Interruption du PS du Zynq pour informer d'un événement dans le PL

G. Goavec-Merou, J.-M Friedt, 7 février 2019

Tous documents autorisés, connexion internet autorisée pour rechercher les documents cités dans le sujet, communications et téléphones portables proscrits. Les url des sites accédés au cours de l'examen sont susceptibles d'être enregistrées. **Correction à** `http://jmfriedt.free.fr/exam2018_M2EEA.pdf`

L'architecture Zynq se prête particulièrement bien aux applications de datation d'évènements, avec par exemple des compteurs implémentés dans le FPGA – similaires aux *timers* des microcontrôleurs – et le processeur généraliste exécutant GNU/Linux en charge de traiter les mesures et les transférer aux utilisateurs, par exemple via internet. Nous nous intéressons à un exemple de datation d'évènements sur l'impulsion 1-PPS (1 impulsion/seconde) issue des récepteurs GPS, caractérisés par une incertitude sur le temps de l'ordre de la centaine de nanosecondes. Notre objectif dans ce sujet est de nous intéresser exclusivement à la communication entre PL et PS, dans lequel une interruption sera déclenchée par le PL par le 1-PPS, à laquelle le PS doit pouvoir réagir, par exemple pour récupérer la date stockée dans le compteur avant qu'un nouvel événement se produise.

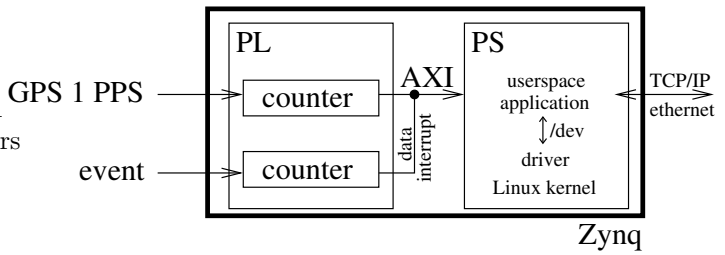


Figure 1: Architecture matérielle mise en œuvre pour ce problème. Dans notre cas, nous éliminons les compteurs, et ne nous intéressons qu'à une des deux voies d'évènements, simulée par un GPIO.

Une archive avec divers squelettes de programmes en C et fichiers de configuration de Vivado est disponible à `/home/jmfriedt/exam2018_M2EEA.tar.gz`.

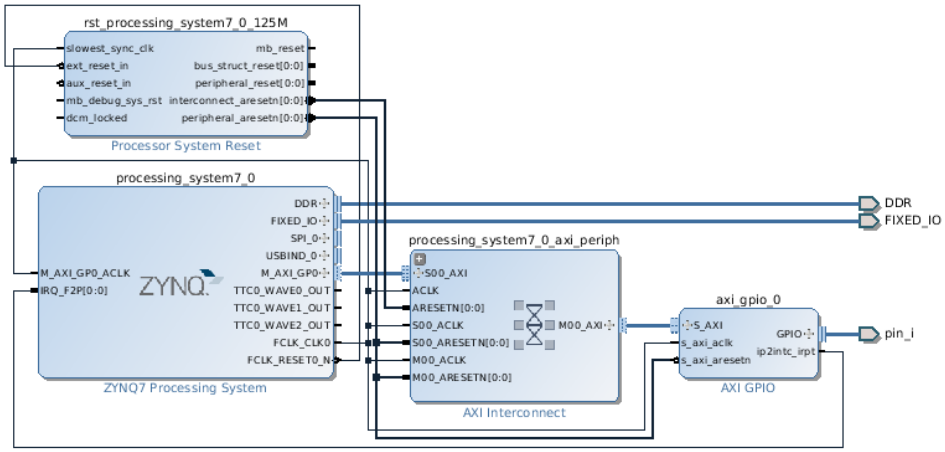
1 Simulation du 1-PPS par GPIO

En l'absence de récepteur GPS, nous simulons le 1-PPS par une impulsion générée de façon logicielle sur un GPIO. Cependant, nombre de GPIOs de la Redpitaya sont déjà utilisés à d'autres fins. Nous devons donc configurer le GPIO pour cette fonction et non une autre fonction étendue sélectionnée par défaut.

1. En étudiant le schéma de la Redpitaya, fichier Red_Pitaya_Schematics_STEM_125-10_V1.0.pdf à `http://downloads.redpitaya.com/doc/`, pouvons nous trouver un GPIO connecté au PS qui soit disponible, i.e. ne soit pas utilisé pour une autre fonction (e.g. communication), pour simuler le 1-PPS ? Si oui, laquelle ? Si non, justifier.
2. Il est classique d'implémenter le signal de déclenchement d'une transaction SPI par un GPIO. Identifier la broche permettant une telle fonction sur le connecteur E2 de la Redpitaya : à quelle GPIO du PS est-ce que cette broche est liée ?
3. Quelle est l'adresse de base à partir de laquelle tous les registres de configuration des GPIOs sont situés ? Quelle est l'adresse de base à partir de laquelle les configurations générales du système sont sont accessibles – en particulier l'horloge cadençant les GPIOs et la fonction attribuée à chaque broche ?
4. Quel registre du Zynq permet de configurer la fonction de la broche `XX` associée au CS# du SPI ? On pourra étudier pour cela `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf` et ainsi fournir l'offset par rapport à l'adresse de base de ce registre, sachant que le nom du registre est de la forme `MIO_PIN_XX`.
5. La section 14.5 de UG585 explique comment configurer une broche en GPIO : quelle est la valeur contenue dans le registre de configuration de la broche sélectionnée après mise sous tension de la Redpitaya ? On pourra accéder à cette information soit depuis le shell linux de la redpitaya, soit par un programme en C. Est-ce que cette configuration correspond à un GPIO ? si non, quel(s) bit(s) modifier afin d'atteindre la fonction GPIO ?
6. En s'inspirant du programme `gen_sig.c`, compléter les éléments manquant pour activer cette broche. On pourra s'appuyer sur `ledlib.c` qui aide à accéder aux registres de configurations du système en débloquant leur accès. Démontrer la capacité à faire changer d'état, une fois par seconde, le niveau de la broche – soit depuis le shell, soit depuis un programme en C. On démontrera sur oscilloscope ou avec une LED le bon fonctionnement du programme **(3 points)**.

2 Déclenchement d'une interruption du PL

Nous avons vu que le PL peut être configuré pour supporter des blocs reconfigurables de GPIO. Parmi les fonctionnalités de ces blocs, il est possible d'associer une interruption à chaque GPIO interne au PL. Notre objectif est désormais d'informer le PS du déclenchement d'un évènement (interruption) dans le PL.



7. En étudiant le diagramme en blocs dans Vivado, identifier la broche du Zynq à laquelle est connectée la broche du GPIO dans le PL. Est-ce que cette broche est accessible sur un des connecteurs de la Redpitaya ? si oui, quelle broche de quel connecteur ? **(2 points)**
8. Ayant identifié la broche sur laquelle nous connecterons le 1 PPS (ou sa simulation que nous venons de concevoir dans le chapitre précédent), nous allons proposer un module noyau pour gérer cette interruption et transmettre l'information de déclenchement d'une interruption à l'espace utilisateur en débloquant une lecture bloquante. Pourquoi devons nous appréhender ce problème au niveau du noyau, et ne pouvons nous pas écrire un gestionnaire d'interruption en espace utilisateur ?
9. En étudiant la documentation de l'IP GPIO communiquant sur bus AXI fournie par Xilinx, quels sont les registres permettant d'activer les interruptions de ce GPIO ? à quelle adresse absolue se trouvent ces registres ? (une réponse de la forme `base+offset` est acceptable).
10. Un squelette de module est proposé dans `axi_gpio_it_drv.c`. Nous y démontrons en particulier l'enregistrement de l'interruption auprès du noyau et l'association d'une fonction dédiée (*handler*) à l'acquittement de l'évènement. Ajouter un mécanisme de blocage de la fonction `read` qui ne rende la main que lorsque l'interruption s'est déclenchée. Comment s'appelle le mécanisme utilisé ?
11. Compiler le module et l'insérer dans le noyau. Pour cette seconde étape, on passera par un greffon au *devicetree* qui garantit la cohérence entre le *bitstream* configurant le FPGA et le pilote noyau associé. En particulier, le mécanisme proposé crée automatiquement dans */dev* un point d'entrée du nom du nœud ajouté dans le *devicetree overlay*. Quel est le nom de ce point d'entrée nouvellement créé au chargement de l'overlay/bitstream/pilote ? **(3 points)**
12. Rappeler la différence entre un périphérique (*device*) et un pilote (*driver*) dans la séquence des ajouts de fonctionnalités au noyau ? De quelle partie le *devicetree* prend-t-il la place ? Justifier de cette stratégie de description du matériel par rapport à une description figée de chaque plateforme matérielle.
13. Proposer un programme en espace utilisateur qui ouvre ce pseudo-fichier de communication avec le noyau, et y lit une valeur. Compiler et exécuter ce programme : démontrer que la lecture bloque tant que l'interruption n'est pas déclenchée **(2 points)**.
14. Connecter la sortie GPIO de la première section à l'entrée interruption de la seconde, et démontrer que la lecture se débloque à chaque changement d'état du GPIO simulant le 1 PPS.

3 Conclusion

Nous avons pu mettre en œuvre un mécanisme d'interruption pour prévenir PS qu'un évènement s'est produit dans le PL. Cependant, la norme du 1 PPS du GPS définit la seconde sur le front montant du signal, alors que la date du front descendant n'est pas définie. Il faut donc garantir que le message ne concerne que le front montant et non le front descendant de la transition du GPIO.

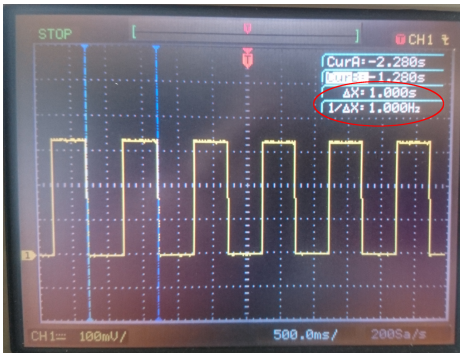
15. Y-t-il un mécanisme pour sélectionner le front de déclenchement de l'interruption d'un GPIO configuré dans le FPGA ? Si oui, quel est-il ? Si non, quel mécanisme pourrions nous imaginer pour que le logiciel ne débloque la lecture du processus utilisateur que sur un front montant ?

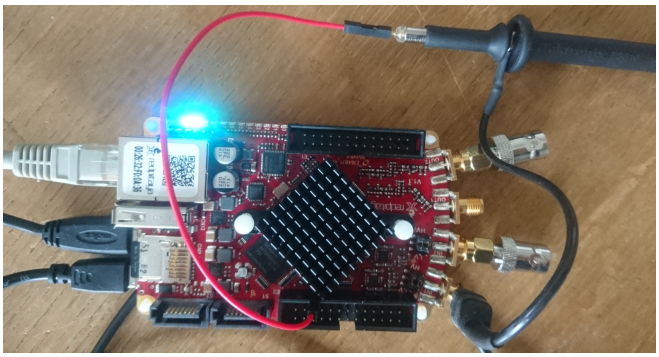
Correction :

1. aucun GPIO du MIO n'est disponible : toutes les broches liées aux PS sont configurées dans leur fonction étendue de communication par défaut.
2. CS# de SPI1 est la broche 6 du connecteur E2, aussi nommée MIO13 sur le Zynq.
3. 0xE000A000 pour les GPIOs, mais avant cela il faut passer le CS# SPI en mode GPIO, qui s'obtient avec les registre MIO_REG_XX qui sont à l'échelle du système, donc en 0xF8000000+0x700
4. MIO_PIN_13 est un registre qui configure la broche MIO13 routée sur SPI1_CS#. Son adresse est base+0x734 avec base=0xF8000000 (p.1583 de UG585).
5. L0=L1=L2=0 et L3=000 tel que décrit en page 1658 selon la procédure de configuration décrite en page 389 (section 14.5 : pour activer un MIO, L0=L1=L2=0 et L3=000)
6. depuis un programme C :

```c
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <sys/time.h>
#include "ledlib.h"

#define TIMESLEEP 500000

int status;         // declenche' a chaque appel du timer => exterieur
unsigned char *h;

void test(__attribute__ ((__unused__))int signum)
{red_gpio_output(h, 13, status);
 status ^= 0xff;
}

int main(void)
{ struct sigaction sa;
  struct itimerval timer;
  status = 0;

  h = red_gpio_init(13);  // PS_MIO13=E2-6=SPI_CS\#
  red_gpio_set_cfgpin(h, 13);

  memset(&sa, 0, sizeof(sa));
  sa.sa_handler = &test;
  sigaction(SIGVTALRM, &sa, NULL);
[...]
  while(1) {}
  red_gpio_cleanup();
  return 0;
}
```

Le résultat d'un tel programme, en plaçant correctement le point de mesure sur E2-6, est

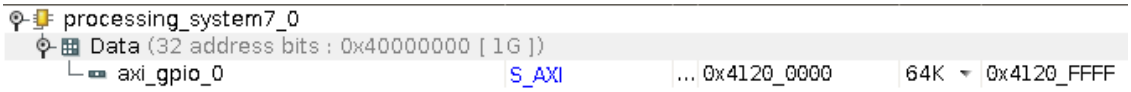




7. contrainte dans Vivado : pin_i=H17=DIO1_N sur le connecteur E1 d'après le schéma de la Redpitaya

```
set_property IOSTANDARD LVCMOS33 [get_ports {pin_i_tri_i}]
set_property SLEW SLOW [get_ports {pin_i_tri_i}]
set_property DRIVE 8 [get_ports {pin_i_tri_i}]

set_property PACKAGE_PIN H17 [get_ports {pin_i_tri_i}]
```

L'adresse de base des registres du bloc GPIO est donné par Vivado :



8. seul le noyau a accès à la ressource interruptions, et distribue cette information aux processus. Une gestion des interruptions matérielles au niveau de l'espace utilisateur ne saurait garantir l'acquittement par un processus donné sans être perturbé par les autres processus.
9. l'étude de `AXI GPIO v2.0 -- LogiCORE IP Product Guide` disponible sous le nom de `pg144-axi-gpio.pdf` nous indique en page 13 que deux registres configurent les interruptions. D'une part GIER est le registre d'activation global des interruptions, et d'autre part chaque interruption individuellement associée à chaque GPIO se configure par IPIER (IP Interrupt Enable). Cette configuration s'applique dans le pilote par

```
int axi_gpio_it_drv_open(struct inode *inode, struct file *filp)
{ [... creation du device ...]
  writel(GIER_GIE,  dev->membase + GPIO_GIER);   // globale interrupt
  writel(IER_CH1_IE,dev->membase + GPIO_IER);
 [...]
}
```

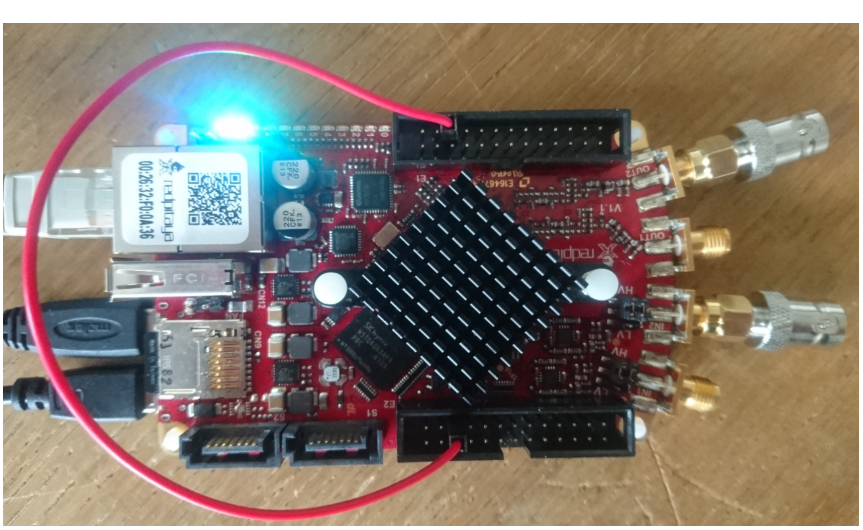
qu'on pensera à relacher en quittant le pilote par

```
int axi_gpio_it_drv_release(struct inode *inode, struct file *filp)
{struct axi_gpio_it_drv_dev *dev = filp->private_data;
 writel(0, dev->membase + GPIO_GIER);
 writel(0, dev->membase + GPIO_IER);
 filp->private_data = NULL;
 return 0;
}
```

10. les mutex permettront de bloquer la méthode `read()` jusqu'au déclenchement de l'interrution dont le handler se charge de débloquer le `mutex`. Ce mécanisme a l'avantage de sortir la tâche de la file des processus de l'ordonnanceur et ainsi libérer des ressources tant que l'interruption n'est pas déclenchée.
11. `/dev/jmf` est apparu lors du chargement du pilote par le chargement de l'overlay du devicetree
12. un device peut appeler une instance du pilote en lui passant des arguments. Plusieurs devices peuvent faire appel à plusieurs instances du même pilote en lui passant en argument des ressources différentes. Le devicetree prend la place du device en appelant le pilote auquel il passe les ressources occupées.
13. le programme suivant lit le pseudo-fichier `/dev/jmf` qui se débloque à chaque interruption puisque nous avons acquitté le mutex dans le pilote

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdint.h>
#include <sys/time.h>

int main(void)
{uint32_t c;
 struct timeval tv;
 int fd = open("/dev/jmf", O_RDWR); // cf name du noeud dans le .dts
 if (fd < 0) {printf("erreur d'ouverture\n");return 1;}
 while(1) {
   read(fd, &c, sizeof(uint32_t));
   gettimeofday(&tv, NULL);
   printf("hello %ld %ld\n", tv.tv_sec, tv.tv_usec);
 }
 close(fd);
 return 0;
}
```

14. La démonstration du bon fonctionnement du programme en connectant la sortie simulant le 1 PPS sur l'entrée interruption (Fig. ci-dessous, gauche) s'observe par la datation des transistion (Fig. ci-dessous, droite).



```
redpitaya> ./test_it
hello 2730 432246
hello 2730 940613
hello 2731 440657
hello 2731 940597
hello 2732 440641
hello 2732 940599
hello 2733 440633
hello 2733 940597
hello 2734 440646
hello 2734 940596
hello 2744 941627
```

```
redpitaya> ./gen_sig
1000 f8000000
734
dfff2000
^CCTRL-C
redpitaya>
```

Nous vérifions qu'il y a bien deux transitions d'état par seconde.

15. aucun mécanisme n'est disponible au niveau du matériel pour sélectionner le front qui déclenche une interruption. Nous devons donc lire l'état du GPIO juste après le déclenchement de l'interruption (transition d'état), et selon le niveau décider de débloquer ou non la lecture. Si le niveau est haut, la transition s'est faite du bas vers le haut et il s'agit d'une transition valide du 1 PPS, sinon (niveau bas lu) nous ne déclenchons pas le déblocage de la lecture.