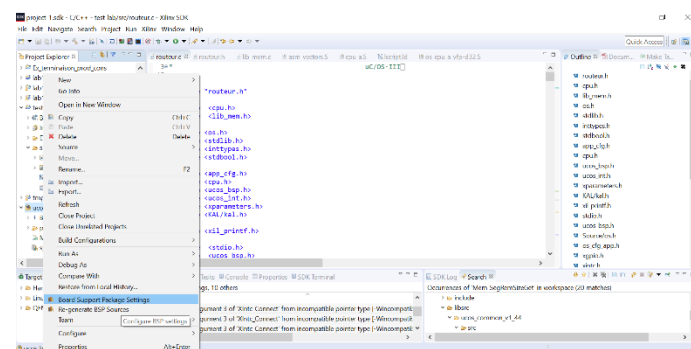


Figure 6 Pour atteindre certains paramètres du BSP

Board Support Package Settings					
Control various settings of your Board Support Package.					
Overview	Configuration for library: ucos_osiii				
ucos					
ucos_common					
ucos_osiii					
ucos_standalone					
drivers					
ps7_cortexa9_0					
Name	Value	Default	Type	Description	
01. MISCELLANEOUS					
02. EVENT FLAGS					
03. MEMORY PARTITIONS					
04. MUTEXES					
05. QUEUES					
06. SEMAPHORES					
07. STATISTICS TASK					
08. TASKS					
09. TIME					
10. TLS					
11. TIMERS					
12. TRACE					
13. APPLICATION					
OS_CFG_IDLE_TASK_STK_SIZE	256	256	integer	Stack size of the idle task (number of bytes)	
OS_CFG_ISR_STK_SIZE	512	512	integer	Stack size of ISR stack (number of bytes)	
OS_CFG_MSG_POOL_SIZE	6000	48	integer	Maximum number of messages in the message pool	
OS_CFG_STAT_TASK_Prio	62	62	integer	Priority of the statistic task	
OS_CFG_STAT_TASK_RATE_HZ	10	10	integer	Rate of execution of the statistic task	
OS_CFG_STAT_TASK_STK_SIZE	256	256	integer	Stack size of the statistic task (number of bytes)	
OS_CFG_TASK_STK_LIMIT_PCT_EMP	10	10	integer	Internal Task stack limit in percentage	
OS_CFG_TICK_RATE_HZ	1000	1000	integer	Tick rate in Hz	
OS_CFG_TMR_TASK_Prio	61	61	integer	Priority of the timer task	
OS_CFG_TMR_TASK_RATE_HZ	10	10	integer	Rate for timers in OS Ticks per second	
OS_CFG_TMR_TASK_STK_SIZE	256	256	integer	Stack size of the timer task (number of bytes)	

Figure 7 Deux paramètres en particulier : nombre d'éléments maximum quand on somme tous les fifos et fréquence de l'horloge de uC/os-III (tick ou quantum en HZ)

Finalement, comme deuxième configuration, il faut augmenter le *heap*. En effet, puisque nous allons utiliser le *malloc* lors de la génération de paquets, il est possible qu'après avoir alloué un grand nombre de paquets avant de faire des *free*, vous ayez un manque de mémoire pour allouer de nouveaux paquets³. Prenez note que *malloc* vous renvoie dans ce cas un pointeur *nul*.

Pour augmenter le heap éditer le fichier *lscript.ld* (pour linker script et situé dans le même répertoire que *router.h* et *routeur.c*) puisse faite passer le *heap size* à *0xA00000* (plutôt que *0x2000*) comme illustré à la figure 8. Si parfois vous en manquez (pointeur nul lors de *malloc*).

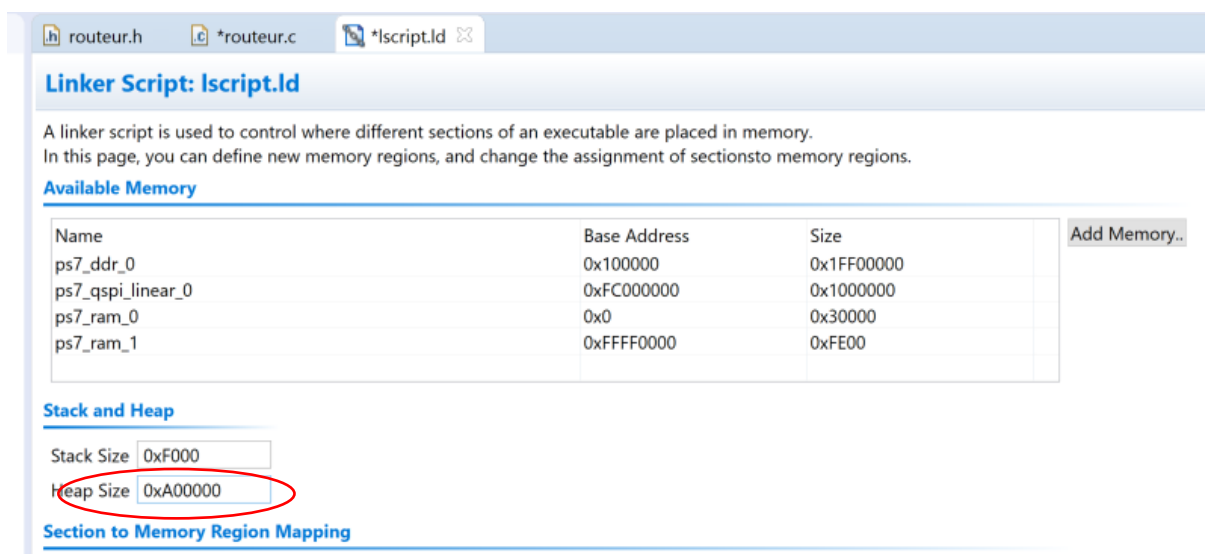



Figure 8

5.3.1.2 Validation fonctionnelle du code

Cette section vous sera d'abord utile pour comprendre le fonctionnement du routeur. **De plus, quand par la suite vous ajouterez de nouvelles fonctionnalités, vous pourrez revenir à cette section pour valider votre routeur modifié.**

D'abord, nous allons nous assurer que le délai alloué pour vider les différents fifos lors du mode attente de *TaskGenerate* est suffisant. Remarquez les variables *delai_pour_vider_les_fifos_sec* et *delai_pour_vider_les_fifos_msec* de *routeur.h* égalent à ~~0~~1 et ~~500~~0 respectivement. Ce qui permet 2 secondes comme temps d'attente. Ce délai de ~~2~~1 secondes est demandé à la **ligne 245** de *TaskGenerate*. Voici donc ce que vous devez faire pour vous assurer que les fifos se vident bel et bien.

³ Par exemple quand on garde dans le fifo de *TaskStats* un certain nombre de paquets rejetés avant de les rendre (*free*).

- i. Activez la macro *safeprint* c'est-à-dire enlever les commentaires en faisant passer le *define* de FULL_TRACE de 0 à 1 (permettant d'activer les traces dans votre programme) et recompilez.
- ii. Mettre **point d'arrêt** à la **ligne 250** de la tâche TaskGenerate c.-à-d. à :
safeprintf("\n***** DEMARRAGE \n\n").
- iii. Mettre **point d'arrêt** à la **ligne 240** de la tâche TaskGenerate c.-à-d. à :
isGenPhase = false;
- iv. Démarrez l'exécution avec *Debug Configuration* et dans l'onglet *Application* de la fenêtre *Debug Configuration* en vous assurant d'avoir activé *Stop at program entry*
- v. Assurez-vous que votre SDK terminal est bien connecté.
- vi. Démarrez une première fois l'exécution de votre routeur avec *Resume*  qui devrait s'arrêter à la ligne 87 (1^{er} instruction du *main()*).
- vii. Cliquez sur *Resume* une 2^e fois et vous devriez voir apparaître l'entête pour affichage des statistiques et le programme arrêté à la **ligne 250**. Vous devriez aussi voir l'impression de :

***** RAFALE No 1 DE 57 PAQUETS DURANT LES 57 PROCHAINES MILLISECONDES
- viii. Cliquez pour un 3^e *resume* et le code devrait s'exécuter jusqu'à la **ligne 250** et vous devriez voir la trace :

***** FIN DE LA RAFALE No 1.

L'idée ici est de comprendre ce qui s'est passé entre le 2e et 3e resume. Pour cela, commencez en défilant l'écran vers le haut et revenez à ***** RAFALE No 1 DE 57 PAQUETS.

À partir d'ici voici 10 points à bien comprendre (et n'hésitez pas à poser des questions au chargé de laboratoire si ce n'est pas clair pour vous):

1. Vous devriez voir les 57 paquets générés avec pour chaque paquet généré la trace qui donne le pointeur sur le paquet, le id du paquet, la source du paquet et la destination du paquet et finalement la priorité du paquet via type (voir p.e. *Generation du Paquet # 1* à la Figure 9). Cette information vous sera utile quand vous voudrez vérifier qu'un paquet est bien rendu à destination plus bas à l'étape 6 (c.-à-d. dans la bonne interface). Vous vous apercevrez que bien *TaskGenerate* a la plus haute priorité par rapport à *TaskComputing*, ce dernier arrive quand même à s'exécuter et remplir les fifos du routeur. Rappelez-vous que nous sommes dans un monde préemptif et que dès que *TaskGenerate*

bloque, *TaskComputing* peut démarrer s'il a suffisamment de priorité. **Répondez à la question no 1.**

```
***** RAFALE No 1 DE 57 PAQUETS DURANT LES 57 PROCHAINES MILLISECONDES

***** DEMARRAGE

TaskGenerate : *****Generation du Paquet # 1 *****
ADD 18C108
  ** id : 1
  ** src : 1ED5B4A6
  ** dst : CC33F38
  ** type : 0
```

Figure 9

2. Vous devriez donc observer de temps en temps une trace qui indique qu'un paquet a été mis (produit) dans le fifo associé à *TaskComputing* appelé *inputQ* avec :
Nb de paquets dans inputQ - apres production de TaskGenerate...
3. Vous devriez également observer de temps en temps une trace qui indique qu'un paquet a été lu (consommé) du fifo associé à *TaskComputing* avec :
Nb de paquets dans inputQ - apres consommation de TaskComputing...
4. Vous devriez également observer qu'un paquet obtenu précédemment en 3 a été transféré dans une des 3 fifos de priorités avec par exemple :
Nb de paquets dans highQ - apres production de TaskComputing:...
5. Finalement juste avant la fin de la trace (Figure 10), vous devriez voir que le fifo *inputQ* contient 0 paquet, alors que *highQ* contient 17 paquets, *mediumQ* 17 paquets et *lowQ* 16 paquets.
6. Nous allons maintenant vérifier si un délai de 2 secondes est bel et bien suffisant pour vider les fifos. Cliquez à nouveau sur *Resume*. Vous devriez vous rendre jusqu'à l'impression de ***** RAFALE No 2 DE 94 PAQUETS. À partir de là, remontez l'écran vers le haut, jusqu'à l'endroit où vous étiez avant ce dernier *Resume* c'est-à-dire ***** FIN DE LA RAFALE No 1 (juste après la génération du paquet #57).
7. D'abord on voit à la figure 11 que le nombre de paquets à vider n'est pas comme vu au point 5 de 17 paquets pour *highQ*, 17 paquets pour *mediumQ* et 16 paquets pour *lowQ*, mais plutôt 18, 17 et 17 paquets respectivement. C'est sans doute à cause de *xil_printf* retardataires...

Nb de paquets dans mediumQ - apres production de TaskComputing: 17

Nb de paquets dans inputQ - apres production de TaskGenerate: 1

Nb de paquets dans inputQ - apres consommation de TaskComputing: 0

TaskGenerate : *****Generation du Paquet # 55 *****
ADD 199DC8
** id : 55
** src : ED5A038A
** dst : 2443216E
** type : 0

Nb de paquets dans lowQ - apres production de TaskComputing: 15

Nb de paquets dans inputQ - apres production de TaskGenerate: 1

Nb de paquets dans inputQ - apres consommation de TaskComputing: 0

TaskGenerate : *****Generation du Paquet # 56 *****
ADD 199E18
** id : 56
** src : 4CA3BC5A
** dst : 7D34D8B4
** type : 2

Nb de paquets dans highQ - apres production de TaskComputing: 17

Nb de paquets dans inputQ - apres production de TaskGenerate: 1

Nb de paquets dans inputQ - apres consommation de TaskComputing: 0

TaskGenerate : *****Generation du Paquet # 57 *****
ADD 199E68
** id : 57
** src : 9323A8BC
** dst : E6A652A4
** type : 0

Nb de paquets dans lowQ - apres production de TaskComputing: 16

Nb de paquets dans inputQ - apres production de TaskGenerate: 1

Nb de paquets dans inputQ - apres consommation de TaskComputing: 0

***** FIN DE LA RAFALE No 1

Figure 10

```

***** FIN DE LA RAFALE No 1

--TaskForwarding: paquets 1 envoyes

Nb de paquets dans highQ - apres production de TaskComputing: 18

Nb de paquets dans HighQ - apres consommation de TaskFowarding: 17

Nb de paquets dans MediumQ - apres consommation de TaskFowarding: 17

-- dispatch_packet - Paquet arrive dans interface 3

-- dispatch_packet - Paquet arrive dans interface 1

```

Figure 11

8. Ensuite vous devriez voir en descendant la fenêtre que *highQ* passe de 18 à 16 (Figure 13), puis à 15 puis si vous continuez à défiler vers le bas, *highQ* va passer à 13, etc. Et toujours en défilant vers le bas, une fois que *highQ* est à 0, ce sera au tour de *mediumQ* de se vider, puis plus loin de *lowQ*. Si vous ne le voyez pas, demandez au chargé de laboratoire.

Notez que pendant que *highQ* se vide, il n'est pas impossible que *mediumQ* se vide aussi à l'occasion. Encore une fois, rappelez-vous bien que nous sommes dans un monde préemptif, dès que la tâche qui vide *HighQ* (*TaskFIFOForwardingTCB[PACKET_VIDEO]*) est en attente de quelque chose, il est possible que la tâche qui vide *mediumQ* (*TaskFIFOForwardingTCB[PACKET_AUDIO]*) démarre le temps que *TaskFIFOForwardingTCB[PACKET_VIDEO]* reprenne son exécution. Le même phénomène est possible entre *mediumQ* et *lowQ*.

Finalement, vous devriez aussi observer qu'au fur et à mesure que les fifos se vident selon leur priorité, les paquets arrivent à destination dans la bonne interface (p.e. *Paquet reçu en 2* sur la figure 12 devrait être identique au paquet 2 généré par *TaskGenerate*) et sont libérés. Il est aussi possible qu'un même paquet soit envoyé aux 3 interface si son adresse était dans l'intervalle broadcast.

9. Si vous poursuivez la descente vers le bas (jusqu'à ***** RAFALE No 2 DE 213 PAQUETS), vous devriez observer que la trace *Nb de paquets dans HighQ - apres consommation de TaskFowarding: 0* est atteinte en premier, puis *Nb de paquets dans MediumQ - apres consommation de TaskFowarding: 0*, puis finalement rendu presque en bas rendu, vous verrez *Nb de paquets dans LowQ - apres consommation de TaskFowarding: 0*.

```

Nb de paquets dans highQ - apres production de TaskComputing: 18

Nb de paquets dans HighQ - apres consommation de TaskFowarding: 17

Nb de paquets dans MediumQ - apres consommation de TaskFowarding: 17

-- dispatch_packet - Paquet arrive dans interface 3

-- dispatch_packet - Paquet arrive dans interface 1

Paquet reçu en 2
** id : 2
>> src : 5DED9056
>> dst : A29DC096
>> type : 0

Paquet reçu en 0
** id : 4
>> src : 6EF33EC2
>> dst : 1755C2AC
>> type : 1

--TaskForwarding: paquets 3 envoyes

--TaskForwarding: paquets 3 envoyes

Nb de paquets dans HighQ - apres consommation de TaskFowarding: 16

Nb de paquets dans MediumQ - apres consommation de TaskFowarding: 16

-- dispatch_packet - Paquet BC arrive dans tous les interfaces

-- dispatch_packet - Paquet arrive dans interface 3

Paquet reçu en 0
** id : 3
>> src : C749E8F2
>> dst : E8BB2A04
>> type : 0

```

Figure 12

En conclusion: ça donc dire que durant la génération d'une rafale de *TaskGenerate*, *TaskComputing* a le temps⁴ de faire de commencer à aiguiller des paquets dans le fifo entre *TaskGenerate* et *TaskComputing*. Par contre, *TaskFIFOForwarding* et *TaskOutputPort* ne peuvent s'exécuter durant cette rafale, mais ensuite durant un délai

⁴ Notez aussi que pour la première rafale on a 57 paquets , or 255 >> 57 donc le fifo devrait être davantage sollicité dans les prochaines rafales.

passif de 2 secondes de TaskGenerate, *highQ*, *mediumQ* et *lowQ* se vident. Et c'est précisément à ça que sert le délai d'attente de 2 secondes⁵. On verra plus loin qu'il est possible de diminuer ce dernier dans l'ordre des ms.

Évidemment vous pourriez poursuivre la trace de manière similaire pour la rafale #2, etc.

10. Dernier point à vérifier : que le nombre et la liste de paquets rejetés pour mauvaise adresse source fonctionnent correctement. Vous allez d'abord remettre FULL_TRACE à 0. Puis mette *print_paquets_rejetes* = 1 dans *routeur.h*. Si vous exécutez à nouveau votre code, après chaque impression de statistiques, vous devrez voir apparaître la liste de paquets rejetés (si c'est le cas). Ces paquets sont ensuite remis en mémoire (*free*). Vous pouvez ensuite remettre FULL_TRACE à 1.

⁵ Ligne 213 OSTimeDlyHMSM(0, 0, delai_pour_vider_les_fifos_sec, delai_pour_vider_les_fifos_msec, OS_OPT_TIME_HMSM_STRICT, &err);

5.3.2 1^{ère} fonctionnalité à ajouter : le *timestamp*

On souhaite ici connaître le délai maximum que prend un paquet entre le moment où il entre dans routeur (*TaskComputing*) et qu'il arrive à destination de l'interface (*TaskOutputPort*). On pourrait utiliser le tick d'horloge de uC/OS-III (e.g. 1 ms) et donc mettre un champ *timestamp* à paquet avec la valeur de *OSTimeGet()* à sa création, pour qu'à l'arrivée du paquet à sa destination (interface) il ne reste qu'à faire la différence entre la valeur courante (à nouveau via *OSTimeGet()*) et le champ *timestamp*.

Mais on désire avoir quelque chose de plus précis encore que la milliseconde. On va donc utiliser le *timestamp* (*ts*) offert par l'horloge du ARM (qui est dans l'ordre des us).

- 1) Par conséquent, si vous ajoutez dans la tâche *StartupTask* (juste avant la création des tâches uC du routeur) l'appel suivant :

```
freq_hz = CPU_TS_TmrFreqGet(&err); /* Get CPU timestamp timer frequency. */  
xil_printf("\nfreq du timestamp: %d\n", freq_hz);,
```

vous verrez lors du *xil_printf* que vous disposez d'une fréquence encore plus fine que le tick d'horloge de uC qui est de 1/325 MHz. Elle provient de l'horloge du ARM (650 MHz) divisée par 2. Il faut s'avoir qu'à 325 MHz un registre 32 bits va vite se remplir (*overflow* en quelques minutes), on va travailler avec du 64 bits (on va donc utiliser le type uC CPU_TS64).

- 2) Dans le fichier *routeur.c*, assurez-vous de bien comprendre le rôle des variables suivantes:

```
#define LLONG_MAX 9223372036854775807    // valeur maximum pour 64 bits  
CPU_TS_TMR_FREQ freq_hz;                // Pour connaître la fréquence de mis à  
                                          // jour du timestamp au départ  
                                          // (sert aussi au code décrit plus haut dans startup)  
CPU_TS64 max_delay_video = 0L;           // On part avec le délai minimum  
float max_delay_video_float;             // Equivalent float pour le xil_printf  
CPU_TS64 max_delay_audio = 0L;           // On part avec le délai minimum  
float max_delay_audio_float;             // Equivalent float pour le xil_printf  
CPU_TS64 max_delay_autre = 0L;           // On part avec le délai minimum  
float max_delay_autre_float;             // Equivalent float pour le xil_printf
```

- 3) Également notez bien la présence du timestamp la définition du paquet dans les fichiers *routeur.h*:

```
typedef struct {
    unsigned int src;
    unsigned int dst;
    unsigned int type;
    CPU_TS64 timestamp;
    unsigned int crc
    unsigned int data[10];
} Packet;
```

Notez que le paquet a au total 16 mots.

- 4) Maintenant allez dans TaskGenerate et juste après avoir créé le paquet (ou juste avant ++nbPacketCrees), initialisez la valeur du *timestamp* avec :

```
packet->timestamp = CPU_TS_Get64();
```

- 5) Puis, juste après la fonction ***create_events()*** créer et compléter (voir les 2 commentaires **À compléter**) la fonction *Update_TS* :

```
void Update_TS(CPU_TS64 TS, PACKET_TYPE type) {
    CPU_TS64 delay;

    delay = CPU_TS_Get64() - TS; // Valeur courante - valeur initiale
                                // à la création du paquet

    if (delay < 0) {xil_printf("Attention overflow\n");}
    else {
        À compléter
    }
}
```

- 6) Dans ***dispatch_packet()*** juste avant de déposer le paquet dans la FIFO de *TaskOutputPort*, appelez la fonction *Update_TS()* du point 6.

- 7) Dans *TaskStats*, ajouter après la série de *xil_printf* les 2 lignes suivantes :

```
max_delay_video_float = (float) max_delay_video / (float) freq_hz;
xil_printf("21- Pire temps video ");
printf("%.10f", max_delay_video_float);
xil_printf("\r\n");

Similaire pour audio et autre.
```

5.3.3 Analyse de performance avec timestamp

La section 5.3.1 est ce qu'on appelle une validation fonctionnelle. Ici, nous allons nous intéresser à une validation de performance. Le délai de 1 seconde pour vider les différents fifos lors du mode attente de *TaskGenerate* à la section 5.3.1 vous a été suggéré dans le code de départ, car nous savions que les traces (*xil_printf*) ralentissent l'exécution. Toutefois, ceci est acceptable pour faire un test de validation fonctionnelle, mais pas dans un test de performance. Par conséquent, sachant que votre routeur fonctionne correctement grâce à la validation de la section 5.3.1, assurez-vous avant les tests de performance d'avoir FULL_TRACE à 0.

L'objectif de ce test de performance est d'observer 3 délais pour vider les différents fifos (200 ms, 100ms et 50ms) tout en observant pour chaque délai les statistiques suivantes : nombres de paquets maximums dans les différents fifo (no 10 à 13) et les pire temps d'exécution (no 20 à 23).

Conservez donc les 3 captures d'écran de l'exécution comme celui de la figure 13 pour lequel on a testé un délai de 500 ms.

Répondez aux question 2 et 3 (voir plus bas).

```
----- Affichage des statistiques -----  
Delai pour vider les fifos sec: 0  
Delai pour vider les fifos msec: 500  
Frequence du systeme: 1000  
1 - Nb de packets total crees : 100270  
2 - Nb de packets total traitees : 89755  
3 - Nb de packets rejetees pour mauvaise source : 373  
4 - Nb de packets rejetees pour mauvaise source Total: 9068  
5 - Nb de packets rejetees pour mauvais CRC : 41  
6 - Nb de packets rejetees pour mauvais CRC total : 868  
7 - Nb de paquets rejetees dans fifo d'entree: 0  
8 - Nb de paquets rejetees dans 3 Q: 0  
9 - Nb de paquets rejetees dans l'interface de sortie: 0  
  
10 - Nb de paquets maximum dans le fifo d'entree : 251  
11 - Nb de paquets maximum dans highQ : 84  
12 - Nb de paquets maximum dans mediumQ : 85  
13 - Nb de paquets maximum dans lowQ : 80  
  
14 - Nb de paquets maximum dans port0 : 0  
15 - Nb de paquets maximum dans port1 : 0  
16 - Nb de paquets maximum dans port2 : 0  
  
17- Message free : 4425  
18- Message used : 575  
19- Message used max : 641  
20- Nombre de ticks depuis le début de l'execution 782760  
21- Pire temps video '0.2519957721'  
22- Pire temps audio '0.2523134947'  
23- Pire temps autre '0.2525891662'  
----- Task stop suspend all tasks -----  
----- Flags: 0 -----
```

Figure 13

5.3.4 2^{ème} fonctionnalité à ajouter : allocation de blocs de mémoire statique et test de performance

Une application peut allouer et libérer de la mémoire dynamique à l'aide respectivement des fonctions *malloc()* et *free()* de n'importe quel compilateur ANSI C. Cependant, utiliser *malloc()* et *free()* dans un système temps réel embarqué avec contraintes durs peut être risqué. À terme, il pourrait ne pas être possible d'obtenir une seule zone mémoire contiguë en raison de la fragmentation. Comme vu en INF2610 (Fig. 14), la fragmentation est le développement d'un grand nombre de zones libres distinctes (c'est-à-dire que la mémoire libre totale est fragmentée en petits morceaux non contigus). Le temps d'exécution de *malloc()* et *free()* est généralement non déterministe étant donné les algorithmes utilisés pour localiser un bloc contigu de mémoire libre suffisamment grand pour satisfaire une requête *malloc()*. μ C/OS-III fournit une alternative à *malloc()* et *free()* en permettant à une application d'obtenir des blocs mémoire de taille fixe à partir d'une partition constituée d'une zone mémoire contiguë, comme illustré dans la figure ci-dessous. Tous les blocs de mémoire ont la même taille et la partition contient un nombre entier de blocs. L'allocation et la désallocation de ces blocs mémoires s'effectuent en temps constant et sont déterministes. La partition elle-même est généralement allouée de manière statique (sous forme de tableau), mais peut également être allouée à l'aide de *malloc()* tant qu'elle n'est jamais libérée.

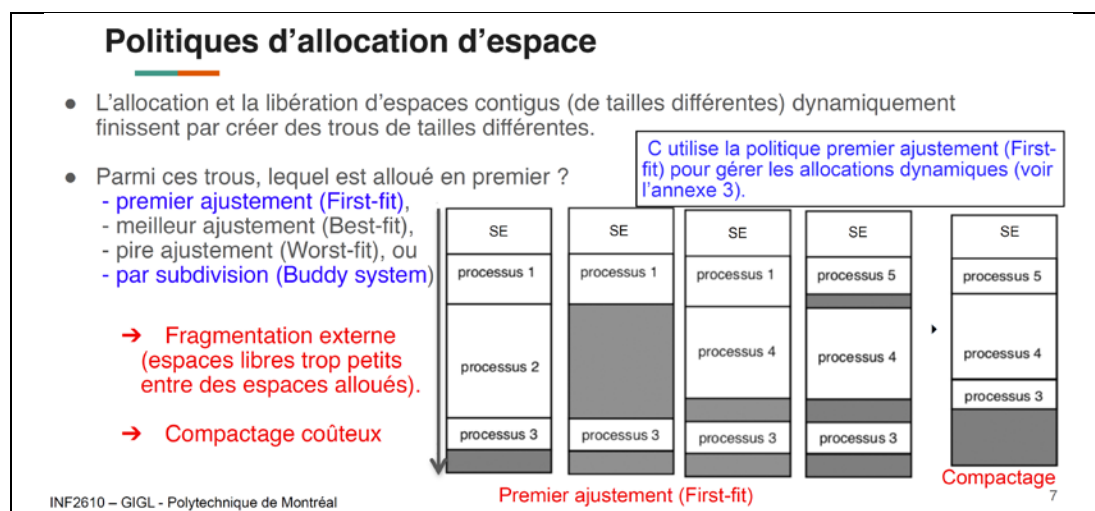


Figure 14

Regardez attentivement l'exemple donné en classe *Ex_Prod_Cons_Mem_Static_Allocation* afin de bien comprendre comment on peut remplacer *malloc* et *free* par les 3 fonctions μ C/OS-III *OSMemCreate()*, *OSMemGet()* et *OSMemPut()*⁶.

Utilisez 10000 paquets de chacun 16 mots de 32 bits :

```
OSMemCreate(&BBlockMem, "BBlockMem", &Tab_Block[0][0], 10000, sizeof(Packet) *  
sizeof(CPU_INT32U), &err);
```

Puis, remplacer tous les *malloc()* par *OSMemGet()* et tous les *free()* par *OSMemPut()*.

⁶ Page 407 à 418 du User Manuel de μ C/OS-III (disponible sur le site web)

Attention ! pour vous assurez de ne pas allouer un pointeur nul au pointeur *packet* vous devez utiliser un sémaphore compteur (voir page 417).

De manière similaire à la section 5.3.3, refaites le test pour un délai de 100 ns pour vider les différents fifos et comparez votre résultat avec celui de la section 5.3.3. Répondez à la **question no 4.**

6. Questions pour le rapport

Question 1 À la dernière phrase de l'étape 1 (page 11), on dit : *Rappelez-vous que nous sommes dans un monde préemptif et que dès que TaskGenerate bloque, TaskComputing peut démarrer s'il a suffisamment de priorité.*

- a) Proposez une manière de vérifier que TaskGenerate est bloqué (*Pend*)
- b) Dans l'affirmative de a), qu'est-ce qui peut bloquer TaskGenerate

Question 2 Concernant la section 5.3.2, justifiez l'endroit où vous avez inséré la fonction `Update_TS()`. Plus précisément, pourquoi ne pas l'avoir mise dans la tâche `TaskOutput()` ?

Question 3 Concernant la section 5.3.3, vous aurez remarqué que le pire temps (que ce soit video, audio ou autre) est de plus de 1 sec.

- a) Est-ce possible que ce délai provienne de l'attente active qui simule du temps de calcul. Expliquez.
- b) Mettez en commentaire cette attente active (ou à 0) et refaites les 3 tests de la section 5.3.3 (200 ms, 100ms et 50ms). Présenter les 3 résultats dans le format de la figure 13.
- c) Toujours avec un délai d'attente active 0 pour TaskComputing refaites le test mais de 100 ms mais avec un tick d'horloge de 3000 Hz. Comparez votre résultat avec b), est- plus rapide ? P.S. Ramenez ensuite l'horloge à 1000Hz.
- d) Dans la vraie vie, 1 tick est probablement trop pour simuler un temps de calcul (attente active), comment aurait-on pu faire pour simuler par exemple un temps actif par exemple 1/10 de tick.

Question 4 Concernant la section 5.3.4.

- a) Pourquoi ne voit-on pas de grosse différence dans TaskStats entre l'utilisation de la mémoire dynamique et statique ?
- b) Que se passe-t-il si au lieu d'avoir 10,000 paquets dans mon tas, j'en ai 5,000. Montrez comment la performance est alors dégradée. En ce sens peut-on aussi dire que 10,000 paquets est une bonne approximation ?

7. Barème et date de remise

Barème Lab 1 Partie 1

Questions	Notes	Commentaires
Q1 sur x point		
Q2 sur y point		
Q3 sur z point		
Q4 sur u point		
Fonctionnement sur v point		
Total sur 8		

Date de remise : Jeudi 15 février avant minuit