

SYSTÈMES D'EXPLOITATION ET
ENVIRONNEMENTS D'EXÉCUTION
EMBARQUÉS

Rapport de laboratoire
Master HES-SO

Émilie GSPONER, Grégory EMERY

3 mai 2016
version 1.0

Table des matières

1	Introduction REPTAR	3
1.1	Mise en place de l'environnement, utilisation de git	3
1.2	Démarrage de Qemu	4
1.3	Tests avec U-boot	6
1.4	Tests avec Linux	8
1.5	Tests sur la plate-forme réelle	11
1.6	Accès aux périphériques REPTAR	14
2	Émulation de périphériques	16
2.1	Environnement Qemu et machine Reptar	16
2.2	Émulation de la FPGA Spartan6	17
2.3	Émulation des devices de type LED (output)	18
2.4	Émulation de type boutons (input)	20
2.5	Gestion des interruptions (IRQ) avec les boutons	21
2.6	Émulation de l'afficheur 7 segments	23
2.7	Mini-application utilisant les boutons et l'afficheur 7 segments	25
3	Drivers	29
3.1	Environnement Qemu et plate-forme Reptar	29
3.2	Driver de type caractère	30
3.3	Pilotage des LEDs	32
3.4	Pilotage des boutons	33

1 Introduction REPTAR

1.1 Mise en place de l'environnement, utilisation de git

a) **Donnée** : Il faut tout d'abord récupérer le dépôt étudiant pour les laboratoires SEEE à l'aide de la commande suivante (via une fenêtre de terminal) :

```
1 $ git clone firstname.lastname@eigit.heig-vd.ch:/home2/reds/seee/seee_student
```

Travail réalisé : Nous n'avons pas les droits d'accès pour le dépôt git, nous l'avons donc téléchargé, puis extrait depuis le lien : <https://drive.switch.ch/index.php/s/TbHxQZtm09IVdkb>. Le dossier seee_student a ensuite été placé dans : /home/redsuser/

b) **Donnée** : Lancez Eclipse et ouvrez le workspace seee_student. Vous devriez obtenir la liste des projets (à gauche). Chaque projet a un lien symbolique dans la racine du workspace.

Travail réalisé : En introduisant le path du dossier seee_student comme workspace au lancement d'Eclipse, nous obtenons la liste de projets suivante :

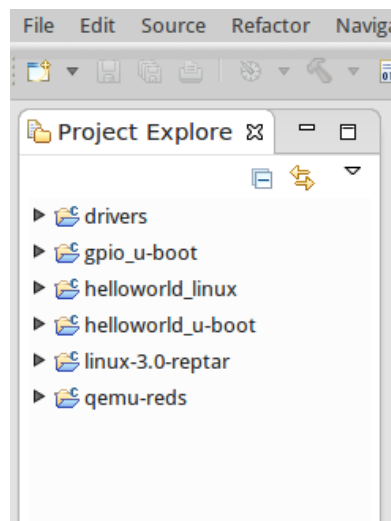


FIGURE 1 – Liste des projets

c) **Donnée** : Compilez maintenant l'émulateur Qemu. Dans une fenêtre de terminal, lancez la commande suivante à partir de votre répertoire seee_student :

```
1 $ make qemu
```

Travail réalisé : Vu que nous n'avons pas téléchargé le dossier de projets depuis git, il faut nettoyer le contenu du dossier avec clean ou distclean avant de compiler qemu. Le make qemu prend quelques instants.

```
1 $ cd ~/seee_student
2 $ make clean
3 $ make qemu
4 ...
5 make[1]: Leaving directory '/home/reduser/seee_student/qemu-reds'
6 $
```

1.2 Démarrage de Qemu

a) **Donnée :** Depuis Eclipse, lancez le debugger avec la configuration de debug « qemu-reds Debug ». Dans la fenêtre Console, vous pourrez entrer directement des commandes de U-boot (tapez help par exemple).

Travail réalisé :

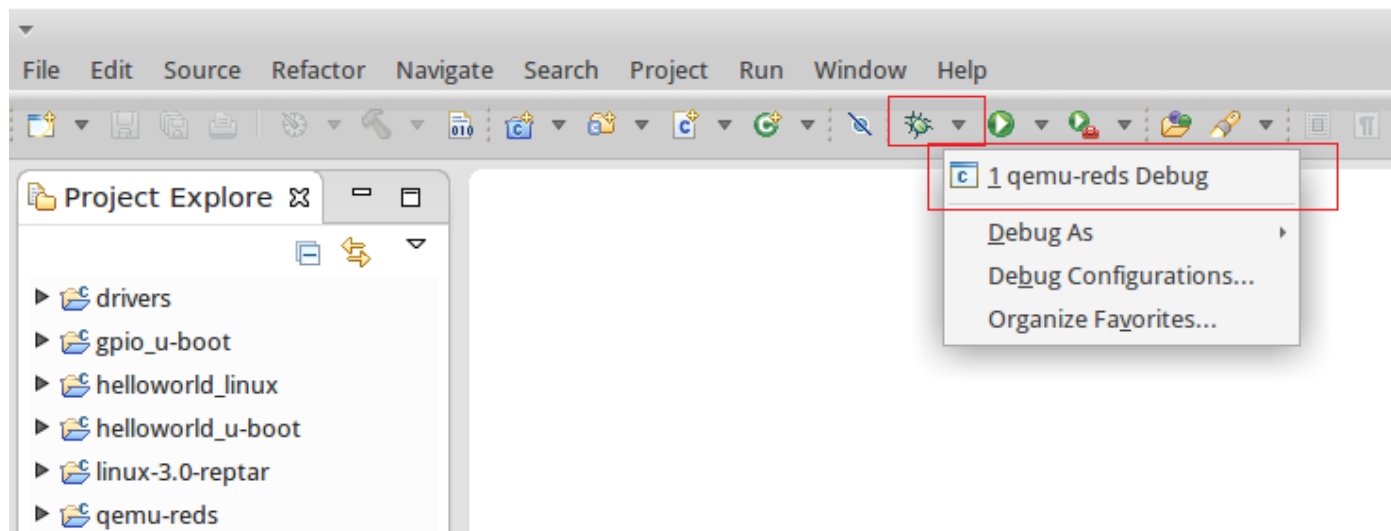


FIGURE 2 – Lancement d'Eclipse en mode Debug

Remarque : Après le lancement du Debug, il faut changer d'onglet en haut à droite en choisissant *Debug* pour avoir la console. Ce changement d'onglet ne se fait pas automatiquement.

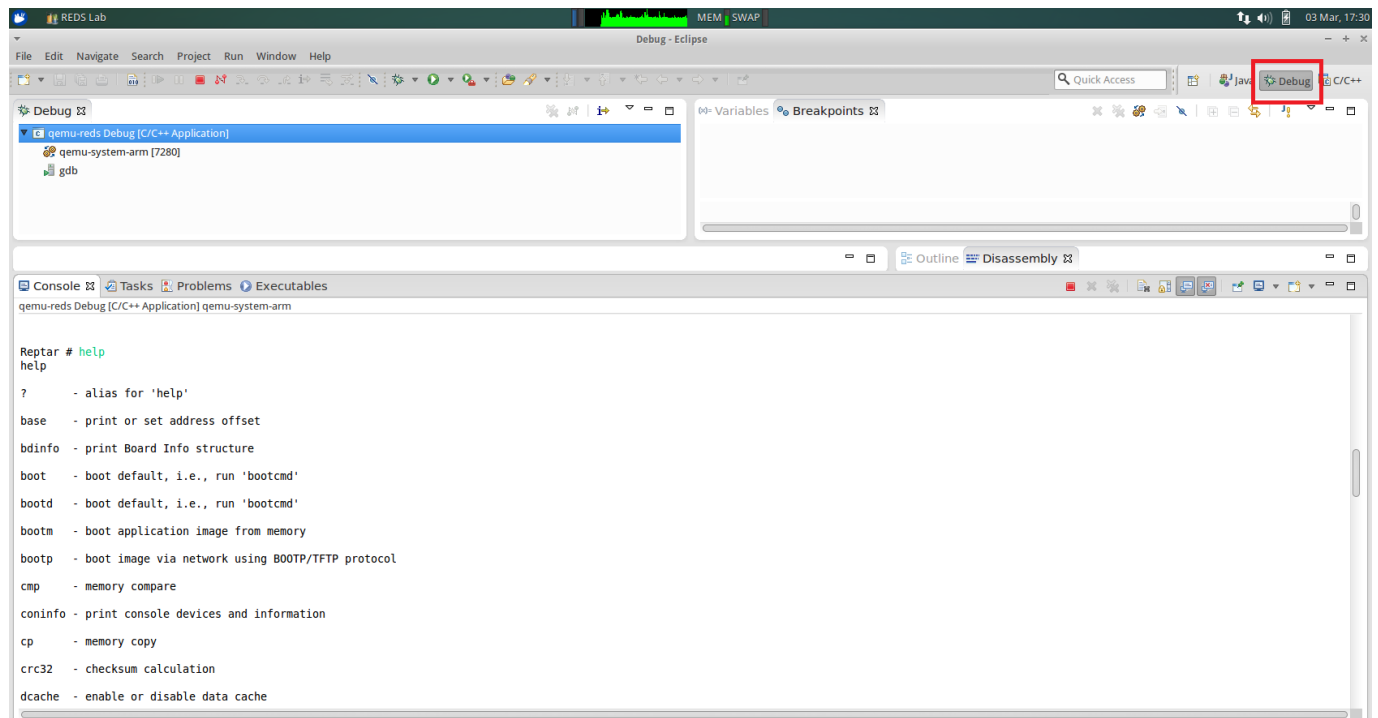


FIGURE 3 – Command help dans l'U-boot

b) **Donnée** : Interrompez l'exécution du programme en cliquant sur l'icône pause. Identifiez la ligne en cours d'exécution dans le code source.

Travail réalisé : En interrompant le programme avec le bouton *suspend*, on obtient la vue assembleur ci-dessous. L'environnement essaie d'ouvrir le fichier `ppoll.c`, on est donc en attente d'un événement. Le programme est interrompu après un `syscall`.

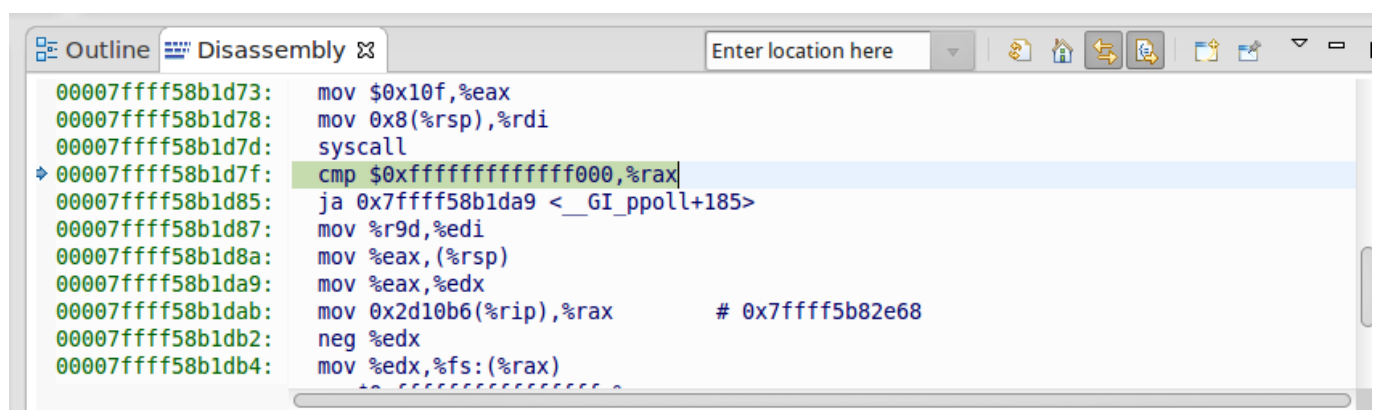


FIGURE 4 – Command help dans l'U-boot

c) **Donnée** : Stoppez l'exécution, et dans une fenêtre de commande, démarrez `qemu` à l'aide du script `stf` (en tapant `./stf`) dans le répertoire racine. Vous arrivez dans U-boot.

Travail réalisé : Cette partie n'a plus rien avoir avec Eclipse, on peut le fermer et lancer un terminal. Avec la commande *stf* tapée à la racine du répertoire *see_student*, on arrive au même point qu'en lançant le Debug dans Eclipse. On peut également essayer la commande *help*

```

1 $ cd ~/see_student
2 $ ./stf
3 WARNING: Image format was not specified for 'filesystem/flash' and probing
   guessed raw.
4 ...
5
6 Reptar # help
7 ?      - alias for 'help'
8 base   - print or set address offset
9 binfo  - print Board Info structure
10 boot   - boot default, i.e., run 'bootcmd'
11 bootd  - boot default, i.e., run 'bootcmd'
12 bootm  - boot application image from memory
13 bootp  - boot image via network usi
14 ...

```

1.3 Tests avec U-boot

a) Donnée : Dans U-boot, listez les variables d'environnement avec la commande *printenv*. Observez les variables prédéfinies « *tftp1*, *tftp2* et *goapp* ». Ces variables définissent des commandes U-boot qui peuvent être exécutées à l'aide de la commande *run* (par exemple *run tftp1*). La commande *go <addr>* permet de lancer l'exécution à l'adresse physique *<addr>*. Vous pouvez définir/modifier vos propres variables et les sauvegarder dans la flash émulée avec la commande *saveenv* (seulement avec le lancement via *stf*).

Travail Réalisé : Après être entré dans l'U-boot avec *stf*, nous avons pu lister les variables d'environnement suivantes :

```

1 $ cd ~/see_student/
2 $ ./stf
3 WARNING: Image format was not specified for 'filesystem/flash' and probing
   guessed raw.
4 Reptar # printenv
5 ...
6 goapp=go 0x81600000
7 ...
8 tftp1=tftp helloworld_u-boot/helloworld.bin
9 tftp2=tftp gpio_u-boot/gpio_u-boot.bin
10
11 Environment size: 930/4092 bytes
12 Reptar #

```

Les variables *tftp1* et *tftp2* sont des alias permettant de lancer des applications, la variable *goapp* est un alias permettant de lancer l'exécution de l'adresse physique 0x81600000. Elle définit l'adresse de début des applications. Voici un exemple d'utilisation de ces variables :

```
1 $ cd ~/seee_student/helloworld_u-boot/
2 $ make
3 ...
4 $ cd ../gpio_u-boot/
5 $ make
6 ...
7 $ cd ..
8 $ ./stf
9 WARNING: Image format was not specified for 'filesystem/flash' and probing
   guessed raw.
10 ...
11 Reptar # run tftp1
12 smc911x: detected LAN9118 controller
13 smc911x: phy initialized
14 smc911x: MAC e4:af:a1:40:01:fe
15 Using smc911x-0 device
16 TFTP from server 10.0.2.2; our IP address is 10.0.2.10
17 Filename 'helloworld_u-boot/helloworld.bin'.
18 Load address: 0x81600000
19 Loading: #
20 done
21 Bytes transferred = 776 (308 hex)
22
23 Reptar # run goapp
24 ## Starting application at 0x81600000 ...
25 Example expects ABI version 6
26 Actual U-Boot ABI version 6
27 Hello World
28 argc = 1
29 argv[0] = "0x81600000"
30 argv[1] = "<NULL>"
31 Hit any key to exit ...
32 ## Application terminated, rc = 0x0
33
34 Reptar # run tftp2
35 smc911x: detected LAN9118 controller
36 smc911x: phy initialized
37 smc911x: MAC e4:af:a1:40:01:fe
38 Using smc911x-0 device
39 TFTP from server 10.0.2.2; our IP address is 10.0.2.10
40 Filename 'gpio_u-boot/gpio_u-boot.bin'.
41 Load address: 0x81600000
42 Loading: #
43 done
44 Bytes transferred = 3080 (c08 hex)
45
46 Reptar # run goapp
47 ## Starting application at 0x81600000 ...
48 Start of the GPIO U-boot Standalone Application
49 Stop of the GPIO U-boot Standalone Application
50 ## Application terminated, rc = 0x0
51 Reptar #
```

La commande `run tftp<x>` charge une application à l'adresse 0x81600000, tandis que `run`

goapp va exécuter l'application depuis cette adresse comme le montre l'exemple ci-dessus.

b) Donnée : La production de l'exécutable helloworld_u-boot s'effectue en tapant la commande make dans le répertoire contenant les sources du programme. Ensuite, vous pouvez transférer le fichier (extension .bin) dans U-boot et exécuter le binaire (aidez-vous des variables d'environnement prédéfinies).

Travail réalisé : Ce point a été fait en même temps que le précédent. Il ne faut rien faire de particulier pour que le .bin soit accessible dans l'U-boot. Il suffit d'utiliser les variables d'environnement prédéfinies.

c) Donnée : Testez le debugger dans Eclipse avec le projet helloworld_u-boot. Mettez un breakpoint dans le code source au démarrage du programme, et lancez le debugger avec la configuration de debug « helloworld_u-boot Debug ».

Travail Réalisé : Il faut que qemu soit lancé dans un terminal externe avec *stf* pour que la manipulation fonctionne avec Eclipse.

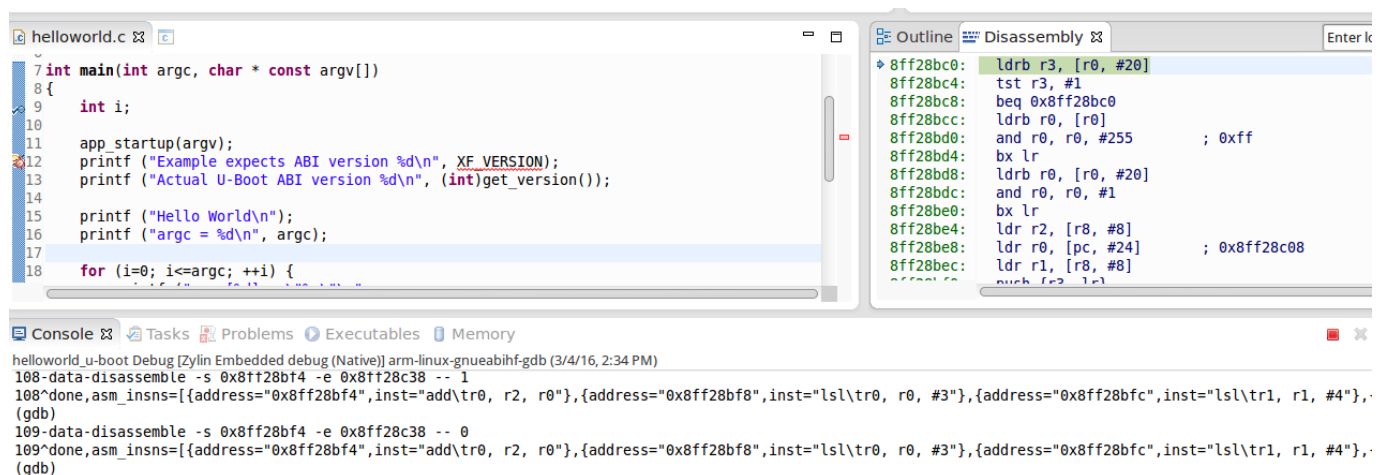


FIGURE 5 – Debug d'hello_world_u-boot

Remarque : Qemu contient un serveur GDB, ce qui permet à Eclipse (avec un plug-in) de communiquer avec lui et de debugger/lancer des applications à distance. On peut ainsi debugger des applications de l'environnement émulé depuis la machine hôte.

1.4 Tests avec Linux

a) Donnée : Lancez le script `./deploy` qui permettra de déployer le noyau Linux dans la sdcard virtuelle (ignorez l'erreur due à l'absence de certains fichiers).

Travail réalisé :

```
1 $ cd ~/seee_student
2 $ ./deploy
```



```

3 Deploying into reftar rootfs ...
4 Mounting filesystem/sd-card.img...
5 [sudo] password for redsuser:
6 SD card partitions mounted in 'boot_tmp' and 'filesystem_tmp' directories
7 cp: cannot stat 'drivers/sp6.ko': No such file or directory
8 cp: cannot stat 'drivers/usertest': No such file or directory
9 cp: cannot stat 'drivers/buttons_test': No such file or directory
10 Unmounting SD card image...
11 Synchronizing .img file
12 Unmounting 'boot_tmp' and 'filesystem_tmp' ...
13 Done !
14 $

```

b) Donnée : Poursuivez ensuite en cross-compilant l'application helloworld pour Linux (via make).

Travail réalisé :

```

1 $ cd ~/seee_student/helloworld_linux/
2 $ make
3 ...
4 $

```

c) Donnée : Copiez l'exécutable dans le rootfs

Travail réalisé :

```

1 $ cd ~/seee_student
2 $ ./mount-sd.sh
3 Mounting filesystem/sd-card.img...
4 SD card partitions mounted in 'boot_tmp' and 'filesystem_tmp' directories
5
6 $ sudo cp helloworld_linux/helloworld filesystem_tmp/root
7
8 $ ./umount-sd.sh
9 Unmounting SD card image...
10 Synchronizing .img file
11 Unmounting 'boot_tmp' and 'filesystem_tmp' ...
12 Done !
13 $

```

Remarque : Pour accéder au rootfs de la sdcard virtuelle, on va d'abord la monter son image sur la machine hôte. On peut ensuite copier l'exécutable de notre application à la racine du système de fichiers de la carte sd. Pour terminer, il faut démonter la carte Sd.

d) Donnée : Lancez le script stq suivi de la commande boot dans U-boot pour amorcer le démarrage de Linux

Travail réalisé : Avec la commande stq, une représentation graphique de la carte se lance en plus de la console U-boot.

```

1 $ cd ~/seee_student
2 $ ./stq
3 libGL error: failed to authenticate magic 1
4 libGL error: failed to load driver: vboxvideo
5 Running QEMU
6 ...
7 Warning: smc911x-0 MAC addresses don't match:
8 Address in SROM is      52:54:00:12:34:56
9 Address in environment is e4:af:a1:40:01:fe
10 Reptar # boot
11 reading uImage
12 ...
13 *** Welcome on REPTAR (HEIG-VD/REDS): use root/root to log in ***
14 reptar login: root
15 Password:
16 #

```

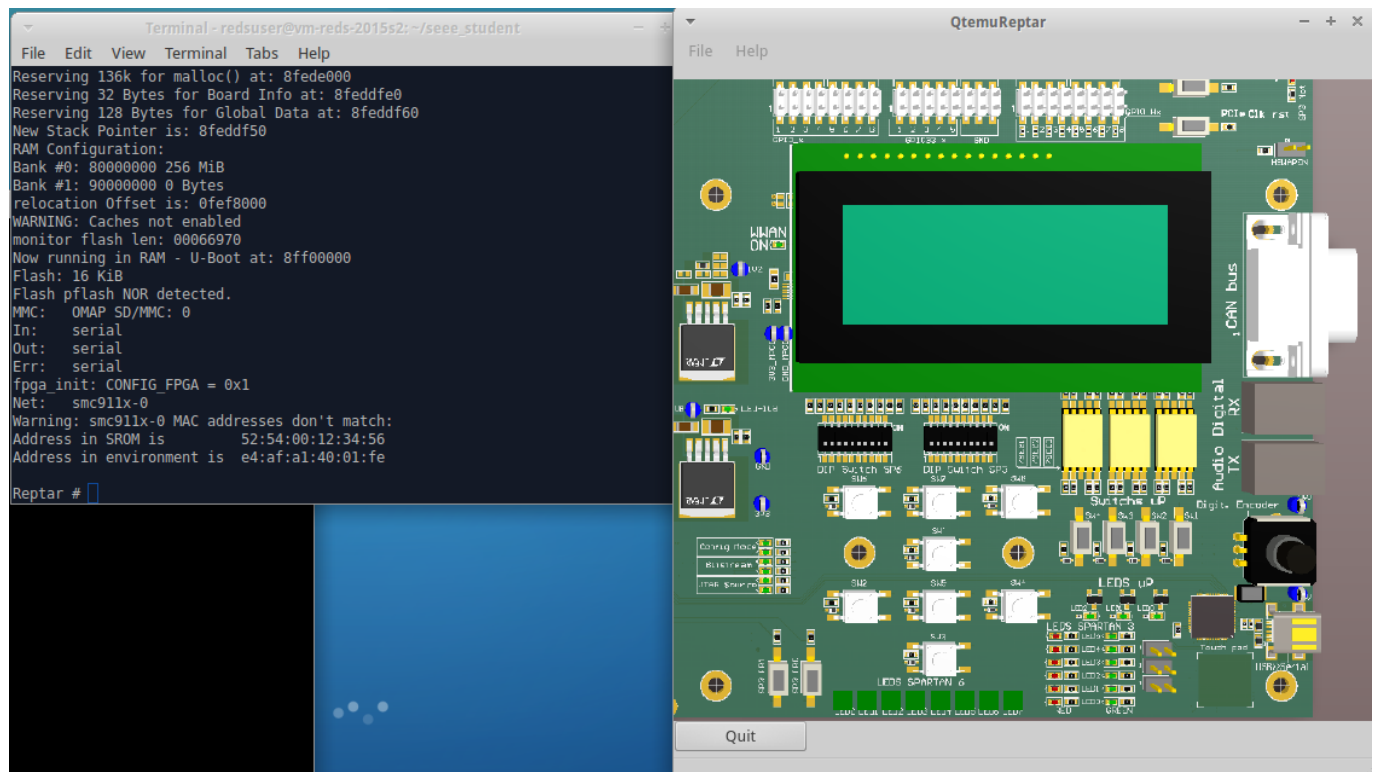


FIGURE 6 – Environnement émulé

e) **Donnée** : Lancez votre application

Travail réalisé : Avec la commande `ls`, on voit que l'application que nous avons chargée dans le rootfs est bien présente.

```

1 # ls
2 Settings          fs                  helloworld         rootfs_domU.img

```

```

3 # ./helloworld
4 Hello world within Linux
5 argv[0] = ./helloworld
6 #

```

f) Donnée : Dans Linux, tapez la commande suivante :

```

1 $ /usr/share/qt/examples/effects/lighting/lighting -qws &

```

Travail réalisé : Cette commande permet de lancer une application pré installée de l'émulateur.

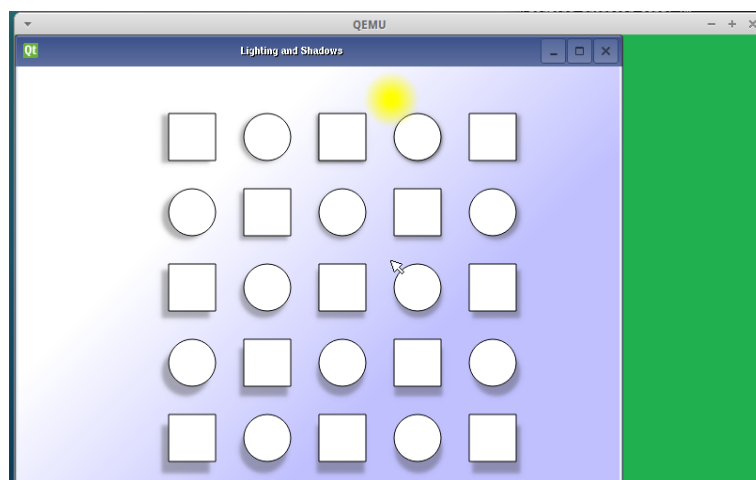


FIGURE 7 – Lancement d'une application

1.5 Tests sur la plate-forme réelle

a) Donnée : Déployez l'application helloworld dans U-boot sur la plate-forme REPTAR avec l'interface réseau. Le transfert peut s'effectuer avec la commande tftp. Il est nécessaire d'exécuter la commande suivante pour mettre à jour les adresses IP et MAC de la plate-forme REPTAR :

```

1 # run setmac setip

```

Travail réalisé : Avec la commande tftp il faut donner comme paramètre le *.bin* de l'application ainsi que l'adresse physique où charger le programme. Cette adresse est 0x81600000 comme dans les exercices précédents.

Pour charger notre application, il faut placer le *.bin* dans le répertoire tftpboot de la machine hôte. On va utiliser la connexion série (USB) pour exécuter la commande tftp et transférer l'exécutable sur la plateforme.

```

1 $ cd ~/seee_student/helloworld_u-boot
2 $ make

```

```
3 $ cp helloworld.bin /home/redsuser/tftpboot
4 $ sudo picocom -b 115200 /dev/ttyUSB0
5 [sudo] password for redsuser:
6 picocom v1.7
7 ...
8 Terminal ready
9
10 Reptar # run setmac setip
11 Reptar # tftp 0x81600000 helloworld.bin
12 smc911x: detected LAN9220 controller
13 smc911x: phy initialized
14 smc911x: MAC e4:af:a1:40:01:fe
15 Using smc911x-0 device
16 TFTP from server 192.168.1.1; our IP address is 192.168.1.254
17 Filename 'helloworld.bin'.
18 Load address: 0x81600000
19 Loading: T #
20 done
21 Bytes transferred = 776 (308 hex)
22
23 Reptar # go 0x81600000
24 ## Starting application at 0x81600000 ...
25 Example expects ABI version 6
26 Actual U-Boot ABI version 6
27 Hello World
28 argc = 1
29 argv[0] = "0x81600000"
30 argv[1] = "<NULL>"
31 Hit any key to exit ...
```

Remarque : La commande tftp ne fonctionnera pas tant que la configuration réseau n'est pas correcte. Il faut impérativement que l'adresse IP de la connexion par pont de la VM soit 192.168.1.1.

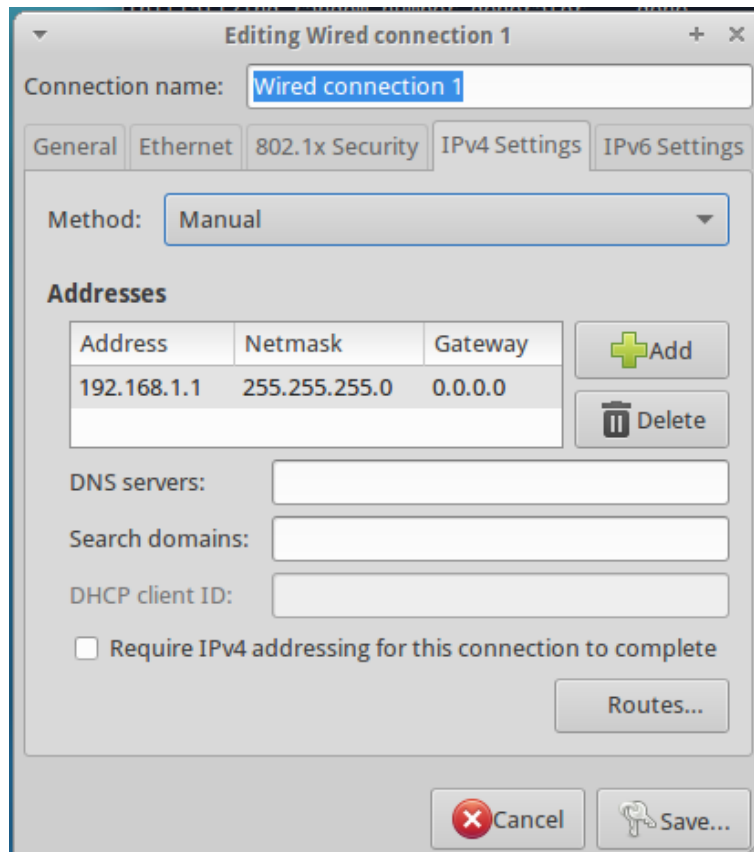


FIGURE 8 – Configuration réseau

b) Donnée : Déployez l'application helloworld dans Linux à l'aide du réseau et de la commande scp.

Travail réalisé : Nous avons découvert que l'adresse IP de la carte n'était pas celle attendue, nous avons donc dû adapter scp pour l'adresse IP 192.168.1.254.

```

1 Reptar # boot
2 reading uImage
3 ...
4 *** Welcome on REPTAR (HEIG-VD/REDS): use root/root to log in ***
5 reptar login: root
6 Password:
7 # ifconfig
8 eth0      Link encap:Ethernet  HWaddr E4:AF:A1:40:01:FE
9 inet addr:192.168.1.254  Bcast:192.168.1.255  Mask:255.255.255.0
10 ...

```

La commande scp permet de transférer le helloworld_linux sur la carte reptar par l'interface réseau depuis la machine hôte.

```

1 $ scp helloworld root@192.168.1.254:helloworld
2
3 The authenticity of host '192.168.1.254 (192.168.1.254)' can't be established.

```

```

4 RSA key fingerprint is fb:59:a3:73:97:9d:b7:b9:8a:40:e8:bc:19:ab:ab:70.
5 Are you sure you want to continue connecting (yes/no)? yes
6 Warning: Permanently added '192.168.1.254' (RSA) to the list of known hosts.
7 root@192.168.1.254's password:
8 helloworld                                     100% 6877      6.7KB/s   00:00

```

L'application helloworld est maintenant chargée sur la cible, il ne reste plus qu'à l'exécuter.

```

1 # ls
2 bitstreams  helloworld      tests
3 # ./helloworld
4 Hello world within Linux
5 argv[0] = ./helloworld
6 #

```

1.6 Accès aux périphériques REPTAR

a) Donnée : Sur la base de l'exemple gpio_u-boot., vous devez développer une application permettant d'interagir avec les LEDs et les switchs présents sur la carte CPU de la plate-forme REPTAR. Le but de l'application est d'allumer une LED lorsqu'on appuie sur un switch.

1. La LED 0 doit s'allumer lorsqu'on appuie sur le SWITCH 0.
2. La LED 1 s'allume si l'on appuie sur le SWITCH 1.
3. Et ainsi de suite pour les LEDs et switchs 0..3 de la carte CPU.

Le switch numéro 4 sert à quitter l'application. Aidez-vous des fichiers d'en-tête (`#include`) déjà présents dans le chablon fourni.

L'application gpio_u-boot est à déployer dans U-boot via la commande tftp.

Emplacement du code : /gpio_u-boot.c

Ce code est très basique, mais implémente correctement les points exigés par la donnée. Les commandes suivantes ont permis de lancer l'application sur la cible réelle dans l'U-boot. Une pression sur le switch numéro 4 permet de terminer l'application.

```

1 $ cd ~/seee_student/gpio_u-boot
2 $ make
3 ...
4 $ cp gpio_u-boot.bin /home/redsuser/tftpboot
5 $ sudo picocom -b 115200 /dev/ttyUSB0
6 [sudo] password for redsuser:
7 picocom v1.7
8 ...
9 Terminal ready
10
11 Reptar # tftp 0x81600000 gpio_u-boot.bin
12 smc911x: detected LAN9220 controller
13 smc911x: phy initialized
14 smc911x: MAC e4:af:a1:40:01:fe

```

```
15 Using smc911x-0 device
16 TFTP from server 192.168.1.1; our IP address is 192.168.1.254
17 Filename 'gpio_u-boot.bin'.
18 Load address: 0x81600000
19 Loading: T #
20 done
21 Bytes transferred = 776 (308 hex)
22
23 Reptar # go 0x81600000
24 ...
25 Stop of the GPIO U-boot Standalone Application
26 Reptar #
```

2 Émulation de périphériques

2.1 Environnement Qemu et machine Reptar

Cette étape vous permet de vous familiariser avec l'environnement que nous utiliserons pour l'émulation de périphériques. Dans cette étape, il est nécessaire de travailler avec l'application graphique Qtemu, qui constitue le frontend graphique de Qemu. L'application est développée en C++ et utilise la librairie Qt.

a) Donnée : A partir du répertoire `seee_student`, lancez le frontend graphique avec le script `stq`

Travail réalisé :

```
1 $ cd ~/seee_student/
2 $ ./stq
3 ...
4 Reptar #
```

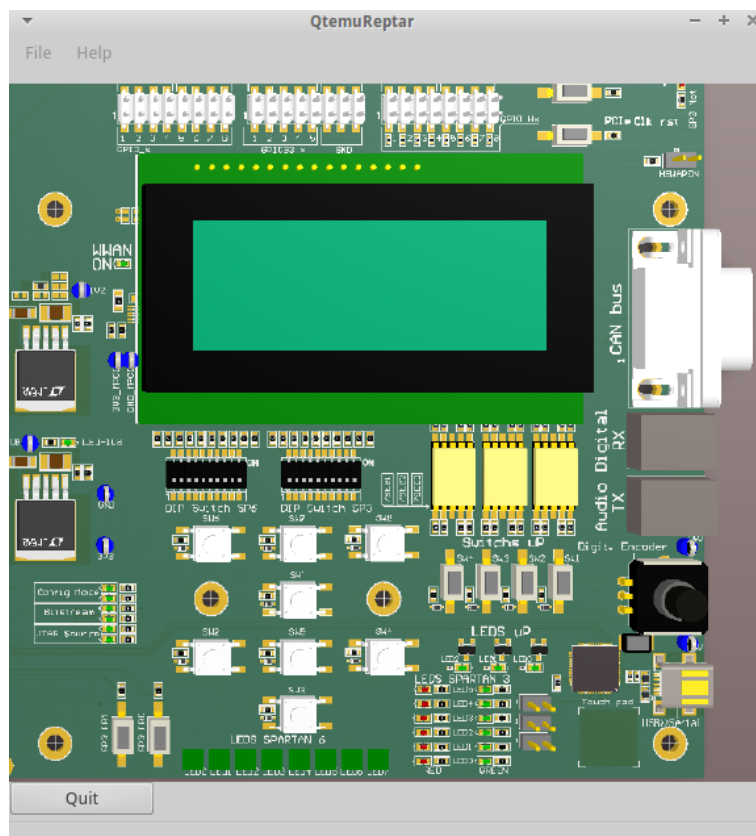


FIGURE 9 – Frontend graphique de Qemu

b) Donnée : Les fichiers sources de Qemu se trouvent dans le répertoire `qemu-reds`. Examinez les fichiers suivants :

1. hw/arm/reptar/reptar.c Emulation plate-forme REPTAR
2. hw/reptar_sp6.c Emulation de la FPGA
3. hw/reptar_sp6_clcd.c Emulation gestion du LCD4x20
4. hw/reptar_sp6_buttons.c Emulation gestion des boutons
5. hw/reptar_sp6_emul.c Gateway entre Qemu et Qtemu

Vous trouverez également toute la documentation nécessaire sur la plate-forme Reptar dans le répertoire doc.

Remarque : Ces différents fichiers implémentent ce qui ressemble à des modules noyaux.

c) Donnée : La compilation de Qemu pourra s'effectuer dans le répertoire qemu-reds directement, avec la commande `make` (utilisez `make -j4` ou `-j8` pour aller plus vite!).

Travail réalisé : Par la suite, seule la commande *make* sera nécessaire pour recompiler l'émulateur.

En lançant `./qtemu` et Eclipse, on pourra debugger l'émulateur Qemu.

```

1 $ cd ~/seee_student/qemu-reds /
2 $ ./configure --target-list=arm-softmmu --enable-debug --disable-attr --disable
   -docs
3 ...
4 $ make -j8
5 ...
6 $

```

2.2 Émulation de la FPGA Spartan6

Dans cette étape, il s'agit de mettre en place la structure nécessaire à l'émulation de la FPGA intégrée à la plate-forme. La FPGA implémente des registres associés aux périphériques externes. Dans cette étape, il s'agit de s'assurer que l'accès aux adresses I/O en lecture et écriture fonctionne.

a) Donnée : Complétez l'émulation de la FPGA afin de tester l'écriture et la lecture à l'une ou l'autre adresse dédiée à la FPGA (affichez simplement un message).

Emplacement du code :

`/emulationSpartan6_part2/reptar_sp6.c`
`/emulationSpartan6_part2/reptar.c`

Travail réalisé : Nous avons modifié les fichiers *reptar_sp6.c* et *reptar.c*

Le point crucial de cette partie du labo a été de trouver l'adresse de base de la FPGA qui est **0x18000000**. Nous avons en effet besoin de cette adresse pour lier/créer le *reptar_sp6* dans la partie reptar.

```

1 sysbus_create_simple("reptar_sp6", 0x18000000, NULL);

```

Pour le reste de l'implémentation, nous nous sommes basés sur le diagramme de séquence du support de cours et avons pris le document *versatilepb.c* comme exemple pour le contenu des méthodes callback.

Le bon fonctionnement du code a été "testé" premièrement en réussissant la compilation sans erreurs, puis le lancement sans crash. Nous avons également ajouté des messages affichés dans la console dans les différentes méthodes callback pour suivre l'initialisation. L'exécution a été faite de la manière suivante :

```
1 $ cd ~/seee_student/qemu-reds /
2 $ make
3 ...
4 $ cd ..
5 $ ./stq
6 ...
7 sp6 init
8 reptar-sp6-emul: sp6_emul_init
9 sp6 initfn
10 ...
11 Reptar #
```

b) Donnée : Testez les accès en lecture-écriture avec U-Boot.

Travail réalisé : Pour l'instant, les callback de lecture/écriture que nous avons implémentés contiennent uniquement des messages d'indication qui sont affichés dans la console. À l'aide des commandes suivantes, nous avons pu tester leur bon fonctionnement. Les commandes suivantes tentent de lire, puis écrire à l'adresse de base de la FPGA.

```
1 Reptar # md.l 0x18000000 1
2 18000000:sp6 read
3 00000000 ....
4 Reptar # mw.l 0x18000000 1
5 sp6 write
6 Reptar #
```

2.3 Émulation des devices de type LED (output)

Donnée : La FPGA est connectée à des LEDs qui sont visibles sur l'interface graphique. Cette étape consiste à implémenter le code d'émulation précédent afin de gérer l'accès aux LEDs reliées à la FPGA. Les interactions entre la FPGA et l'interface graphique doivent être gérées proprement.

Emplacement du code : */emulationSpartan6_part3/reptar_sp6.c*

Travail réalisé : Pour cette partie, il a fallu rechercher l'offset du registre des LEDs qui est *0x003A*. Il faut donc ajouter cet offset à l'adresse de base de la FPGA.

Nous avons défini une variable qui garde la valeur écrite dans le registre des LEDs pour permettre la relecture de la valeur.

Pour la lecture et l'écriture, on teste si l'offset correspond et si l'on lit/écrit des données de la bonne taille, soit 16bits. Des messages ont été implémentés pour indiquer si la lecture/écriture est faite correctement.

La valeur lue est simplement affichée dans la console. Pour l'écriture, la valeur écrite est transmise à l'aide d'un message JSON conformément aux directives du *Guide d'utilisation de l'infrastructure*.

Nous avons testé le bon fonctionnement du code avec le test 10 de itbok¹. Le test montre que l'on arrive à lire et écrire correctement sur les leds.

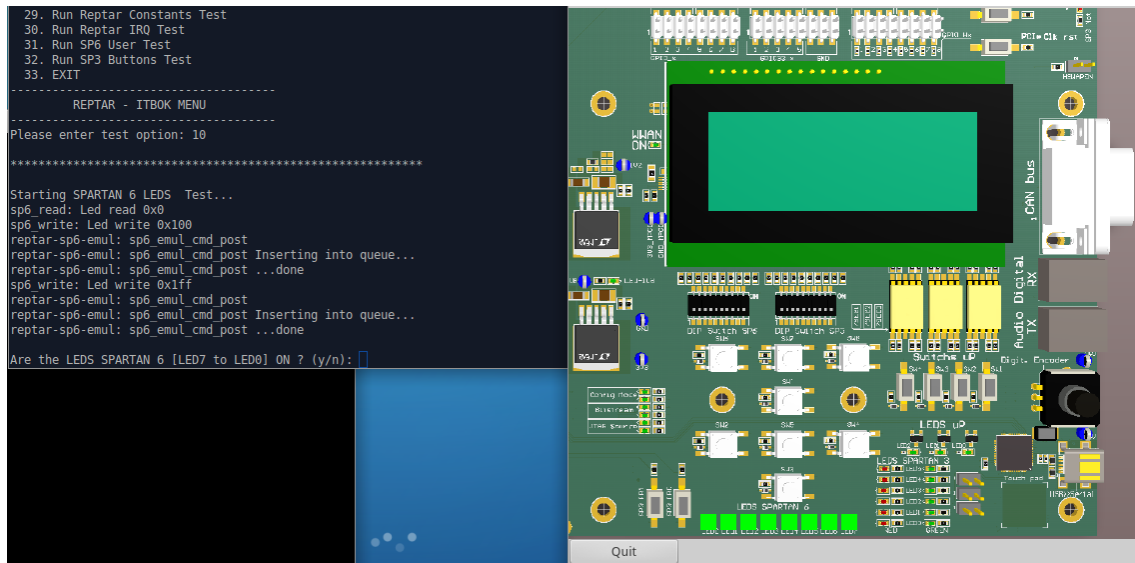


FIGURE 10 – Test d'allumage des LEDs

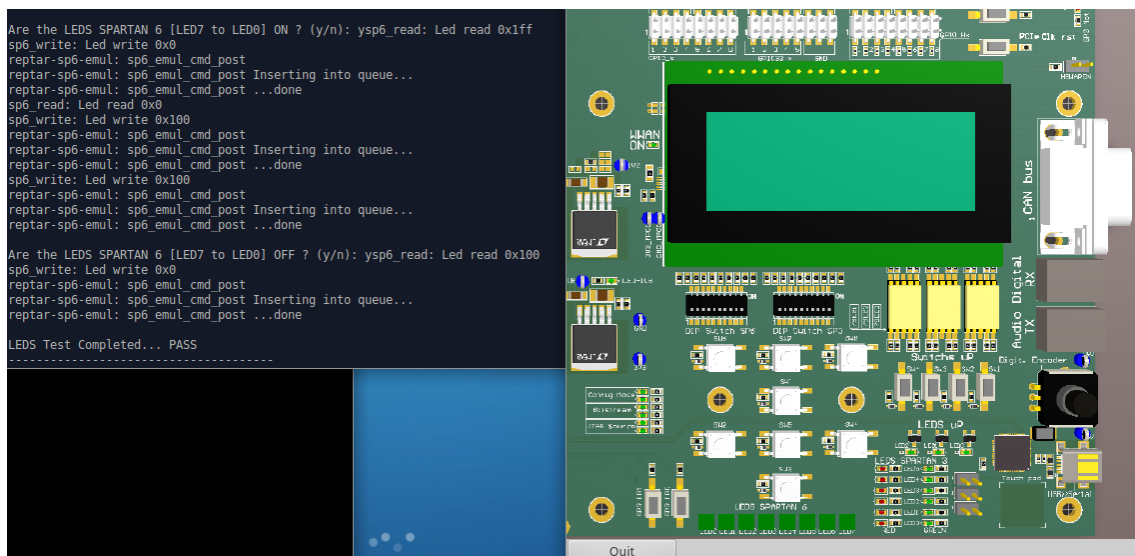


FIGURE 11 – Test d'extinction des LEDs

1. itbok, signifie Is The Board OK

2.4 Émulation de type boutons (input)

La FPGA est connectée à une série de boutons (switches) sur la plate-forme Reptar. Cette étape consiste à mettre en place la structure nécessaire à la gestion de ces boutons.

a) Donnée : Adaptez les fichiers nécessaires afin que l'émulation de votre périphérique (FPGA) puisse détecter la pression d'une touche, sans vous préoccuper pour le moment des interruptions.

Emplacement du code :

```
/emulationSpartan6_part4/reptar_sp6.c
/emulationSpartan6_part4/reptar_sp6_buttons.c
```

Travail réalisé : Le code de cette partie est inspiré du *Guide d'utilisation de l'infrastructure*. L'offset pour lire la valeur des boutons est *0x0012*. Lorsqu'un bouton est pressé, le handler du fichier *sp6_button* est appelé et la valeur du registre est mémorisée. La valeur des boutons peut également être récupérée lors d'une lecture du registre correspondant.

b) Donnée : Le projet *sp6_buttons_u-boot* contient une application permettant de tester vos boutons (en mode polling). Compilez l'application et effectuez quelques tests.

Travail réalisé : L'application a été compilée avec *make*. Il faut ensuite lancer l'émulateur avec la commande *stq*. Une fois dans l'U-boot, on peut lancer l'application testant les boutons. Elle est enregistrée dans les variables d'environnements sous le nom *tftp3*. Les lignes ci-dessous démontrent le bon fonctionnement des boutons. Le bouton *exit* arrête l'application.

```
1 $ cd ~/seee_student
2 $ ./stq
3 ...
4 Reptar # run tftp3
5 smc911x: detected LAN9118 controller
6 smc911x: phy initialized
7 smc911x: MAC e4:af:a1:40:01:fe
8 Using smc911x-0 device
9 TFTP from server 10.0.2.2; our IP address is 10.0.2.10
10 Filename 'sp6_buttons_u-boot/sp6_buttons.bin'.
11 Load address: 0x81600000
12 Loading: #####
13 done
14 Bytes transferred = 34512 (86d0 hex)
15
16 Reptar # run goapp
17 ## Starting application at 0x81600000 ...
18 Start of the SP6 buttons standalone test application.
19 Button ONE pressed
20 ...
21 Button ONE pressed
22 Button ONE pressed
23 Button ONE pressed
24 reptar-sp6-emul: sp6_emul_event_handle: read 29
25 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
26 Button status : 0x0
```

```

27
28 Button LEFT pressed
29 Button LEFT pressed
30 Button LEFT pressed
31 reptar-sp6-emul: sp6_emul_event_handle: read 29
32 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
33 Button status : 0x0
34
35 reptar-sp6-emul: sp6_emul_event_handle: read 30
36 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
37 Button status : 0x10
38 Button EXIT pressed
39 SP6 buttons standalone test application exit.
40 ## Application terminated , rc = 0x0
41 Reptar # reptar-sp6-emul: sp6_emul_event_handle: read 29
42 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
43 Button status : 0x0

```

2.5 Gestion des interruptions (IRQ) avec les boutons

Complétez votre émulateur avec le code nécessaire à la gestion d'une interruption à niveau émise par la FPGA lorsqu'un bouton est pressé. L'interruption est censée être acquittée par le driver. Il faut donc gérer l'état interne associé à cette interruption.

a) Donnée : Commencez par adapter le code d'initialisation de la plate-forme (*reptar.c*) afin d'instancier une interruption en provenance de la FPGA ; l'interruption sera de type niveau.

Emplacement du code :

```

/emulationSpartan6_part5/reptar_sp6.c
/emulationSpartan6_part5/reptar.c
/emulationSpartan6_part5/reptar_sp6_buttons.c

```

Travail réalisé : La première étape a consisté à assigner le *reptar_sp6* sur la pin GPIO 10 dans le fichier *reptar.c*.

Il a ensuite fallu faire en sorte de générer une interruption de type niveau lors d'une pression sur un bouton. Cela a été fait dans le fichier *reptar_sp6_buttons.c*. Notre code génère l'interruption uniquement si les IRQ ont été préalablement autorisées en configurant les registres de contrôle. Nous avons choisi de ne pas générer d'interruptions lors de la relâche du bouton (0x0).

Finalement, le fichier *reptar_sp6.c* a été adapté pour permettre la lecture et l'écriture du registre d'IRQ des boutons. Lorsque le registre est lu, on va lire la valeur des bouton et l'ajouter dans le registre de status de l'IRQ, puis retourner le tout. Pour l'écriture, un masquage est fait afin de savoir si l'on veut activer et/ou quittancer les interruptions et dans ce cas repasser la GPIO10 à l'état bas.

b) Donnée : Testez que l'interruption fonctionne en configurant le contrôleur GPIO et en

interrogeant le registre d'état, dans U-Boot. Les registres du microcontrôleur à utiliser sont les suivants : GPIO_RISINGDETECT, GPIO_IRQENABLE1 et GPIO_IRQSTATUS1. De plus, l'interruption doit aussi être activée au niveau de la FPGA (cf documentation).

Travail réalisé : Notre code a dans un premier temps été testé à l'aide d'*itbok* avec le test numéro 30. Cela a permis de valider le bon fonctionnement des interruptions avec tous les boutons. On peut voir que le message *IRQ RAISE* ne s'affiche plus si l'on désactive les interruptions.

```

1  Press on SW7...
2  reptar-sp6-emul: sp6_emul_event_handle: read 29
3  reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
4  Button status : 0x0
5  reptar-sp6-emul: sp6_emul_event_handle: read 30
6  reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
7  Button status : 0x40
8  IRQ RAISE
9  sp6_read: Button irq status read 0x8d (button value 0x7)
10 sp6_write: Button irq status write 0x81
11 Enable IRQ
12 Clear IRQ
13 IRQ detected:
14 - button: 7 ..... Test PASSED
15 Press on SW8...
16 reptar-sp6-emul: sp6_emul_event_handle: read 29
17 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
18 Button status : 0x0
19 reptar-sp6-emul: sp6_emul_event_handle: read 31
20 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
21 Button status : 0x80
22 IRQ RAISE
23 sp6_read: Button irq status read 0x8f (button value 0x8)
24 sp6_write: Button irq status write 0x81
25 Enable IRQ
26 Clear IRQ
27 IRQ detected:
28 - button: 8 ..... Test PASSED
29 sp6_write: Button irq status write 0x1
30 Disable IRQ
31 Clear IRQ
32
33 IRQ test complete. Press Enter to exit
34 reptar-sp6-emul: sp6_emul_event_handle: read 29
35 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
36 Button status : 0x0
37 reptar-sp6-emul: sp6_emul_event_handle: read 31
38 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
39 Button status : 0x80
40 reptar-sp6-emul: sp6_emul_event_handle: read 29
41 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
42 Button status : 0x0

```

Notre code a ensuite été testé avec les registres GPIO du microcontrôleur (*GPIO_RISINGDETECT*,

GPIO_IRQENABLE1, *GPIO_IRQSTATUS1*). Il faut utiliser pour ce faire le banque GPIO1, ce qui donne les adresses de registre correspondantes : 0x48310048, 0x4831001C et 0x48310018. Comme les boutons sont sur la GPIO 10, il faut aller autoriser l'interruption en activant le bit 10 des registres *GPIO_RISINGDETECT* et *GPIO_IRQENABLE1*. Il ne faut pas oublier d'aller ensuite autoriser les interruptions sur la FPGA. La capture ci-dessous présente cette configuration. Après chaque interruption, il faut aller quittancer le bit 10 du *GPIO_IRQSTATUS1* et également intervenir au niveau de la FPGA.

```

1 #Enable interrupts in GPIO
2 Reptar # mw.l 0x4831001C 0x00000400
3 Reptar # mw.l 0x48310048 0x00000400
4
5 #Enable interrupts in FPGA
6 Reptar # mw.w 0x18000018 0x0081
7 sp6_write: Button irq status write 0x81
8 Enable IRQ
9 Clear IRQ
10
11 #The IRQ are now available , the status register doesn't have detect interrupt
12   yet
13 md.l 0x48310018 1
14 48310018: 00000000      ....
15
16 #Generate an interrupt
17 Reptar # reptar-sp6-emul: sp6_emul_event_handle: read 30
18 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
19 Button status : 0x40
20 IRQ RAISE
21 reptar-sp6-emul: sp6_emul_event_handle: read 29
22 reptar-sp6-emul: sp6_emul_event_handle: cJSON_Parse done
23 Button status : 0x0
24
25 #Check status register , interrupt has been detected
26 md.l 0x48310018 1
27 48310018: 00000400      ....
28
29 #Need to ack the interrupt
30 Reptar # mw.l 0x48310018 0x00000400
31 Reptar # mw.w 0x18000018 0x0081
32 sp6_write: Button irq status write 0x81
33 Enable IRQ
34 Clear IRQ
35
36 #... Repeat operations

```

2.6 Émulation de l'afficheur 7 segments

La FPGA est connectée à un afficheur 7 segments, visible sur l'émulateur. Cette étape consiste à mettre en place la gestion de cet afficheur 7 segments.

a) Donnée : Adaptez les fichiers nécessaires afin que l'émulation de votre périphérique

(FPGA) puisse gérer les trois digits de l’afficheur 7 segments.

Emplacement du code : */emulationSpartan6_part6/reptar_sp6.c*

Travail réalisé : Pour cette partie, nous avons simplement ajouté le code pour écrire et lire dans le registre de chacun des trois digits de l’affichage 7 segments. La valeur de chaque affichage est stocké dans une variable. Pour l’écriture, il faut spécifier dans le message Json le nom du périphérique, le numéro du digit ainsi que la valeur à afficher.

b) Donnée : Le dossier 7seg_u-boot contient une application permettant de tester l’afficheur 7 segments : les chiffres de 0 à 9 doivent défiler progressivement : 012, puis 123, 234, 456, 567, ..., 901, 012, etc. Compilez l’application et effectuez quelques tests.

Travail réalisé : Une fois l’application compilée, il a fallu la lancer dans l’U-boot. Comme elle n’est pas définie dans les variables d’environnement, il faut utiliser la commande complète. Une fois lancée, l’application va incrémenter la valeur des digits. L’image ci-dessous démontre le bon fonctionnement.

```
1 $ cd ~/seee_student
2 $ ./stq
3 ...
4 Reptar # tftp 7seg_u-boot/7seg_u-boot.bin
5 smc911x: detected LAN9118 controller
6 smc911x: phy initialized
7 smc911x: MAC e4:af:a1:40:01:fe
8 Using smc911x-0 device
9 TFTP from server 10.0.2.2; our IP address is 10.0.2.10
10 Filename '7seg_u-boot/7seg_u-boot.bin'.
11 Load address: 0x81600000
12 Loading: #####
13 done
14 Bytes transferred = 34932 (8874 hex)
15 Reptar # run goapp
```

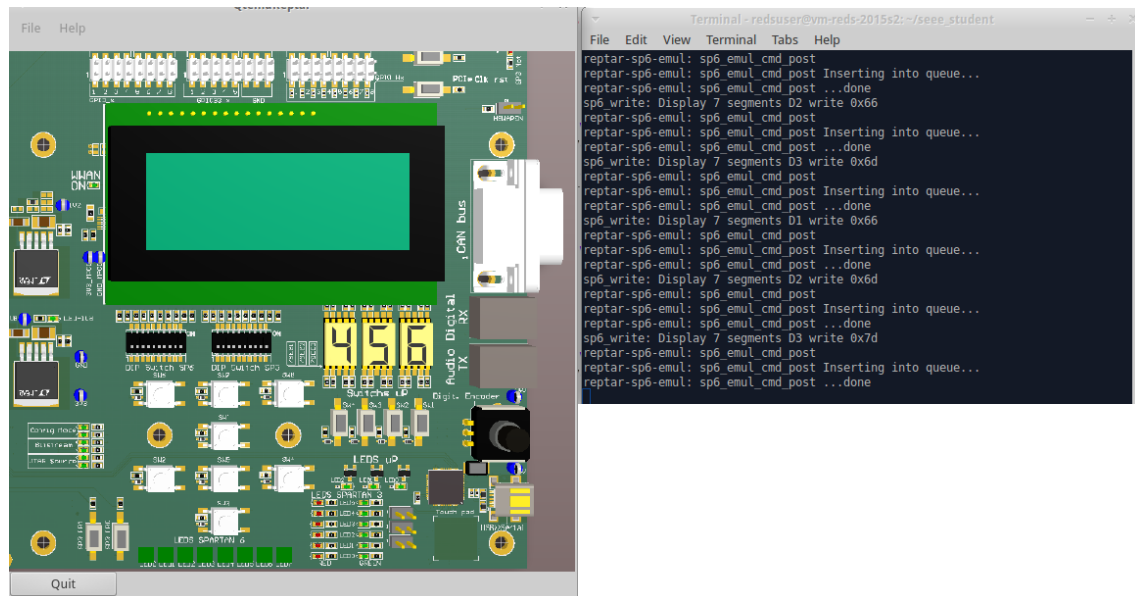



FIGURE 12 – Frontend graphique de Qemu avec affichage 7 segments

2.7 Mini-application utilisant les boutons et l’afficheur 7 segments

a) **Donnée :** Le dossier `miniapp_u-boot` contient un chablon. Complétez-le afin de créer une application qui utilise les boutons SW2, SW5, SW4 et SW3, ainsi que l’afficheur 7 segments.

1. Lors d’un appui sur SW2, SW5 ou SW4, le digit respectivement à gauche, au centre ou au milieu est incrémenté de 1, modulo 10. Si un digit atteint 9, il reviendra à 0.
2. La valeur initiale de chaque digit, au démarrage de l’application, est 0 (on affichera 000).
3. Un appui sur SW3 quitte l’application.
4. Vous devrez gérer l’anti-rebond : le digit ne devra être incrémenté que si le bouton est pressé puis relâché (comme un appui sur une touche de sonnette par exemple).

Emplacement du code : `/miniapp_test_emulation/miniapp_u-boot.c`

Travail réalisé : L’application a été codée avec une boucle `while` exécutant de manière répétitive les étapes suivantes :

- On lit le registre d’état des boutons et on stocke sa valeur dans une première variable.
- On attend un petit moment.
- On relit le registre d’état des boutons et on stocke cette nouvelle valeur dans une seconde variable.
- Ensuite, on teste les deux variables avec des masques pour savoir si :
 - Dans la première variable, le masquage retourne une valeur supérieure à 0 et indique que le bouton testé est enfoncé.
 - Dans la seconde variable, le masquage retourne une valeur égale à 0 et indique que le bouton a été relâché.

Ces tests sont effectués pour les trois boutons d'incrémentations des 7 segments et pour le bouton d'arrêt de l'application.

- Si un des tests passe :
 - Si c'est un des boutons d'incrémentations, on appelle la fonction *incr_7seg* avec en paramètre l'index du 7 segments à incrémenter.
 - Si c'est le bouton d'arrêt, on quitte la boucle avec un *break*.

Nous avons remarqué une chose lors de la compilation. Il s'avère que le compilateur prend comme point d'entrée la première instruction du programme, avec le Makefile fourni. Lors des premiers essais, la fonction *incr_7seg* était implémentée avant le *main*, ce qui en faisait le point d'entrée du programme. Après correction, c'est à dire, définition du prototype de la fonction avant le *main* et implémentation après, tout fonctionne correctement.

b) Donnée : Testez votre application sur l'émulateur

Travail réalisé :

La figure 13 montre l'application qui vient de démarrer. Tous les 7-segments sont initialisés à zero.

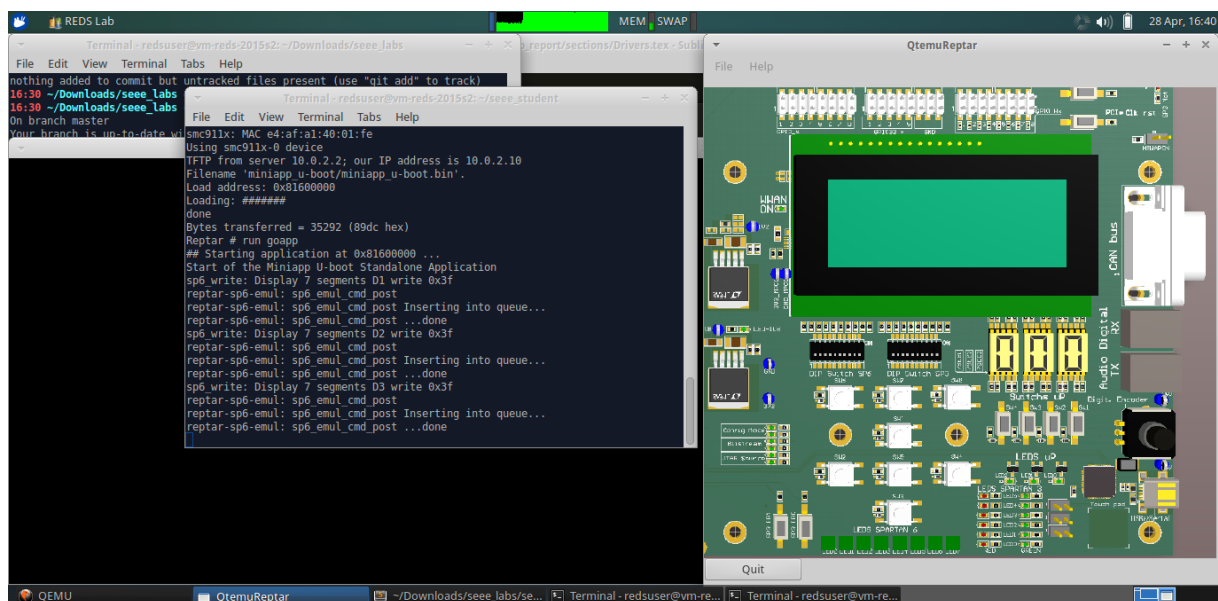


FIGURE 13 – Initialisation des 7-segments durant le lancement de l'application sur le frontend d'émulation

La figure 14 montre l'application avec plusieurs 7-segments incrémentés.

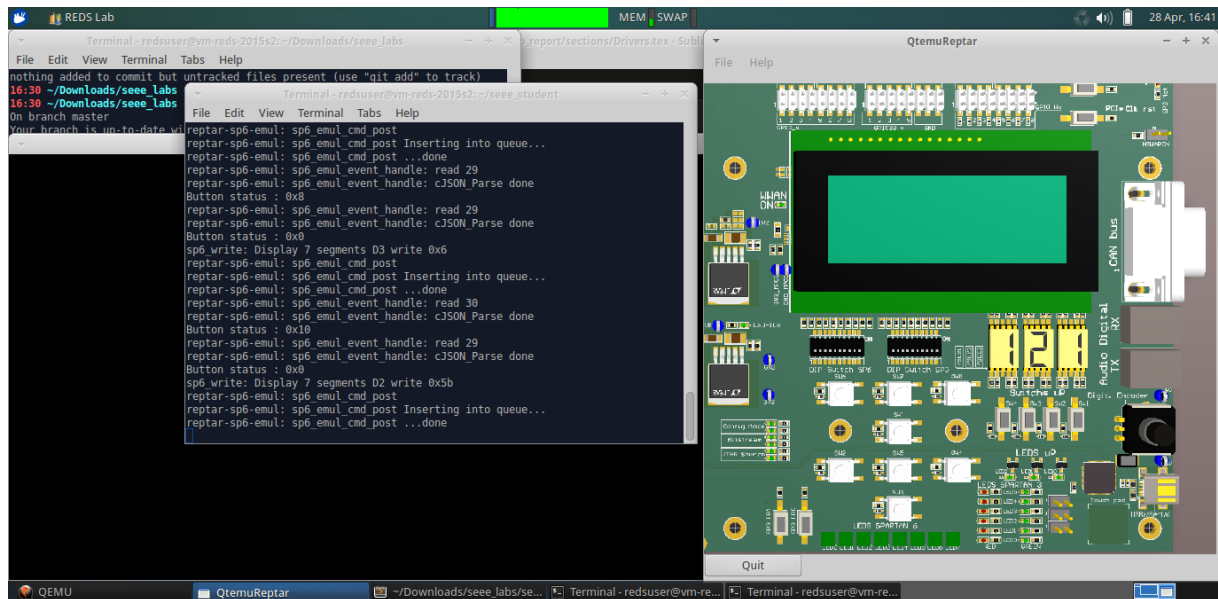


FIGURE 14 – Incrémentation des 7-segments du frontend d’émulation

La figure 15 montre la fin de l’application. Le bouton tout en bas de la croix a été pressé pour l’arrêter.

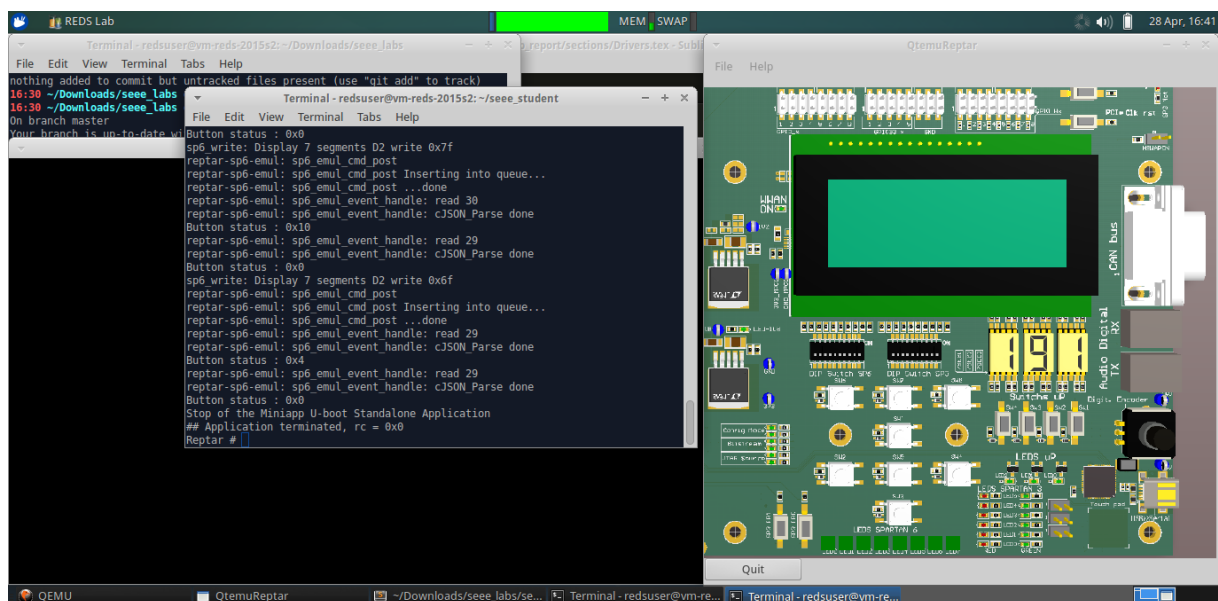


FIGURE 15 – Frontend graphique de Qemu avec affichage 7 segments

c) **Donnée** : Déployez et testez votre application sur la plate-forme réelle

Travail réalisé : Une fois l’application compilée, il a fallu copier le `.bin` dans le dossier `tftpboot`. On peut ensuite lancer l’application depuis l’U-boot en accédant la carte Reptar avec `picocom`. La pression sur le switch 3 réussit ici aussi à terminer l’application.

```

1 $ cd ~/seee_student
2 $ sudo picocom -b 115200 /dev/ttyUSB0
3 [sudo] password for redsuser:
4 ...
5 Terminal ready
6
7 Reptar # tftp 0x81600000 miniapp_u-boot.bin
8 smc911x: detected LAN9220 controller
9 smc911x: phy initialized
10 smc911x: MAC e4:af:a1:40:01:0a
11 Using smc911x-0 device
12 TFTP from server 192.168.1.1; our IP address is 192.168.1.10
13 Filename 'miniapp_u-boot.bin'.
14 Load address: 0x81600000
15 Loading: T ###
16 done
17 Bytes transferred = 35292 (89dc hex)
18
19 Reptar # go 0x81600000
20 ## Starting application at 0x81600000 ...
21 Start of the Miniapp U-boot Standalone Application
22 Stop of the Miniapp U-boot Standalone Application
23 ## Application terminated, rc = 0x0
24 Reptar #

```

Et ici, une photo de l'application en train de tourner sur la plateforme :

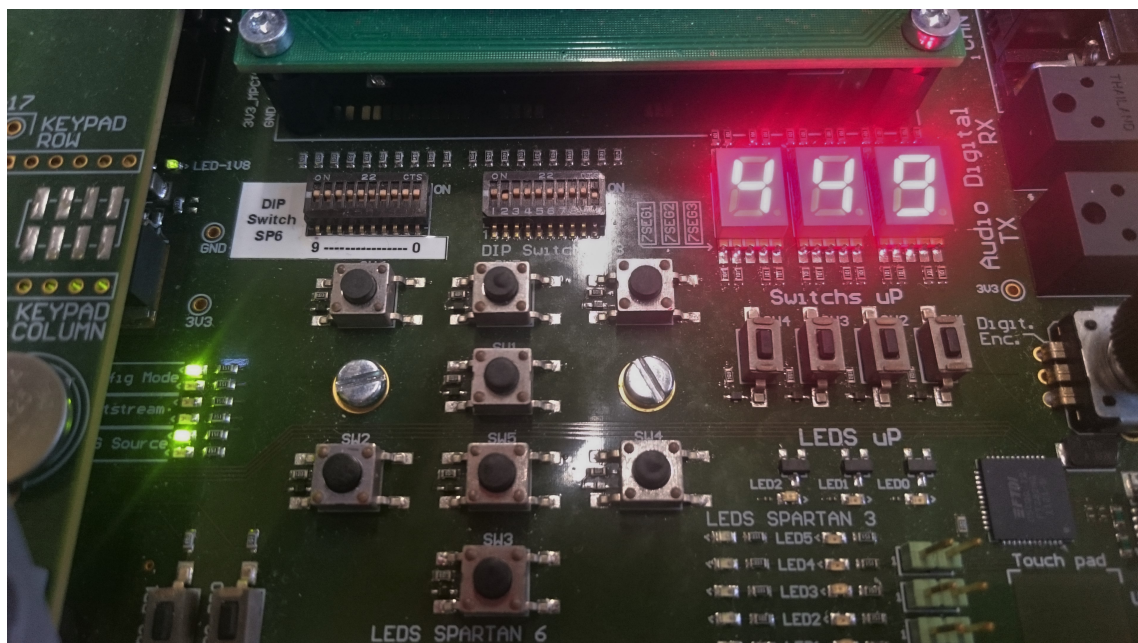


FIGURE 16 – Programme miniapp en fonctionnement sur la plateforme physique

3 Drivers

3.1 Environnement Qemu et plate-forme Reptar

Lors de cette étape, nous allons déployer l'environnement de la cible à partir d'une image de carte MMC (sdcard) et nous nous familiariserons avec l'insertion dynamique de module. Le projet concerné est le projet drivers (répertoire du même nom à la racine du workspace).

a) Donnée : Lancez make dans le répertoire drivers/

Travail réalisé :

```

1 $ cd ~/seee_student/drivers/
2 $ make
3 ...
4 make[1]: Leaving directory '/home/reduser/seee_student/linux-3.0-reptar'
5 arm-linux-gnueabi-gcc -marm -I../linux-3.0-reptar -static buttons_test.c -o
6   buttons_test
7 $

```

b) Donnée : A la racine du workspace, lancez les scripts suivants, puis la commande boot dans U-boot :

```

1 $ ./deploy
2 $ ./stf
3 Reptar # boot

```

Travail réalisé :

```

1 $ cd ~/seee_student/drivers
2 $ ./deploy
3 Deploying into reptar rootfs ...
4 Mounting filesystem/sd-card.img...
5 [sudo] password for reduser:
6 SD card partitions mounted in 'boot_tmp' and 'filesystem_tmp' directories
7 Unmounting SD card image...
8 Synchronizing .img file
9 Unmounting 'boot_tmp' and 'filesystem_tmp' ...
10 Done !
11 $ ./stf
12 ...
13 Reptar # boot
14 reading uImage
15 ...
16 *** Welcome on REPTAR (HEIG-VD/REDS): use root/root to log in ***
17 reptar login: root
18 Password:
19 #

```

c) Donnée : A la racine du rootfs (cd /), insérez le module avec la commande suivante :

```
1 # insmod sp6.ko
```

Vérifiez qu'il n'y ait aucun message d'erreur. La liste des modules chargés dynamiquement est obtenue avec la commande `lsmod` et le retrait du module avec la commande `rmmod`

```
1 # lsmod
2 # rmmod sp6
3 reptar_sp6: bye bye!
4 #
```

Travail réalisé : Avec la commande `lsmod`, on peut vérifier que notre module est correctement chargé. Si on le retire, il n'apparaît plus dans la liste.

```
1 # cd /
2 # pwd
3 /
4 # insmod sp6.ko
5 reptar_sp6: module starting...
6 Probing FPGA driver (device: fpga)
7 input: reptar_sp6_buttons as /devices/platform/fpga/reptar_sp6_buttons/input/
  input1
8 reptar_sp6: done.
9 # lsmod
10 Module                Size  Used by    Not tainted
11 sp6                    4606   0
12 # rmmod sp6
13 reptar_sp6: bye bye!
14 # lsmod
15 Module                Size  Used by    Not tainted
16 #
```

3.2 Driver de type caractère

Cette étape consiste à travailler sur un driver de type caractère au niveau FPGA. Le code de cette partie se trouve dans les fichiers `reptar_sp6.h` et `reptar_sp6.c`. Sur la base du code existant, on souhaite pouvoir écrire et lire une chaîne de caractères contenant la version du bitstream (hypothétique) dans la FPGA, stockée dans la variable globale `bitstream_version`.

a) Donnée : Complétez les callbacks `read()` et `write()` afin qu'une application utilisateur puisse lire et écrire une chaîne de (80 max.) caractères.

Travail réalisé : Les callbacks `read` et `write` ont été implémentés très sommairement dans le module avec la fonction `copy_from/to_user`. Il faut faire attention à bien retourner le nombre de bytes qui ont été lus ou écrits, pour que l'application puisse être au fait et détecter le cas échéant, si une erreur s'est produite.

```
1 # cd /
2 # insmod sp6.ko
3 reptar_sp6: module starting...
4 Probing FPGA driver (device: fpga)
5 input: reptar_sp6_buttons as /devices/platform/fpga/reptar_sp6_buttons/input /
   input1
6 reptar_sp6: done.
7 # ./usertest
8 Device ID : 0
9 Inode number : 5
10 Protection mode : 0
11 Num of hard links : 680
12 User ID of owner : 8624
13 Group ID of owner : 1
14 Device ID (spec files only): 0
15 Total size [bytes] : 0
16 Setting bitstream version to : mais coucou mon petit
17 Device ID : 0
18 Inode number : 5
19 Protection mode : 0
20 Num of hard links : 680
21 User ID of owner : 8624
22 Group ID of owner : 1
23 Device ID (spec files only): 0
24 Total size [bytes] : 0
25 Normally read buffer
26 Bitstream version : mais coucou mon petit
```

b) Donnée : Pour identifier le nom de l'entrée dans `/dev/` qui sera créée automatiquement, examinez la fonction `probe()` du driver.

Réponse aux questions :

1. Comment l'entrée dans `/dev` est-elle générée ?
2. Quel sera le nom de l'entrée dans `/dev` ?

Travail réalisé :

c) Donnée : Afin de tester votre driver, écrivez une application `usertest` (fichier `usertest.c`) qui écrira puis relira la chaîne de version en utilisant l'entrée dans `/dev` évoquée ci-dessus. Une application a aussi été codée pour tester les callbacks. Celle-ci affiche les stats du fichier créé dans le dossier `/dev`, écrit une version de bitstream totalement fabuliste, relit les stats et enfin lit le bitstream pour confirmer le succès de l'écriture.

Réponse aux questions :

1. Recherchez les valeurs du major et minor attribuées à ce driver. Expliquez votre démarche

Travail réalisé :

3.3 Pilotage des LEDs

Le code de pilotage des LEDs se trouve dans le fichier `reptar_sp6_leds.c`. La réalisation du driver des LEDs.

1. L'application graphique `qtemu` sera utilisée pour l'environnement émulé.
2. Le driver devra être également testé sur la plate-forme réelle.

Pour le pilotage des LEDs, on souhaite utiliser le sous-système `leds` présent dans le noyau Linux. Effectuez un mappage du registre des LEDs à l'aide de la fonction `ioremap()`, en vous servant de la structure `fpga_resource`.

a) Donnée : Enregistrez le device comme un device de type `leds` à l'aide de la fonction `led_classdev_register()`.

Travail réalisé :

b) Donnée : Cherchez et implémentez le(s) callback(s) gérant l'enclenchement/déclenchement des LEDs.

Travail réalisé :

Réponse aux questions :

1. Combien y a-t-il de devices de type LED gérés par notre driver ?

Il y a 6 LEDs selon la déclaration dans `reptar_sp6.h` et il y a bien 6 LEDs dans la structure `reptar_sp6_leds_pdata[]` du fichier `reptar_sp6.h`

```
1 /* Only LEDs 0 to 5 are under CPU control. 6 and 7 are used by the FPGA itself
   */
2 #define SP6_NUM_LEDS 6
```

c) Donnée : Testez le driver LED dans l'environnement `qtemu` (application graphique). Lancez le script `ledstest.sh`.

Travail réalisé :

d) Donnée : Testez votre driver sur la plate-forme (réelle) `Reptar`.

Travail réalisé :

3.4 Pilotage des boutons

Lors de cette étape, nous travaillerons sur le driver gérant la pression des boutons. L'objectif est de contrôler une application dans l'espace utilisateur à l'aide des boutons.

a) Donnée : Complétez la fonction `probe()` pour l'enregistrement des deux callbacks d'interruption (traitement immédiat + traitement différé) à l'aide de la fonction `request_threaded_irq()`.

Travail réalisé :

b) Donnée : Implémentez le traitement immédiat lié à l'interruption. Il devra :

1. stocker la valeur du registre bouton dans le champ `current_button` de la structure privée (l'adresse du registre contenant cette information est également disponible dans la structure privée)
2. acquitter l'interruption.

Travail réalisé :

c) Donnée : Testez votre driver boutons à l'aide de l'application `buttons_test`. Dans l'environnement émulé, l'application devra ouvrir le fichier `/dev/input/event1`. Il faudra taper la commande suivante :

```
1 # ./buttons_test -e1
```

Testez les boutons un par un.

Travail réalisé :