Polytechnique de Montréal - DGIGL

Laboratoire #1 : Routeur sur puce FPGA

INF3610 – Hiver 2024

Séance no 2 : Programmation d'interruptions

1. Objectifs de la séance no 2

Nous avons vu dans l'énoncé du laboratoire de la séance (ou partie) no 1, les objectifs généraux du lab 1, la mise en contexte et le fonctionnement du routeur, auquel vous avez modifié la fonctionnalité pour finalement réaliser un test de performance.

L'**objectif** de cette 2e séance est de se concentrer sur la couche bas niveau (*firmware*) du routeur dans laquelle vous allez programmer les ISR pour le *Fit Timer*, le *GPIO* et le *Timer* pour ainsi faire le lien avec votre code de la partie 1.

2. Quelques précisions sur l'objectif 1

La figure 1 illustre le fonctionnement du système qu'on souhaite réaliser dans cette partie 2.

Pour obtenir ce fonctionnement, vous devrez modifier les **2 tâches *TaskReset* et *TaskStop*** et ajouter **4 ISRs** (*gpio_isr1, gpio_isr2, fit_timer1_isr* et *fit_timer2).*

2.1) Initialisation et démarrage du système et reprise de service (sera assuré par l'*ISR gpio_isr 0* et *TaskReset*)

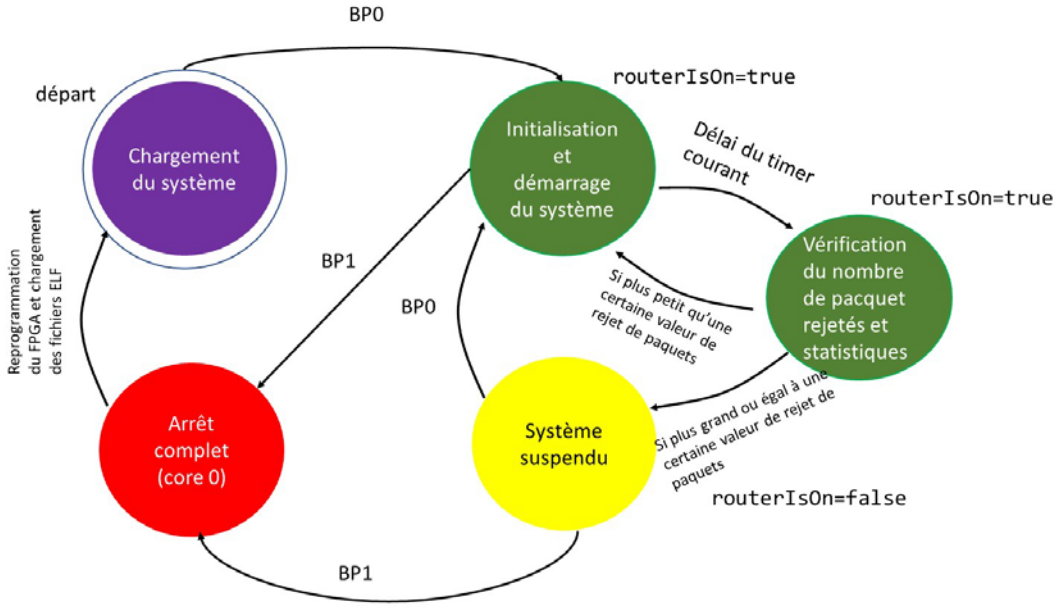


Figure 1. États du routeur pour la partie 2

Au départ, vous devrez concevoir une sous-routine d'interruption ISR (nommé *gpio_isr0*) liée au bouton pressoir (en anglais *press button*) qui lorsque le bouton est activé (pressée) va réaliser un

rendez-vous bilatéral avec la tâche *TaskReset* qui elle va à son tour démarrer le système c'est-à-dire les tâches du routeur tel que réalisé dans la partie 2.

TaskReset va donc permettre de mettre en fonction le système (état *Initialisation et démarrage du système* à la figure 1). Ce démarrage se fera une première fois (juste après l'état *Chargement du système*), mais aussi plus loin quand la tâche *TaskStop* (section 2.2) va suspendre le système (état *Système suspendu*). Donc, que ce soit pour démarrer ou redémarrer le système (routeur), vous utiliserez le bouton 0 (BP0). Le détail est donné plus bas à la section 4.3.

Finalement, à partir de l'état *Initialisation et démarrage* du système vous pourrez également appuyer sur le bouton 1 (BP1) afin de mettre fin à l'exécution du système. Dans ce dernier cas, on passe alors à l'état *Arrêt complet*. Le contrôle devra alors revenir à *StartupTask* et se suspendre et afficher. Nous décrivons plus bas à la section 4.6 ce qui est exactement demandé.

2.2) Vérification du nombre de paquets rejetés et statistiques (sera assuré par *fit_timer_isr0, fittimer_isr1, gpio_isr1 et TaskStop*)

Cet état permettra de faire une vérification du système et suspendre ce dernier en bloquant les tâches *TaskGenerate*, *TaskComputing*, *TaskFowarding*, *TaskOutputPort* et *TaskStats* après qu'un certain nombre de paquets rejetés ont été rencontrés (variables *nbPacketSourceRejete*)[1]. Plus précisément, ces interruptions, via les ISR de *fit_timer1 ou fit_timer2*, vont réaliser un rendez-vous bilatéral avec *TaskStop* [2]. Cette dernière va alors afficher les statistiques (*TaskStats*) [3] et regarder si le *nbPacketCRCRejete* a dépassé un certain nombre de paquets (p.e. 50). Dans l'affirmative, *TaskStop* va stopper les tâches du système (pour ainsi bloquer à nouveau *TaskGenerate*, *TaskComputing*, *TaskFowarding*, *TaskOutputPort* et *TaskStats*). On tombe alors dans l'état *Système suspendu* (en jaune) de la figure 2. Sinon (cas avec moins de 50 paquets rejetés), on revient à l'état *Initialisation et démarrage du système*.

Pour faire le choix du délai courant qui va ordonnancer TaskStats (*fit_timer1* ou *fit_timer2*) ou encore pour ne pas ordonnancer TaskStats, vous utiliserez les *switchs* (SW0 et SW1) selon les configurations suivantes :

[1] Attention dans la partie 2, on stoppait sur le *nbPacketSourceRejeteTotal*, cette fois ce sera sur *nbPacketSourceRejete*.
[2] Donc il faut en mettre moins que 1000 sinon ça ne stoppera jamais…
[3] Par rapport à la partie 2, cette fois, le rendez-vous sera inversé de TaskStop à TaskStats. De plus, vous pouvez mettre en commentaire OSTimeDLYHMSM(). On dit que TaskStats est remplacé par une minuterie matérielle.

01 : On fait un rendez-vous avec *TaskStats* pour faire afficher les statistiques à la fréquence de *fit_timer0*,

10 : On fait un rendez-vous avec *TaskStats* pour faire afficher les statistiques à la fréquence de *fit_timer1*, et

00 ou 11: on ne prend pas compte des ISRs de *fit_timer1* ou *fit_timer2* (les interruptions rentrent, mais on n'en tient pas compte) et donc on ne fait pas de rendez-vous avec *TaskStats*.

La variable globale contenant une de ces 4 valeurs se nommera *Stat_Period* (initialisée à 0) et elle sera mise à jour par les masques *NO_STAT*, *SWITCH1* et *SWITCH2* définis dans *interrup.h* (lignes 47 à 49).

À propos du nombre de paquets à considérer pour passer à l'état suspendu

Durant l'état suspendu (Fig. 1), seul la tâche système *idle* doit rouler. 50 paquets rejetés (print no 3 des statistiques) semblent un bon nombre pour atteindre l'état suspendu: vous devriez ainsi pouvoir voir **quelques affichages** de statistiques, avant de tomber dans l'état Système suspendu. Si à l'exécution de votre code, vous n'arrivez pas à voir du tout l'affichage des statistiques, augmenter à plus haut de 50. Juste avant de s'arrêter, le système devra afficher une dernière fois les statistiques.

Finalement, quand on passe à cet état suspendu, 2 possibilités pourront par la suite survenir:

1) vous appuyez sur le bouton 0 (BP0) afin de repartir le système et vous retournez alors à l'état *Initialisation et démarrage du système* (section 2.1), ou
2) vous pourriez également appuyer sur le bouton 1 (BP1) afin de mettre fin à l'exécution du système. Dans ce dernier cas, et tout comme en 2.1) lorsqu'on appuie sur BP1, on passe alors à l'état *Arrêt complet*. Le contrôle devra alors revenir à *StartupTask* pour préparer la fin. Encore une fois, nous décrivons plus bas ce qui sera exactement demandé pour cette fin.

Remarque : une fois cette fonctionnalité validée (c'est-à-dire le système est suspendu après quelques affichages de statistiques) et que vous pouvez le repartir ou l'arrêter, vous pourrez augmenter à 1000 paquets pour passer à d'autres validations du système.

3. À propos des boutons pressoirs

Vous devez donc réserver 2 boutons (BTN0 et BTN1) pour déterminer le fonctionnement. Vous utiliserez aussi les *switchs* (SW0 et SW1).

La figure 2 illustre ces boutons sur la PYNQ-Z2.

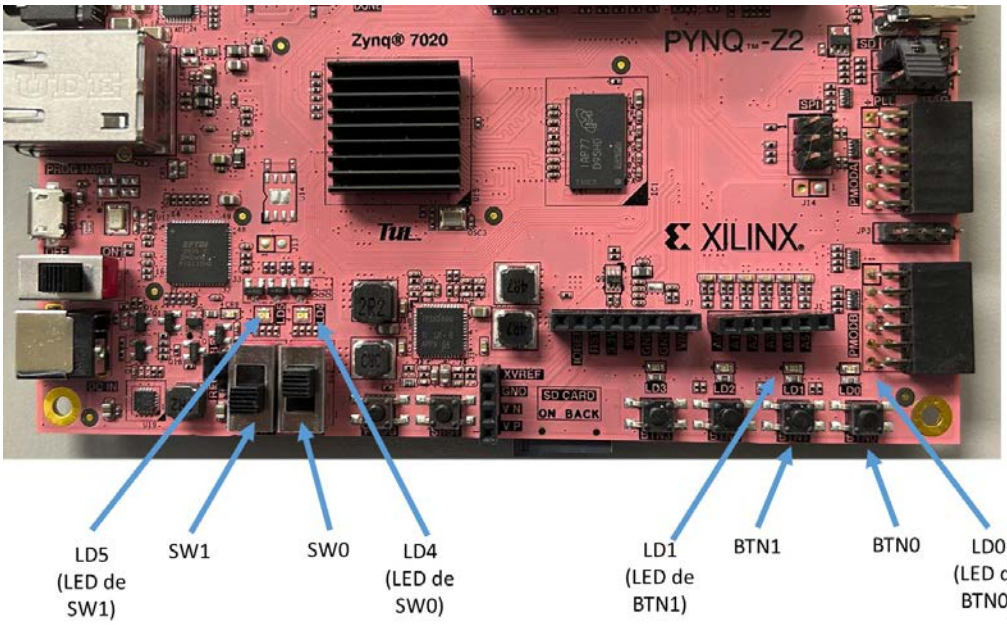


Figure 2 Emplacements des boutons et des switchs sur votre carte

À chaque fois que le système (re)démarre, le LED correspondant c.-à-d. *LED0* devra s'afficher pour indiquer que le système (re)-démarre. Même chose avec le LED1 quand on décide de terminer.

Tel qu'illustré sur la figure 3, on peut lire le numéro du bouton sur lequel on a appuyé via *XGpio_DiscreteRead* (en spécifiant le canal 1) alors qu'on peut allumer le LED correspondant à un bouton par *Xgpio_DiscreteWrite* (en spécifiant le canal 2). Plus précisément :

button_data = Xgpio_DiscreteRead(&gpButton, 1); //get button data from

Xgpio_DiscreteWrite(&gpButton, 2, button_data); //write switch data to the corresponding LEDs

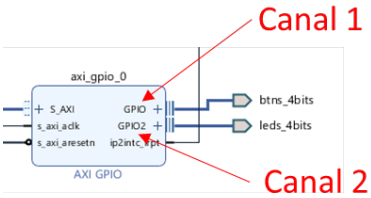


Figure 3 Un canal qui s'occupe de lire le bouton et un canal qui s'occupe d'allumer le LED correspondant

De manière similaire, on peut lire le numéro de la switch sur lequel on a appuyé via *XGpio_DiscreteRead* alors qu'on peut allumer le LED correspondant à une switch par *Xgpio_DiscreteWrite*. Plus précisément :

button_data = Xgpio_DiscreteRead(&gpSwitch, 1); //get button data
Xgpio_DiscreteWrite(&gpSwitch, 2, button_data); //write switch data to the corresponding LEDs

4 Travail à effectuer

4.1 Modifiez votre schéma Vivado

Tel que montrez à la figure 4 (page suivante), ajoutez dans votre schéma courant 1 gpio pour la switch et 1 fittimer1 pour une période de 12.000005 secondes. Puis, générez le *bitstream*, faites le *export* et *lauch SDK*. Votre lab 1 partie 1 va alors être mise à jour.

De plus, vous devez ajouter le composant timer (axi_timer_0) différent du fittimer dont nous verrons plus loin les différences. Pour le moment, allez chercher le composant axi_timer, faites le Run Connection Automation et ajouter une cinquième entrée aux interruptions à in4[0:0] (voir fig. 4).

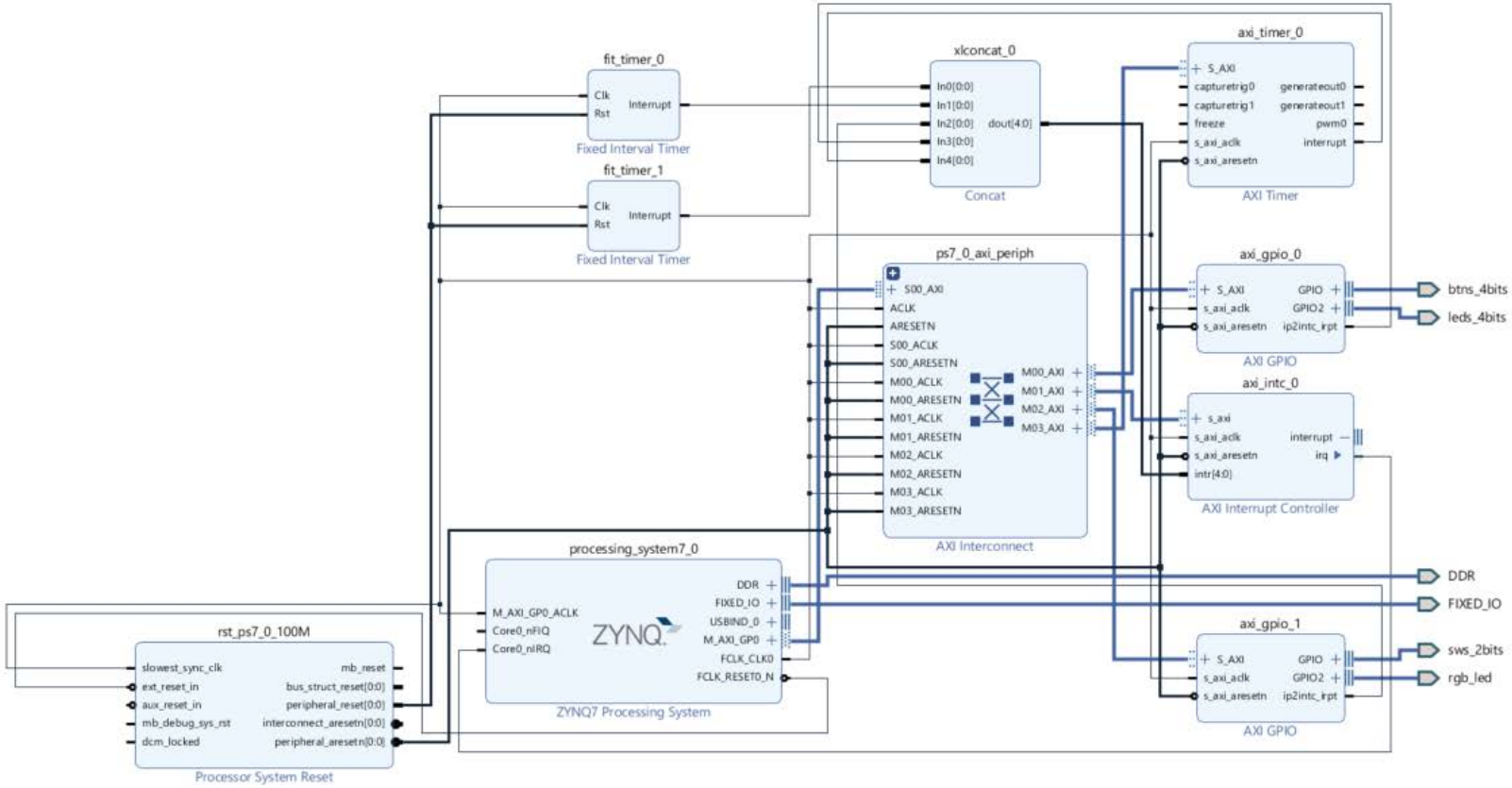


Figure 4. Mise à jour du schéma Vivado

4.2 Réorganisation des .h et ajouts de *interrupts.h* et *interrupts.c pour compiler sans erreur*

Fonctions pilotes (drivers) fournies

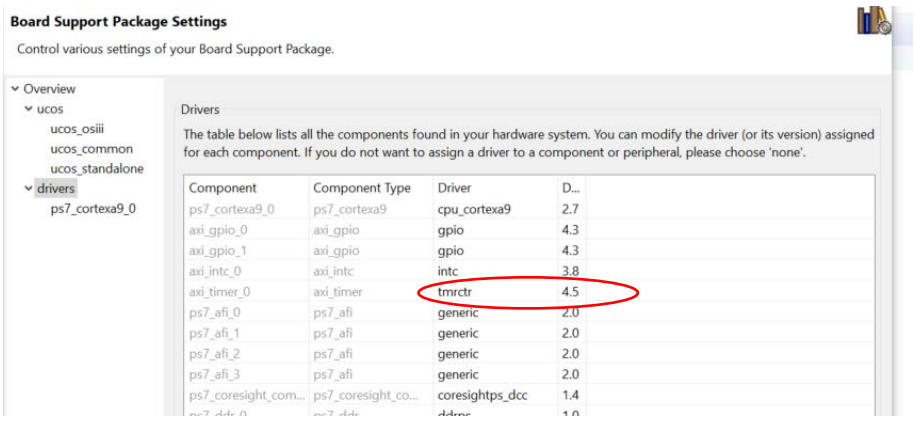
Les 15 fonctions suivantes seront données dans un code de départ sous le nom d'interruptions.c et interruptions.h que vous déposerez avec votre routeur.c et routeur.h. Prenez le temps de comprendre leurs rôles.

```
void initialize_gpio0();
void initialize_gpio1();
void initialize_timer();

int initialize_axi_intc();

int connect_fit_timer_irq0();
int connect_fit_timer_irq1();
int connect_timer_irq();

int connect_gpio_irq0();
int connect_gpio_irq1();

void connect_axi();

void cleanup();

void disconnect_intc_irq();

void disconnect_fit_timer_irq0();
void disconnect_fit_timer_irq1();

void disconnect_timer_irq();
```

Vous trouverez de la documentation sur ces fonctions en cliquant sous BSP Documentation (p. ex. , dans *gpio_v4_3* et *intc_v3_8*) juste en dessous de *ucos_bsp_0* (fenêtre de gauche dans SDK).

Changements dans routeur.h et routeur.c

- Créer d'abord une nouvelle application nommée lab1_part2 (avec le hello world uC/OS-III)
- Détruire le fichier *app.c*
- Copier les fichiers *routeur.c* et *routeur.h* de la partie 1
- Importer les fichiers *interruptions.c* et *interruptions.h* du code de départ
- Ajouter les drivers pour le axi_timer_0 dans le BSP :



- Puis effectuer les changements suivants dans *routeur.h* et *routeur.c* :

  1) Dans *routeur.h*, ajouter les 4 masques suivants (liés aux ISRs) juste après ceux liés aux tâches (aussi disponible dans le code de départ):

```
// Evenements (masques) lies aux ISRs
#define TASK_RESET_RDY          0x20    // RDV entre gpio_isr0 et TaskReset pour démarrer le système (le bit du masque est mis à 1)
#define TASK_SHUTDOWN           0x40    // RDV entre gpio_isr0 et StartUp  pour suspendre définitivement le système (le bit du masque est remis à 0)
#define TASK_STOP_RDY           0x80    // RDV entre fittimer0 ou fittimer1 et TaskStop pour suspendre temporairement le système (le bit du masque est remis à 0)
#define TASK_STATS_PRINT        0x100   // RDV entre TaskStop et TaskStats pour afficher les statistiques
```

  2) Couper le bloc suivant dans *routeur.h* :

```
/*DECLARATION DES COMPTEURS POUR STATISTIQUES*/
int nbPacketCrees = 0;                          // Nb de packets total crees
int nbPacketTraites = 0;                        // Nb de paquets envoyes sur une interface
int nbPacketSourceRejeteTotal = 0;              // Nb de packets total rejetés pour mauvaise source
int nbPacketSourceRejete = 0;                   // Nb de packets rejete pour mauvaise source pour 30 sec
int nbPacketCRCRejete;                          // Nb de packets total rejetés pour mauvaise CRC
int nbPacketCRCRejeteTotal;                     // Nb de packets total rejetés total pour mauvaise CRC
int Nb_of_ticks_in_CRC_total = 0;
int nb_calls_CRC = 0;
int packet_rejete_fifo_pleine_inputQ = 0;       // Utilisation de la fifo d'entrÃ©e
int packet_rejete_output_port_plein = 0;        // Utilisation des MB
int packet_rejete_fifo_pleine_Q = 0;
int delai_pour_vider_les_fifos_sec = 0;
int delai_pour_vider_les_fifos_msec = 50;
int print_paquets_rejetes = 0;
int limite_de_paquets= 100000;
```

     puis transférer le dans *routeur.c* juste avant la macro de *safeprint*

  3) Ajouter à ce bloc l'initialisation la variable *int routerIsOn = 0* qui indiquera si le système est en fonction ou pas.

  4) Ajouter à ce bloc l'initialisation la variable *int Stat_Period = 0* (définie à la section 2.2 en haut de la page 3) qui indiquera si le système affiche ou pas les statistiques et dans le cas où il affiche, s'il s'agit de *NO_STAT*, *SWITCH1* ou *SWITCH2*.

  5) Détruire les lignes suivantes de routeur.h :

```
#include <os.h>
#include <stdlib.h>
#include <inttypes.h>
#include <stdbool.h>
```

  6) Couper le bloc suivant dans routeur.c :

```
#include  <cpu.h>
#include  <lib_mem.h>

#include <os.h>
#include <stdlib.h>
#include <inttypes.h>
#include <stdbool.h>

#include <app_cfg.h>
#include <cpu.h>
#include <ucos_bsp.h>
#include <ucos_int.h>
#include <xparameters.h>
#include <KAL/kal.h>

#include <xil_printf.h>

#include  <stdio.h>
#include  <ucos_bsp.h>

#include <Source/os.h>
#include <os_cfg_app.h>

#include <xgpio.h>
#include <xintc.h>
#include <xil_exception.h>
```

Puis transférer le au tout début de *routeur.h* c'est-à-dire juste après:

```
/* * router.h

#ifndef SRC_ROUTEUR_H_EXT_
#define SRC_ROUTEUR_H_EXT_
```

Finalement n'oubliez pas d'ajouter à la fin : #include <xtmrctr.h>

7) Dans *routeur.c* ajouter un #include "interrupts.h"

8) Finalement, créez les fonctions ISRs *gpio_isr0*, *gpio_isr1*, *fit_timer_isr0*, *fit_timer_isr1* et *timer_isr* dans chacun de ces ISRs, et pour des raisons de débogages[4], mettez pour le moment uniquement un *xil_printf*:

```
xil_printf("---------------gpio_isr0---------------\n");    // dans l'ISR gpio_isr0
xil_printf("---------------gpio_isr1---------------\n");    // dans l'ISR gpio_isr1
xil_printf("---------------fit_timer_isr0---------------\n");  // dans l'ISR fit__isr0
xil_printf("---------------fit_timer_isr1---------------\n");  // dans l'ISR fit__isr1 et
xil_printf("---------------timer_isr---------------\n"); / dans l'ISR timer_isr
```

Puis ajouter XGpio_InterruptClear(&gpButton, XGPIO_IR_MASK); et XGpio_InterruptClear(&gpSwitch, XGPIO_IR_MASK); dans gpio_isr0 et gpio_isr1 resp. Sinon ça va partir en boucle car l'interruption n'est jamais remise à 0.

Dans la fonction startup, vous devrez appeler 4 fonctions de drivers afin d'initialiser les différents périphériques juste après *UCOS_IntInit();*:

[4] Une fois ce débogage terminer on doit enlever ces xil_printf car il amène une composante non déterminisme non souhaité dans un ISR.

```
UCOS_Print("Programme initialise - \r\n");

initialize_gpio0();
initialize_gpio1();
initialize_timer();
initialize_axi_intc();
connect_axi();
```

N'oubliez pas non plus de mettre assez de mémoire dans le heap (0xA00000) du fichier leaker script. À partir de là ça devrait compiler sans erreur et vous devriez pouvoir exécuter votre routeur avec la fonctionnalité de la partie 1. *TaskStats* sera encore ordonnancer par *OSTimeDlyHMSM(0, 0, 30, 0, OS_OPT_TIME_HMSM_STRICT, &err);* et vous verrez aussi afficher à la fréquence de 20sec le print de *fit_timer_isr0* et à la fréquence de 12.00005 sec le print de *fit_timer_isr1*. De même, quand vous appuyez sur un des boutons pressoir vous verrez afficher le print de *gpio_isr0* et finalement quand vous bougerez une switch, le print de *gpio_isr1*.

Le timer quant à lui ne démarrera pas car on ne l'a pas démarré. On y reviendra plus bas.

Dans ce qui suit, on va y aller étape par étape. On va d'abord mettre en service le bouton pressoir pour démarrer le système c'est-à-dire le compléter le code de ISRs *gpio_isr0* et adapter la tâche *TaskReset.*

N.B. Mettre l'implémentation des 5 ISRs juste avant TaskReset. Ça va faciliter le travail pour la correction du chargé.

4.3 ISR *gpio_isr0* et *tâche TaskReset.*

Voici le pseudo-code que vous devez compléter en vous inspirant de la section 3.2:

ISR gpio_isr0 (à mettre juste avant TaskGenerate dans routeur.c)

```
void gpio_isr0(void *p_int_arg, CPU_INT32U source_cpu) {
   CPU_TS ts;
   OS_ERR err;
   OS_FLAGS  flags;
   int button_data = 0;

   // N.B. Inspirez-vous du no 34 (fig. 5 plus bas) pour le DiscreteRead et
   // DiscreteWrite

   xil_printf("---------------gpio_isr0---------------\n");

   // On regarde quel bouton a été activé (BP0 ou BP1)[5] en le mettant dans
   // button_data
```

[5] Si parfois le bouton est à 0, consultez la section 4.7

```
        // Si button_data == BP0 on :
        // 1) affiche le LED correspondant
        // 2) on fait un rendez unilatéral avec OSFlagPost pour débloquer TaskReset
        //    via Flag (masque TASK_RESET_RDY) qui va (re)démarrer le système.
        //    Attention ne pas appeler l'ordonnanceur puisque que ce même
        //    ordonnanceur sera de toute façon appelé à la toute fin du ISR via
        //    OSIntExit().

        // Si button_data == BP1 on :
        // 1) affiche le LED correspondant
        // 2) on fait un rendez unilatéral pour débloquer StartupTask via flag
        //    (masque TASK_SHUTDOWN) qui va alors s'occuper d'arrêter le système
        //    Attention ne pas appeler l'ordonnanceur…

        // On met à 0 les interruptions du GPIO avec l'aide du masque
        //  XGPIO_IR_MASK
}
```

Notez que pour l'affichage on a créé une macro *TurnLED(color)* dans le fichier de départ *interrupt.h*. Ici *color* est la sortie de *XGpio_DiscreteRead* (section 3.1).

Tâche TaskReset (à adapter)

```
void TaskReset(void* data) {

        CPU_TS ts;
        OS_ERR err;
        int i;
        while (true) {

        // mettre ici le code du RDV unilatéral avec gpio_isr0 avec OSFlagPend via
        // flag (masque TASK_RESET_RDY).
        // Attention de bien consommer afin que TaskReset bloque de nouveau à la
        // prochaine itération.

       xil_printf("--------------------- Task Reset --------------------\n");

        routerIsOn = 1; // Var. globale qui indique que le système (re)démarre

        // Réutiliez le code de la partie 1 qui fait un rendez unilatéral pour
        // débloquer via le masque TASKS_ROUTER  les tâches TaskGenerate,
        // TaskComputing, TaskFowarding, TaskOutputPort et TaskStats
        }
}
```

À propos du no 34

Si vous prenez le temps d'écouter le vidéo du no 34 (chapitre 2, bloc 5 du site web), vous verrez en détail l'explication du code du *gpio_isr* dont le code est le suivant :

```
150 void gpio_isr(void *p_int_arg, CPU_INT32U source_cpu) {
151         OS_ERR err;
152         CPU_TS ts;
153         int button_data = 0;
154 
155         button_data = XGpio_DiscreteRead(&gpButton, 1); //get button data
156 
157         XGpio_DiscreteWrite(&gpButton, 2, button_data); //write botton data to the LEDs
158 
159         XGpio_InterruptClear(&gpButton, XGPIO_IR_MASK);
160 
161         OSSemPend(&Sem, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err);
162         UCOS_Print ("gpio isr \r\n");
163         OSSemPost(&Sem,  OS_OPT_POST_1 + OS_OPT_POST_NO_SCHED, &err);
164     }
```

Figure 5

Vous remarquerez sur cette figure 5 que *gpio_isr* va faire un rendez-vous unilatéral avec une tâche uC via un sémaphore *Sem*. On a donc ici un rendez-vous similaire entre *gpio_isr0* et *TaskReset* mais via *OSFlagPost* et *OSFlagPend* respectivement.

Dans la prochaine section, nous allons mettre en service les 2 minuteries matérielles qui remplaceront la minuterie logicielle *OSTimeDlyHMSM(0, 0, 30, 0, OS_OPT_TIME_HMSM_STRICT, &err);* dans TaskStats **(donc à partir de maintenant mettre cette ligne en commentaire)**. Autrement dit, *fit_timer_isr0* ou *fit_timer_isr1* se synchroniseront avec *TaskStop* qui elle affichera les statistiques en se synchronisant avec *TaskStats* et suspendra temporairement le système si le nombre de paquets rejetés depuis le dernier affichage (variable *nbPacketSourceRejete* == 50 paquets).

4.4 Création des ISRs fit_timer_isr0 et fit_timer_isr1, ainsi que l'adaptation de la tâches TaskStop.

On va d'abord créer les 2 minuteries matérielles.

ISR fit_timer_isr0 et ISR fit_timer_isr1 (à mettre juste après ISR gpio_isr0 dans routeur.c)

```
void fit_timer_isr0(void *p_int_arg, CPU_INT32U source_cpu); {

	OS_ERR perr;
	CPU_TS ts;
	OS_FLAGS  flags;

xil_printf("------------------ FIT TIMER 0 ------------------\n");

// Si Stat_Period == SWITCH1
//On fait le rendez-vous avec TaskStop  via les flags avec le masque
//TASK_STOP_RDY
//Attention ne pas appeler l'ordonnanceur…

}

void fit_timer_isr1(void *p_int_arg, CPU_INT32U source_cpu); {

OS_ERR perr;
CPU_TS ts;
OS_FLAGS  flags;

xil_printf("---------------FIT TIMER 1 ---------------\n");

// Si Stat_Period == SWITCH2
//On fait le rendez-vous avec TaskStop  via les flags avec le masque
//TASK_STOP_RDY
//Attention ne pas appeler l'ordonnanceur…

}
```

Puis, on va adapter les 2 minuteries à TaskStop :

Tâche TaskStop (à adapter)

```
void TaskStop(void* data){
	CPU_TS ts;
	OS_ERR err;
	while (true) {

		// mettre ici le code du RDV unilatéral avec fit_timer_isr0 ou
		// fit_timer_isr1 via flag (masque TASK_STOP_RDY). Attention de bien
		// consommer afin de bloquer de nouveau la prochaine itération…

		if (!routerIsOn) {
			continue ;	// fit_timer0 et fit_timer_1  poursuivent même
					//  quand le système est arrêté,
					// mais on fait rien (voir Fig. 1)
```

```
            }

            // xil_printf("-------- Statistics & Task stop check ---------\n");

            // Rendez unilatéral pour débloquer TaskStats pour 1 affichage avec
            // OSFlagPost et le masque TASK_STATS_RDY[6]
            //

            // Test sur nbPacketCRCRejete
            // Si nbPacketSourceRejete a pas dépassé la valeur limite (50)
            // alors 3 choses:
                  1) safeprintf("-------- Task stop suspend all tasks -------
                     \n");
                  2) routerIsOn = 0;
                  3) // Mettre le code de la partie 1 qui avec OSFlagPost fera
                     // bloquer TaskGenerate, TaskComputing,
                     // TaskFowarding,TaskOutputPort et TaskStats
                     // en mettant le masque TASKS_ROUTER à 0
            }
      }
}
```

Également, dans TaskStats, ajouter le code suivant à la suite des xil_printf qui permettra de savoir sur quelle fréquence nous sommes :

```
            if (stat_Period == SWITCH1       ) {

                  xil_printf("26- Frequence des stats de 20 sec :\n");

            };

            if (stat_Period == SWITCH2       ) {

                  xil_printf("26- Frequence des stats de 12.000005 sec :\n");

            };
```

Dans ce qui suit, nous allons créer le ISR du 2e gpio qui pilote la switch.

[6] Dans le code de départ de la partie 2 j'ai redéfini DEFINE TASK_STATS_RDY à 0x100 plutôt que 0x010 comme dans la partie 1. La raison est que maintenant la tâche statistique ne dépend que d'un rendez-vous de TaskStop. Le masque qui permet le démarrage et l'arrêt du système (#define TASKS_ROUTER 0x0F) ne couvre donc plus Task_Stats. Assurez-vous donc de détruire l'ancien DEFINE i.e. DEFINE TASK_STATS_RDY à 0x10.

4.5 Création de ISRs gpio_isr1 (à mettre juste après ISR fit_timer_isr1 dans routeur.c)

```
void gpio_isr1(void *p_int_arg, CPU_INT32U source_cpu) {
	CPU_TS ts;
	OS_ERR err;
	int switch_data = 0;

	// N.B. Inspirez-vous du no 34 (fig. 4) pour le DiscreteRead et
	// DiscreteWrite…

	// On regarde quelle switch a été activée via switch_data

	// Si switch_data == SWITCH1 on affiche le LED correspondant  et la
	variable globale Stat_Period = SWITCH1

	// Si switch_data == SWITCH2 on affiche le LED correspondant  et la
	variable globale Stat_Period = SWITCH2

	// Si switch_data == NO_STAT ou switch_data == SWITCH1and2 on affiche
	NO_STAT pour éteindre le LED précédent  et la variable globale Stat_Period
	= NO_STAT

	// On met à 0 les interruptions du GPIO avec l'aide du masque
	//  XGPIO_IR_MASK
}
```

Attention vous pouvez utiliser de belles couleurs comme COLOR_DOUBLE_PURPLE ou COLOR_DOUBLE_GREEN pour SWITCH1 et SWITCH2 (XGpio_DiscreteWrite).

Finalement, il nous reste à adapter *StartupTask* afin de reprendre le contrôle quand on va suspendre définitivement le système (quand button_data == BP1 dans gpio_isr0).

4.6 Modifications à la tâche *StartupTask pour terminaison*

Dan StartupTask (dernière ligne), remplacez *OSTaskSuspend((OS_TCB *)0,&err)* par :

```
	UCOS_Print("Router initialized - Ready to launch - Hit push button\r\n");

	Mettre ici le OSFlagPend() qui bloque sur TASK_SHUTDOWN

	// Dans la partie 3 nous procéderons à la destruction des tâches ici

	UCOS_Print("Prepare to shutdown System - \r\n");

	while (1) {	// indique que le système est en arrêt permanent
			TurnLEDButton(0b1111);   // mettre 4 bits
			OSTimeDlyHMSM(0, 0, 1, 0, OS_OPT_TIME_HMSM_STRICT, &err);
			TurnLEDButton(0b0000);
			OSTimeDlyHMSM(0, 0, 1, 0, OS_OPT_TIME_HMSM_STRICT, &err);
	}
```

4.7 Création du ISR *timer_isr*

Le *fittimer* est une minuterie à sa plus simple expression. On lui donne dans Vivado un load (champs Number of Clocks) et une interruption est levée quand le compteur a exécuté la valeur du load à la vitesse de 100 MHz.

Le *timer* lui est plus sophistiqué que le *fittimer*. Chaque composant offre 2 timers de 32 bits (Figure). Il utilise plusieurs modes de fonctionnement dont un mode génération similaire au *fittimer*, mais avec la possibilité d'incrémenter ou décrémenter et d'offrir des fonctions pour lire la valeur du compteur. Il offre aussi une configuration PWM pour permettre la génération d'onde carrée avec rapport cyclique autre que 50%.

La minuterie compteur de 32 bits roule à 100 MHz donc incrémente 100 000 000 fois par seconde. Si vous le laissez compter 42 secondes (4 200 000 000) il sera proche du débordement. Nous verrons dans la partie 3 son utilisation dans une configuration 64 bits (2 timers en cascade), mais pour le moment nous allons simplement le mettre en service dans une configuration 32 bits et vérifier son fonctionnement en mesurant la valeur de 1 tick d'horloge de uC/OS-III.

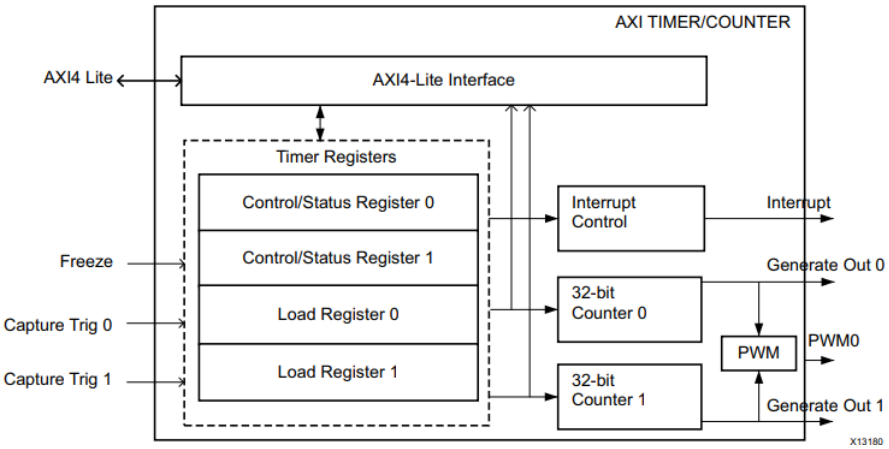


Figure 6 AXI-Timer 2.0

Vous avez 2 choses à faire ici:

1) Dans routeur.c juste complétez le ISR *timer_isr* en fonction de la configuration *initialize_timer()* dans interrupts.c. Si votre ISR est bon, cette interruption fera son print à toute les 42 secondes et se réinitialisera.

2) après la ligne le bloc de code *xil_printf("\nfreq du timestamp: %d\n", freq_hz);* ajoutez le code suivant qui calibre la minuterie:

```
// Calibrate HW timer
XTmrCtr_Start(&timer_dev, XPAR_AXI_TIMER_DEVICE_ID);

unsigned int init_time = XTmrCtr_GetValue(&timer_dev,
XPAR_AXI_TIMER_DEVICE_ID);
unsigned int curr_time = XTmrCtr_GetValue(&timer_dev,
XPAR_AXI_TIMER_DEVICE_ID);
unsigned int calibration =  init_time - curr_time;
```

3) Puis, juste après le code de 2) faites le code qui va calculer la valeur exacte de 1000 ticks en attente active. Inspirez-vous ajouter du code fait dans la partie pour le timestamp (différence de temps entre le moment ou le paquet est créé et celui ou il est mis dans le fifo de TaskOutputPort). Ce code va servir à montrer la bonne fonctionnalité correct du timer que nous ré-utiliserons dans la partie 3. N'oubliez pas d'ajouter la calibration.

4.8 Autres points à faire attention :

- Quand vous exécuté *gpio_isr0* et que vous testez (via un break point) la valeur de *button_data*, si parfois *button_data* égal à 0 c'est qu'il y a un problème (selon interrupts.h cette valeur devrait donner 8 si vous avez appuyé sur BP0, 4 si vous avez appuyé sur BP1, etc..). Assurez-vous d'appuyer quelques secondes sur le bouton (pas juste l'effleuré) et placez le break point après le test (i.e. dans le if) :

```
If (button_data == 0b1000) {
// mettre le point d'arrêt sur une des instructions de la boucle.
}
```

Si le problème persiste (button_data = 0) consultez l chargé

- Lorsque vous mettrez gpio_isr1 à NO_STAT longtemps et que vous le ramènerez à SWICTH1 ou SWITCH2, ne soyez pas surpris si le système est suspendu temporairement. En effet le nombre de paquet rejeté pourrait alors croître de manière importante. Vous n'avez qu'à repartir le système (BP1).

5. Questions pour le rapport et date de remise

Q1. Expliquez en détail la configuration et la programmation du AXI Timer de la section 4.8. Expliquez comment vous avez fait votre test pour déterminer la valeur de 1000 ticks avec votre timer et le rôle de la calibration Hw dans votre calcul. Indiquez aussi ce qui ce serait passé si on avait fait une attente active de plus de 42 secondes.

Q2 Qu'est-ce qu'on en commun le timer et le timestamp de la partie 1 du lab 1? Expliquez. Auriez-vous été capable de réaliser le calcul du délai d'un paquet de la partie 1 avec le timer? Ou encore résoudre le no 3d.

Barême de correction

Questions	**Notes**	**Commentaires**
Q1		
Q2		
Fonctionnement sur 5		
Total sur		

Date de remise :
Gr. 1 mardi le 29 février 2024 avant minuit

Guy Bois
Responsable du cours