Polytechnique de Montréal - DGIGL

Laboratoire #1 : Routeur sur puce FPGA

INF3610 – Hiver 2021 (Guy Bois)

Séance no 1 : Développement et validation fonctionnel d'un simple routeur s'exécutant sous µC/OS-III

1. Objectif

L'objectif de ce laboratoire est de concevoir une application temps réel pour un système embarqué en ayant recours au RTOS µC/OS-III, pour d'abord le valider sous Visual Studio (partie 1) puis par la suite en faire une implémentation sur FPGA SoC avec la carte Cora Z7 (partie 2).

Les objectifs spécifiques de ce premier laboratoire sont :

- Faire l'apprentissage de µC/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 µC/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 CoraZ7.
- Utiliser des composants matériels tel que MailBox et Mutex

N.B. Tout comme pour µC/OS-III, l'architecture de la Cora Z7 sera présentée en classe (ce qui inclue aussi Zynq SoC et BSP).

2. Laboratoire en 2 séances

Ce laboratoire assure que vous vous êtes déjà familiarisés au laboratoire no 0 avec la création d'une plate-forme pour Cora Z7 avec Xilinx Vivado 2018.3 et d'un BSP uC/OS-III sous Xilinx SDK 2018.3.

Séance 1 : Validation de la fonctionnalité du routeur par une simulation de µC/OS-III sous Visual Studio (Windows)

Séance 2 : Implémentation du routeur sur Cora Z7

3. Mise en contexte

Application

La ***figure 1*** illustre un réseau téléinformatique permettant l'échange 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.

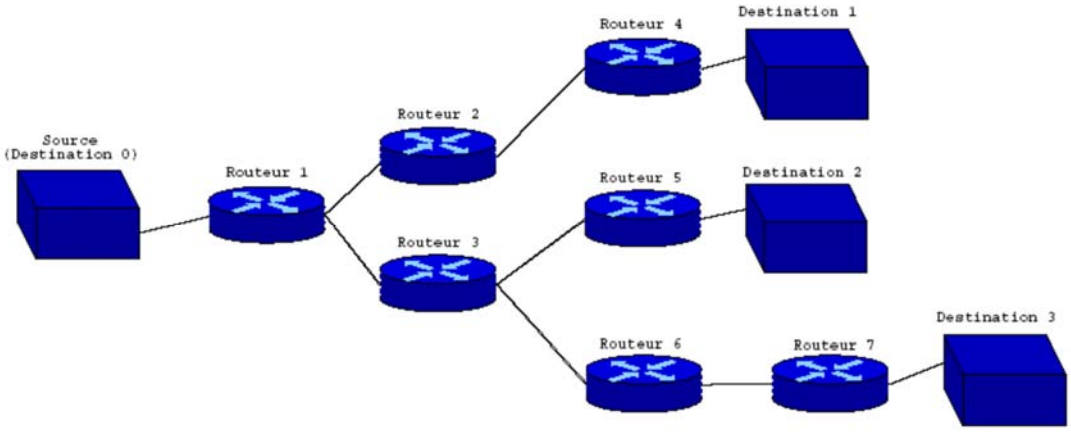

Figure 1 Exemple de Réseau téléinformatique

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 Octets	4 Octets	4 Octets	52 Octets
Source	Dest	Type	DATA

Ces paquets sont de taille fixe (***64 octets***) et possèdent quatre champs, soit une **source** indiquant la provenance du paquet, une **destination**, un **type** pour la qualité de service, et finalement les **données** transportées. La définition de cette structure vous sera fournie.

Plateforme matérielle ciblée

Bien que dans le la partie 1 vous passerez une bonne partie du temps sur le simulateur Visual Studio pour d'abord valider la fonctionnalité, nous présentons ici la cible finale envisagée sur la Cora Z7 pour vous donner une vue globale du laboratoire.

La flexibilité des puces multiprocesseurs configurables Zynq utilisées en laboratoire nous permet de faire différents partitionnements matériels/logiciels. La ***figure 2*** nous montre le partitionnement final que nous souhaitons réaliser. Il s'agit d'une architecture avec un cœurs de ARM (Cortex A9) et un processeur softcore uBlaze[1]. Chaque processeur ayant son système d'exploitation, il s'agit donc d'une architecture AMP (Asymmetric Multi-Processing). Les composants (de la librairie Logicore de Vivado) Mailbox[2] et Mutex sont des réalisations physiques de mécanismes pour faire la communication et la synchronisation entre processeurs (par opposition à tâches comme c'est le cas avec uC/OS-III).

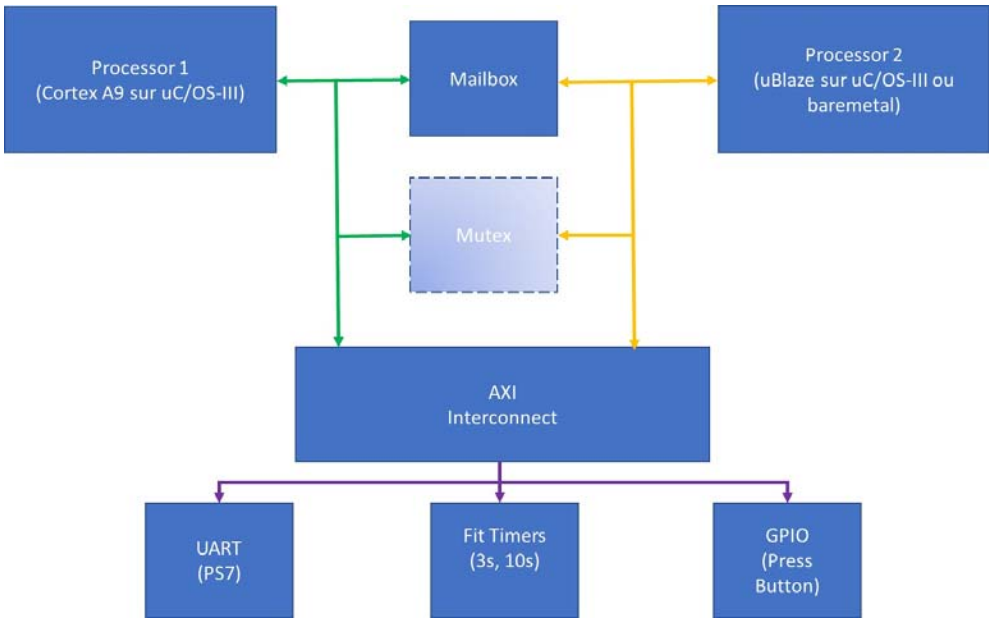

Figure 2 Architecture ciblée

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 le mutex, le rendez-vous unilatéral et la gestion de Queue de messages. Vous pouvez vous référez au programme *RDVUni_Queue_SemCompteur.c*

[1] Nous verrons dans la deuxième partie du lab si le uBlaze peut s'intégrer sans trop de travail, sinon des alternatives existent pour simplifier le travail.

[2] Pour le moment voyer le mailbox comme un fifo. De plus, il est possible que nous n'ayons pas besoin du mutex selon qu'on utilise ou pas les interruptions du mailbox, si c'est le cas (on a pas besoin) je prendrai quand même le temps de le présenter dans la partie 2, car il permet de faire de l'attente active (test and set).

Voici une brève description des différentes tâches (sur la fig. 3, on 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)

La tâche est fournie. Cette tâche s'occupe de générer des paquets aléatoires qui seront traités par le routeur. Cette tâche fait partie du banc de test puisqu'elle pourrait être par la suite remplacée par la vraie interface d'entrée. À titre d'exemple, les paquets pourraient provenir d'un autre processeur (uBlaze) via un partage de zone mémoire de DDR ou encore à travers un mailbox (Figure 2). 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 2 sec[3] (*OSTimeDlyHMSM(0, 0, 0, 2, OS_OPT_TIME_HMSM_STRICT, &err);*). Durant le mode génération, les paquets sont générés avec des valeurs aléatoires pour la source, la destination, le type et les données. Chaque génération de paquet est entrecoupé d'un délai de 0.5 ms [4] , c'est-à-dire *OSTimeDlyHMSM(0, 0, 0, 5, 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 (2s vs 0.5ms) sont des exemples,c'est-à-dire qu'on peut ensuite faire varier ces valeurs, d'une part pour faciliter le déverminage mais surtout 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[5].

Calcul (TaskComputing)

La tâche est à compléter. 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 (defines commençant par `REJECT_`).
- Ensuite, les paquets doivent être envoyés dans différentes files selon le type de paquets (vidéo, audio et autres resp. 0, 1 et 2). Si jamais cette file est pleine, vous devez détruire le paquet.

La fonctionnalité de TaskComputing est ici grandement simplifiée. D'autres types de calcul pourraient se faire à ce niveau comme par exemple un calcul de checksum ou CRC. Afin de simuler un temps de

[3] Valeur suggérée mais à valider à la section 5

[4] Valeur suggérée mais à valider à la section 5

[5] Il s'agit d'un cas particulier d'usage de fifo beaucoup plus compact que l'usage du QPend. Le chargé en glissera un mot au début de la séance de laboratoire. Pour l'instant, vous pouvez consulter le user manual p. 394 à 400.

traitement plus significatif (imaginez un calcul de CRC), une attente active de 3 ticks est donc effectuée à l'intérieur de la tâche de calcul.

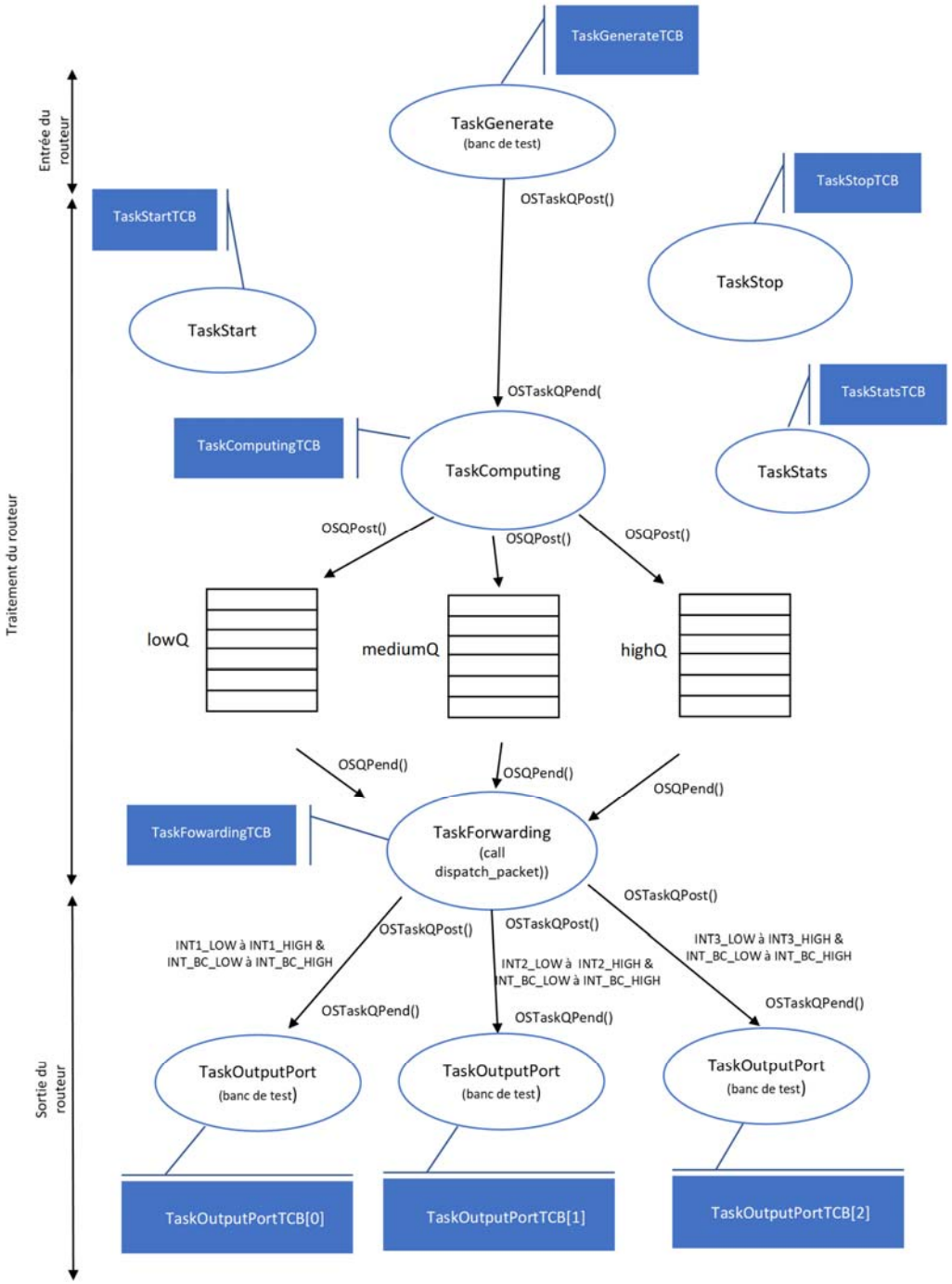

Figure 3 Flot de données dans le routeur

Forwarding (TaskForwarding)

La tâche est à compléter et doit répondre aux critères suivants :

- La tâche doit 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. Attention la queue vidéo (ou audio) est vide la tâche ne doit pas bloquer (mode non bloquant).
- La tâche doit 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 un *broadcast* sur les 3 interfaces.

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

    NB : Pour les paquets broadcastés, faites bien attention à allouer de l'espace pour les paquets nouvellement créés
- La tâche va finalement écrire dans la bonne interface. Ici chaque 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 TaskForwarding va écrire le paquet (similaire à l'écriture de TaskGenerate).

Interface de sortie (TaskOutputPort)

La tâche est à compléter. Cette tâche peut représenter le périphérique d'arrivée ou un autre routeur. Elle lit donc les paquets et de les impriment. Pour cette raison, on dit ici que l'interface de sortie fait partie du banc de test. Tel que 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

Arrêt de service (TaskStop)

Cette tâche sera complétée dans la partie 2. Elle permet de suspendre les activités des tâches après qu'un certains nombres de paquets rejetés ont été rencontrés.

Démarrage du système et reprise service (TaskReset)

Cette tâche sera complétée dans la partie 2. Elle permet au système de démarrer (évenuellement à l'aide d'un *press button*) une première fois et par la suite de redémarrer quand la TaskStop a suspendu le système.

Statistiques (TaskStats)

La tâche est à compléter. Pour le moment (c'est-à-dire sur Visual Studio), cette tâche est réveillée à toutes les 10 secondes et procure un certain nombre d'information depuis le début de l'exécution:

1. Nb de paquets total créés
2. Nb de paquets total traités
3. Nb de paquets rejetés pour mauvaise source (adresse)
4. Nb de paquets rejetés dans la fifo d'entrée
5. Nb de paquets rejetés dans l'interface de sortie
6. Nb de paquets maximum dans le fifo d'entrée
7. Nb de paquets maximum dans highQ
8. Nb de paquets maximum dans mediumQ
9. Nb de paquets maximum dans lowQ
10. Pourcentage de temps CPU Max de TaskGenerate
11. Pourcentage de temps CPU Max TaskComputing
12. Pourcentage de temps CPU Max TaskFowarding
13. Pourcentage de temps CPU Max TaskOutputPort no 1
14. Pourcentage de temps CPU Max TaskOutputPort no 2
15. Pourcentage de temps CPU Max TaskOutputPort no 3

Remarque : Les statistiques 7 à 15 sont réalisés automatiquement par uC/OS-III via sa tâche statistique OSStatTask (priorité 62 juste avant *Idle* quand on a 64 taches). Cette dernière échantillonne des données à la fréquence de 1/10Hz (fichier os_cfg_app.h). De l'information supplémentaire sur la mise en marche de la tâche OSStatTask vous seront données au laboratoire.

En ce qui concerne l'assignation des priorités aux tâches

Il faut s'assurer de perdre le moins de paquets possibles dans le fifo d'entrée et de minimiser l'utilisation des différents fifo (taille de la fifo). Dans ce contexte, vous devrez proposer et justifier une assignation de priorité aux tâches.

5. Travail à effectuer – séance no 1 :

Validation de la fonctionnalité du routeur par simulation sous Windows

Vous devez réaliser la spécification de la section 4 à partir d'un code de départ, plus précisément les tâches TaskComputing, TaskForwarding, TaskOutputPort et TaskStats. Les taches TaskReset et TaskStop ainsi que l'utilisation d'un mailbox (hardware) se feront dans la partie 2.

Quelques corrections ou considérations sur Windows (Visual Studio) :

1. À la ligne 124 du code de départ, remplacer &TaskOutputPortTCB[i] par &Port[i]
2. Pour comparer des pommes avec des pommes lors des 3 questions plus bas, assurez-vous de mettre OS_CFG_MSG_POOL_SIZE à 1024 sur VS (512 est la valeur du code de départ)
3. Il existe 2 façons pour initialiser *TaskStats* et les 2 cas conduisent au même résultat[6]. **La première approche est celle utilisée dans Visual Studio**:

   Utilisation de *OSSuspend()* et *OSResume()*

   Vous pouvez créer toutes les tâches (incluant ceux de l'application) comme c'est le cas actuellement dans code de départ. Mais, juste avant d'appeller *OSStatTaskCPUUsage()*[7], vous devez suspendre *TaskGenerate*, *TaskComputing*, *TaskForwarding* et *TaskOutputPort* et une fois *OSStatTaskCPUUsage()* complété, vous faites un resume sur *TaskComputing*, *TaskForwarding* et *TaskOutputPort*.
   Cette approche est donc celle utilisée dans le code de départ de VS, mais elle contient une erreur : pour corriger cette erreur enlever le code entre la déclaration des variables de TaskStats et la boucle while et remplacer par le code de la Table 1.

[6] Dans un cas comme dans l'autre les tâches TaskComputing, TaskForwarding, TaskOutputPort et TaskStats ne viennent pas biaiser le calcul du jalon par idle.
[7] Pour calculer le nombre maximum d'incrémentation (jalon) de la tâche idle,

```
OSTaskSuspend(&TaskGenerateTCB, &err);
OSTaskSuspend(&TaskComputingTCB, &err);
OSTaskSuspend(&TaskForwardingTCB, &err);
for (i = 0; i < NB_OUTPUT_PORTS; i++) {
        OSTaskSuspend(&TaskOutputPortTCB[i], &err);
};
OSStatTaskCPUUsageInit(&err);
OSStatReset(&err);

/* On peut repartir le système */
OSTaskResume(&TaskGenerateTCB, &err);
OSTaskResume(&TaskComputingTCB, &err);
OSTaskResume(&TaskForwardingTCB, &err);
for (i = 0; i < NB_OUTPUT_PORTS; i++) {
        OSTaskResume(&TaskOutputPortTCB[i], &err);
```

Table 1 Code qui suspend les taches applicatives, initialise et démarre les statistiques, et finalement redémarre les tâches applicatives.

Implémentation sur Cora Z7 (SDK)

Le code de départ pour la Cora Z7 est disponible sur Moodle. Une fois que votre code fonctionne sur VS vous n'avez qu'à porter vos tâches *TaskComputing*, *TaskForwarding* (et la fonction *dispatch_packet*) et *TaskOutputPort* aux bons endroits et remplacer les *printf* par des *xil_printf*. En ce qui concerne le code de TaskStats, porter aussi votre code mais **retirer le code de la Table 1 (l'initialisation de TaskStats va se faire différemment, voir point 2 plus bas).**

Quelques corrections ou considérations sur SDK:

1. Vous devrez aussi modifier 3 variables à la compilation. Ces variables se trouvent dans les fichiers *os_cfg_app.h* et *os_cfg.h* situé dans le BSP (ucos_bsp_0 dans la fenêtre de gauche de SDK) dans ucos_bsp_0\ps7_cortexa9_0\libsrc\ucos_osiii_v1_44\src:
    i. dans os_cfg_apph.h
        a. # define OS_CFG_MSG_POOL_SIZE 1024
    ii. dans os_cfg.h

```
#define OS_CFG_STAT_TASK_EN                    1
#define OS_CFG_TASK_PROFILE_EN                 1
```

2. **Sur SDK on a utilisé une 2e approche pour initialiser la tâche *TaskStats*.** L'idée est de ne créer qu'une seule tâche au départ, il s'agit ici de la tâche *StartUpTask* qui elle va faire le test avec *OSStatTaskCPUUsage()*, puis va procéder à la création des tâches TaskGenerate, TaskComputing, TaskForwarding, TaskOutputPort et TaskStats.
3. Les traces 10 à 15 (Pourcentage de temps CPU Max des tâches de l'application) donnent 0 sur la Cora Z7[8]. Par contre, le pourcentage de temps CPU courant global et le maximum (pour toutes les tâches de l'application) fonctionnent. Ajoutez donc à la tâche statistique[9]:
   - printf("16- Pourcentage de temps CPU : %d \n", OSStatTaskCPUUsage / 100);
   - printf("17- Pourcentage de temps CPU Max : %d \n", OSStatTaskCPUUsageMax / 100);
4. Si parfois la fenêtre SDK Terminal (uart) s'emballe (après un certains temps la fenêtre statistique ne s'arrête plus surtout si vous avez beaucoup de traces), fermez la fenêtre (en tapant sur le X). Repartir le système (bitstram et elf) et aller dans *Windows -> Perspective -> Reset Perpective* et dites oui à *Do you want to reset the current Debug perspective to its defaults*? Ainsi une nouvelle fenêtre de SDK terminal apparaitra et vous devrez vous rebrancher sur le port.

6. Barème et rendu suite à la séance no 1

Évaluation de la fonctionnalité de la partie 1 :

- En séance de laboratoire le 19 février (Gr 2), nous souhaitons voir les fifos se remplir et se vider pour chacune des implémentations (Visual Studio et SDK) avant la fin de la séance. 3pts sur 20 seront données pour ces fonctionnalités.
- En séance de laboratoire le 26 février (Gr 1), nous souhaitons voir les fifos se remplir et se vider pour chacune des implémentations (Visual Studio et SDK) avant la fin de la séance. 3pts sur 20 seront données pour ces fonctionnalités.

[8] Bug à corriger éventuellement
[9] Pour comparer des pommes avec des pommes, vous devriez aussi ajouter ces statistiques pour répondre aux 3 questions plus bas.

Pour le rapport répondez aux 3 questions ci-après (valant au total 4 pts sur 20) :

Mais d'abord, désactivez les traces safeprint ainsi que les *printf* (*xil_printf*) On vous suggère de garder les traces de génération de paquets pour savoir où vous en êtes lors de l'exécution:

```
OSMutexPend(&mutPrint, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
//if (shouldSlowThingsDown) {
xil_printf("GENERATE : ********Generation du Paquet # %d ******** \n", nbPacketCrees);
xil_printf("ADD %x \n", packet);
xil_printf("        ** src : %x \n", packet->src);
xil_printf("        ** dst : %x \n", packet->dst);
xil_printf("        ** type : %d \n", packet->type);
OSMutexPost(&mutPrint, OS_OPT_POST_NONE, &err);
```

Ainsi avec moins de trace la fonction TaskStats() sera plus précise (souvenez-vous que l'approche de uC/OS-III est intrusive).

Question 1: Voyez ce qui se passe sur chacune des implémentations (VS et Cora) si vous diminuer la période de génération à moins de 2 sec. Plus précisément pour chaque implémentation, déterminez la valeur minimum de rafale (entre 0 et 2 sec) pour que les fifos se remplissent et se vident (i.e., packet_rejete_fifo_pleine_inputQ = 0 et les 4 fifos i.e. 6 à 10 des statistiques sont inférieurs à 1024 et stables).

Faites une recherche binaire sur des exécution de 10K paquets ou plus en commencant par 1 sec de rafale puis aller à 0.5 sec si pas de débordement de fifo sinon allez à 1.5, etc.). Imprimer les captures d'écran. Justifiez bien votre résultat de valeur minimum de rafale. Comparez également entre VS et Cora Z7 les différentes tailles de fifo, ainsi que le pourcentage de CPU maximum (global) pour de mêmes périodes de génération.

Question 2: Refaire le même exercice que pour la question 1, mais avec 1 tick d'horloge de 1000Hz (#define OS_CFG_TICK_RATE_HZ 1000 dans os_cfg_app.h).

Encore une fois, imprimer les captures d'écran. Comparer les résultats avec ceux de la question 1 pour une même valeur de rafale (avons-nous les mêmes statistiques pour une même valeur de rafale?) et s'il y a lieu tentez d'expliquer les différences.

Question 3 : Expliquez pourquoi la fifo de sortie (pourtant de profondeur 1) ne déborde jamais (statistique 5). Également, en assumant une distribution uniforme de paquets générés par rand() pour highQ, mediumQ et lowQ expliquez pourquoi est-ce toujours lowQ qui déborde en premier (si le code est bien conçu).

- Le groupe 2 devra remettre la réponse aux questions au plus tard à 23h59 le 23 février
- Le groupe 1 devra remettre la réponse aux questions au plus tard à 23h59 le 1er mars

Autres questions pour vous-mêmes (mais n'est pas à remettre):

1. Assurez-vous que le fifos se remplissent et se vident à chaque 2 sec (ligne 246 de OSTaskGenerate). Pour cela utiliser les trace de la macro safeprint() (ligne 28).
2. Si tout fonctionne en 1), peut-on diminuer (ou devrait-on augmenter) la taille des fifos? Aidez-vous des statistiques (mais laissez rouler un bout avant de vous en inspirer). Expliquez
3. Changer l'ordre des priorité TaskComputing, TaskForwarding, TaskOutputPort et voir l'effet sur la taille des fifos et les périodes de génération.
4. Expliquez la structure de la fonction TaskForwarding. Pourquoi un un if else else (on aurait pu aussi mettre un switch) est-il important plutôt par exemple que 3 instructions if sans else? Expliquez.
5. Expliquez plus en détails le fonctionnement de la tâche OS_StatTask.
6. Justifiez l'utilisation des mutex mutRejete, mutAlloc et même mutPrint?
7. On dit que le profilage fait par uC/OS-III est intrusif. Expliquez ce terme et donnez un exemple.