

Polytechnique de Montréal - DGIGL

Laboratoire #1: Routeur sur puce FPGA

INF3610 – Automne 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

- Validation fonctionnelle et analyse de performance
- Utilisation d'un timestamp pour le calcul du délai maximum d'un paquet dans le routeur
- Évaluation de performance pour le délai maximum d'un paquet avec différents ticks d'horloge
- Calcul d'un délai minimum pour vider le fifo Queuing
- Utilisation de sections critiques avec Mutex et Sémaphore

Séance no 2 :

- Interruption et écriture de drivers (détails à venir)

Séance no 3 :

- Mécanismes de partage de tâches du routeur sur 2 processeurs en mode AMP (détails à venir)

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.

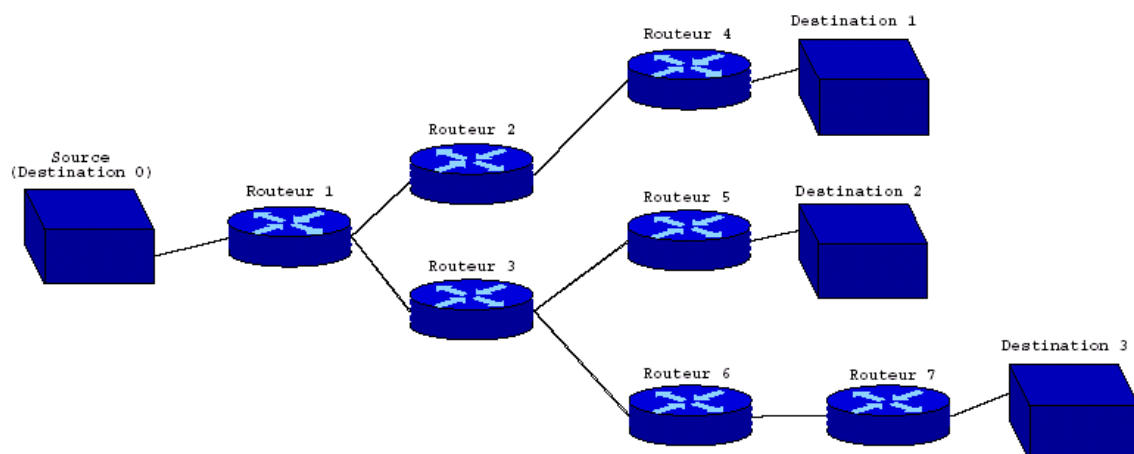


Figure 1 Exemple d'un routeur de paquets simplifié

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.

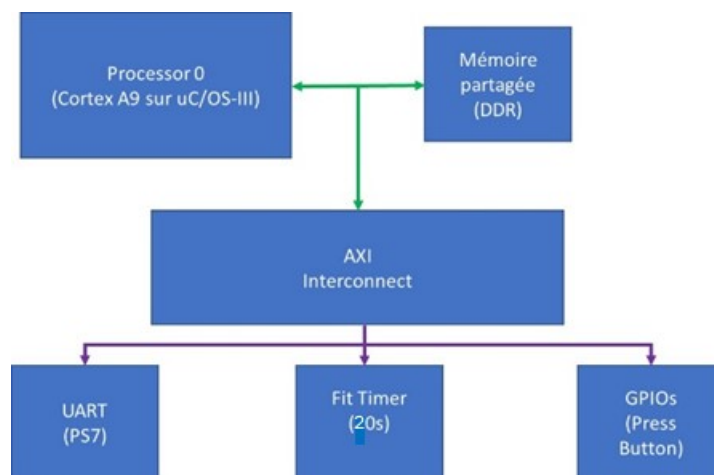


Figure 2 Architecture ciblée

4. Fonctionnement général et description des tâches

La **figure 3** illustre le flot des paquets à l'intérieur du routeur pour lequel il y a 3 niveaux de qualité de services et 3 nœuds de destination. 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.

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:

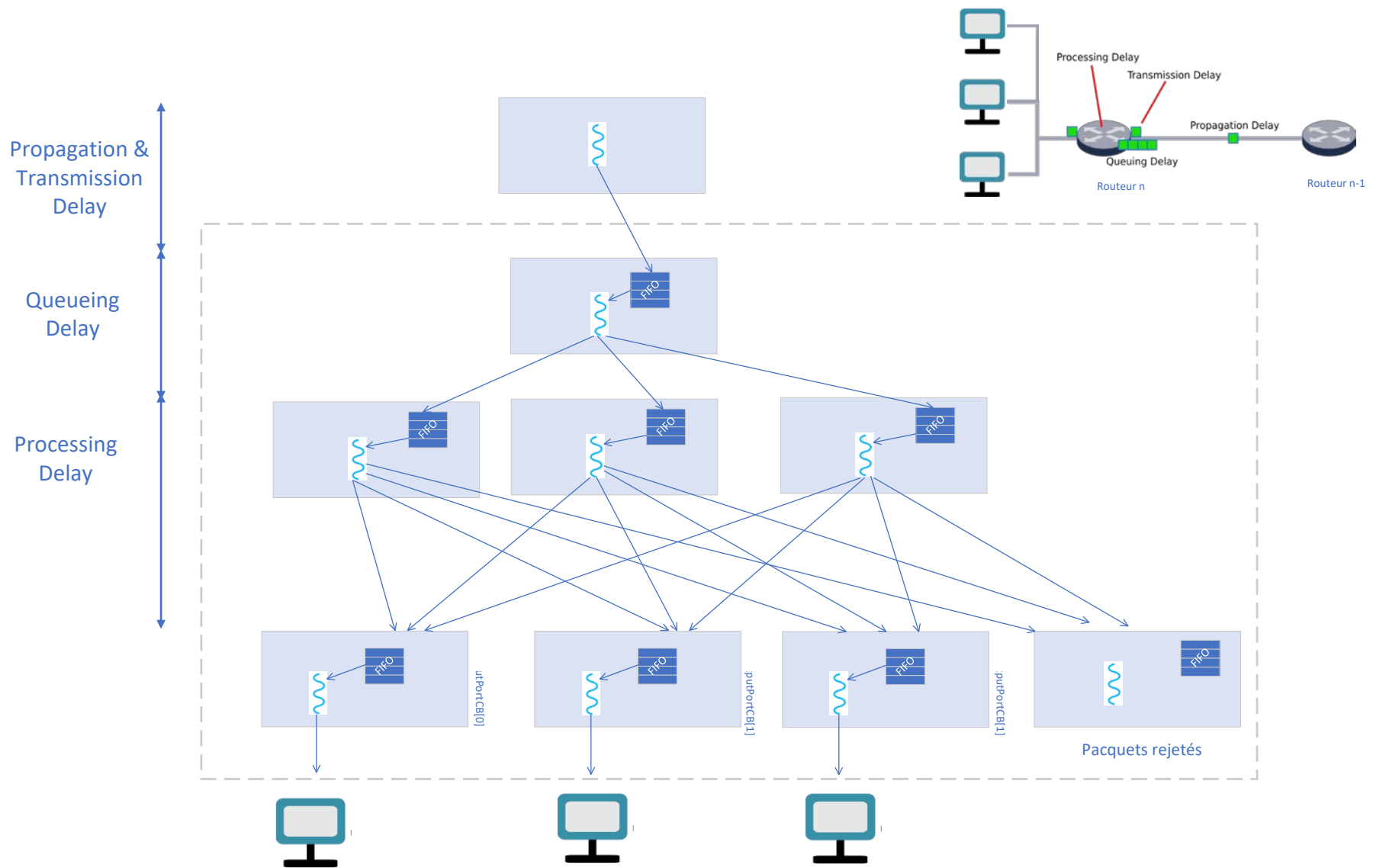


Figure 3 Flots de données dans le routeur

Interface d'entrée/génération des paquets (TaskGenerate)

Cette tâche s'occupe de générer des paquets aléatoires qui seront traités par le routeur. Elle 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 sec ¹ (*OSTimeDlyHMSM(0, 0, 1, 0, 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.

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

Queuing (TaskQueuing)

Ici *TaskQueuing* s'occupe d'envoyer les paquets dans les différents fifos d'un tableau de 3 TCB en ordre de priorité de traitement via le type de paquets (vidéo, audio et autres, respectivement *TaskComputing[0]*, *TaskComputing[1]* et *TaskComputing[2]*).

Selon la variable de compilation *FULL_TRACE*, si jamais la fifo sélectionnée est pleine, les paquets sont détruits immédiatement (valeur de 0) et sinon (valeur de 1) les paquets sont mis dans le fifo de *TaskStats* pour impression (si la variable *print_paquets_rejetes* est à vrai³) lors de l'exécution périodique de la tâche *TaskStats* (toutes les 30 secondes). Dans tous les cas, le nombre de paquets rejetés est mémorisé dans une variable globale et celui-ci sera affiché par la tâche statistique.

Computing (TaskComputing)

Selon son ordre de priorité, chaque instance de la tâche *TaskComputing* dans le tableau *TaskComputing[]* devra réaliser les étapes suivantes :

- Dans un premier temps, vérifier que le paquet n'a pas été corrompu à l'aide du CRC.
- Ensuite, 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_*). Tout comme pour les fifos pleins de dans *TaskQueuing*, selon la variable de compilation *FULL_TRACE*, les paquets sont détruits immédiatement (valeur de 0) et sinon (valeur de 1) les paquets sont mis dans le fifo de *TaskStats* pour impression (si la variable *print_paquets_rejetes* est à vrai) lors de l'exécution périodique de la tâche *TaskStats* (toutes

¹ Valeur suggérée pour démarrer, mais à minimiser à la section 5.2.3.1

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.

³ Peut servir au débogage...

les 30 secondes). Dans tous les cas, le nombre de paquets rejetés est mémorisé dans une variable globale et celui-ci sera affiché par la tâche statistique.

- 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
```

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

- Et 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*). Encore une fois, selon la variable de compilation *FULL_TRACE*, si jamais la fifo de *TaskOutputPort* sélectionnée est pleine, les paquets sont détruits immédiatement (valeur de 0) et sinon (valeur de 1) les paquets sont mis dans le fifo de *TaskStats* pour impression (si la variable *print_paquets_rejetes* est à vrai) lors de l'exécution périodique de la tâche *TaskStats* (toutes les 30 secondes). Dans tous les cas, le nombre de paquets rejetés est mémorisé dans une variable globale et celui-ci sera affiché par la tâche statistique.

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 box⁴. 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)

Mentionnez à 3 reprises ci-haut, cette tâche est réveillée toutes les 30 secondes et procure un certain nombre d'informations depuis le dernier affichage ou encore depuis le début de l'exécution dont les paquets rejetés depuis les 30 dernières secondes (*nbPacketSourceRejete*), mais aussi le nombre total de paquets rejetés depuis le début de l'exécution (*nbPacketSourceRejeteTotal*).

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) tout en respectant les priorités de paquet. Nous avons fourni au départ une assignation que vous devrez justifier à la question no 1, c'est-à-dire après avoir bien compris le fonctionnement du routeur à partir du code de départ.

⁴ Nom donné à un fifo de profondeur 1

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 5.2.1 à 5.2.3.

5.2.1 Bien comprendre le code de départ

5.2.1.1 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).

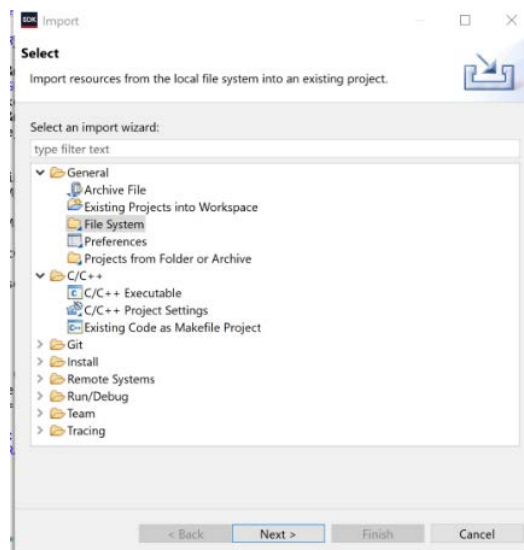


Figure 4

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

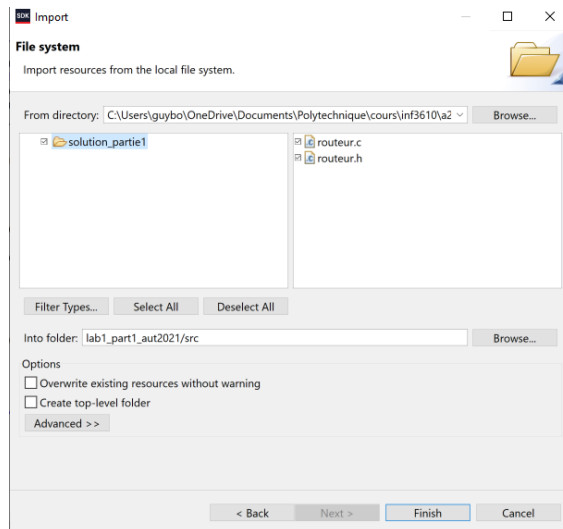


Figure 5

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.

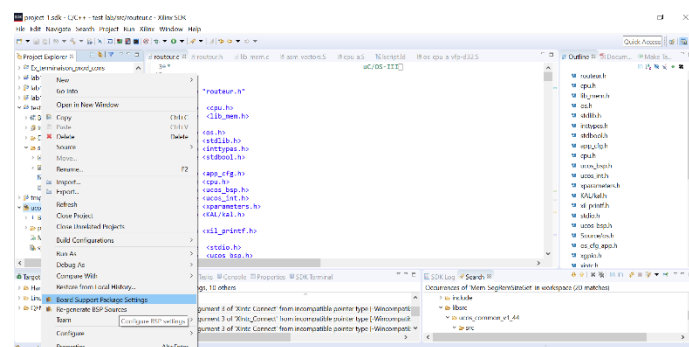


Figure 6 Pour atteindre certains paramètres du BSP

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 words)
OS_CFG_ISR_STK_SIZE	512	512	integer	Stack size of ISR stack (number of words)
OS_CFG_MSG_POOL_SIZE	6000	48	integer	Maximum number of messages
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 words)
OS_CFG_TASK_STK_LIMIT_PCT_EMP	10	10	integer	Internal Task stack limit in percent
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 words)


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)

5.2.1.2 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 1 et 0 respectivement. Ce qui permet 1 secondes comme temps d'attente (voir ligne 277 de *TaskGenerate*). Voici donc ce que vous devez faire pour vous assurer que les fifos se vident bel et bien.

- Mettre le *define* de FULL_TRACE de 0 à 1 dans *router.h* (permettant d'activer les traces dans votre programme) et recompilez.
- Mettre **point d'arrêt** à la ligne 282 de la tâche *TaskGenerate* c.-à-d. à :
`safeprintf("\n***** DEMARRAGE \n\n").`
- Mettre **point d'arrêt** à la ligne 272 de la tâche *TaskGenerate* c.-à-d. à :
`isGenPhase = false;`
- 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*
- 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 84 (1^{er} instruction du *main()*).
- vii. Cliquez sur *Resume* une 2^e fois et vous devriez voir apparaître l'entête pour **affichage des statistiques** (dont nous expliquerons le rôle plus loin) et le programme arrêté à la ligne 277.
- viii. Cliquez pour un 3^e *resume* et le code devrait s'exécuter jusqu'à la ligne 282 et vous devriez voir la trace :

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

- ix. Cliquez pour un 4^e *resume* et le code devrait s'exécuter jusqu'à la ligne 272 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 3^e et 4^e *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 9 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.

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

***** TaskGenerate: DEMARRAGE

TaskGenerate : *****Generation du Paquet # 1 *****
ADD 1850F4
** 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 *TaskQueueing* avec par exemple :

TaskGenerate: nb de paquets dans fifo de TaskQueueing - apres production: x (où x est le nombre d'éléments courant dans le fifo)

(n.b. : pour une raison qu'on a pas eu trop de temps pour investiguer, on remarque le nombre d'éléments dans la fifo est en retard de 1, c'est-à-dire on a 0 après 1 production et 1 après 2 productions, etc.).

3. Vous devriez également observer qu'un paquet obtenu précédemment en 3 a été transféré dans une des 3 fifos de priorités avec par exemple :

TaskQueueing: nb de paquets dans TaskComputing HIGHQ apres production : x

4. Vous devriez également observer de temps en temps une trace qui indique qu'un paquet a été lu (consommé) du fifo associé à *TaskQueueing* avec :

TaskQueueing: nb de paquets apres consommation du fifo: x

5. Cliquez à nouveau sur *Resume*. Vous devriez vous rendre jusqu'à l'impression de ***** RAFALE No 2 DE 213 PAQUETS. À partir de là, remontez l'écran vers le haut, jusqu'à l'endroit où vous étiez avant ce dernier *Resume* (étape 4) c'est-à-dire ***** FIN DE LA RAFALE No 1 (juste après la génération du paquet #57). Juste après la fin de la trace (Figure 10)⁵, vous devriez voir que le fifo de *TaskQueueing* contient 0 paquet, alors que *highQ* contient 18 paquets, *mediumQ* 19 paquets et *lowQ* 17 paquets.

N.B. $18+19+17 = 54$ paquets est inférieur à 57 paquets mais souvenez-vous que des paquets peuvent être rejetés (p.e. mauvaise adresse de destination).

6. Nous allons maintenant vérifier si un délai de 1 seconde est bel et bien suffisant pour vider les 3 fifos de *TaskComputing* et ainsi acheminer les paquets aux bonnes adresses (ports).

⁵ Remarquez que l'affichage de *TaskComputing HighQ* et *MediumQ* arrive après la fin de la rafale. Pourquoi ? Voir question no 1

```

TaskQueueing: nb de paquets dans TaskComputing HIGHQ apres production : 17
TaskGenerate: nb de paquets dans fifo de TaskQueueing - apres production: 1

TaskQueueing: nb de paquets apres consommation du fifo: 0

TaskGenerate : *****Generation du Paquet # 57 *****
ADD 188FF4
  ** id : 57
  ** src : 9323A8BC
  ** dst : E6A652A4
  ** type : 0

TaskQueueing: nb de paquets dans TaskComputing LOWQ apres production: 17
TaskGenerate: nb de paquets dans fifo de TaskQueueing - apres production: 1

TaskQueueing: nb de paquets apres consommation du fifo: 0

***** TaskGenerate: FIN DE LA RAFALE No 1

TaskQueueing: nb de paquets dans TaskComputing HIGHQ apres production : 18
TaskComputing HighQ: nb de paquets apres consommation du fifo: 18
TaskComputing MediumQ: nb de paquets apres consommation du fifo: 19

TaskComputing HighQ: paquet envoye dans interface 1
TaskComputing MediumQ: paquet envoye dans interface 1

```

Figure 10

7. Vous devriez voir en descendant la fenêtre que *highQ* passe de 18 à 17 (Figure 11), puis à 16 puis si vous continuez à défiler vers le bas, *highQ* va passer à 14, 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 0 sur la figure 11 devrait être identique au paquet 0 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.

8. 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*.

```

TaskGenerate : *****Generation du Paquet # 57 *****
ADD 188FF4
  ** id : 57
  ** src : 9323A8BC
  ** dst : E6A652A4
  ** type : 0

TaskQueueing: nb de paquets dans TaskComputing LOWQ apres production: 17
TaskGenerate: nb de paquets dans fifo de TaskQueueing - apres production: 1

TaskQueueing: nb de paquets apres consommation du fifo: 0

**** TaskGenerate: FIN DE LA RAFALE No 1

TaskQueueing: nb de paquets dans TaskComputing HIGHQ apres production : 18
TaskComputing HighQ: nb de paquets apres consommation du fifo: 18

TaskComputing MediumQ: nb de paquets apres consommation du fifo: 19

TaskComputing HighQ: paquet envoye dans interface 1

TaskComputing MediumQ: paquet envoye dans interface 1

Paquet reçu en 0
  ** id : 1
  >> src : 1ED5B4A6
  >> dst : CC33F38
  >> type : 0

TaskComputing HighQ: nb de paquets apres consommation du fifo: 17

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

TaskComputing HighQ: paquet envoye dans interface 3

```

Figure 11

En conclusion: ça donc dire que durant la génération d'une rafale de *TaskGenerate*, *TaskComputing* a le temps⁶ 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 passif de 2 secondes de *TaskGenerate*, *highQ*, *mediumQ* et *lowQ* se vident. Et c'est

⁶ 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.

précisément à ça que sert le délai d'attente de 2 secondes⁷. 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.

9. 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.

Vous devriez être maintenant en mesure de répondre à la **question 1**.

À partir d'ici vous pourrez toujours revenir à ces traces lors du débogage mais pour la suite mettre dans `router.h` FULL_TRACE à 0 mais laissez PERFORMANCE_TRACE à 1 :

```
36 #define FULL_TRACE 0
37 #define PERFORMANCE_TRACE 1
```

À la section précédente, on a utilisé 1 seconde de délai pour vider les fifos afin de s'assurer que celles-ci se vidaient à chaque rafale, mais on va montrer par la suite qu'on peut diminuer de beaucoup moins que 1 seconde en débutant à 128 ms:

```
169 int delai_pour_vider_les_fifos_sec = 0;
170 int delai_pour_vider_les_fifos_msec = 128;
```

5.2.2 Comprendre l'utilisation du timestamp pour le calcul du délai maximum d'un paquet dans le routeur et évaluation de performance pour le délai maximum d'un paquet avec différents ticks d'horloge

Prenez le temps d'abord de regarder comment le calcul de délai de chaque paquet est calculer avec l'aide du type double mot CPU_TS64. Le démarrage du calcul se fait dans TaskGenerate à la ligne 230 et termine dans TaskComputing (aux ligne 520 ou 528 ou 543 dépendamment de la destination) en calculant le délai maximum (souvenez-vous qu'en temps réel, c'est souvent le pire cas qui nous intéresse...). Finalement, notez que l'impression de ces 3 valeurs maximum se font dans TaskStats à partir de la ligne 665.

⁷ Ligne 213 `OSTimeDlyHMSM(0, 0, delai_pour_vider_les_fifos_sec, delai_pour_vider_les_fifos_msec, OS_OPT_TIME_HMSM_STRICT, &err);`

D'autre part, en classe, nous avons vu comment on peut émuler une attente active générée aléatoirement et bornée par une certaine valeur. Par exemple pour une valeur entre ~~0,9~~ [0, 9] on aura:

```
WAITFORTICKS = (rand() %10);  
actualticks = OSTimeGet(&err);  
while(WAITFORTICKS + actualticks > OSTimeGet(&err));
```

Vous devez donc ajoutez un code qui se placera juste avant le test sur la destination du paquet (ligne 497) et qui fera une attente active sur une valeur aléatoire dans l'intervalle ~~0,2~~ [0, 2] tick. Utilisez une structure de switch sur packet->type et générez une valeur aléatoire entre ~~0,2~~ [0, 2] tick pour chaque type de paquet. Notez que la structure switch sera utile afin que par la suite on puisse plus loin avoir des délais et sections critiques (section 5.2.3.2) différents pour chaque type de paquet.

Ensuite, testez votre code sur 100,000 paquets générés⁸. Nous allons nommer cette manipulation **manip 1**). À toutes les 30 secondes vous devriez voir un affichage comme celui de la figure 12 qui représente le tout premier affichage (d'où la raison pourquoi toutes valeurs sont à 0). Pour avoir 100,000 paquets et un peu plus, cela peut prendre quelques minutes... Une fois l'exécution terminée, faites un copier/coller de la dernière fenêtre d'affichage.

Finalement, complétez avec une 2e autres manipulations :

- **Manip 2** : refaites le même exercice mais en changeant les paramètre du BSP afin d'avoir la période du tick à .125 ms ou 8000 Hz et en ajustant les délais d'attente (8 fois plus long) dans la structure de switch sur packet->type de TaskComputing fait à la page précédente. Faites un copier/coller de la dernière fenêtre d'affichage.
- **Manip 3** : ici on suppose que TaskComputing peut faire son traitement généré aléatoirement 8 fois plus vite que pour la manip 1. Refaites donc le même exercice à 8000 Hz, mais en remettant les délai ~~0,2~~ [0, 1] de la manip 1. Faites un copier/coller de la dernière fenêtre d'affichage.

Vous devriez être maintenant en mesure de répondre à la **question 2**.

⁸ limite_de_paquets = 100000 dans router.h ce qui va faire un total de 100,000 paquets créés durant l'exécution. Notez que comme la tache statistique s'exécute à toutes les 30 secondes vous aurez un peu plus que 100,000 paquets... mais c'est correct !

```

Delai pour vider les fifos sec: 0
Delai pour vider les fifos msec: 128
Frequence du systeme: 1000
1 - Nb de packet-s total crees : 0
2 - Nb de packets total traites : 0
3 - Nb de packets rejetes pour mauvaise source : 0
4 - Nb de packets rejetes pour mauvaise source total: 0
5 - Nb de packets rejetes- pour mauvais CRC : 0
6 - Nb de packets rejetes pour mauvais CRC total : 0
7 - Nb de paquets rejetes fifo : 0
8 - Nb de paquets rejetes fifo total : 0
9 - Nb de paquets rejetes mauvaise -priorites : 0
10 - Nb de paquets rejetes mauvaise priorites total : 0
11 - Nb de paquets maximum dans le fifo de Queueing : -0
12 - Nb de paquets maximum dans le fifo HIGHQ de TaskComputing: 0
13 - Nb de paquets maximum dans fifo MEDIUMQ de TaskComputing: 0
14 - Nb de paquets maximum dans fifo LOWQ de TaskComputing: 0
15 - Nb de paquets maximum dans port0 : 0
16 - Nb de paquets maximum dans port1 : 0
17 - Nb de paquets maximum dans port2 : 0

18- Message free : 6000
19- Message used :- 0
20- Message used max : 0
21- Nombre de ticks depuis le début de l'exécution 120
22- Pire temps video '0.0000000000'
23- Pire temps audio '0.0000000000'
24- Pire temps autre '0.0000000000'

```

Figure 12

5.2.3 Analyse de performance

La section 5.2.1.2 que vous devriez avoir complétée avec la question 1, est ce qu'on appelle une validation fonctionnelle. Ici, nous allons nous intéresser à une validation de performance. Nous allons faire 2 tests aux sections 5.2.3.1 et 5.2.3.2 suivantes.

5.2.3.1 Faire diminuer le temps du délai

Reprenez les mêmes paramètres que la manip 3 de la page précédente (8000 Hz avec des délais de aléatoire de ~~0,2~~ 0,1 tick dans la structure de switch sur packet->type de TaskComputing).

L'objectif de ce test de performance est de voir si on peut faire mieux qu'un délai de 128 ms pour vider les différents fifos en faisant une recherche binaire. Plus précisément, on descend en divisant par 2 à chaque itération tant que les **2 conditions** ci-après sont respectées :

- 1) **Aucun fifo ne déborde** (print no 11 à 14 dans les statistiques) pour plus de 100 000 paquets créés.

2) Chaque fifo ne dépasse pas en moyenne 50%⁹

Si par exemple 64 respecte les 2 conditions faites la recherche binaire entre 0 et 64 et sinon entre 64 et 128, etc. En principe vousdevriez pouvoir trouvez ce délai x (entre 1 et 64) en moins de 7 itérations.

Gardez la capture d'écran de l'exécution de la valeur **x** trouvée ainsi que celle pour **x-1**.

Vous êtes en mesure de répondre à la **question 3**.

5.2.3.2 Utilisation de sections critiques Mutex vs Sémaphore

Pour cette manipulation, gardez la valeur du meilleur délai **x** obtenu à la section précédente. L'idée ici est de vous familiariser avec le mutex et sémaphore de uC/OS-III et de valider le concept d'héritage de priorité.

i) *Mutex*

On va supposer que les 2 tâches TaskComputing pour gérer HighQ et LowQ partagent une section critique et que l'héritage de priorité avec un mutex TaskComputing est utilisé (donc MediumQ ne partage pas la section critique). Les appels au mutex doivent se faire dans HighQ et LowQ avant et après le délai d'attente active sur une valeur aléatoire dans l'intervalle ~~0,2~~ [0, 1] tick. Notez que MediumQ n'a pas de section critique et possède toujours un délai d'attente active sur une valeur aléatoire dans l'intervalle ~~0,2~~ [0, 1] tick. **Gardez la capture d'écran de l'exécution pour plus de 100,000 paquets générés.**

ii) *Sémaphore*

Refaire les manipulations de la section précédente mais en remplaçant par des appels de fonctions sémaphore pour protéger la section critique. Utilisez *SemTaskComputing* déjà défini. **Gardez la capture d'écran de l'exécution pour plus de 100,000 paquets générés.**

Vous êtes en mesure de répondre à la **question 4**.

⁹ Il s'agit d'une moyenne, p.e. si tous les fifos sont en bas de 50% d'utilisation maximum et qu'une seule dépasse légèrement 50%, le résultat peut être acceptable.

6. Questions pour le rapport

Question 1 En vous référant aux priorités assignées aux tâches et en analysant le code de TaskComputing, répondez aux questions suivantes :

- a) Pourquoi durant la génération des paquets, le nombre d'éléments de fifo de TaskQueueing est toujours maintenu assez bas (proche de 0)?
- b) À la figure 10, l'affichage des fifos de TaskComputing HighQ et MediumQ (resp. 18 et 19 éléments) arrive après la fin de la rafale. Pourquoi ?
- c) À la figure 11, pourquoi l'affichage du paquet reçu id 1 de type HighQ (venant de TaskOutputPort[0]) arrive avant la mise à jour de TaskComputing de HighQ (i.e. TaskComputing HighQ: nb de paquets apres consommation du fifo: 17)?
- d) Comment expliquez-vous que les fifos HighQ se vide toujours avant MediumQ et que MediumQ se vide toujours avant LowQ?
- e) Justifiez le choix de la priorité des tâches dans routeur.h. Selon vous, est-ce le meilleur choix, y a-t-il d'autres assignations possibles et comment le comportement serait alors modifié.

Question 2

- a) Que signifie ~~[0, 2]~~ [0, 1] tick exactement, c'est-à-dire pourquoi le 0 n'est pas inclus et comment peut-on avoir .5 tick ?
- b) Affichez le résultat obtenu après 100,000 paquets créés (similaire à la figure 12) pour les manip 1 à 3.
- c) À partir des 3 figure obtenue en b), l'utilisation moyenne des 4 fifos (lignes 11 à 14) dépassent-elle la limite ?
- d) À partir des 3 figures obtenue en b), est-ce que le temps maximum pour traiter un paquet à travers les 3 fifos de TaskComputing HighQ, MediumQ et LowQ respectant les priorités demandées ?
- e) À partir des 3 figures obtenue en b), est-ce que le temps maximum pour traiter un paquet à travers les 3 fifos de TaskComputing HighQ, MediumQ et LowQ respectant les priorités demandées ?
- f) À partir des figure obtenue en b), pourquoi selon vous le temps maximum de la manip 2 est nettement différent de celui des manip 1 et manip 3.

Question 3

Présentez votre capture d'écran de l'exécution de la valeur x trouvée ainsi que celle pour $x-1$.

Question 4

- a) Affichez d'abord les performances obtenus : capture d'écran de la valeur x trouvé à la question 3 (qui sera la solution sans section critique) + 2 les captures d'écran de la section 5.2.3.2 (i.e. avec section critique pour HIGHQ et LOWQ protégé par mutex, puis avec section critique pour HIGHQ et LOWQ protégé par sémaphore).
- b) Comparez les 3 captures d'écran de a) en focusant sur le temps maximum d'un paquet HIGHQ. Que peut-on conclure?
- c) Calculez le blocage maximum de la tâche HIGHQ de TasComputing. Expliquez bien vos calculs.
- d) Supposons qu'on augmente le délai aléatoire de MEDIUMQ à 3 ticks et qu'on l'inclus maintenant dans la section critique. Calculez le blocage maximum de la tâche HIGHQ de TasComputing. Expliquez bien vos calculs.
- e) Supposons que l'on remplace la gestion de la section critique par un fifo qui au départ contient 1 seul élément. Lorsqu'une tâche rentre dans cette section critique elle consomme cet élément et quand la même tâche quitte la section critique elle produit cet élément.
 - c.1) Montrez qu'on a bel et bien un comportement capable de gérer une section critique.
 - c.2) Sans faire l'implémentation, à quel mécanisme (mutex ou sémaphore) devrait s'approcher les performance après plus de 100,000 paquets générés.

Question 5

Expliquez comment se fait la gestion de mémoire de uC/OS-III et quels sont les avantages de cette gestion par rapport à des appels de *malloc* et *free*, quand on fait du temps réel.

7. Barème et date de remise

Barème Lab 1 Partie 1

Questions	Notes	Commentaires
Q1 sur 1 point		
Q2 sur 1 point		
Q3 sur 1 point		
Q4 sur 1 point		
Q5 sur 1 point		
Fonctionnement sur 3 points		
Total sur 8		

Dates de remise :

Gr. 1 : 7 octobre 2024 8h30 AM

Gr. 2 : 30 septembre 2024 8h30 AM

Gr. 3 : 9 octobre 2024 8h30 AM