Polytechnique de Montréal - DGIGL

Laboratoire #1 : Routeur sur puce FPGA

INF3610 – Hiver 2024

Séance 3 : Partitionnement de l'application de routage sur 2 cœurs en mode AMP

1. Objectif de la séance no 3

Une architecture multiprocesseur où tous les processeurs ne sont pas traités de la même façon par le système d'exploitation est dite asymétrique (AMP pour Asymmetric MultiProcessing). Par exemple, un système peut autoriser (que ce soit au niveau du matériel ou du système d'exploitation) à un seul processeur d'exécuter le code du système d'exploitation ou peut autoriser un seul processeur à effectuer des opérations d'E/S. Un autre exemple de système AMP est un système multiprocesseur où chaque processeur roule son propre OS ou RTOS plutôt qu'un seul OS qui gère tous les processeurs[1]. Dans le cas de la séance no 3 on aura 2 ports (BSP) de uC/OS-III qui roulent chacun sur un core (core 0 et core 1)[2].

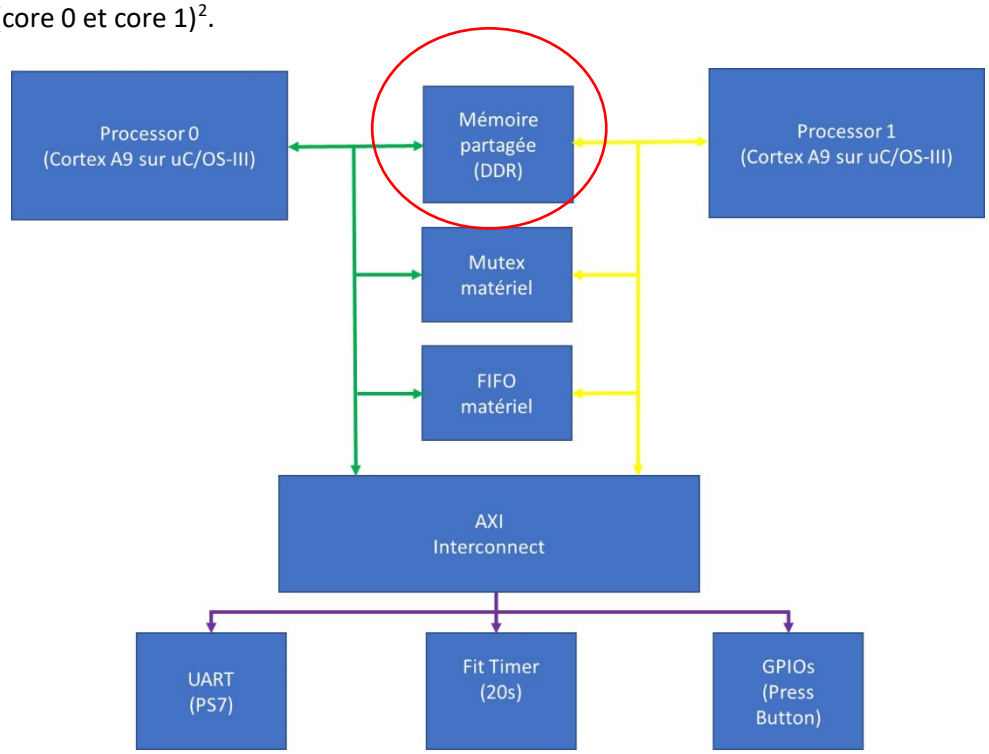


Figure 1. Architecture ciblée

Comme avec 2 processeurs on a 2 mémoires distinctes, la difficulté des systèmes AMP (mais aussi vrai pour SMP) est l'échange d'information entre les cores (ici core 0 et core 1). La figure 1 illustre 3 mécanismes pour faire cette communication sur PYNQ-Z2: 1) par mémoire partagée DDR, 2) par mutex matériel et 3) par FIFO matériel. Dans ce laboratoire, nous nous concentrerons sur une mémoire partagée dans la DDR (cercle en rouge sur la figure 1).

[1] On parle alors d'un système SMP (Symmetric MultiProcessing) tel que Linux sur un i7.
[2] Notez qu'on aurait pu aussi avoir comme système AMP core 0 qui roule baremetal et core 1 qui roule uC/OS-III ou encore core 0 qui roule FreeRTOS et core 1 qui roule uC/OS-III. Un système AMP peut donc rouler des OS différents.

L'objectif de cette 3e séance se concentre principalement sur 2 manipulatons :

Réalisation 1) Passage de la tâche *TaskGenerate* du core 0 au core 1 et programmation AMP en fonction de ce nouveau partitionnement (c'est-à-dire core 0 avec tout les tâches et ISRs de la séance no 2 moins la tâche *TaskGenerate* et core 1 avec *TaskGenerate*) à l'aide d'une mémoire partagée.

Réalisation 2) Synchronisation inter core pour l'arrêt du système complet

Réalisation 3) Terminaison propre

2. Ce que vous devez savoir pour la réalisation 1 (incluant un tutorial important pour la compréhension)

Concernant les adresses de programmes ELF sur core 0 et core 1

La figure 2 illustre l'assignation (mapping) des adresses sur une Zynq SoC série 7000. La PYNQ utilise seulement 512 MB puisque la DDR est limitée à cette valeur. D'autres cartes comme la ZYBO possèdent 1 GB.

Table 4-1: **System-Level Address Map**

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000_0000 to 0003_FFFF[2]	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU[3]
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF[4]	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF[2]	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Figure 2 Mapping des adresses sur une Zynq SoC série 7000

Maintenant, les figures 3 et 4 illustrent les zones mémoires qu'utilise chaque core (core 0 et core 1). Sur la figure 3, on remarque d'abord qu'on a encore besoin du *heap size* à 0xA00000 (10 MEG). On remarque aussi que l'adresse de base (Base Address) choisie de la DDR pour le core 0 est 0x1000000 et que sa longueur est 0x1000000 (16 MB). Qu'est-ce que SDK va mettre (charger) dans ce 16 MB? La réponse est plus bas sur la figure 3, il s'agit du fameux fichier ELF[3] qui est en fait le résultat de la compilation. Notez que si on avait voulu loger une partie du programme ELF dans la RAM (p.e. RAM_0) on aurait pu le faire en remplaçant ps7_ddr_0 par ps7_ram_0.

[3] ELF pour *Executable and Linking Format* est un format de fichier binaire populaire (Linux) utilisé pour l'enregistrement de code compilé (objets, exécutables, bibliothèques de fonctions).

Du côté du core 1, on aura le linker script de la figure 4. Que remarquez-vous comme seul changement? Et oui, l'adresse de base (Base Address) fait suite à Base Adress de core 0 plus le size de core 0, ce qui donne l'adresse 0x2000000. Ceci nous assure que les 2 programmes peuvent s'exécuter en parallèle[4].

Dernier point, comme *TaskGenerate* ne fera pas de malloc, heap size pourra rester à 0x2000.

[4] SDK donne par défaut une valeur peu importe le core. C'est au concepteur de s'assurer qu'il n'y a pas de recouvrement entre 2 programmes ELF. Un recouvrement risque fort de faire planter l'exécution.

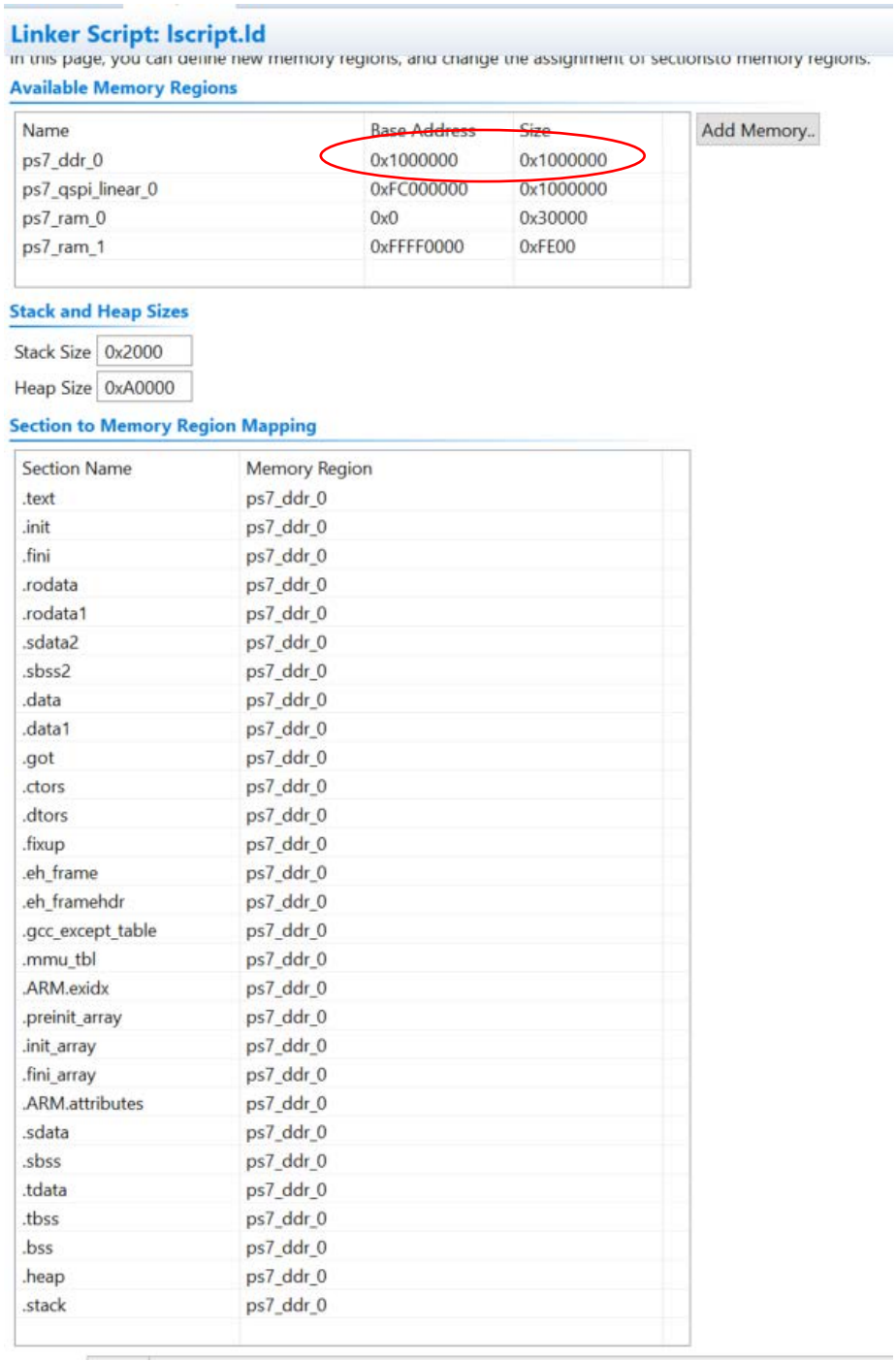


Figure 3 Linker script du core 0 qui contiendra tout le système de la séance no 2 moins *TaskGenerate*

Figure 4 Linker script du core 1 qui contiendra *TaskGenerate*

Comment créer la mémoire partagée DDR

On doit déterminer la plage d'adresse de la mémoire partagée (cercle rouge sur la figure 1). Celle-ci se fera forcément en dehors de celle des programmes de core0 et core 1. On pourrait par exemple décider de prendre une zone partagée à partir de l'adresse 0x3000000. Ce partage doit être annoncé dans un *define* sur chaque core :

```
const uint32_t BASEADDR = 0x3000000;
```

Comment vérifier que 0x3000000 est une bonne valeur? Deux choses : 1) puisqu'il s'agit d'un mapping direct sur le mapping des adresses de la figure 3, on observe que 0x3000000 est bel et bien réservé à la DDR et 2) les programmes ELF des figures 3 et 4 occupent de 0x1000000 à 0x2FFFFFF n'utilisent pas la plage 0x3000000 et plus (encore une fois pour éviter tout recouvrement). Finalement, nous allons

créer une zone mémoire partagée entre core0 et core1 pour échanger une rafale d'au maximum 255 paquets, donc 4K mots au maximum. À 4K mots on est inférieur à 3FFF FFFF, donc on est OK.

Dernier point, vous devez désactiver l'utilisation de la mémoire cache pour la zone qui sera partagée. Dans notre cas, pour des raisons de simplification nous désactiverons toute la cache juste avant OSStart() avec la commande *Xil_DCacheDisable()*. **Important chaque programme ELF qui accède à la mémoire partagée doit faire cette désactivation.**

Comment synchroniser core 0 et core 1 lors d'un partage de rafale– tutorial du producteur/consommateur

Nous allons utiliser le protocole de type *handshacking* (poignée de main). Cela nécessite 2 signaux de contrôle *ack* et *req*. La figure 5 illustre le protocole entre un consommateur sur core 0 et un producteur sur core 1.

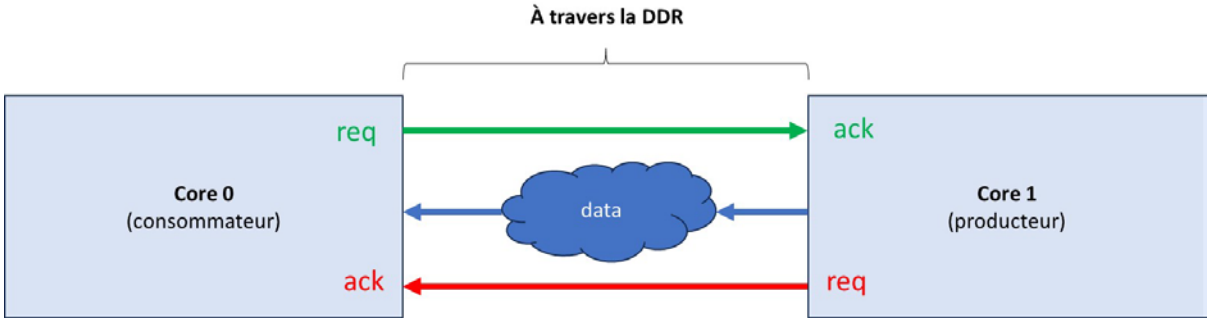


Figure 5 *Handshacking (échange avec poignée de main)* entre un consommateur et un producteur à travers la DDR : req de *core 0* et ack de *core1* sont la même adresse en DDR et même chose pour req et ack.

On suppose au départ que les variables *req*, *ack* définis en zone partagée dans la DDR sont à 0 et que la donnée (data) partagée est aussi dans la DDR. Un cycle de transfert se fera de la façon suivante :

i. *core 1* va produire une donnée et va attendre de manière active, avec *while (!ack)*, que *core 0* soit prêt à la consommer.
ii. Quand *core 0* est prêt à consommer la donnée, il met son *req* à 1 (ce qui débloque *core 1*) et il attend à son tour de manière active, avec *while (!ack)*, que core 1 lui donne le feu vert pour consommer.
iii. *core 1* met son *req* à 1 (ce qui débloque *core 0*) et va attendre de manière active que *core 0* est fini de consommer avec *while (ack)*.
iv. *core* 0 va consommer et puis il remet son *req* à 0 (ce qui débloquer core 1) pour indiquer qu'il a consommé et qu'on peut passer à une autre valeur.
v. core 0 attends de manière active, avec *while (ack)*, que core 1 lui dise de passer à une prochaine valeur à consommer.
vi. *core 1* met son *req* à 0 (ce qui débloque *core 0*) pour indiquer que c'est terminer et qu'il va passer à une prochaine donnée.
vii. Et on retourne à l'étape i pour une prochaine donnée.

Les programmes **producteurs** et **consommateurs** distribués comme code de départ illustrent un exemple de ce *handshacking*. Le transfert de *TaskGenerate* sera calqué sur cet exemple de producteur/consommateur mais au lieu de transférer des entiers (valeur i), vous transférez une rafale de paquets (de grandeur aléatoire entre 1 et 255). Par conséquent, prenez 1 heure pour bien comprendre comment exécuter ce code du producteur/consommateur sur 2 cores en lisant bien les figures 6 à 18 et le texte qui les accompagnent. N'hésitez pas à poser des questions au chargé de laboratoire. Si vous comprenez bien le fonctionnement de ce producteur/consommateur et comment le programmer sur 2 cœurs, le travail de transférer TaskGenerate sur core 1 sera simple, puisque vous pourrez ré-utiliser le même découpage. Plus précisément, comme vous aurez créé 2 BSPs (1 pour core 0 et 1 autre pour core 1), vous pourrez reprendre ces 2 BSPs.

Voici dans ce qui suit comment exécuter ces 2 programmes en mode AMP, chaque programme sur un cœur avec son propre port de uC/OS-III :

1) Le consommateur sera sur core 0 et il sera ***slave***. Vous devez créer une nouvelle application que vous nommerez *Ex_consommateur*. **Attention! créez du même coup le BSP en cliquant l'option *Create New*** telle qu'illustrée en rouge à la figure 6. Comme à l'habitude ajoutez le *Hello World* de uC/OS-III sur la fenêtre suivante.

2) Cliquez avec le bouton de droite pour faire le setting du BSP (flèche rouge sur Figure 7) i.e. :
    a. Les drivers avec int 3.8,
    b. *ucos_standalone* pour *stdout* o*ps7_uart_0* (PYNQ-Z2) ou *ps7_uart_1* (Zedboard),
    c. mettre consommateur comme un slave, i.e. *UCOS_AMP_MASTE = false* (Figure 8) et
    d. finalement, profitez-en pour mettre les bons paramètres de profondeur de FIFO (*OS_CFG_MSG_POOL_SIZE*) et de fréquence (*OS_CFG_TICK_RATE_HZ*) comme montré à la figure 9 (peut-être pas utile pour l'exemple producteur/consommateur, mais utile pour votre application qui roulera plus tard sur le core 0 (5000 de pool size).

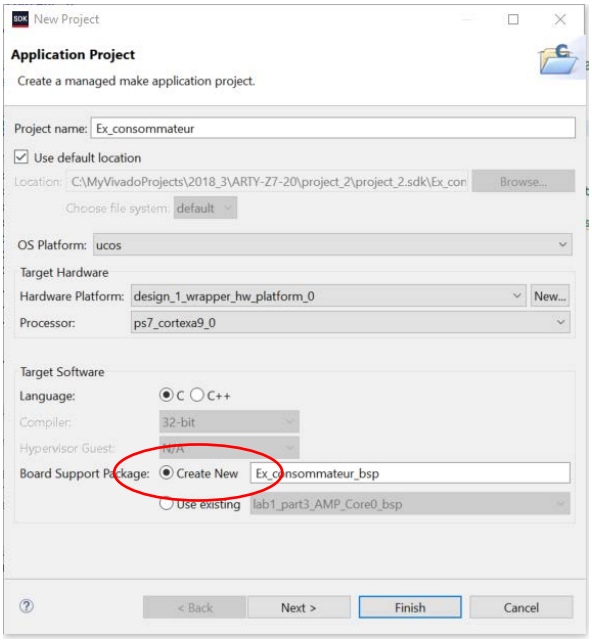


Figure 6

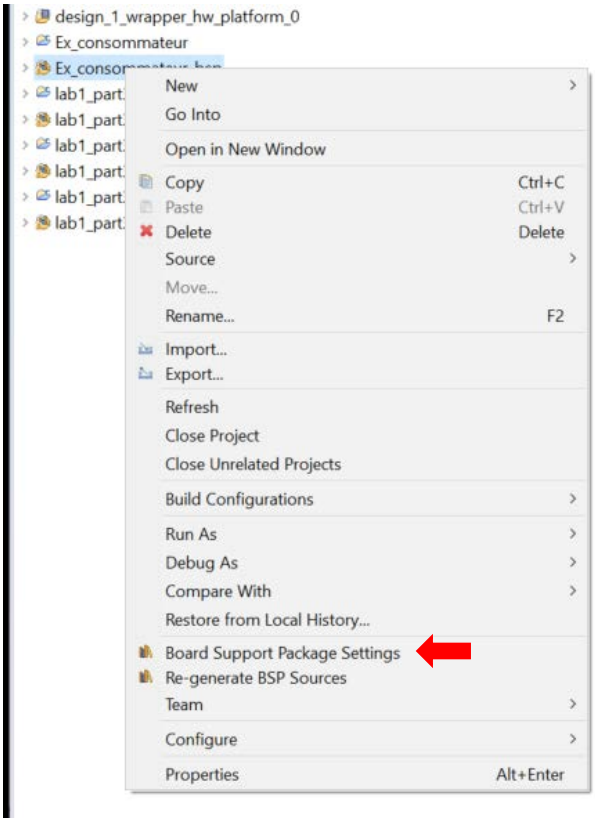


Figure 7

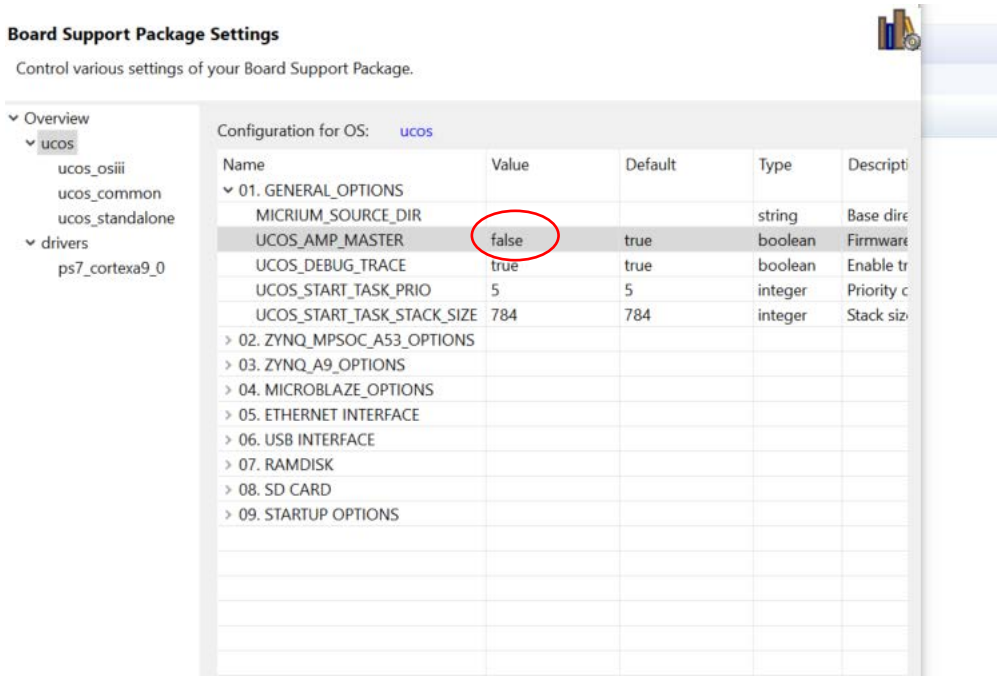


Figure 8

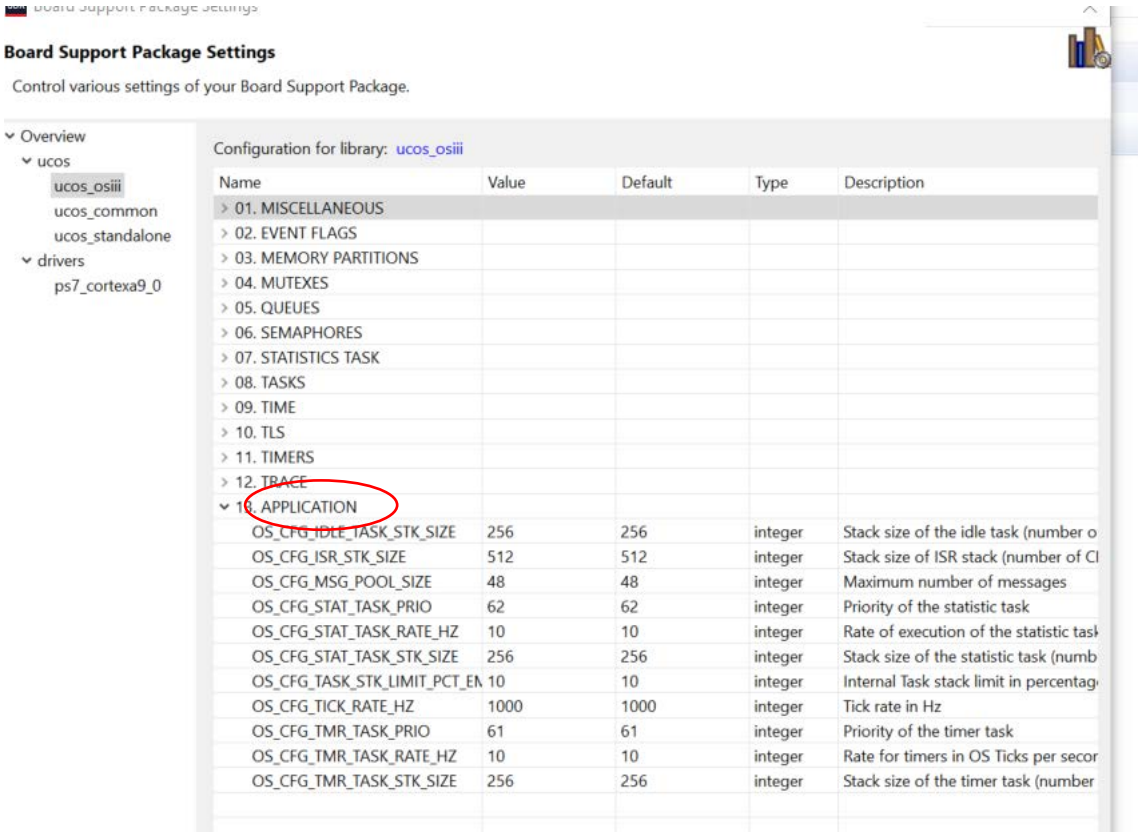


Figure 9

3) Remplacer le fichier *app.c* par *consommateur.c* et modifier le linker script pour avoir celui de la figure 3.

4) Le producteur sera sur core 1 **et il sera maître**. Vous devez créer une nouvelle application que vous nommerez *Ex_producteur*. Refaire les mêmes manipulations que 1) et 2) ci-haut, mais avec 2 différences importantes :

   i) Quand vous créez l'application, **mettez-la sur ps_cortex9_1** (Figure 10) et non sur ps_cortex9_0.

   ii) Ne pas appliquer l'option 2c) (figure 9). On doit ici avoir UCOS_AMP_MASTE = true

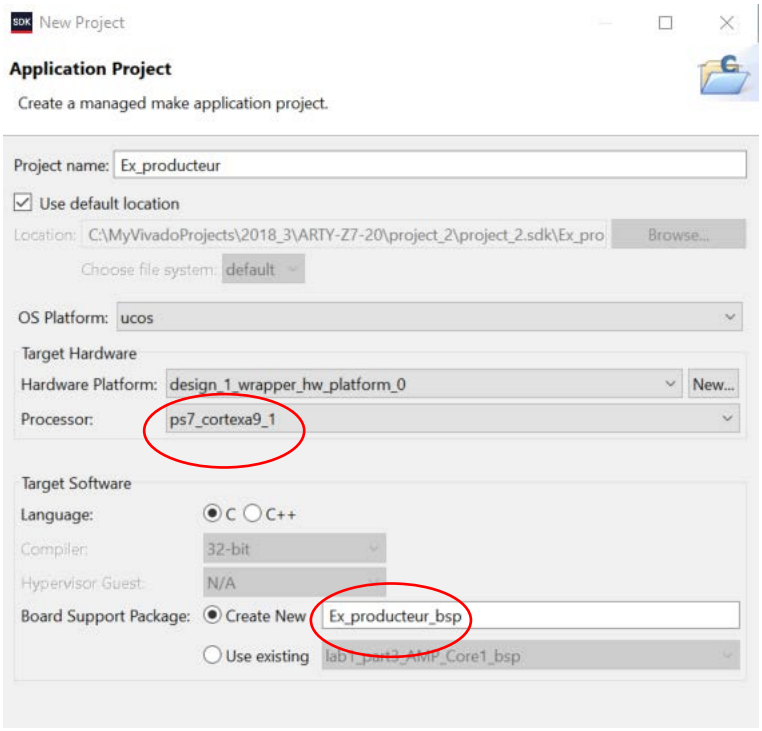


Figure 10

5) Remplacer le fichier *app.c* par *producteur.c* et modifier le linker script pour avoir celui de la figure 4.
6) Il ne vous reste qu'à configurer votre carte ce qui implique programmer la partie matérielle (PL) et la partie logiciel (PS avec les ELF) avec la fenêtre de *Debug Configurations* (onglet Run dans la barre horizontale).
    a. La figure 11 montre l'onglet *Target Setup* pour programmer le FPGA (partie PL). **Cette fois, il ne faut pas mettre l'option *Reset entire system***. Il faut laisser la partie PS s'initialiser elle-même.
    b. La partie PS demande deux chargements : la figure 12 montre le chargement de l'application *EX_consommateur.elf* , alors que la figure 13 montre le chargement de *EX_producteur.elf*

       Attention : *Reset processor* reset les 2 cores (on choisit ici de le faire seulement sur core 0).

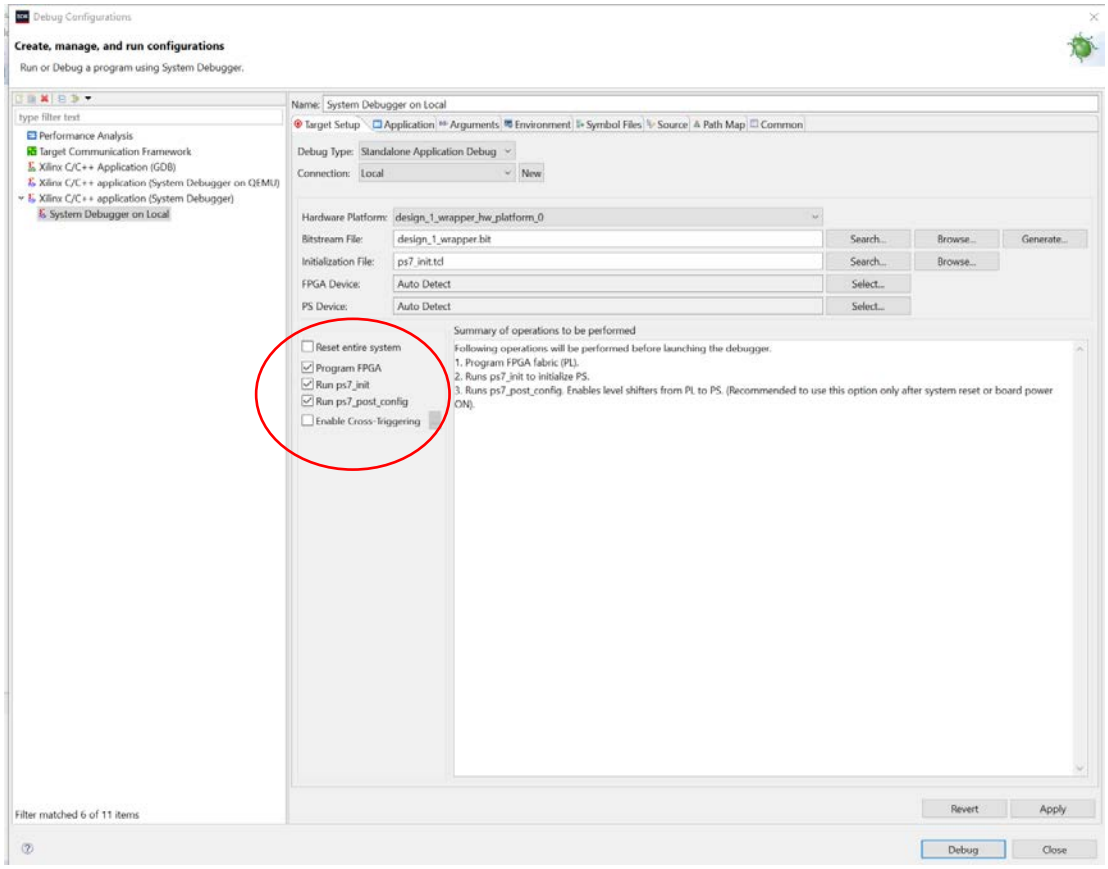


Figure 11 Programmation de PL sans reset

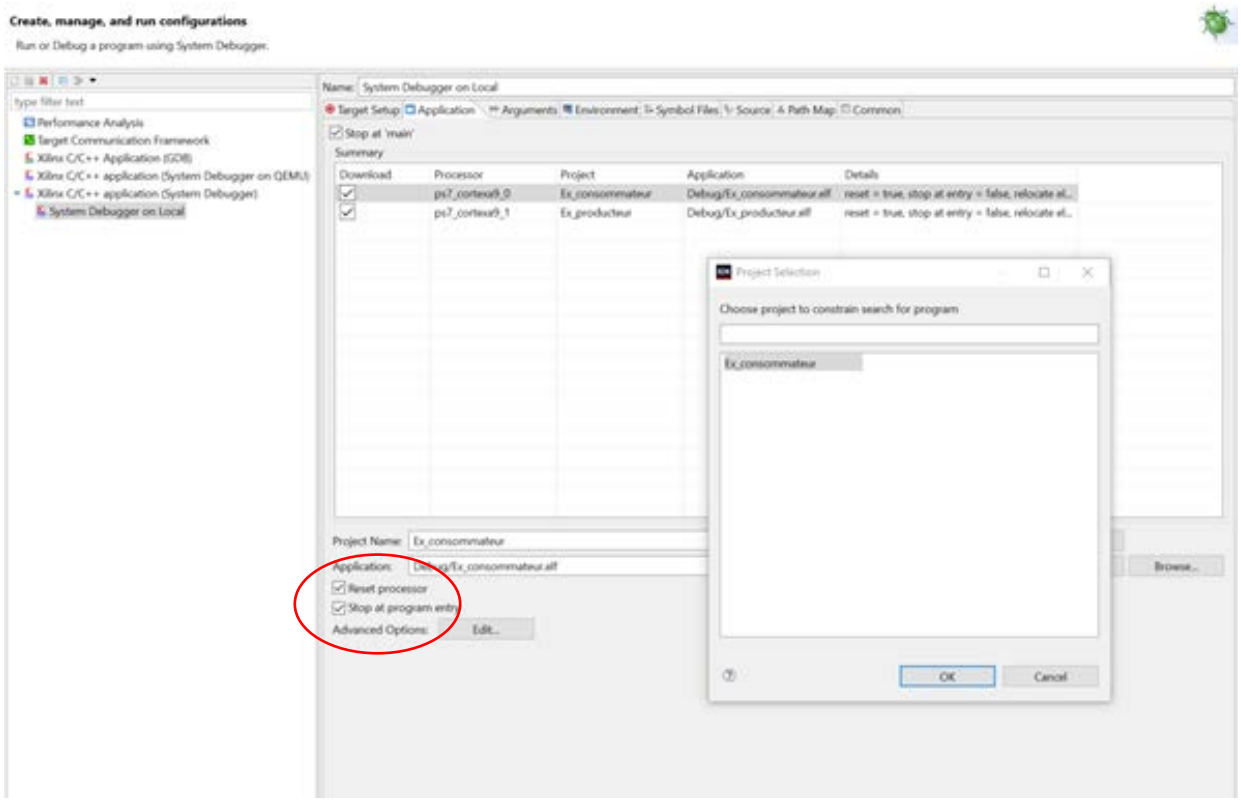


Figure 12 Programmation du core 0 (ELF) avec un reset des 2 cores

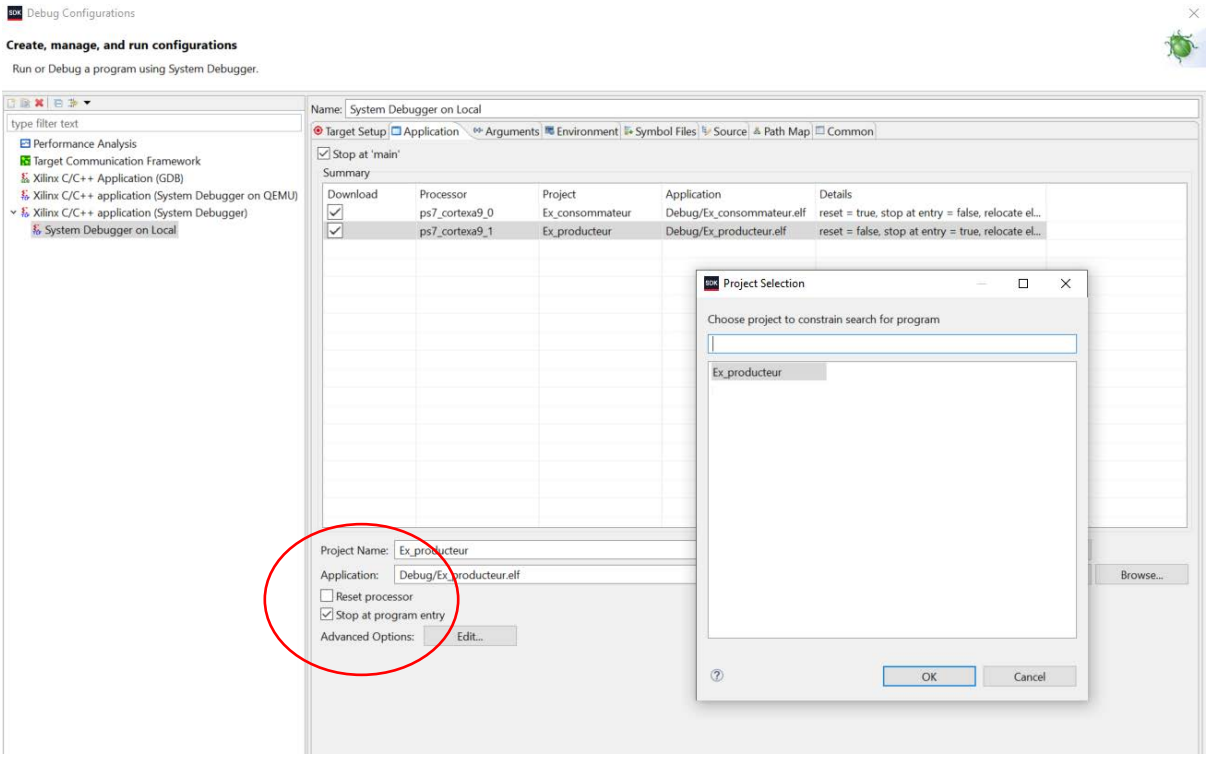


Figure 13 Programmation du core 1 (ELF) mais pas de reset cette fois (un 2e reset aurait fait annuler la configuration sur core 0 fait à la figure 12)

7) Cliquez ensuite sur *Debug* dans le bas de la fenêtre (voir figure 11).
8) Vous avez maintenant le choix de démarrer soit le core 0 ou le core 1. Comme notre application démarre par le producteur qui fait une mise à 0 des signaux *ack* et *req* (Figure 5), nous allons d'abord démarrer le ARM Cortex-A9 MPCore # 1 en premier. Comme montré à la figure 14 :
    i. cliquez sur ce dernier, puis
    ii. cliquez sur Resume. Le core 1 va alors s'arrêter à la ligne 109. Préparez alors la fenêtre SDK terminal puis faites un autre Resume. Vous devriez voir l'initialisation du core 1 complété dans la fenêtre SDK terminal (Figure 15). Toujours sur la figure 15, vous devriez aussi voir que core 1 est en mode *Running*. Cliquez sur *Suspend* (Figure 16) et vous devriez voir que core 1 a stoppé à la ligne 151. On vous expliquera au laboratoire le protocole mais disons que pour l'instant considérez que core 1 (producteur) est en attente active du signal *ack* (adresse 0x3000014 dans la mémoire partagée). Or ce signal *ack* est en fait le signal *req* du consommateur sur core 0 (voir fig. 5).

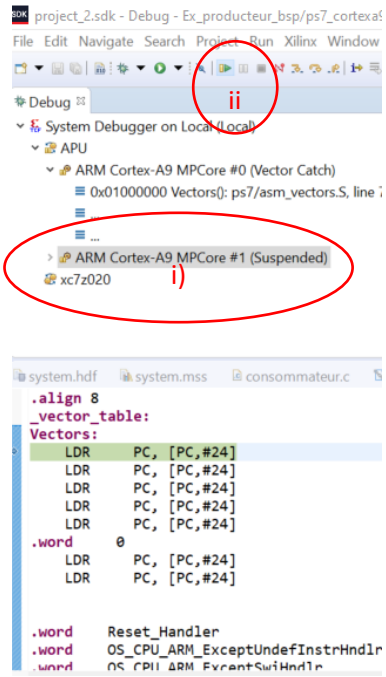


Figure 14

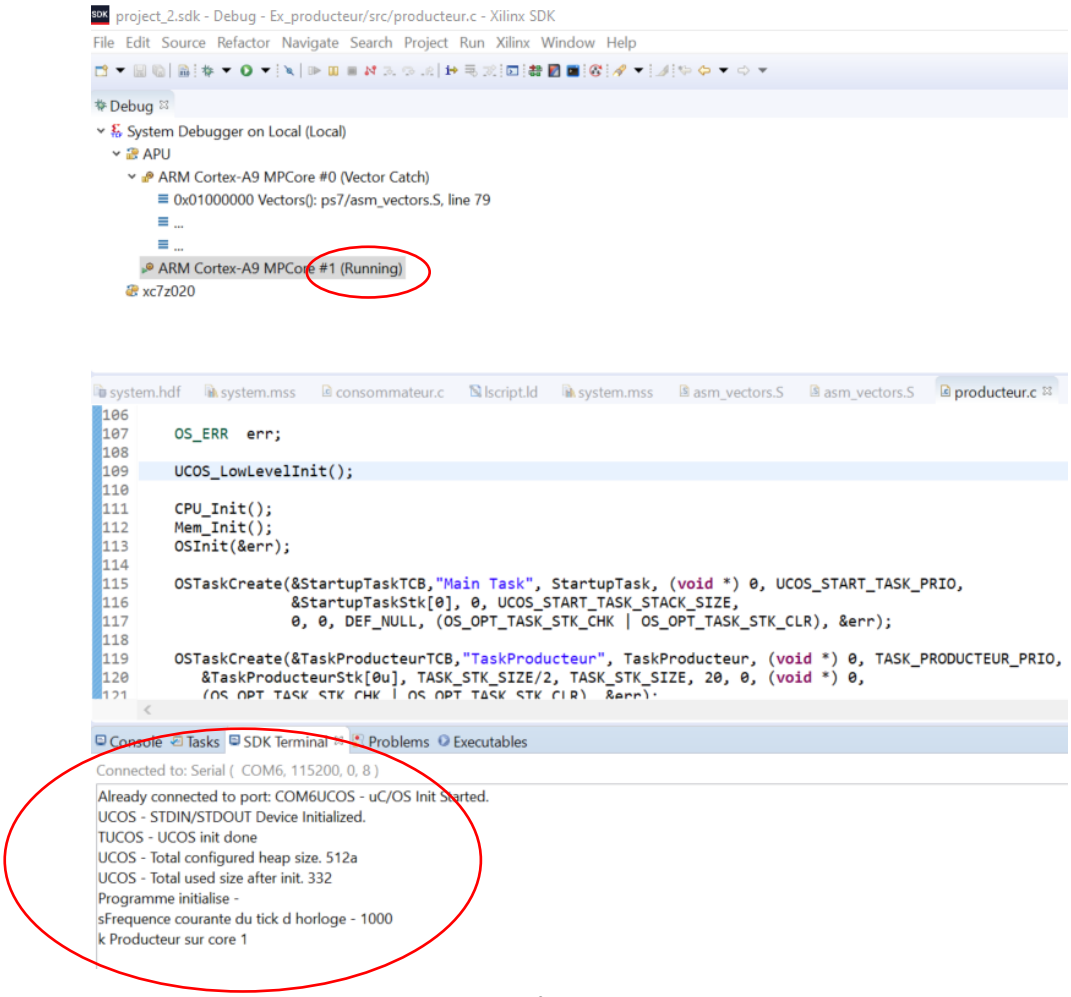


Figure 15

(Remarque suite à l'étape 8)

Notez que lorsque vous faites ainsi un *Suspend*, vous pouvez observer tout le contenu des variables de core 1. Prenez le temps de noter quelque part les adresses et le contenu de *ack* et *req* en mémoire partagée. Laissez pour l'instant core 1 en mode Suspend.

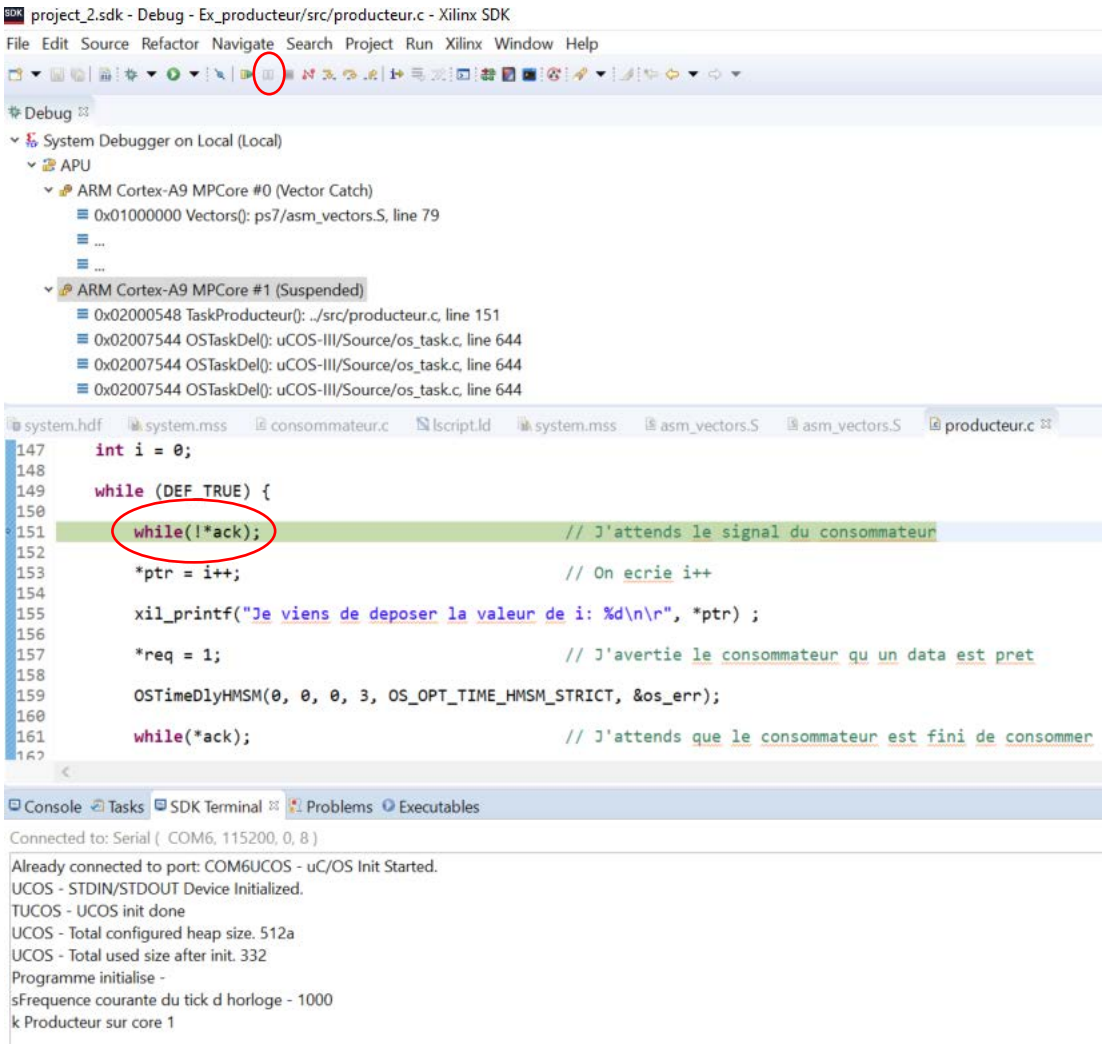


Figure 16

9) Nous allons maintenant démarrer core 0. Procédez comme à l'étape 8 mais en cliquant sur *Cortex-A9 MPCore # 0* puis sur *Resume*. Faites ensuite un deuxième *Resume*. Le système devrait s'être initialisé puis être en mode Running (Figure 17). À nouveau si vous cliquez sur *Suspend* core 0 va s'arrêter à la ligne 148 lui aussi en attente active du signal *ack*. Vous devriez aussi pouvoir observer que core 0 a mis ***req* à 1** juste avant de se mettre en attente de *ack*. **Bonne nouvelle! Nous ne sommes pas dans un deadlock.**

Remarque : Faire attention, car parfois on croit observer les variables de core 0 mais on observe celles de core 1, si on a par exemple cliqué sur *Cortex-A9 MPCore # 1 Suspended* et non sur *Cortex-A9 MPCore # 0 Suspended*.

10) Redémarrez maintenant core 1 en cliquant *ARM Cortex-A9 MPCore # 1* et cliquez sur *Resume* puis à nouveau sur *Suspend*. Vous verrez que core 1 attend maintenant activement à la ligne 161. Il attend que le consommateur ait consommé (***req* à 0**). Faites un *Resume* sur core 1, puis un *Resume* sur core 0. Votre système AMP devrait maintenant être en fonction!

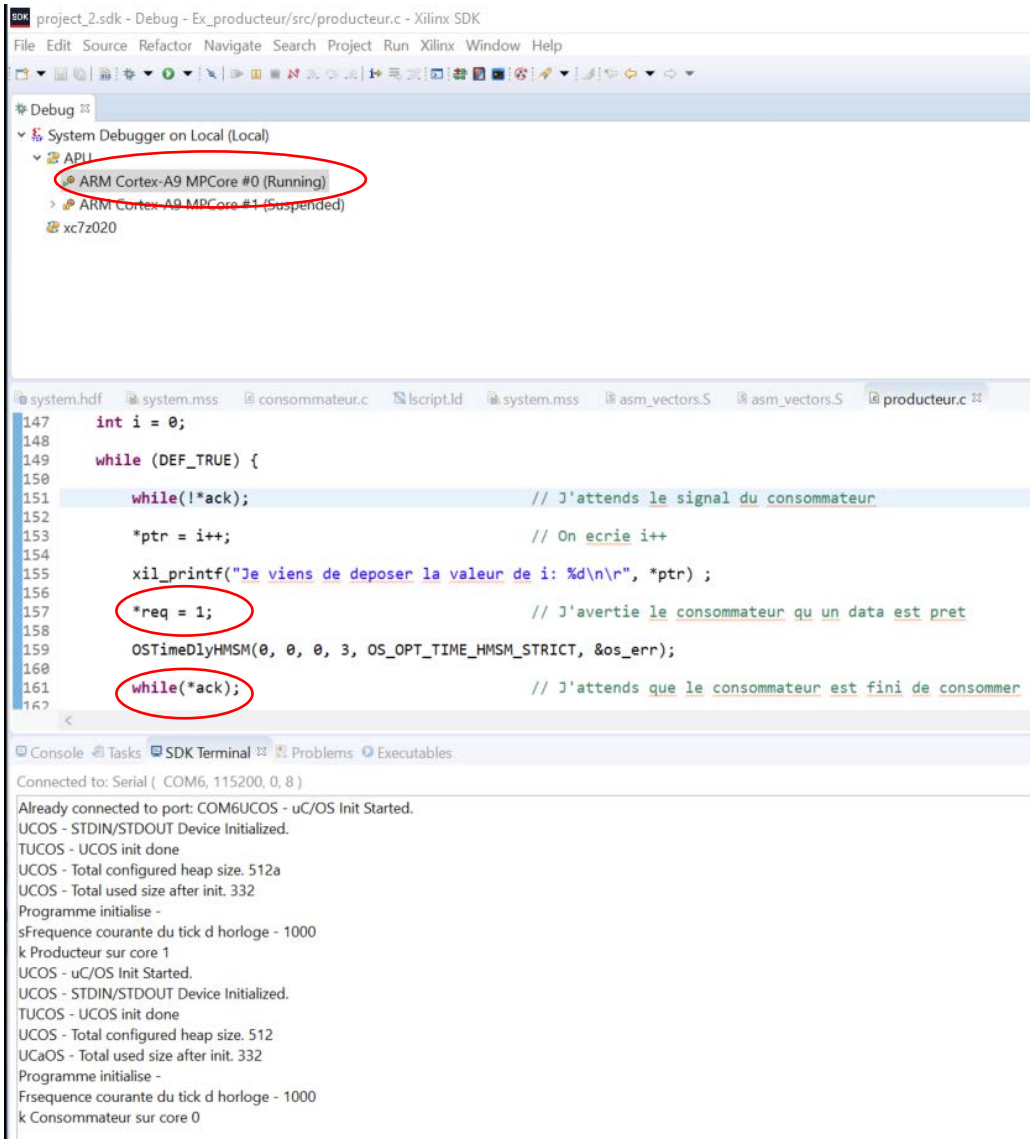
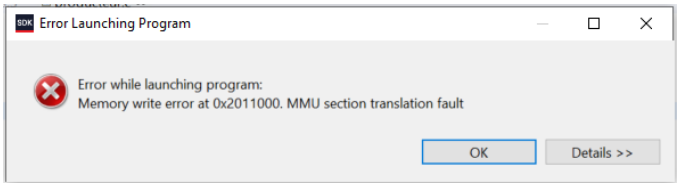


Figure 17

Autres remarques importantes :

1) Entre chaque exécution de Debug Configurations, assurez-vous de faire un reset (voir bouton SRST sur figure 18) sur la carte. Surtout si le message suivant apparait :



Notez que vous pouvez aussi mettre la switch à OFF et ensuite à ON mais vous perdrez alors le UART du SDK Terminal (ce qui n'est le cas avec SRST).

2) Ne laissez pas votre système fonctionner trop longtemps (plusieurs minutes). Ça engorge le UART du SDK terminal et peut faire planter SDK. Le problème est que la prochaine fois que vous démarrez SDK, la fenêtre de départ ne s'ouvre plus… Pour être certain d'arrêter la trace utilisez les boutons reset du point 1).

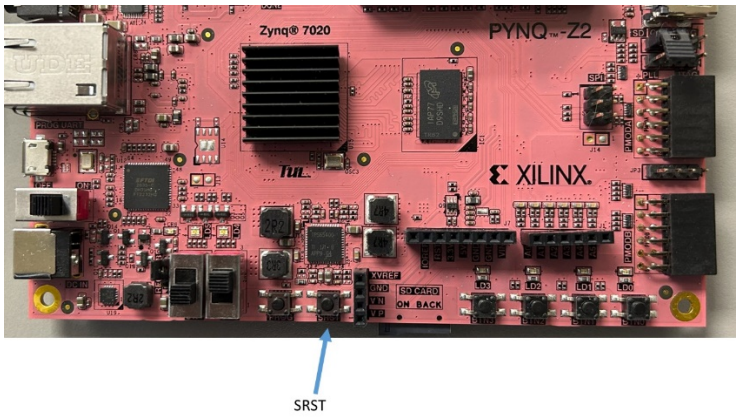


Figure 18

3. Réalisation 1)

Tâche à faire : partitionner le routeur en vous inspirant du producteur/consommateur

Après vous avoir familiarisez avec l'exemple du producteur/consommateur, vous êtes en mesure de compléter une implémentation AMP du routeur selon l'assignation suivante :

- core 0 = tout le système de la séance no 2 moins *TaskGenerate*
  - Créez une application lab1_partie3_A0 sur ps_cortexa9_0 pour laquelle vous allez utilisé le BSP *ex_consommateur_bsp* créé plus haut.
  - Copier dans le source votre lab 1 partie 2 et extraire (ou mettre en commentaire) de routeur.c et routeur.h tout ce qui concerne la tâche Task_Generate
  - Poursuivre plus bas avec la présentation du pseudo code du côté de TaskComputing (core 0)

- core 1 = *TaskGenerate*
  - Créez une application lab1_partie3_A1 sur ps_cortexa9_1 pour laquelle vous allez utilisé le BSP *ex_producteur_bsp* créé plus haut.
  - Les fichiers *generateur.h* et *generateur.c*[5] sont en grande partie donnés, mais vous devez bien les comprendre (en autres si vous voulez compléter ensuite la réalisation 2). De plus, ne pas oublier de copier la fonction *_computeCRC* pour créer le CRC (ligne 193), toujours en injectant 1 fois sur 10 une erreur de CRC.
  - Poursuivre plus bas avec la présentation du pseudo code du côté de TaskGenerate (core 1)

Ré-utilisez les 2 BSPs créés précédemment : BSP du consommateur pour core 0 et BSP du producteur pour core 1

Voici comment sera organisé la DDR vu de core 1 (dans les déclarations de *TaskGenerate*) :

```
// Variables partagees entre core 0 et core 1, au total:
// 4 mots de controle suivi d'au maximum 255 paquets de 16 mots
// Les 2 premiers mots servent à la synchronisation
// Le  3e mot sert a indiquer le no du burst
// Le  4e mot indique le nombre de paquet dans le burst (max de 255)
// Puis vont suivre les paquets (entre 1 et 255 paquets selon la fonction random de TaskGenerate)
// Attention ack de Task Generate doit être jumelé avec req de Task Computing
//        et req de Task Generate doit être jumelé avec ack de Task Computing

volatile uint32_t *req = (uint32_t*)(BASEADDR + 0x00);          // 1er mot: signal comme quoi on est prêt à recevoir un paquet
volatile uint32_t *ack = (uint32_t*)(BASEADDR + 0x04);          // 2e mot: signal comme quoi on est prêt à envoyer un paquet
volatile uint32_t *burst_no = (uint32_t*)(BASEADDR + 0x08);
volatile uint32_t *number_of_packets = (uint32_t*)(BASEADDR + 0x0C);

Packet *ppacket;
ppacket = BASEADDR + 0x10;  // C'est à partir de cet adresse qu'on va lire le 1er  paquet de 16 mots

*req = 0;
*ack = 0;
```

[5] Le code de TaskGenerate est légèrement différent du votre en ce sens qu'il n'utilise pas *isGenPhase* mais détermine tout de suite le nombre de paquet à générer et fait un for. Mais bref la fonctionnalité est identique à votre code.

Voici comment sera organisé la DDR vu de core 0 (dans les déclarations de types et variables de *TaskComputing*) qui servira à mettre en place le protocole du *handshacking :*

```
// Variables partagees entre core 0 et core 1, au total:
// 4 mots de controle suivi d'au maximum 255 paquets de 16 mots
// Les 2 premiers mots servent à la synchronisation
// Le  3e mot sert a indiquer le no du burst
// Le  4e mot indique le nombre de paquet dans le burst (max de 255)
// Puis vont suivre les paquets (entre 1 et 255 paquets selon la fonction random de TaskGenerate)
// Attention ack de Task Generate doit être jumelé avec req de Task Computing
//        et req de Task Generate doit être jumelé avec ack de Task Computing

volatile uint32_t *req = (uint32_t*)(BASEADDR + 0x04);          // 1er mot: signal comme quoi on est prêt à recevoir un paquet
volatile uint32_t *ack = (uint32_t*)(BASEADDR + 0x00);          // 2e mot: signal comme quoi on est prêt à envoyer un paquet
volatile uint32_t *burst_no = (uint32_t*)(BASEADDR + 0x08);
volatile uint32_t *number_of_packets = (uint32_t*)(BASEADDR + 0x0C);

Packet *ppacket;
ppacket = BASEADDR + 0x10;  // C'est à partir de cet adresse qu'on va lire le 1er  paquet de 16 mots

*req = 0;
*ack = 0;
```

Voici le pseudo-code du côté de *TaskComputing* (core 0):

1) On vérifie avec le flag que le système est bien initialisé et démarré via BP1 (Fig. 2)
2) *TaskComputing* indique avec req = 1 qu'il est prêt à consommer une rafale de paquet de *TaskGenerate* situé maintenant sur core 1.
3) On attend de manière active avec *while(!*ack)* que *TaskGenerate* est envoyé une rafale. À partir de 4) on va consommer la rafale.
4) ppacket = BASEADDR + 0x10; // Toujours partir une rafale à 0x10
5) Ici plutôt que de lire dans le fifo de TaskComputing avec OSTaskQPend, on va aller lire la rafale produite dans la DDR par TaskGenerate :

   Pour i = 1 jusqu'à **number_of_packets* lire la rafale complète de la mémoire partagée :

```
for (int i = 1; i <= *number_of_packets; ++i) {

    OSSemPend(&Sem_MemBlock,0, OS_OPT_PEND_BLOCKING, &ts, &err);
    Packet *packet = (Packet *)OSMemGet(&BLockMem, &err);
    packet->src = (ppacket->src);
    packet->dst = (ppacket->dst);
    packet->type = (ppacket->type);

    for(int i = 0; i < ARRAY_SIZE(packet->data); ++i)
        packet->data[i] = (ppacket->data[i]);

    ++nbPacketCrees;
```

   On fait ensuite le traitement que vous avez réalisé dans la partie 1 du lab 1 : vérification de l'espace d'addressage, vérification du CRC et si tout est OK on envoie dans le fifo de *TaskFIFOForwarding* selon la priorité du paquet.

   Puis on passe au prochain paquet de la rafale dans la DDR avec

```
ppacket++; // +17
```

6) On indique qu'on a fini de consommer avec *req = 0* et qu'on pourra passer à une autre valeur.
7) On attend de manière active avec *while(*ack)* que *TaskGenerate* est également prêt à passer à la prochaine rafale.

8) *TaskComputing* fait une pause avec un délai suffisant pour vider les fifos et pendant ce temps *TaskGenerate* peut préparer (en **parallèle**) la prochaine rafale.

Voici le pseudo-code du côté de *TaskGenerate* (core 1) avec au départ *int burst_number* = 1 :

1) Calcul aléatoire du nombre de paquets entre 1 et 255 avec *packGenQty* et mettre la valeur de p*ackGenQty* dans le 4e mot (number_of_packets)
2) ppacket = BASEADDR + 0x10; // Toujours partir une rafale après ses valeurs initiales à 0x10
3) Pour i = 1 jusqu'à packGenQty mettre dans la rafale complète dans la mémoire partagée :

```
for (int i = 1; i <= packGenQty; ++i) {

ppacket->src = rand() * (UINT32_MAX/RAND_MAX);
ppacket->dst = rand() * (UINT32_MAX/RAND_MAX);
ppacket->type = rand() % NB_PACKET_TYPE;

for (int j = 0; j < ARRAY_SIZE(ppacket->data); ++j)
    ppacket->data[i] = (unsigned int)rand();
ppacket->data[0] = ++nbPacketCrees;

ppacket++; // +16

OSTimeDlyHMSM(0, 0, 0, 1, OS_OPT_TIME_HMSM_STRICT, &err);

}
```

4) On incrémente de 1 *burst_number*
5) On attend de manière active que routeur (*TaskComputing*) sur core 0 soit prêt
6) On indique alors qu'une rafale est prête.
7) On attend de manière active que *TaskComputing* ait terminé le transfert de la rafale
8) On indique qu'on a fini l'envoie d'une rafale et on retourne à 1) pour préparer la prochaine.

Remarques importantes :

- Encore une fois, vous pourriez créer 2 nouveaux BSPs, mais comme indiqué précédemment, ***réutilisez les 2 BSPs créés pour l'exemple du producteur/consommateur : BSP du consommateur pour core 0 et BSP du producteur pour core 1.***
- N'oubliez pas non plus de modifier les linker scripts et finalement n'oubliez pas de désactiver la cache avant *OSStart()* de chaque programme.
- Votre timestamp venant de TaskGenerate (packet->timestamp = CPU_TS_Get64()) va maintenant calculer le temps d'un échange entre 2 cœurs + le temps de traitement des FIFOs.

Pour valider votre système, revenez au besoin sur les différentes traces utilisées dans les séances no 1 et no 2.

Réalisation 2)

Proposez un moyen de mettre fin à l'exécution du cœur 0 et du cœur 1. La fin de l'exécution du cœur 0 étant déjà partiellement en place (partie 2) :

i) Soit par le bouton BP1 qui mettra fin à l'exécution

ii) Si i) n'est pas arrivé après 60 secondes du axi_timer0. Le ISR de ce dernier (timer_isr) devra tourner 2 fois 30 secondes et faire la même chose que gpio_isr0 quand BP1 est activé.

En résumé la condition d'arrêt est soit BP1 ou soit 60 secondes d'exécution.

Une fois core 0 arrêté vous devez réaliser l'arrêt du cœur 1. Pour cela ajouter une variable au bloc DDR (BASEADDR = 0x3000000) dans lequel core 0 va indiquer que c'est terminé et que core 1 va périodiquement consulté afin de voir s'il poursuit ou pas la génération.

Réalisation 3)

Suite à la réalisation 2, il faudra faire une terminaison que je nommerais *propre.* Ça veut dire ici :

1) Vider les queues en rendant la mémoire (free)

   Ici je vois deux manières de faire :

   - en utilisant les fonctions uC/OS-III pertinente ou
   - encore de manière beaucoup plus simple avec :

   OSTimeDlyHMSM(0, 0, delai_pour_vider_les_fifos_sec, delai_pour_vider_les_fifos_msec, OS_OPT_TIME_HMSM_STRICT, &err);

   En choisissant un délai suffisamment long pour vider toutes les queues…

2) Créer une fonction *delete_events* qui va détruire ce qui a été créé dans *create_events* tout en utilisant les fonctions uC/OS-III pertinente.
3) Détruire toutes les tâches en utilisant les fonctions uC/OS-III pertinente.

5. Question pour le rapport

À venir

6. Barême de correction

Questions	**Notes**	**Commentaires**
Fonctionnement sur 5		
Total sur 8		

Date de remise :
xxx

Guy Bois
Responsable du cours