

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 μ C/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 μ C/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 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 CoraZ7.
- Utiliser des composants matériels tel que MailBox et Mutex

N.B. Tout comme pour μ C/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 μ C/OS-III sous Xilinx SDK 2018.3.

Séance 1 : Validation de la fonctionnalité du routeur par une simulation de μ C/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.

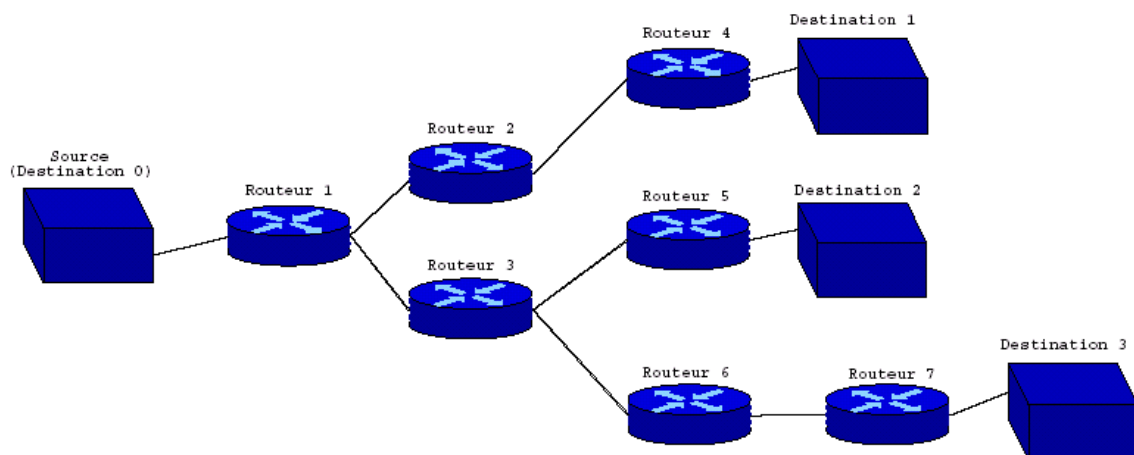


Figure 1 Exemple de Réseau téléinformatique

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 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 cœur de ARM (Cortex A9) et un processeur softcore uBlaze¹. 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) Mailbox² 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).

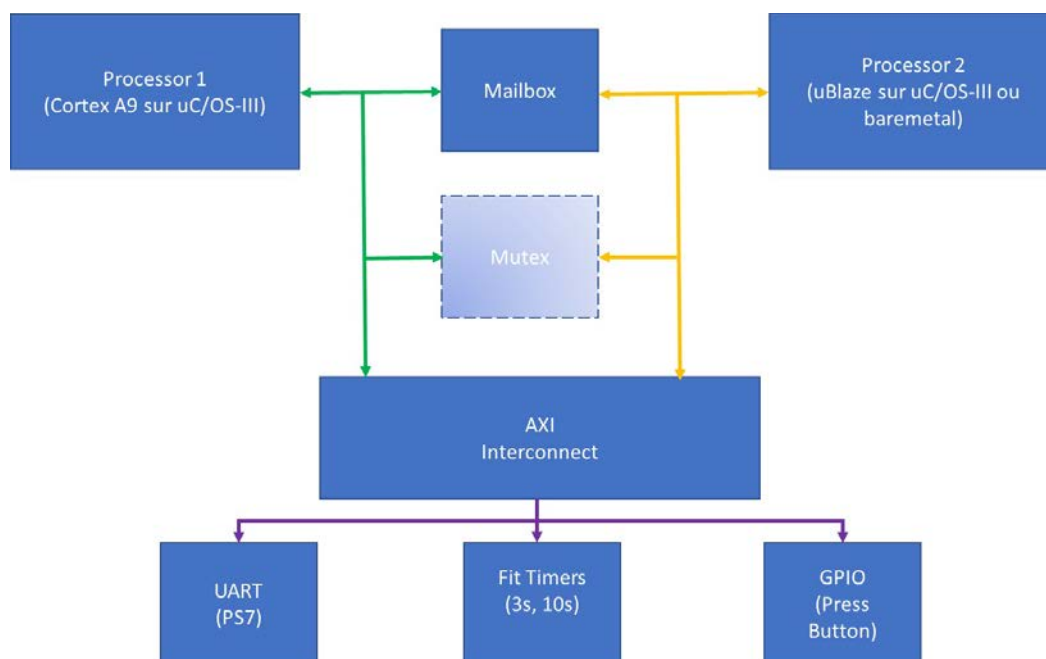


Figure 2 Architecture ciblée

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 référer 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 voyez 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 sec³ (*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 1 ms⁴, 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.

Finalement, notez que TaskGenerate écrit directement dans le fifo du TCB de la première tâche du routeur, c'est-à-dire TaskComputing⁵.

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 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.

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.

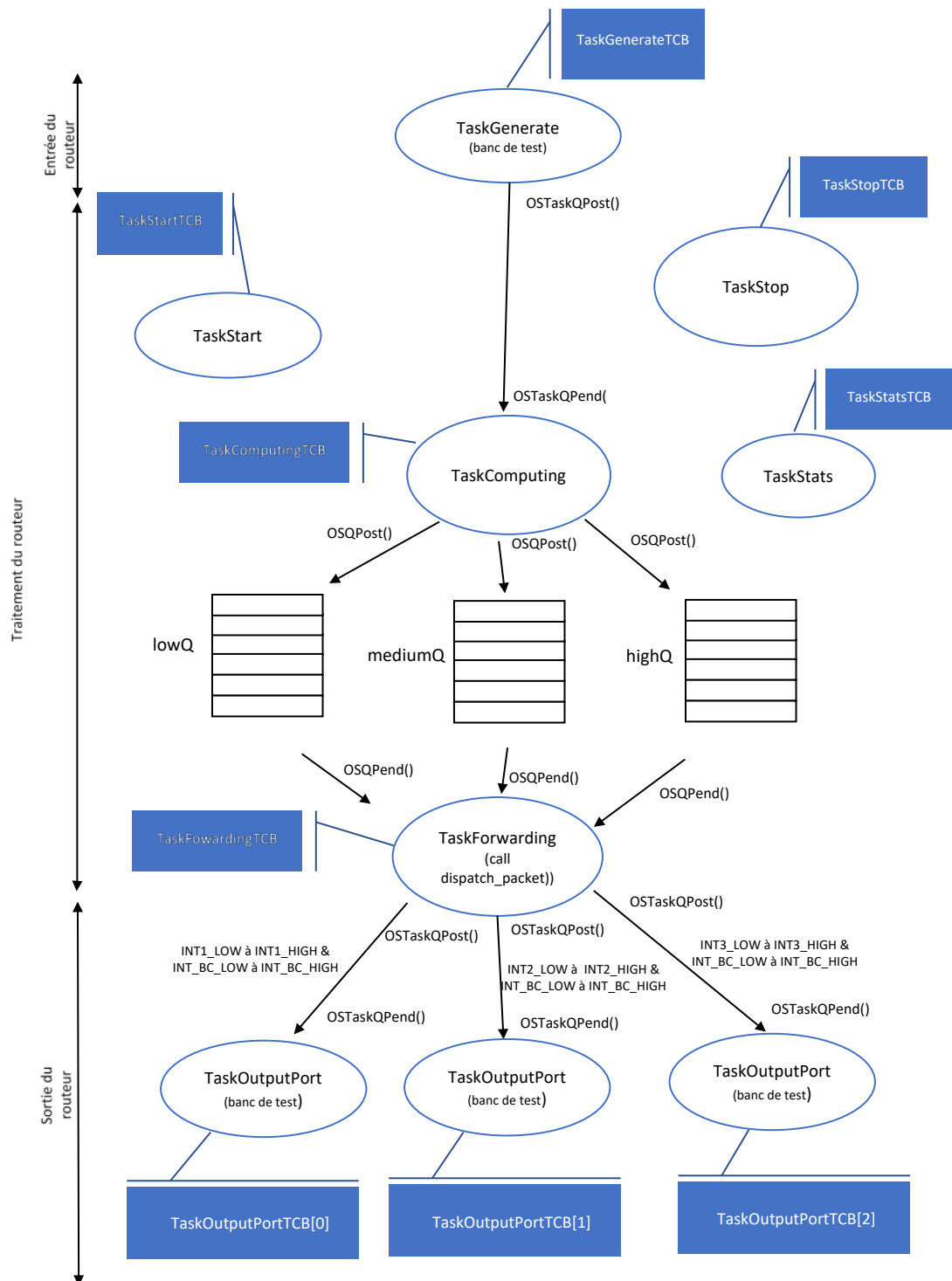


Figure 3 Flot de données dans le routeur

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 (éventuellement 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 tâches TaskReset et TaskStop ainsi que l'utilisation d'un mailbox (hardware) se feront dans la partie 2.

Chose à bien valider et questions :

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. Si tout fonctionne en 1), voyez ce qui se passe si maintenant vous diminuer la période de génération à moins de 2 sec ou encore en diminuant le temps entre 2 générations consécutives de paquets. Décrivez ce qui le comportement en vous aidant des statistiques (mais laissez rouler un bout avant de vous en inspirer).
4. La fréquence de la minuterie est de 100Hz (OS_CFG_TICK_RATE_HZ, fichier os_cfg_app.h), que se passe-t-il avec les questions 1 à 3 si on passe à 1000Hz?
5. Justifiez les priorités données avec le code de départ (hint : y-a-t-il des paquets rejetés à cause de fifo trop pleine)?
6. Changer l'ordre des priorité TaskComputing, TaskForwarding, TaskOutputPort et voir l'effet sur la taille des fifos et les périodes de génération.
7. 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.

8. Expliquez plus en détails le fonctionnement de la tâche OS_StatTask.
9. Justifiez l'utilisation des mutex mutRejete, mutAlloc et même mutPrint?
10. On dit que le profilage fait par uC/OS-III est intrusif. Expliquez ce terme et donnez un exemple.

Implémentation sur Cora Z7

Une fois que vous aurez terminé cette section porter votre code sur la Cora Z7 en utilisant le modèle du projet *Attente Active Vs Non Active* disponible sur le site web (section laboratoire) et refaire la question 3 (d'autres questions suivront).

6. Barème et rendu pour la séance de laboratoire no 1

À venir

