Réalisation du système pour Red Pitaya sur SD-card

G. Goavec-Mérou, J.-M Friedt, 14 août 2022

L'approche proposée n'exploite pas de paquet binaire pré-compilé mais vise, par soucis de cohérence des outils, à fabriquer soi-même tous ses outils à partir des sources, avec l'aide de `buildroot`. Cet outil propose un environnement de développement **cohérent** qui fournit *bootloader*, *toolchain*, noyau et système de fichiers *rootfs*.

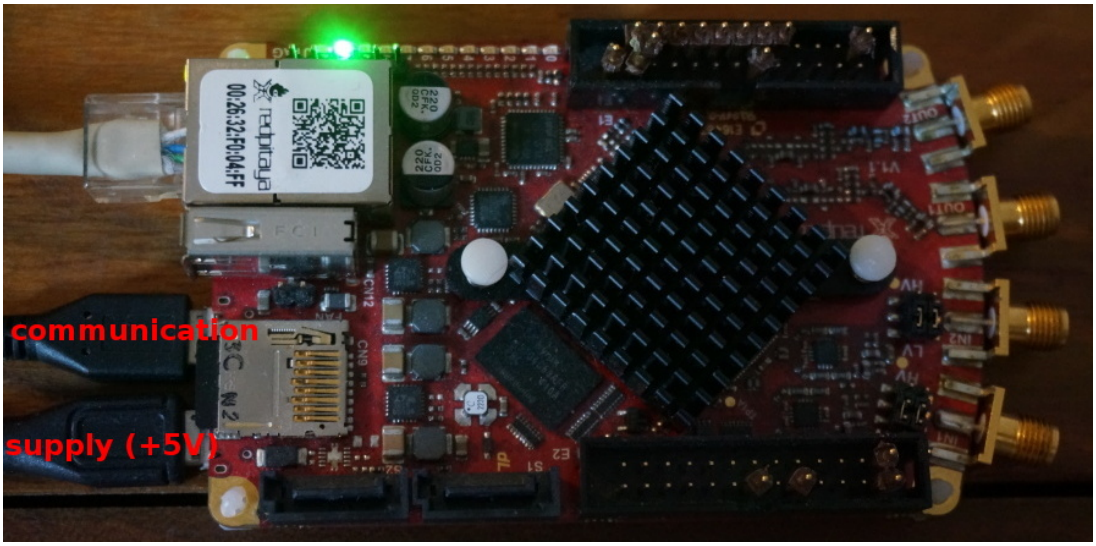


FIGURE 1 – Carte Red Pitaya. Le connecteur USB micro-B au bord de la carte, opposé au connecteur RJ45 ethernet, sert à l'alimentation. Le connecteur USB micro-B du milieu fournit un terminal sur port série virtuel.

1 Pré-requis

```
# apt-get install build-essential libncurses5-dev u-boot-tools
# apt-get install git libusb-1.0-0-dev pkg-config
```

Les commandes précédées d'un $ doivent être exécutées en mode utilisateur.
Les commandes précédées d'un # doivent être exécutées en mode super-utilisateur (`root` – précéder la commande de `sudo`).

2 Génération de tous les outils depuis les sources : `buildroot`

Buildroot [1] est un outil intégré permettant de

1. compiler sa toolchain pour la cible appropriée (dans `output/host/usr/bin/`),
2. compiler son noyau Linux,
3. compiler ses outils GNU.

Plusieurs choix se sont offerts à nous au moment de la mise en place de la chaîne de développement : un processeur aussi récent que le Zynq qui équipe la plateforme Red Pitaya force une évolution du noyau Linux qui s'efforce d'intégrer les nouvelles technologies – en particulier la dualité processeur (PS)-FPGA (PL) qui caractérise le Zynq – et les méthodes de travail associées. Xilinx, fabriquant du Zynq, a produit une branche de développement du noyau parallèle à la branche officielle. À l'heure actuelle, seule la branche de Xilinx, nommée `linux-xlnx`, permet de configurer le FPGA, selon des méthodes qui seront à terme intégrées dans le noyau officiel. Cependant, la seconde contrainte est de pouvoir expérimenter avec l'extension temps-réel de Linux – Xenomai – qui lui n'est supporté que sur les noyaux officiels. Le dilemme porte donc sur le choix d'un noyau officiel sans accès au FPGA, ou le choix du noyau divergent de Xilinx sans le support de Xenomai, avec une pérennité incertaine. Dans l'espoir d'une fusion future de `linux-xlnx` avec le noyau officiel, et compte tenu de la taille de Xilinx qui devrait garantir une certaine continuité des développements, nous avons fait le choix d'intégrer le noyau Linux dans `buildroot` et d'adapter les modifications nécessaires au support

de Xenomai sur ce noyau – version évoluée du 4.29.0 au moment de la scission du noyau officiel. L'extension de buildroot ci-dessous comportera donc un noyau spécifique pour le Zynq, supportant la configuration du FPGA ainsi que l'extension temps-rélle Xenomai, et les outils de développements associés, à l'exclusion de l'outil propriétaire de Xilinx pour configurer les FPGAs nommé Vivado.

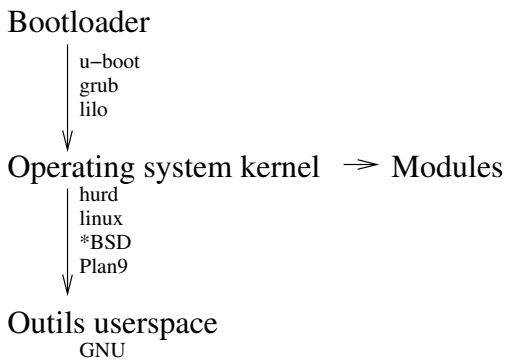


Figure 2: Éléments nécessaires au fonctionnent d'un système d'exploitation : les outils générant toutes ces étapes doivent être cohérents.

L'avantage de buildroot [1], comme de open-embedded [2], est de fournir un environnement intégré et cohérent qui cache un certain nombre de détails au développeur (Fig. 2). L'inconvénient de buildroot, comme de open-embedded, est de fournir un environnement intégré et cohérent qui cache un certain nombre de détails au développeur (!). Nous allons profiter de cet environnement de travail pour compiler tous les outils nécessaires au travail sur Red Pitaya sans dépendre d'une distribution binaire externe, puis pour ajouter l'extension temps-réelle de Linux nommée Xenomai [2]. Cette extension nécessite, en plus d'une mise à jour du noyau, des outils en espace utilisateur que buildroot peut gérer. Le port de Xenomai à Red Pitaya sera décrit en détail dans la section 7.

Buildroot, disponible à `https://github.com/buildroot/buildroot`, ne supporte pas la plateforme Red Pitaya. Nous devons exploiter un mécanisme d'ajout de fonctionnalité au travers de la variable d'environnement `BR2_EXTERNAL` qui référence le répertoire absolu contenant les fichiers manquant dans la distribution officielle pour le support d'une plateforme. Nous téléchargerons donc un second dépôt en complément de la distribution officielle de `buildroot` afin d'ajouter les fonctionnalités manquantes en faisant pointer `BR2_EXTERNAL` vers une arborescence cohérente avec l'organisation des fichiers attendue par `buildroot`.

3 Compiler une image Linux sans support Xenomai (sans support temps réel)

Nous partons d'une version connue de `buildroot`, au lieu de la dernière version disponible sur `github`, afin de connaître la version de chaque outil :

```
wget https ://buildroot.org/downloads/buildroot-2021.02.1.tar.gz
```

Nous complétons la distribution par défaut par les fonctionnalités additionnelles pour supporter Red Pitaya :

```
git clone https ://github.com/trabucayre/redpitaya.git
```

qui crée le répertoire `redpitaya` avec les fichiers nécessaires. Nous aurons besoin d'expliquer à `buildroot` où ces fichiers additionnels se trouvent, et ce au travers de la variable d'environnement `BR2_EXTERNAL`. Nous pourrons soit faire pointer cette variable vers le chemin absolu du dépôt `redpitaya`, soit exécuter le script `sourceme.ggm` dans le répertoire `redpitaya` que nous venons de cloner :

```
cd redpitaya
source sourceme.ggm
```

On prendra soin de recharger le script dans chaque nouveau terminal, la définition de `BR2_EXTERNAL` n'étant que locale au terminal actif et n'étant pas mémorisée entre diverses sessions de travail.

Entrer dans le répertoire contenant `buildroot` que nous avons décompressé et désarchivé par

```
tar zxvf buildroot-2021.02.1.tar.gz
```

Dans ce répertoire, configurer `buildroot` pour la Red Pitaya :

```
cd buildroot-2021.02.1
make redpitaya_defconfig
```

Ensuite, `make` génère une première image fonctionnelle en téléchargeant toutes les sources nécessaires et en les compilant. À l'issue de ce travail (quelques heures de compilation et 5 GB plus tard), l'ensemble des binaires pour la Red Pitaya se trouvent dans `output/images`.

⚠ La ligne qui va suivre peut **corrompre le disque dur** si le mauvais périphérique est sélectionné. Toujours **vérifier** le nom du périphérique associé à la carte SD (`dmesg | tail`) avant de lancer la commande `dd`.

1. `buildroot.uclibc.org/`
2. `www.openembedded.org`

L'image résultant de la compilation est transférée sur la carte SD par
`sudo dd if=output/images/sdcard.img of=/dev/sdc` nous avons volontairement choisi le périphérique `/dev/sdc` dans cet exemple car rarement utilisé : en pratique, la carte SD sera souvent nommée `/dev/sdb` (second disque dur compatible avec le pilote SCSI de Linux) ou `/dev/mmcblk0` (cas du lecteur SD interne).

> ⚠ **Attention** : le contenu de la carte SD, ou de tout support pointé par le dernier argument de cette commande, sera **irrémédiablement** perdu. Vérifier à deux fois le nom du périphérique cible de l'image issue de buildroot.

Une fois l'image flashée sur la carte SD, nous constaterons deux partitions : une première en VFAT (format compatible Microsoft Windows) avec le devicetree, le noyau linux et le bootloader, et une seconde partition contenant le système GNU/Linux. Cette seconde partition est dimensionnée *exactement* à la taille nécessaire pour stocker le système, et aucun espace n'est disponible pour nos propres ajouts (nous verrons plus tard que nous voudrons par exemple placer les bitstreams pour configurer le FPGA sur le système de fichiers). Pour pallier à cette déficience, monter la carte SD et redimensionner la seconde partition par

1. effacer la partition au moyen de `fdisk` : lancer ce programme en tant que root, afficher la configuration actuelle (nous en aurons besoin ci-dessous) par `p`, puis `d` pour effacer la partition 2,
2. re-créer cette partition avec tout l'espace disponible sur la carte : `n` pour créer une nouvelle partition, `p` pour une partition primaire, fournir le premier secteur de la partition tel que nous l'avions affiché auparavant (dans la capture d'écran ci-dessous, ce premier secteur est 32769), et accepter le choix par défaut d'occuper tout l'espace. Conclure par `w` pour écrire cette nouvelle configuration,
3. après avoir quitté `fdisk`, lancer `resize2fs /dev/sdd2` pour redimensionner le système de fichiers à la taille de la partition.

À l'issue de ces opérations, le système GNU/Linux occupe tout l'espace disponible. Dans l'exemple ci-dessous, nous avions commencé avec une configuration de la forme

```
Device     Boot Start     End Sectors  Size Id Type
/dev/sdd1  *        1   32768   32768   16M  c W95 FAT32 (LBA)
/dev/sdd2       32769 1081344 1048576  512M 83 Linux
```

et après redimensionnement nous finissons avec

```
Device     Boot Start     End Sectors  Size Id Type
/dev/sdd1  *        1   32768   32768   16M  c W95 FAT32 (LBA)
/dev/sdd2       32769 7796735 7763967  3.7G 83 Linux
```

L'émulation de la liaison asynchrone (RS232) est fonctionnelle sur le port USB en milieu de carte : lancer `minicom` après avoir branché le câble USB-micro B, en 115200 bauds, 8N1. Un prompt de la forme `redpitaya>` doit apparaître à l'appui de la touche ENTER.

Diverses fioritures additionnelles telles qu'un serveur d'interface graphique (X11, voir annexe B) ne sont pas développées plus en détail dans le corps du texte mais reléguées en annexe.

3.1 Configuration du réseau

La configuration du réseau entre l'hôte et la cible sont pris en charge par les commandes de configuration des interfaces `ifconfig` pour assigner une adresse logique (IP) à une interface physique (adresse MAC) et établir la liaison point à point entre les ordinateurs d'un sous-réseau et leur passerelle. Cette passerelle sera chargée de transmettre les paquets de données vers l'extérieur du sous-réseau, voir vers internet, si l'adresse de destination n'est pas un des ordinateurs du sous-réseau. La commande pour établir ces règles de routages est `route` : nous n'en aurons pas besoin car GNU/Linux assigne automatique la règle de router les paquets vers l'interface du sous-réseau associé à l'adresse du destinataire.
Le réseau est par défaut configuré pour communiquer sur le port Ethernet, avec l'adresse IP de la carte accessible par `ifconfig`.

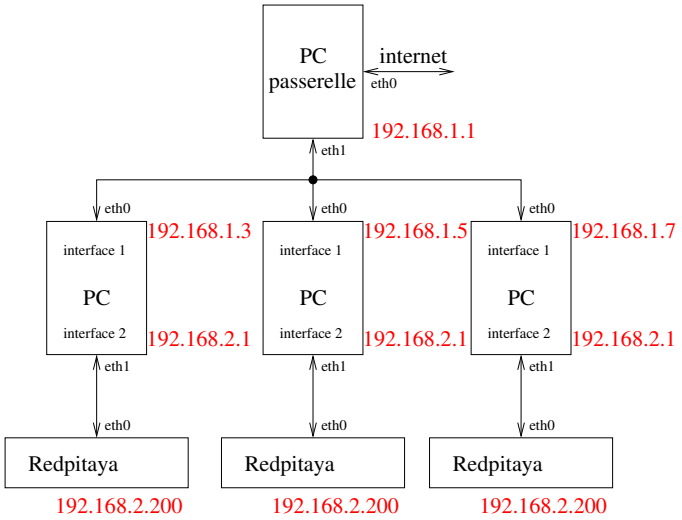


Figure 3: Architecture du réseau TCP/IP

Nous constatons

```
redpitaya> ifconfig
eth0      Link encap:Ethernet  HWaddr 22:3D:C4:D9:DB:CD
          inet addr:192.168.2.200  Bcast:192.168.1.255  Mask:255.255.255.0
...
```

que l'adresse par défaut est 192.168.2.200, soit un réseau local [3].

Pour configurer une interface réseau (sur PC ou sur Red Pitaya), la commande `ifconfig` permet de manipuler la configuration des interfaces. Elle prend comme argument le nom de l'interface et l'adresse IP correspondante :

```
ifconfig eth0 192.168.2.1
```

Avant de configurer une interface, la liste des interfaces, incluant celles qui n'ont pas été configurées, s'obtient par `ifconfig -a`. Cette commande permet aussi d'identifier l'adresse matérielle (MAC) dans le champ `ether`, information transmise à la passerelle lors de la transmission des paquets vers internet.

`ifconfig` tend à devenir obsolète et être remplacée par `ip`. Le même résultat que ci-dessus s'obtient par

```
ip address add 192.168.2.1 dev eth0
ip link set dev eth0 up
```

La configuration réseau entre deux ordinateurs se valide par `ping IP_dest` en demandant une réponse (`ping`) de l'ordinateur distant d'adresse IP `IP_dest`.

Une fois la configuration réseau établie et validée manuellement, il est possible d'automatiser cette démarche de configuration en l'inscrivant dans `/etc/network/interfaces`. Nous ajoutons les règles de configuration de l'adresse IP sur la Red Pitaya par

```
auto eth0
iface eth0 inet static
  address 192.168.2.200
  netmask 255.255.255.0
  gateway 192.168.2.1
```

Cette configuration est activée sur la Red Pitaya par `ifup eth0`. En cas de configuration antérieure de l'interface, il peut être utile de la déconfigurer par `ifdown eth0`. Si une telle configuration des interfaces du réseau est aussi utilisée sur PC, vérifier que Network Manager n'entre pas en conflit avec `/etc/network/interfaces` : on pourra dire à Network Manager de ne pas modifier les interfaces décrites dans `/etc/network/interfaces` en indiquant `managed=false` dans `/etc/NetworkManager/NetworkManager.conf` du PC.

Nous aurons par la suite besoin d'un serveur web : `lighttpd` est une solution légère et stable. Pour ajouter une nouvelle application, `make menuconfig` et rechercher l'emplacement où le serveur web s'active, soit par une recherche (`/` suivi de `lightt`), soit en activant directement `Target packages` → `Networking applications` → `lighttpd`.

On se facilitera par ailleurs la vie en installant un éditeur de texte sur la cible. Le plus efficace, `vim`, nécessite d'activer le support `wchar` et `curses` par `Toolchain` → `Enable WCHAR support` et `Target packages` → `Libraries` → `Text and terminal handling` → `ncurses`. Une fois ces bibliothèques activées, nous faisons apparaître le support pour `vim` de `busybox` par `Target packages` → `Show packages that are also provided by busybox` puis `Target packages` → `Text editors and viewers` → `vim`. De nombreux autres éditeurs alternatifs sont disponibles dans ce même menu. Une fois le choix fini, quitter `make menuconfig`, lancer `make` puis une fois la compilation achevée, transférer l'image sur la carte SD à nouveau.

Nous validons le résultat de cette nouvelle image en lançant la Red Pitaya et nous assurant que les nouveaux outils sont disponibles :

```
redpitaya> which lighttpd
/usr/sbin/lighttpd
```

3. les plages d'adresses de réseaux privés, qui ne sont pas routées sur internet, sont décrites dans le RFC1918 disponible à `https://tools.ietf.org/html/rfc1918`

3.1.1 Correspondance adresse IP-nom de domaine

Le serveur de nom de domaine (qui effectue la traduction entre nom de domaine et adresse IP) – DNS – est informé dans le fichier `/etc/resolv.conf`. Cette information n'est nécessaire que pour accéder à un domaine sur internet (par exemple `free.fr` ou `google.com`) et est inutile tant que nous nous limitons à une connexion locale où l'interlocuteur est renseigné par son adresse IP. Cette information est déjà renseignée sur les PCs, mais la Red Pitaya ne sait pas faire la correspondance entre un nom (e.g. `sequanux.org`) et son adresse IP correspondante (188.165.36.56). Pour ce faire, il faut compléter le fichier de configuration : les informations dans
`/etc/resolv.conf` sont de la forme

```
nameserver 194.57.91.200
nameserver 194.57.91.201
```

avec 194.47.91.20{0,1} les serveurs de nom de domaine de l'université de Franche Comté. Sur un autre site, on s'informera du DNS en consultant `/etc/resolv.conf` du PC. Une solution générique est de renseigner `9.9.9.9` comme DNS [4].

Lors de la connexion par le Shell Sécurisé `ssh` depuis le PC vers la Red Pitaya, il faut fournir le mot de passe pour se connecter sur l'unique compte qu'est l'administrateur `root`.

Le mot de passe par défaut de `root` est **root**.

3.1.2 NFS (*Network File System*)

Dans le cas particulier de la salle 235B, le répertoire autorisé au partage NFS est `/home/etudiant/nfs`. Ce répertoire est l'équivalent de la référence à `/home/utilisateur/target` dans la description plus générale qui suit.

Le partage de répertoire par NFS permet de faire apparaître une partie du disque dur du PC (hôte) comme un répertoire de la carte embarquée Red Pitaya (cible). Ce partage virtuel de fichiers au travers d'une connexion réseau fournit un environnement souple de développement puisqu'à l'issue de la cross-compilation d'un binaire sur le PC, l'exécutable résultant est immédiatement disponible dans le répertoire partagé pour exécution sur la plateforme embarquée.

Créer pour ce faire sur le PC, un répertoire `/home/utilisateur/target` qui sera le répertoire partagé, et dans le fichier `/etc/exports` de l'hôte, indiquer que nous voulons autoriser le partage de ce répertoire par l'ajout de la ligne : `/home/utilisateur/target *(rw,sync)`

Relancer NFS sur le PC pour prendre en compte cette configuration :

```
#/etc/init.d/nfs-kernel-server restart
```

Ce répertoire permettra d'échanger des fichiers entre l'hôte et la cible, par exemple les exécutables que nous aurons cross-compilés sur le PC et que nous désirons exécuter sur Red Pitaya. Sur la carte Red Pitaya, le répertoire est monté par `mount -o nolock IP_PC:/home/utilisateur/target /mnt` : le contenu de `/home/utilisateur/target` de l'hôte apparaît désormais dans `/mnt` de la cible.

Connexion à internet en configurant le PC comme passerelle (optionnel)

Nous ne faisons que survoler ici la possibilité d'utiliser le PC comme une passerelle qui se charge de la traduction d'adresses IP afin de permettre à la carte Red Pitaya de se connecter à internet : cette fonctionnalité est dans un premier temps inutile et on pourra s'en passer. L'outil qui permet de faire cette traduction se nomme `iptables` : un exemple de configuration qui transfère tous les paquets entre les interfaces `eth0` et `eth1` est fournie ci-dessous. Cette configuration s'appliquer sur le PC puisque nous supposons que `eth0` est sur le sous réseau de la passerelle connectée à internet, et `eth1` sur le sous-réseau de la Red Pitaya, et nécessite les droits d'administration (`root`).

```
monpath=/sbin
echo "1" > /proc/sys/net/ipv4/ip_forward
echo "1" > /proc/sys/net/ipv4/ip_dynaddr
${monpath}/iptables -P INPUT ACCEPT
${monpath}/iptables -F INPUT
${monpath}/iptables -P OUTPUT ACCEPT
${monpath}/iptables -F OUTPUT
${monpath}/iptables -P FORWARD ACCEPT
```

4. `https://www.quad9.net/`

```
${monpath}/iptables -F FORWARD
${monpath}/iptables -t nat -F

${monpath}/iptables -A FORWARD -i eth0 -o eth1 -m state --state ESTABLISHED,RELATED -j ACCEPT
${monpath}/iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
${monpath}/iptables -A FORWARD -j LOG

${monpath}/iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

3.2 Exécution dans un émulateur

En l'absence de matériel, l'image issue de la compilation de `buildroot` peut être exécutée depuis un émulateur, ici `qemu` émulant un processeur ARM. Nous utilisons dans un premier temps, tant que l'interface réseau `eth0` n'est pas nécessaire, la version en paquet sous Debian GNU/Linux fourni comme `qemu-system-arm`. Une fois ce paquet installé, se placer dans le répertoire `output/images` de Buildroot et lancer

```
qemu-system-arm -append "console=ttyPS0,115200 root=/dev/mmcblk0 rw earlyprintk" -M xilinx-zynq-a9 \
-m 1024  -serial mon:stdio -dtb zynq-red_pitaya.dtb -nographic -kernel zImage  -sd rootfs.ext4
```

pour booter Linux, charger le système de fichier (`rootfs`) et finir le chargement du système par l'affichage du prompt. Toutes les commandes système sont fonctionnelles, mais l'accès au matériel nécessite évidemment une émulation spécifique que `qemu` ne fournit pas nécessairement. Les GPIOs, par exemple, semblent bien émulées

```
redpitaya> echo "908" > /sys/class/gpio/export
redpitaya> echo "out"  > /sys/class/gpio/gpio908/direction
redpitaya> echo "1"  > /sys/class/gpio/gpio908/value
```

même si les conséquences de l'écriture dans les registres de contrôle des périphériques ne sont pas évidentes.

Les inconvénients de cette approche sont que 1/ l'interface Ethernet de la Red Pitaya n'est pas reconnue et que 2/ seule le système invité (*guest* avec la Red Pitaya émulée sur `qemu`) peut accédér à l'hôte (le PC sur lequel tourne `qemu`) mais que les connexions de l'hôte vers guest ne sont pas supportées sans définir explicitement une inferface réseau le permettant. Pour pallier ces déficiences, nous compilons `qemu` avec la branche fournie par Xilinx tel que décrit à `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842060/QEMU` et en particulier pour le Zynq de la Red Pitaya à `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842054/QEMU+-+Zynq-7000`. Noter qu'il est inutile de compiler tout `qemu` et que après avoir ajouté le compilateur de *devicetree* par `git submodule update --init dtc` nous ne compilons que la cible désirée par `$./configure --target-list="aarch64-so`

Grâce à la compilation de l'archive fournie par Xilinx à `https://github.com/Xilinx/qemu` telle que décrite ci-dessus, nous obtenons `aarch64-softmmu/qemu-system-aarch64` avec le support de la machine `arm-generic-fdt-7series` dont l'interface Ethernet est reconnue comme `eth0`. La ligne de commande pour lancer `qemu` avec cette fonctionnalité est

```
sudo .../aarch64-softmmu/qemu-system-aarch64 -append "console=ttyPS0,115200 root=/dev/mmcblk0 \
rw earlyprintk" -M arm-generic-fdt-7series -machine linux=on -smp 2 -m 1024  -serial mon:stdio \
-dtb zynq-red_pitaya.dtb -nographic -kernel zImage  -sd rootfs.ext4  -net nic -net nic -net nic \
-net nic -net tap,downscript=no
```

le tout lancé depuis le répertoire `output/images` de `buildroot` pour éviter de donner le chemin complet de toutes les archives (devicetree `zynq-red_pitaya.dtb`, root file system `rootfs.ext4`, noyau Linux `zImage`). Noter la **nécessité d'être administrateur sur le PC** pour créer l'interface réseau `tap0` sur l'hôte.

Une fois la Red Pitaya virtuelle lancée dans `qemu-system-aarch64`, nous constatons (`ifconfig -a`) la présence de `eth0` qui fournit l'interface réseau côté cible. Sur le PC, une nouvelle interface a été crée au lancement de `qemu` nommée `tap0` : sur le PC, `sudo ifconfig -a | grep tap` indique la présence de cette nouvelle interface. **Attention** : si une ancienne instance de qemu a mal été tuée ou tourne en tâche de fond, il se peut qu'une nouvelle interface `tap` – par exemple `tap1` – soit créée. Prendre soin de bien identifier quelle interface est associée à quelle instance de `qemu`.

L'interface `tap` étant identifiée, il reste sur le PC à la configurer avec une adresse du même sous-réseau – mais évidemment différente sur l'octet de poids faible – que l'adresse de la Red Pitaya. Une fois cette configuration effectuée sur l'hôte et la cible, `ping` permet de vérifier le bon fonctionnement de la liaison.

La figure 4 démontre la configuration de l'interface `eth0` de l'émulation de la Red Pitaya (adresse 192.168.0.10) dans `qemu` de façon cohérente avec l'interface `tap0` créée sur l'hôte (même sous-réseau 192.168.0.x avec le PC assigné arbitrairement à 192.168.0.5) et la capacité d'un client web sur l'hôte d'accéder au serveur `lighttpd` exécuté sur le guest.

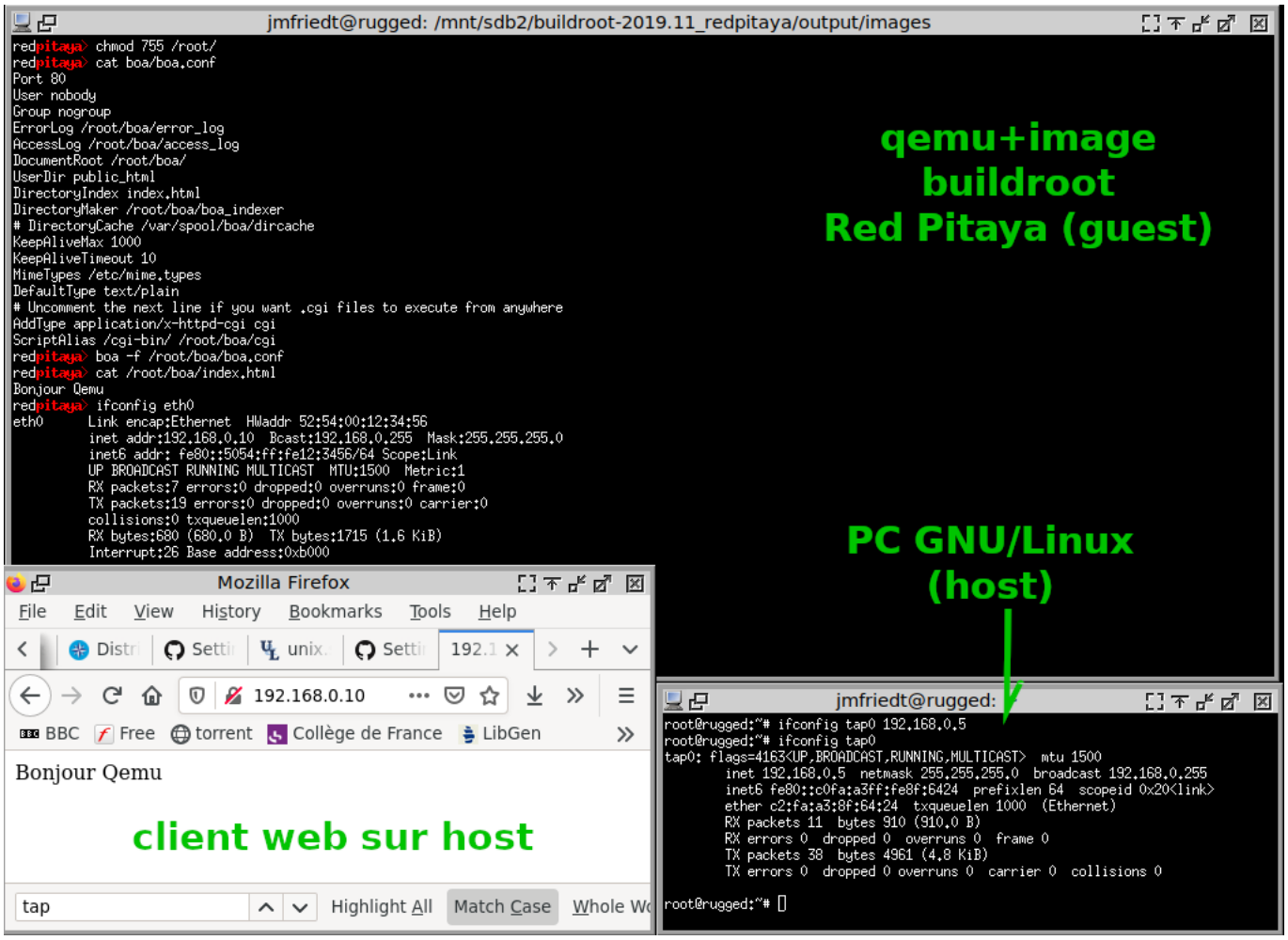


FIGURE 4 – En haut à gauche le guest : `qemu` de Xilinx émulant la Red Pitaya. En bas à gauche le client web exécuté sur l'hôte et connecté au serveur web sur le guest. En bas à droite la configuration de l'interface de l'hôte créée au lancement de `qemu` avec l'option `-net tap`.

Dernier point : si on veut éviter de s'handicaper avec une connexion sécurisée `ssh` de l'hôte vers le guest, un serveur telnet `telnetd` est disponible dans `buildroot` de la Red Pitaya comme fonctionnalité de busybox accessible par `make busybox-menuconfig→networking→telnetd`. De cette façon, la Red Pitaya est accessible par `telnet 192.168.0.10` (selon la configuration visible sur Fig. 4) sans s'handicaper avec les échanges de clés cryptographiques, mais au détriment d'informations circulant en clair sur le réseau.

Les fichiers peuvent se transférer de l'hôte vers la cible par `scp`. Par ailleurs, NFS fonctionne aussi par ce biais, sous réserve d'avoir une connexion TCP/IP fonctionnelle entre hôte et guest et d'avoir exporté le répertoire de l'hôte en renseignant l'adresse IP du guest dans `/etc/exports` (voir plus haut en section 3.1.2).

`qemu` se quitte "proprement" comme `minicom`, par CTRL-a puis x.

3.3 Serveur web

Nous avons vu comment compléter la configuration de buildroot pour Red Pitaya pour ajouter un serveur web léger, qui supporte les scripts CGI, nommé `lighttpd`. Une configuration par défaut dans `/etc/lighttpd/lighttpd.conf` propose le répertoire `/var/www/` comme dépôt par défaut des fichiers accessibles par le web. Le serveur se lance ainsi par `lighttpd -f /etc/lighttpd/lighttpd.conf`.

Le support CGI s'ajoute en fournissant dans le répertoire `/etc/lighttpd/conf.d/cgi.conf` le fichier de configuration

```
server.modules += ( "mod_cgi" )
cgi.assign                  = ( ".pl"  => "/usr/bin/perl",
                                ".cgi" => "/bin/sh",
                                ".py"  => "/usr/bin/python",
                                ".php" => "/usr/bin/php-cgi" )

index-file.names            += ( "index.pl",   "default.pl",
                                 "index.rb",   "default.rb",
                                 "index.erb",  "default.erb",
                                 "index.py",   "default.py",
                                 "index.php",  "default.php" )
```

et en ajoutant dans `/etc/lighttpd/lighttpd.conf` la ligne `include "conf.d/cgi.conf"`.

Une fois la modification effectuée, le serveur web se relance par `/etc/init.d/S50lighttpd restart`. En cas d'erreur au lancement, on vérifiera dans `/var/log/lighttpd-error.log` qu'il n'y a pas d'erreur concernant "pcre support is missing for : dir-listing.exclude". Dans le cas contraire, on pourra désactiver ce module en commentant

```
# include "conf.d/dirlisting.conf"
```

dans `/etc/lighttpd/lighttpd.conf`.

 Ne jamais lancer de service avec les autorisations d'administrateur (`root`). Même si une telle configuration rend l'accès aux ressources matérielles plus facile, elle met en danger le principe de séparation des autorisations d'unix selon les services (exemples de mauvaises configurations de serveurs web : *Exploiting Network Surveillance Cameras Like a Hollywood Hacker*, Black Hat 2013, à `https://www.youtube.com/watch?v=B8DjTcANBx0`). Ici, l'utilisateur et le groupe `www-data` n'ont aucun pouvoir sur le système et une défaillance de sécurité du serveur web ne se traduit pas par une remise en cause de l'intégrité du système.

Pour exécuter un script cgi, nous pouvons créer le sous répertoire `/var/www/cgi` (en accord avec le fichier de configuration) et y placer par exemple un script shell (puisque les extensions `.cgi` sont associées aux scripts bash) :

```
#!/bin/sh
echo -e "Content-type: text/html\r\n\r\n"
echo "Hello CGI"
```

qui sera appelé en faisant pointer le client web vers `http://IP/cgi/script.cgi`. On pensera naturellement à autoriser l'exécution (`chmod 755`) du script shell. Cette méthode de travail permet donc de générer dynamiquement des pages web, par exemple pour représenter la mesure de capteurs.

Il est par ailleurs possible de générer dynamiquement une page au moyen du CGI (*Common Gateway Interface*) qui exécute un script chargé de générer du code HTML interprétable par le client. La spécification du protocole, `http://www.ietf.org/rfc/rfc3875.txt`, indique les méthodes d'échange d'informations entre le client et le serveur.

Le script hébergé sur le serveur peut, par exemple, lire l'état d'un capteur ou d'un GPIO et le retranscrire sur la page affichée. Plus intéressant, la requête HTML peut contenir des noms de variable et leur affectation, en vue d'une utilisation par le script. Deux méthodes répondent à ce principe : POST et GET. La seconde est celle qui sera favorisée ici puisque les données transmises sont contenues dans l'url, une méthode simple pour communiquer entre client et serveur [5]. Ainsi, pour passer un argument au script CGI, nous utiliserons une URL de la forme

```
http://192.168.2.200/cgi/scrpit.cgi?variable1=val1&variable2=val2&var3=val
```

La seule subtilité dans ce contexte est de

1. maîtriser l'exécution du script CGI, quelquesoit le langage dans lequel il est rédigé (la première ligne, comme dans tout script, informe sur la localisation de l'interpréteur),
2. récupérer la liste des arguments transmis au travers de l'url. Les variables contenant ces paramètres sont décrites en page 8 de `http://www.ietf.org/rfc/rfc3875.txt`.

5. noter que l'apparente protection de la méthode POST contre un usage malveillant est trivialement contournée par des outils tels que `curl`

Nous proposons un squelette de script CGI fonctionnel, qui nécessite pour comprendre le sens de la dernière ligne de se référer à la section d'accès au matériel 4, sous la forme de

```
[root@buildroot cgi]# less cgi/script.cgi
#!/bin/sh
echo -e "Content-type: text/html\r\n\r\n"
echo "Hello CGI<br>"
echo $QUERY_STRING "<br>"
led=‘echo $QUERY_STRING | cut -d= -f2‘
echo $led "<br>"
echo $led > /sys/class/leds/led8/brightness
```

qui se situe dans `boa/cgi` (tel que défini dans la variable `ScriptAlias` du fichier de configuration de `boa`). La chaîne de caractères fournie en argument de l'URL est accessible par `$QUERY_STRING` et est ici découpée dans ses éléments individuels par `cut`.

Cependant, nous devons nous assurer que le propriétaire du serveur web, défini comme l'utilisateur `www-data` pour `lighttpd` pour séparer les permissions d'accès et limiter les risques d'intrusion en cas de dysfonctionnement du serveur, a accès aux ressources matérielles que nous sollicitons. Ceci n'est généralement pas le cas, et nous devons prendre soin de modifier les autorisations :

```
# ls -l /sys/class/leds/led*/*
lrwxrwxrwx    1 root     root             0 Jan  1 00:13 brightness
...
# chown www-data.www-data /sys/class/leds/led*/brightness
# ls -l /sys/class/leds/led*/
```

pour `lighttpd`.

Démontrer le contrôle des *deux* LEDs 8 et 9 en fournissant un argument GET permettant de fournir les deux arguments à `lighttpd`.

4 Accès au matériel depuis le shell

4.1 /sys/class/leds

Si nous désirons utiliser le pilote LEDs, alors la séquence suivante

```
redpitaya> cd /sys/class/leds/
redpitaya> echo "1" > led8/brightness   # orange
redpitaya> echo "0" > led8/brightness
redpitaya> echo "0" > led9/brightness   # rouge
redpitaya> echo "1" > led9/brightness
```

permet de vérifier le bon fonctionnement de l'accès aux ressources matérielles par le noyau Linux.

L'interface `leds` est donc plus restrictive que l'interface `gpio` mais plus simple à utiliser. Diverses interfaces sont disponibles depuis l'espace utilisateur pour accéder aux ressources matérielles. La tendance tend à proposer des interfaces au travers de `/sys`. Deux méthodes nous permettent d'accéder aux GPIOs connectés au processeur (MIO dans la nomenclature Zynq, qui s'opposent aux EMIO connectés au PL) grâce aux pilotes fournis par Linux : `/sys/class/leds` et `/sys/class/gpio`.

4.2 /sys/class/gpio

Si ces pilotes sont compilés sous forme de module, nous constatons leur chargement en mémoire par

```
redpitaya> lsmod
Module                  Size  Used by    Not tainted
gpio_zynq               6509  2
leds_gpio               2890  0
led_class               3299  1 leds_gpio
```

Pour accéder aux GPIOs, un pilote fournit accès à `/sys/class/gpio`[6]. Une particularité de `/sys` est de prendre en argument des chaînes de caractère lisibles par un humain (au contraire des valeurs binaires pour les périphériques accessibles dans `/dev`).

Concrètement, le pilote `gpiolib` suppose qu'un port numérique propose au maximum 32 broches d'entrée-sortie. Ainsi, l'indice de la broche `n` sur port `P` est `P`×32+`n`, avec le port A d'indice 0, B d'indice 1 etc ... Dans le cas de la Red Pitaya, les broches associées aux MIO sont indexées à partir de la valeur 906. Ainsi, MIO0 est d'indice 906 et MIO7 d'indice 913 : deux LEDs – orange et rouge – sont connectées à ces deux broches. Nous démontrons l'utilisation de ces périphériques par la séquence suivante.

Dans un premier temps, nous allons requérir la ressource par

```
redpitaya> echo "906" > /sys/class/gpio/export
-sh: write error: Device or resource busy
```

qui échoue car le pilote gérant les LEDs monopolise déjà ces ressources – mettant en évidence le rôle du noyau pour garantir la cohérence de l'accès aux ressources par les diverses applications utilisateur.

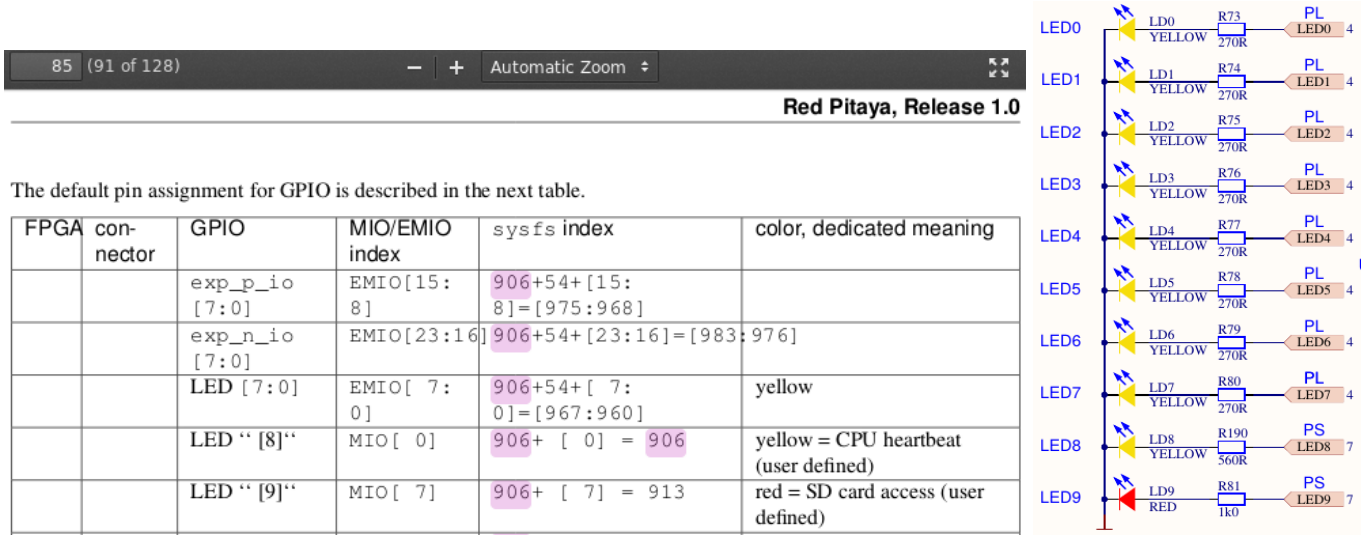


FIGURE 5 – LEDs disponibles sur Red Pitaya : noter que seules LED8 et LED9 sont accessibles depuis le PS (i.e. sans configurer le FPGA du PL). Extraits de `media.readthedocs.org/pdf/rpdocs/latest/rpdocs.pdf` et `downloads.redpitaya.com/doc/Red_Pitaya_Schematics_STEM_125-10_V1.0.pdf` respectivement

.

Nous avons pris soin de configurer les pilotes gérant les ressources matérielles comme modules, ce qui nous permet de libérer la ressource en retirant les pilotes de la mémoire :

```
redpitaya> rmmod leds_gpio
redpitaya> rmmod led_class
redpitaya> echo "906" > /sys/class/gpio/export
```

Cette fois, la requête de la ressource réussit – absence de message d'erreur – et se traduit par la création du répertoire contenant les pseudo-fichiers permettant l'accès aux ressources matérielles : `gpio906`[7]. Les pseudo-fichiers gèrent la direction des transactions (`out` ou `in`) et, pour des broches en sortie, leur état (`1` ou `0`).

```
redpitaya> echo "out" > /sys/class/gpio/gpio906/direction
redpitaya> echo "1" > /sys/class/gpio/gpio906/value
redpitaya> echo "0" > /sys/class/gpio/gpio906/value
```

permet de constater le bon fonctionnement de l'accès aux ressources matérielles par le shell.

6. pour activer cette fonctionnalité, il faut activer le pilote adéquat par `Device Drivers→GPIO Support→/sys/class/gpio/... (sysfs interface)`

7. `http://forum.redpitaya.com/viewtopic.php?f=14&t=115` explique que LED8 = MIO[0]=gpio906 et LED9 = MIO[7]=gpio913

4.3 Concept de `devicetree`

Un SOC (*System On Chip*) fournit de nombreux périphériques autour d'un même cœur de processeur. La description de ces périphériques peut devenir complexe et nécessiter l'héritage de propriétés : sur architecture ARM, la méthode la plus récente consiste à utiliser le `devicetree` (fichier d'extension `dts` pour être éditable par un humain, `dtb` après conversion en format binaire lisible par les pilotes du noyau).

L'accès aux LEDs au travers de l'interface `/sys/class/leds` configurée par `devicetree` se fait en déclarant la volonté d'accéder à une certaine broche, déclaration qui se traduira par la création d'un répertoire contenant des pseudo-fichiers qui donnent accès aux fonctionnalités du noyau pour piloter ces broches.

Nous observons la configuration de MIO0 et MIO7 comme interface led8 et led9 respectivement dans l'extrait de devicetree mis à disposition dans `redpitaya/board/redpitaya/zynq-red_pitaya.dts` du `BR2_EXTERNAL` définissant les périphériques de la Red Pitaya.

```
gpio-leds {
 compatible = "gpio-leds";
 led-8-yellow {
  label = "led8";
  gpios = <&gpio0 0 0>;
  default-state = "off";
  linux,default-trigger = "mmc0";
 };
 led-9-red {
  label = "led9";
  gpios = <&gpio0 7 0>;
  default-state = "off";
  linux,default-trigger = "heartbeat";
 };
};
```

Le *devicetree* sera notre interface privilégiée pour communiquer une configuration spécifique de plateforme au noyau Linux, sur lequel nous reviendrons en détail ultérieurement, en particulier pour la configuration de la partie programmable du SoC (PL).

5 Accès au matériel depuis un programme C

L'accès aux ressources matérielles depuis le shell est certes fonctionnel, mais est 1/ lent car passe à travers toutes les couches d'abstraction entre l'espace utilisateur et le matériel en passant par le noyau et 2/ suppose que quelqu'un avant nous a implémenté le support pour ce matériel. Nous allons ici voir comment accéder au matériel sans passer par le noyau – une méthode efficace de prototypage rapide qui nous permet de revenir aux connaissances acquises sur microcontrôleurs sans système d'exploitation mais qui enfreint les précepts de développement sous GNU/Linux – qui nous sera utile pour qualifier les latences introduites par le système d'exploitation.

`buildroot` nous a fourni un environnement cohérent de développement, et en particulier un compilateur dans `output/host/usr/bin` qui se nomme `arm-buildroot-linux-uclibcgnueabihf-gcc`. La documentation technique qui décrit les registres du Zynq est disponible à `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`. Comme d'habitude, nous y trouvons l'adresse de tous les registres utiles aux accès au matériel ainsi que la description des fonctionnalités. Comme d'habitude, une documentation technique aussi touffue cache la subtilité qui fait la différence entre un programme fonctionnel et un programme qui échoue. Ici la surprise tient, comme dans le STM32, sur la **nécessité d'activer l'horloge qui cadence le périphérique GPIO** : dans le cas contraire, toute lecture ou écriture dans un registre associé à ce périphérique se solde toujours par un résultat nul (0x00). La solution est fournie, pour le MIO, par `https://forums.xilinx.com/t5/Embedded-Linux/Zynq-mmap-GPIO/td-p/368601` qui explique quelle horloge activer (bit 22 du registre APER_CLK_CTRL). Par ailleurs, on se rappellera qu'un pointeur de type `unsigned int *h;` se voit incrémenté par pas de 4 octets, donc `h+4` pointe sur l'adresse de `h` décalée de **16 octets** et non de 4 octets comme nous aurions pu nous y attendre. Mieux vaut donc définir un `unsigned char *h;` qui nous évite toute surprise lors des incréments d'adresses.

L'exemple ci-dessous permet de faire clignoter les deux LEDs connectées aux GPIO liées au processeur – MIO.

```
1 // arm-buildroot-linux-uclibcgnueabihf-gcc -Wall -o led led.c
2
3 #include <stdio.h>
4 #include <fcntl.h>    // O_*
```

```c
#include <unistd.h>   // close
#include <stdlib.h>   // atoi
#include <stdint.h>   // uint32_t
#include <sys/mman.h> // mmap

//#define TYPE int  // si on definit h comme int*, alors l'increment est en multiple de 4 !
#define TYPE char

int main(int argc, char** argv)
{ const int base_addr1= 0xF8000000;
  const int base_addr2= 0xe000a000;
  TYPE *h    = NULL; // int *h = NULL;
  int map_file       = 0;
  unsigned int page_addr,page_offset;
  unsigned page_size=sysconf(_SC_PAGESIZE);

  uint32_t value= 0x80; // 1 ou 128 pour LED orange ou rouge
  if (argc >= 2) value = atoi(argv[1]);
  printf("value = 0x%x\n",value);

  page_addr  = base_addr1 & (~(page_size-1));
  page_offset= base_addr1-page_addr;
  printf("page=0x%x size=0x%x offset=0x%x\n",page_addr,page_size,page_offset);

  // mmap the device into memory
  map_file = open("/dev/mem", O_RDWR | O_SYNC);
  h=mmap(NULL, page_size ,PROT_READ|PROT_WRITE,MAP_SHARED, map_file, page_addr);
  if (h<0) {printf("mmap pointer: %x\n",(int)(h));return(-1);}

  // exemple baremetal pour debloquer acces SLCR registers
  // https://forums.xilinx.com/xlnx/attachments/xlnx/EDK/29780/1/helloworld.c
  printf("@      -> %x %x \n",(unsigned int)h,(unsigned int)(h+0x0c));
  printf("init   -> %x \n",(*(unsigned int*)(h+0x0c/sizeof(TYPE))));
  *(unsigned int*)(h+0x04/sizeof(TYPE))=0x767b; // lock
  printf("lock   -> %x\n",(*(unsigned int*)(h+0x0c/sizeof(TYPE))));
  *(unsigned int*)(h+0x08/sizeof(TYPE))=0xDF0D; // unlock
  printf("unlock -> %x\n",(*(unsigned int*)(h+0x0c/sizeof(TYPE))));

  // IL FAUT L'HORLOGE DE GPIO ! bit 22 de 0xF8000000+0x0000012C, sinon meme lecture echoue (0x00)
  // https://forums.xilinx.com/t5/Embedded-Linux/Zynq-mmap-GPIO/td-p/368601
  printf("ck -> %x\n",(*(unsigned int*)(h+0x12c/sizeof(TYPE))));
  *(unsigned int*)(h+0x12c/sizeof(TYPE))|=(1<<22);
  printf("ck -> %x\n",(*(unsigned int*)(h+0x12c/sizeof(TYPE))));

  page_addr = (base_addr2 & (~(page_size-1)));
  page_offset = base_addr2 - page_addr;
  printf("%x %x\n",page_addr,page_size);
  h=mmap(NULL, page_size ,PROT_READ|PROT_WRITE,MAP_SHARED, map_file, page_addr);
  if (h<0) {printf("mmap pointer: %x\n",(int)(h));return(-1);}
  //write_reg(0xE000A000, 0x00000000, 0x7C020000); //MIO pin 9 value update
  //write_reg(0xE000A000, 0x00000204, 0x200);  //set direction of MIO9
  //write_reg(0xE000A000, 0x00000208, 0x200);  //output enable of MIO9
  //write_reg(0xE000A000, 0x00000040, 0x00000000); //output 0 on MIO9
  //data = read_reg(0xE000A000, 0x00000060);       //read data on MIO

  // https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf p.1347
  *(unsigned int*)(h+page_offset+0)=(value);     // GPIO programming sequence :p.386
  printf("hk: %x\n",*(unsigned int*)(h+page_offset));
  *(unsigned int*)(h+page_offset+0x204/sizeof(TYPE))=value;  // direction
  printf("hk+204: %x\n",*(unsigned int*)(h+page_offset+(0x204)/sizeof(TYPE)));
  *(unsigned int*)(h+page_offset+0x208/sizeof(TYPE))=value;  // output enable
  printf("hk+208: %x\n",*(unsigned int*)(h+page_offset+(0x208)/sizeof(TYPE)));
  *(unsigned int*)(h+page_offset+0x040/sizeof(TYPE))=value;  // output value
  printf("hk+40: %x\n",*(unsigned int*)(h+page_offset+0x40)/sizeof(TYPE));
  close(map_file);
  return 0;
}
```

Cet exemple nous servira à de maintes reprises, qu'il s'agisse de la qualification des latences de Xenomai ou l'accès aux registres implémentés dans la logique reconfigurable du Zynq (partie PL).

Ces accès directs à la mémoire sont possibles depuis l'espace utilisateur – sans reprogrammer en C une application chargée d'ouvrir `/dev/mem` et d'effectuer la conversion entre adresses de la mémoire réelle en adresse de la mémoire virtuelle (`mmap`) – grâce à l'utilitaire `devmem`[8]. Ainsi par exemple pour débloquer l'accès aux registres de configuration

8. sources disponibles dans `buildroot` dans `output/build/busybox-1.30.1/miscutils/devmem.c`

du Zynq, nous pouvons tester

```
redpitaya> devmem 0xf8000004 32 0x767b   # password to lock
redpitaya> devmem 0xf800000c             # check status: 1=locked
0x00000001
redpitaya> devmem 0xf8000008 32 0xdf0d   # password to unlock
redpitaya> devmem 0xf800000c             # check status: 0=unlocked
0x00000000
```

5.1 Reconfiguration de la fonction d'une broche – gestion des interruptions matérielles

La Red Pitaya ne propose que deux GPIO commandés par le PS – les MIO – auxquelles sont connectées des LED. Ces broches ne sont donc pas exploitables en entrée ou pour des fonctions autres qu'indicateur visuel. D'autres broches MIO sont disponibles, mais configurées par défaut (fichier `ps7_init.c` généré par Vivado ou fourni avec le noyau Linux pour le démarrage du système) dans des fonctions auxiliaires (SPI, UART).

Nous désirons avoir accès à un de ces GPIO pour illustrer la gestion d'une interruption matérielle. La fonction de GPIO doit donc être restaurée sur, par exemple, la broche SPI qu'est MIO10, broche 3 du connecteur E2.

La documentation technique de Xilinx est très claires : la fonction d'une broche est déterminée par 3 bits que sont L0_SEL, L1_SEL et L2_SEL. Il y a autant de registre de configuration, et donc de tels bits, que de broches à configurer. Nous identifions l'adresse du registre associé à MIO10, constatons que par défaut il est bien en mode SPI, et le reconfigurons en mode GPIO :

```
redpitaya> cd /sys/class/gpio/
redpitaya> echo "916" > export
redpitaya> echo "out" > direction
redpitaya> devmem 0xF800012C # GPIO clock active
0x00500444
redpitaya> devmem 0xF8000728 # A = SPI
0x000016A0
redpitaya> devmem 0xF8000728 32 0x00001600
redpitaya> devmem 0xF8000728 # 0= GPIO
0x00001600
redpitaya> echo "1" > value
redpitaya> echo "0" > value
```

Register (slcr) MIO_PIN_10

Name	MIO_PIN_10
Relative Address	0x00000728
Absolute Address	0xF8000728
Width	32 bits
Access Type	rw
Reset Value	0x00001601
Description	MIO Pin 10 Control

Register MIO_PIN_10 Details

Field Name	Bits	Type	Reset Value	Description
reserved	31:14	rw	0x0	reserved
DisableRcvr	13	rw	0x0	Operates the same as MIO_PIN_00[DisableRcvr]
PULLUP	12	rw	0x1	Operates the same as MIO_PIN_00[PULLUP]
IO_Type	11:9	rw	0x3	Operates the same as MIO_PIN_00[IO_Type]
Speed	8	rw	0x0	Operates the same as MIO_PIN_00[Speed]

Field Name	Bits	Type	Reset Value	Description
L3_SEL	7:5	rw	0x0	Level 3 Mux Select 000: GPIO 10 (bank 0), Input/Output 001: CAN 0 Rx, Input 010: I2C 0 Serial Clock, Input/Ouput 011: PJTAG TDI, Input 100: SDIO 1 IO Bit 0, Input/Output 101: SPI 1 MOSI, Input/Output 110: reserved 111: UART 0 RxD, Input
L2_SEL	4:3	rw	0x0	Level 2 Mux Select 00: Level 3 Mux 01: SRAM/NOR Data Bit 7, Input/Output 10: NAND Flash IO Bit 5, Input/Output 11: SDIO 0 Power Control, Output
L1_SEL	2	rw	0x0	Level 1 Mux Select 0: Level 2 Mux 1: Trace Port Data Bit 2, Output
L0_SEL	1	rw	0x0	Level 0 Mux Select 0: Level 1 Mux 1: Quad SPI 1 IO Bit 0, Input/Output
TRI_ENABLE	0	rw	0x1	Operates the same as MIO_PIN_00[TRI_ENABLE]

À l'issue de ces manipulations, la broche MIO10 (EP2 broche 3) est bien passée en GPIO, tel que le démontre le programme ci-dessous qui exploite le *timer* du noyau pour faire alterner l'état de la broche. On y notera en particulier que l'argument fourni à `gpio-lib` pour configurer le GPIO est le même que l'argument fourni à `/sys/class/gpio`, à savoir 906+ numéro du GPIO (ici 10) :

```
1 #include <linux/module.h>        /* Needed by all modules */
2 #include <linux/kernel.h>        /* Needed for KERN_INFO */
3 #include <linux/init.h>          /* Needed for the macros */
4 #include <linux/gpio.h>
5
6 struct timer_list exp_timer;
7
```

```
8  // int jmf_gpio=906+0; // 7 en conflit avec heartbeat => virer leds_gpio et led_class (sinon +0)
9  int jmf_gpio=906+10; // ou ('port'-'A')*32+pin dans devicetree
10 volatile int jmf_stat=0;
11
12 static void do_something(unsigned long data)
13 {printk(KERN_INFO "plop");
14  jmf_stat=1-jmf_stat;
15  gpio_set_value(jmf_gpio,jmf_stat);
16  mod_timer(&exp_timer, jiffies + HZ);
17 }
18
19 int hello_start(void);
20 void hello_end(void);
21
22 int hello_start()  // init_module(void)
23 {int delay = 1,err;
24
25  printk(KERN_INFO "Hello\n");
26  err=gpio_is_valid(jmf_gpio);
27  printk(KERN_INFO "err: %d\n",err);
28  err=gpio_request_one(jmf_gpio, GPIOF_OUT_INIT_LOW, "jmf_gpio"); // voir dans gpio.h
29  printk(KERN_INFO "err: %d\n",err);
30
31  // init_timer_on_stack(&exp_timer);
32  // exp_timer.function = do_something;
33  // exp_timer.data = 0;
34  timer_setup(&exp_timer,do_something,0);
35  exp_timer.expires = jiffies + delay * HZ; // HZ specifies number of clock ticks generated per →
        ↪second
36  add_timer(&exp_timer);
37
38  return 0;
39 }
40
41 void hello_end() // cleanup_module(void)
42 {printk(KERN_INFO "Goodbye\n");
43  gpio_free(jmf_gpio);
44  del_timer(&exp_timer);
45 }
46
47 module_init(hello_start);
48 module_exit(hello_end);
49
50 MODULE_LICENSE("GPL");  // NECESSAIRE pour exporter les symboles du noyau linux !
```

De la même façon, cette broche permet de déclencher un évènement (interruption) sur une transition de niveau. Une interruption est gérée par l'exemple de module noyau qui suit

```
1  #include <linux/module.h>       /* Needed by all modules */
2  #include <linux/kernel.h>       /* Needed for KERN_INFO */
3  #include <linux/init.h>         /* Needed for the macros */
4
5  #include <linux/interrupt.h>
6  #include <linux/irq.h>
7  #include <linux/gpio.h>
8
9  static int dummy, irq, jmf_gpio, dev_id;
10
11 void hello_end(void); // cleanup_module(void)
12 int hello_start(void); // cleanup_module(void)
13
14 static irqreturn_t irq_handler(int irq, void *dev_id)
15 {
16         dummy++;
17         printk(KERN_INFO "plip %d",dummy);
18   return IRQ_HANDLED; // etait IRQ_NONE
19 }
20
21 int hello_start()  // init_module(void)
22 {int err;
23
24  printk(KERN_INFO "Hello\n");
25  jmf_gpio =906+10;  // PB2 : version devicetree ; 15 en version script.fex
26  err=gpio_is_valid(jmf_gpio);
27  err=gpio_request_one(jmf_gpio, GPIOF_IN, "jmf_irq");
28  if (err!=-22)
```

```
    {printk(KERN_ALERT "gpio_request %d=%d\n",jmf_gpio,err);
     irq = gpio_to_irq(jmf_gpio);
     printk(KERN_ALERT "gpio_to_irq=%d\n",irq);
     irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
     err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO jmf", &dev_id);
     printk(KERN_ALERT "finished IRQ: error=%d\n",err);
     dummy=0;
    }
 return 0;
}

void hello_end() // cleanup_module(void)
{printk(KERN_INFO "Goodbye\n");
 free_irq(irq,&dev_id);
 gpio_free(jmf_gpio);  // libere la GPIO pour la prochaine fois
}

module_init(hello_start);
module_exit(hello_end);

MODULE_LICENSE("GPL");  // NECESSAIRE pour exporter les symboles du noyau linux !
```

pour ensuite être optimisé avec une gestion de l'interruption en espace noyau visant à communiquer un signal aux processus qui se seraient enregistrés auprès du service :

```
#include <linux/module.h>        /* Needed by all modules */
#include <linux/kernel.h>        /* Needed for KERN_INFO */
#include <linux/init.h>          /* Needed for the macros */
#include <linux/fs.h>            // define fops
#include <linux/uaccess.h>
#include <linux/version.h>

#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/sched/signal.h>        // send_sig_info
#ifdef __ARMEL__
#include <linux/gpio.h>
//#include <linux/signal.h>            // do_send_sig_info
#else
#endif

int hello_start(void);
void hello_end(void);

int pid = 0;
#ifdef __ARMEL__
static int dummy,gpio,id;
static int irq;

static irqreturn_t irq_handler(int irq, void *dev_id)
#else
struct timer_list exp_timer;

static void irq_handler(struct timer_list *t)
#endif
{
#if LINUX_VERSION_CODE < KERNEL_VERSION(5,0,0)
 struct siginfo sinfo; // siginfo
#else
 struct kernel_siginfo sinfo; // siginfo
#endif
 struct task_struct *task;
// alternative 'a send_sig_info :
// struct pid *mypid;
// mypid= find_vpid(pid);
// if (mypid == NULL) pr_info("Cannot find PID from user program\r\n");
//     else kill_pid(mypid, SIGUSR1, 1);
#if LINUX_VERSION_CODE < KERNEL_VERSION(5,0,0)
 memset(&sinfo, 0, sizeof(struct siginfo));  // on cherche PID au cas ou' le process aurait →
     ↪disparu
#else
 memset(&sinfo, 0, sizeof(struct kernel_siginfo));  // on cherche PID au cas ou' le process aurait→
     ↪ disparu
#endif
 sinfo.si_signo = SIGUSR1;                    // depuis son enregistrement
 sinfo.si_code = SI_USER;
```

```
50  task = pid_task(find_vpid(pid), PIDTYPE_PID);
51  if (task == NULL) pr_info("Cannot find PID from user program\r\n");
52     else send_sig_info(SIGUSR1, &sinfo, task);
53 #ifdef __ARMEL__
54  dummy++;
55  printk(KERN_INFO "plip %d",dummy);
56  return IRQ_HANDLED;
57 #else
58  printk(KERN_INFO "plip %ld",jiffies);
59  mod_timer(t, jiffies + HZ);
60 #endif
61 }
62
63 static int dev_open(struct inode *inod,struct file *fil);
64 static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off);
65 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off);
66 static int dev_rls(struct inode *inod,struct file *fil);
67
68 static struct file_operations fops=
69 {.read=dev_read,
70  .open=dev_open,
71  .write=dev_write,
72  .release=dev_rls,};
73
74 int hello_start()  // init_module(void)
75 {int t=register_chrdev(91,"jmf",&fops);  // major = 91
76 #ifdef __ARMEL__
77  int err;
78 #endif
79
80  if (t<0) printk(KERN_ALERT "registration failed\n");
81     else printk(KERN_ALERT "registration success\n");
82  printk(KERN_INFO "Hello\n");
83
84 #ifdef __ARMEL__
85  gpio =906+10;  // PB2 : version devicetree ; 15 en version script.fex
86  err=gpio_is_valid(gpio);
87  err=gpio_request_one(gpio, GPIOF_IN, "jmf_irq");
88  if (err!=-22)
89     {printk(KERN_ALERT "gpio_request %d=%d\n",gpio,err);
90      irq = gpio_to_irq(gpio);
91      printk(KERN_ALERT "gpio_to_irq=%d\n",irq);
92      irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
93      err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO jmf", &id);
94      printk(KERN_ALERT "finished IRQ: error=%d\n",err);
95      dummy=0;
96     }
97 #else
98  timer_setup(&exp_timer,irq_handler,0); // was init_timer_on_stack(&exp_timer); -> replaced since →
        ↪4.14
99  exp_timer.expires = jiffies + HZ; // HZ specifies number of clock ticks generated per second
100 add_timer(&exp_timer);
101 #endif
102  return t;
103 }
104
105 void hello_end() // cleanup_module(void)
106 {printk(KERN_INFO "Goodbye\n");
107 #ifdef __ARMEL__
108  free_irq(irq,&id);
109  gpio_free(gpio);  // libere la GPIO pour la prochaine fois
110 #else
111  del_timer(&exp_timer);
112 #endif
113  unregister_chrdev(91,"jmf");
114 }
115
116 static int dev_rls(struct inode *inod,struct file *fil)
117 {printk(KERN_ALERT "bye\n");
118  return 0;
119 }
120
121 static int dev_open(struct inode *inod,struct file *fil)
122 {printk(KERN_ALERT "open\n");
123  return 0;
```

```
124 }
125
126 static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off)
127 {char buf[15]="Hello read\n\0";
128  int readPos=0;
129  printk(KERN_ALERT "read\n");
130  while (len && (buf[readPos]!=0))
131    {put_user(buf[readPos],buff++);
132     readPos++;
133     len--;
134    }
135  return readPos;
136 }
137
138 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off)
139 {int mylen;
140  char buf[15];
141
142  printk(KERN_ALERT "write ");
143  if (len>14) mylen=14; else mylen=len;
144  if (copy_from_user(buf, buff, mylen) == 0)
145     sscanf(buf, "%d", &pid);
146
147  printk(KERN_ALERT "PID registered: %d",pid);
148  return len;
149 }
150
151 module_init(hello_start);
152 module_exit(hello_end);
153
154 MODULE_LICENSE("GPL");  // NECESSAIRE pour exporter les symboles du noyau linux !
```

en vue de communiquer l'évènement en espace utilisateur :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <unistd.h>
8  #include <fcntl.h>
9
10 void signal_handler(int signum)
11 {
12     if (signum == SIGUSR1)
13         printf("J'ai eu un signal\r\n");
14     return;
15 }
16
17 int main()
18 {FILE *f;
19  printf("Penser a mknod /dev/jmf c 91 0 avant de lancer ce programme\n");
20     signal(SIGUSR1, signal_handler);
21     printf("My PID is %d.\n", getpid());
22     f=fopen("/dev/jmf","w");
23     fprintf(f,"%d\n", getpid()); fflush(f);
24     fclose(f);
25     printf("send: %d\n", getpid());
26     while (1) {};
27     return 0;
28 }
```

6 Programmation noyau : les modules et communication avec l'espace utilisateur

Notre objectif dans cet exposé est de maîtriser la mise en œuvre des interfaces de communication entre l'utilisateur et le matériel au travers du noyau Linux. Bien que nous ayons vu que divers niveaux d'abstraction (`script.bin` ou `devicetree.dtb`) vise à cacher la description du matériel au développeur, nous allons ici nous placer du point de vue de l'électronicien qui connaît son architecture matérielle et désire y accéder au travers de Linux pour en fournir les fonctionnalités à l'utilisateur.

6.1 Pourquoi travailler au niveau du noyau ?

Dans la hiérarchie des niveaux d'abstraction permettant à un programme en espace utilisateur d'atteindre la matériel, le noyau fait office de gestionnaire des ressources. Il s'assure ainsi que chaque tâche a le droit de s'exécuter (ordonnanceur – *scheduler*), que les ressources sont attribuées de façon cohérente, et que les messages venant du matériel sont remontés à tous les programmes utilisateurs qui en ont besoin.

Un module [3] est un bout de code qui vient se connecter au noyau afin d'en compléter les fonctionnalités. Initialement introduit avec le noyau 1.2 de Linux (Mars 1995), il évite de devoir recompiler tout le noyau pour y ajouter une fonctionnalité telle que le support d'un nouveau périphérique matériel. Il est souvent possible d'accéder aux ressources matérielles depuis l'espace utilisateur, mais ce faisant nous perdons les garanties de cohérence d'accès aux ressources proposées par le noyau (interdiction de charger deux fois un même module noyau par exemple). Certaines fonctionnalités (DMA, interruptions) ne sont simplement pas accessibles en dehors de l'espace noyau.

Depuis le noyau linux, la sélection des modules s'obtient en marquant par `M` les éléments accessibles par `make menuconfig`. La compilation des modules s'achève par `make modules` suivi de `make modules_install` qui placera le résultat dans `/var/lib/modules`.

Depuis buildroot, la configuration s'obtient par `make linux-menuconfig` et `make` de buildroot compilera à la fois le noyau et ses modules, pour les placer sur l'images qui sera flashée sur carte SD.

Tout développeur d'interfaces matérielles se doit de comprendre le fonctionnement d'un module noyau pour fournir le support logiciel nécessaire aux utilisateurs qui ne désirent par connaître les subtilités de l'accès au matériel.

Deux grandes classes de modules noyau sont chargés soit de transférer des blocs de données (par exemple vers un disque dur), soit des octets individuels [9] (caractères). Nous nous intéresserons au second cas, plus simple, et surtout moins difficile d'accès au niveau du matériel nécessaire à la démonstration.

Toutes les communications se font au travers de pseudo-fichiers situés dans le répertoire `/dev`. Un fichier peut être ouvert, fermé, et nous pouvons y écrire ou lire des données. Une méthode additionnelle qui n'a pas son équivalent dans un accès aux fichiers est une configuration des transactions : `ioctl` (Input/Output Control).

6.2 Structure et compilation d'un module noyau

Le minimum nécessaire pour attacher un module au noyau Linux est une fonction d'initialisation et une fonction pour libérer les ressources. L'exemple ci-dessous permet de se familiariser d'une part avec l'arborescence du noyau Linux, et d'autre part avec la méthode de compilation au travers d'un `Makefile` qui fait appel aux sources du noyau exécuté sur la plateforme cible.

Un module noyau minimaliste est proposé dans le listing ci-desssous, qui nous permettra de nous familiariser avec sa structure, sa compilation, et son utilisation.

```
1 #include <linux/module.h>        /* Needed by all modules */
2 #include <linux/kernel.h>        /* Needed for KERN_INFO */
3 #include <linux/init.h>          /* Needed for the macros */
4
5 static int __init hello_start(void){printk(KERN_INFO "Hello\n");return 0;}
6 static void __exit hello_end(void) {printk(KERN_INFO "Goodbye\n");}
7
8 module_init(hello_start);
9 module_exit(hello_end);
```

Listing 1 – Premier exemple de module noyau

Le module noyau contient nécessairement un point d'entrée et un point de sortie. Lorsque le modulé est lié au noyau (`insmod mon_noyau.ko`), la méthode `init()` est appelée. Lorsque le module est retiré du noyau (`rmmod mon_noyau`), la méthode `exit` est appelée. Ces méthodes sont des macros vers les fonctions `init_module()` et `cleanup_module()` (voir `linux/init.h` dans les sources d'un noyau Linux).

Alors que historiquement l'objet `.o` était directement lié au noyau, depuis le noyau 2.6 Linux a besoin d'informations supplémentaires pour se lier à un *kernel object* qui différencie le `.o` du `.ko`. Bien que le Makefile déclare la compilation du `.o`, c'est bien le fichier `.ko` qui est chargé par `insmod` tel que décrit à `http://tldp.org/HOWTO/Module-HOWTO/linuxversions.html` : "In Linux 2.6, the kernel does the linking. A user space program passes the contents of the ELF object file directly to the kernel. For this to work, the ELF object image must contain additional information. To identify this particular kind of ELF object file, we name the file with suffix ".ko" ("kernel object") instead of ".o" For example, the serial device driver that in Linux 2.4 lived in the file serial.o in Linux 2.6 lives in the file serial.ko."

9. `https://appusajeev.wordpress.com/2011/06/18/writing-a-linux-character-device-driver/`

Un module est forcément lié à une version particulière d'un noyau, et devra être recompilé lors des mises à jour de ce dernier. Buildroot fournit un environnement cohérent de développement contenant la *toolchain* de cross-compilation et les sources du noyau, que nous trouverons dans `output/build/linux-`*version* de buildroot. Dans notre cas, le noyau linux est obtenu par `git` et contient donc, comme extension, l'identifiant de hashage.

Dans un premier temps, nous nous assurons que le compilateur est dans le chemin de recherche des exécutables :

```
export PATH=$PATH:$HOME/.../buildroot/output/host/usr/bin/
```

puis nous compilons le module – supposé être nommé `mymod.c`[10] pour devenir `mymod.ko` – au moyen du Makefile suivant :

```
1 obj-m += mymod.o
2
3 all:
4   make ARCH=arm CROSS_COMPILE=arm-buildroot-linux-uclibcgnueabihf- -C \
5   /home/jmfriedt/buildroot/output/build/linux-[...] M=$(PWD) modules
6
7 clean:
8   rm mymod.o mymod.ko
```

Ce `Makefile` s'analyse de la façon suivante :

1. il fait lui même appel à un autre Makefile puisqu'exécute la commande `make`
2. cet autre Makefile se trouve dans les sources du noyau sur lequel nous nous lions selon l'option `-C répertoire` qui indique dans quel répertoire aller chercher le fichier de configuration,
3. nous appelons la méthodes `modules` car nous pourrions résumer la commande par `make modules` avec un certain nombre d'autre options ...
4. ... parmi lesquelles le fait de cross-compiler pour une cible ARM, donc `ARCH` et le préfixe du compilation `CROSS_COMPILE`.
5. Enfin, la compilation d'un module nécessite de préciser où se trouvent les sources de ce module : comme ici c'est le répertoire courant, nous indiquons `M=$(PWD)` (ici).
6. Si nous compilons le même module pour le PC (architecture hôte), éliminons `ARCH` et `CROSS_COMPILE` pour ne garder que `make -C kernel_directory M=$(PWD) modules` (voir ci-dessous)

Un module noyau se compile non-seulement pour la plateforme embarquée, mais aussi potentiellement pour l'hôte. Pour connaître la version du noyau qui tourne sur le PC, `uname -a`. S'assurer de la disponibilité des sources du noyau ou au moins des entêtes associés, par exemple par le paquet `linux-headers-amd64` sous Debian GNU/Linux à la date de rédaction de ce document. Sur PC, le Makefile sera de la forme

```
obj-m += t.o
all:
	make -C /usr/src/linux-headers-4.1.0-2-686-pae M=$(PWD) modules
```

Nous observons ici la puissance de `buildroot` qui fournit un environnement cohérent de travail avec une chaîne de compilation (`CROSS_COMPILE=arm-buildroot-linux-uclibcgnueabihf-`) et une arborescence des sources du noyau (`../buildroot-2018.08.1/output/build/linux-4.4.78`) qui est sollicitée lors de la compilation du module. Le principe du `Makefile` est d'ajouter le nom du nouveau module en cours de développement (`mymod`) à la liste des modules du noyau, et lancer la compilation du noyau par `make`. Les autres objets du noyau étant déjà compilés, seul notre module sera ainsi généré.

⚠ **Attention** : en cas d'utilisation d'une carte comportant un noyau autre que celui de son buildroot, il faut synchroniser sa configuration du noyau avec celle de la carte. Pour ce faire, on pourra récuperer le fichier de configuration de Linux dans `/proc/configs.gz`, le placer dans le fichier `.config` du noyau utilisé par buildroot, effectuer un `make linux-menuconfig` pour informer buildroot de la mise à jour, et finalement `make` de buildroot pour refabriquer l'arborescence du noyau. Une fois ces tâches effectuées, le module que nous compilerons sera compatible avec le noyau exécuté sur la carte.

Le module est lié au noyau par `insmod mymod.ko` et retiré par `rmmod mymod`. La liste des modules liés au noyau est affichée par `lsmod`. Les messages du noyau sont soit affichés sur la console (dans le cas de la Red Pitaya, sur le port série) ou dans les *logs* du noyau visualisés par `dmesg` ou `cat /var/log/messages` (ou parfois `/var/log/syslog`).

Le résultat de la compilation est transféré vers la carte Red Pitaya (NFS) pour chargement par `insmod mymod.ko`. Nous en validons le bon fonctionnement en observant les messages fournis par le noyau dans ses logs au moyen de `dmesg` :

10. on se gardera de nommer ce module `kernel.c` : dans ce cas, la compilation se concluera bien, mais par la génération d'un objet qui n'est pas issu de notre code source mais d'un fichier présent dans l'arborescence du noyau !

```
# insmod mymod.ko
# dmesg | tail -1
[  249.879087] Hello
# rmmod mymod.ko
# dmesg | tail -1
[  256.571319] Goodbye
```

Ces informations sont reproduites dans `/var/log/messages` que l'on pourra consulter continuellement par `tail -f`.

L'exemple est extrait des sources du noyau, et accessible à `modules/example/example.c`.

En cas d'échec de la compilation du module, il est fondamental de se remémorer qu'un module noyau est compilé pour une configuration d'un noyau. En cas de perte de cette configuration, il est toujours possible de la retrouver au moyen de `scripts/extract-ikconfig` dont la sortie sera sauvegardée dans un fichier `.config` permettant de régénérer un environnement de travail identique à celui proposé par le noyau en cours d'exécution.

Les capacités de communication de notre module avec le systeme sont cependant restreintes : nous allons les étendre dans le contexte d'un périphérique de caractères (*char device*).

6.3 Communication au travers de `/dev` : interaction avec l'utilisateur

Un module qui se contente de se charger et de se décharger ne présente que peu d'intérêt. Afin d'interagir avec l'utilisateur, nous devons fournir des méthodes de type lecture et écriture. Par convention, les pseudo fichiers permettant le lien entre un module noyau et un programme utilisateur se trouvent dans `/dev`. Ces fichiers se comportent comme des tuyaux reliant les deux espaces (noyau et utilisateur) dans lesquels circulent des caractères (opposés à des blocs de données dans le cas qui nous intéresse ici). La création d'un pseudo-fichier se fait par l'administrateur (`sudo` sur PC) au moyen de la commande `mknod /dev/jmf c 90 0` avec les arguments qui indiquent qu'il s'agit d'un *character device* (`c`) d'identifiant majeur 90 (nature du périphérique) et d'identifiant mineur 0 (indice dans la liste de ce périphérique). Sur PC, on ajouter `-m 777` pour créer le nœud de communication avec les permissions de lecture et d'écriture pour tout utilisateur : `mknod /dev/jmf c 90 0 -m 777`.

 En cas de message de ressource occupée lors de l'insertion du module (erreur -16 – EBUSY), il se peut que le nombre majeur 90 soit déjà occupé : même s'il n'apparaît pas dans la liste de `/dev`, la liste des périphériques dans `/proc/devices` indiquera tous les major numbers occupés. Par exemple sur Red Pitaya, le major number 90 est occupé par le pilote `mtd`. Sur PC, le major 99 est occupé par `ppdev`.

La structure de données qui définit quelle fonction est appelée lors de l'ouverture et la fermeture du tuyau (`open` et `close` en espace utilisateur) ainsi que l'écriture ou la lecture de données dans le tuyau (`write` et `read` respectivement depuis l'espace utilisateur) est nommée `file_operations`.

Dans cet exemple, une écriture dans `/dev/jmf` se traduit par un message dans les logs du noyau contenant la chaîne de caractères écrite.

```
1  // mknod /dev/jmf c 90 0
2  #include <linux/module.h>       /* Needed by all modules */
3  #include <linux/kernel.h>       /* Needed for KERN_INFO */
4  #include <linux/init.h>         /* Needed for the macros */
5  #include <linux/fs.h>           // define fops
6  #include <linux/uaccess.h>
7
8  static int dev_open(struct inode *inod,struct file *fil);
9  static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off);
10 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off);
11 static int dev_rls(struct inode *inod,struct file *fil);
12
13 char buf[15]="Hello read\n\0";
14
15 int hello_start(void); // declaration pour eviter les warnings
16 void hello_end(void);
17
18 static struct file_operations fops=
19 {.read=dev_read,
20  .open=dev_open,
21  .write=dev_write,
22  .release=dev_rls,
23 };
24
```

```
25 int hello_start()  // init_module(void)
26 {int t=register_chrdev(90,"jmf",&fops);  // major = 90
27  if (t<0) printk(KERN_ALERT "registration failed\n");
28     else printk(KERN_ALERT "registration success\n");
29  printk(KERN_INFO "Hello\n");
30  return t;
31 }
32
33 void hello_end() // cleanup_module(void)
34 {printk(KERN_INFO "Goodbye\n");
35  unregister_chrdev(90,"jmf");
36 }
37
38 static int dev_rls(struct inode *inod,struct file *fil)
39 {printk(KERN_ALERT "bye\n");return 0;}
40
41 static int dev_open(struct inode *inod,struct file *fil)
42 {printk(KERN_ALERT "open\n");return 0;}
43
44 static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off)
45 {int readPos=0;
46  printk(KERN_ALERT "read\n");
47  while (len && (buf[readPos]!=0))
48    {put_user(buf[readPos],buff++);
49     readPos++;
50     len--;
51    }
52  return readPos;
53 }
54
55 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off)
56 {int l=0,mylen;
57  printk(KERN_ALERT "write ");
58  if (len>14) mylen=14; else mylen=len;
59  for (l=0;l<mylen;l++) get_user(buf[l],buff+l);
60  // Essayer avec vvv : kernel panic car acces en userspace
61  // for (l=0;l<mylen;l++) buf[l]=buff[l];
62  buf[mylen]=0;
63  printk(KERN_ALERT "%s \n",buf);
64  return len;
65 }
66
67 module_init(hello_start);
68 module_exit(hello_end);
```

nous solicitons ce pilote par

```
# echo "Hello" > /dev/jmf
[  258.408111] open
[  258.414953] write
[  258.416874] Hello
[  258.416874]
[  258.420911] bye
# dd if=/dev/jmf bs=10 count=1
[  282.774521] open
[  282.776557] read
Hello[  282.778454] bye

0+1 records in
0+1 records out
```

```
# echo "toto" > /dev/jmf [  290.922228] open

[  290.924908] write
[  290.926933] toto
[  290.926933]
[  290.930904] bye
# dd if=/dev/jmf bs=10 count=1
[  293.484720] open
[  293.486761] read
toto[  293.488659] bye

0+1 records in
0+1 records out
```

Cet exemple contient une modification au module proposé ci-dessus afin que le message écrit soit celui lu auparavant : implémenter cette modification.

Ce module implémente les communications entre espace utilisateur et espace noyau au travers du pseudo-fichier `/dev/jmf`. Un fichier dans le répertoire `/dev` n'est pas un vrai fichier mais simplement une passerelle permettant de faire transiter des données. Sa création s'obtient par `mknod /dev/jmf c 90 0` avec 90 le l'identifiant (nombre) majeur et 0 l'identifiant mineur de ce pseudo-fichier. L'identifiant majeur reste constant pour les pseudo-fichiers liés à un même type de périphérique, tandis que l'identifiant mineur est incrémenté pour chaque nouvelle instance de ce périphérique. Nous vérifions que le périphérique de communication est bien reconnu par le noyau en consultant `/proc/devices` qui contient l'entrée `jmf` avec le major number que nous lui avons associé (90, qui est libre comme nous le vérifions par `ls`

`-l /dev/`). Cependant, sur nombre de plateformes embarquées, le répertoire `/dev/` est régénéré à chaque lancement (*reboot*) : il faudra donc prendre soin de re-créer l'entrée `/dev/jmf` à chaque réinitialisation de la carte.

Une alternative à la définition explicite du point de communication dans `/dev` et le choix manuel du major number associé est d'utiliser le *misc device*. Pour ce faire, on définit en variable globale

```
struct miscdevice jmfdev;
```

puis dans la fonction d'init :

```
jmfdev.name = "jmf";  // /dev/jmf  de major pris dans la classe misc
jmfdev.minor = MISC_DYNAMIC_MINOR;
jmfdev.fops = &fops;
jmfdev.mode = 0666; // permissions pour user
misc_register(&jmfdev);  // creation dynamique de l'entree /dev/jmf
```

qui se conclut dans la fonction exit par

```
misc_deregister(&jmfdev);
```

6.4 La méthode `ioctl`

Une méthode fort peu élégante – notamment parce qu'elle brise le concept de "tout est fichier" d'unix, est l'ioctl. Cette méthode permet de compléter les fonctions qui ne sont pas accessibles par `read` et `write`, et est notamment utilisée pour la configuration d'un périphérique (par exemple, régler la vitesse d'échantillonnage d'une carte son).

On complètera par exemple le listing précédent par les fonctions

```
// static int dev_ioctl(struct inode *inod,struct file *fil, unsigned int cmd,
//       unsigned long arg); // pre-2.6.35
static long dev_ioctl(struct file *f, unsigned int cmd, unsigned long arg); // actuel

static struct file_operations fops=
{.read=dev_read,
 .open=dev_open,
 .unlocked_ioctl=dev_ioctl,
 .write=dev_write,
 .release=dev_rls,};

// static int dev_ioctl(struct inode *inod,struct file *fil, unsigned int cmd,
//    unsigned long arg) // pre-2.6.35
static long dev_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{switch ( cmd )
 {case 0: printk(KERN_ALERT "ioctl0");break;
  case 1: printk(KERN_ALERT "ioctl1");break;
  default:printk(KERN_ALERT "unknown ioctl");break;
 }
 return 0;
}
```

L'exécution du programme suivant – nous ne connaissons en effet aucune façon d'accéder aux méthodes `ioctl()` en shell sans passer par un programme en C – que nous nommerons `ioct`

```
#include <fcntl.h>      /* open */
#include <unistd.h>     /* exit */
#include <sys/ioctl.h>  /* ioctl */
#include <stdio.h>
#include <stdlib.h>

#define IOCTL_SET_MSG 0
#define IOCTL_GET_MSG 1
#define DEVICE_FILE_NAME "/dev/jmf"

ioctl_set_msg(int file_desc, char *message)
{ int ret_val;
  ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);
  printf("set_msg message: %s\n", message);
}

ioctl_get_msg(int file_desc,char *message)
{ int ret_val;
  ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);
  printf("get_msg message: %s\n", message);
}

main(int argc,char **argv)
{ int file_desc, ret_val;
```

```
25   char msg[30] = "Message passed by ioctl\n";
26
27   file_desc = open("/dev/jmf", 0);
28   printf("%d\n",file_desc);
29   if (argc>1) sprintf(msg,argv[1]);
30   msg[4]=0;
31   ioctl_set_msg(file_desc,msg);
32
33   sprintf(msg,"Hello World");
34   ioctl_get_msg(file_desc,msg);
35   close(file_desc);
36 }
```

se traduit par la séquence suivante :

```
# lsmod
Module                  Size  Used by
g_ether                44895  0
# insmod mymod.ko
# ./ioctl
# tail /var/log/messages
Jan  1 00:11:45 buildroot kern.alert kernel: [  705.740703] registration success
Jan  1 00:11:45 buildroot kern.info kernel: [  705.745730] Hello
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.805798] open
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.808491] ioctl1
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.818059] ioctl0
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.820838] bye
```

D'un point de vue utilisateur, la fonction `ioctl()` s'appelle exactement comme `read()` ou `write()` mais prend 3 arguments : le descripteur de fichier, le numéro de l'ioctl (en cohérence avec la déclaration dans le noyau, ici 0 ou 1) et éventuellement un argument. La page 2 du manuel de `ioctl` est explicite sur ces points. Depuis le noyau, le passage d'argument [11] vers l'espace utilisateur s'obtient au moyen de `put_user()` pour une valeur scalaire ou `copy_to_user` pour un tableau (zone mémoire).

6.5 Passer de l'adressage virtuel à l'adressage physique

Bien que de nombreux systèmes embarqués ne s'encombrent pas de gestionnaire matériel de mémoire (*Memory Management Unit* – MMU), un système d'exploitation multitâche tel que Linux en fait pleinement usage et justifie donc l'utilisation d'un microcontrôleur suffisamment puissant pour être muni d'un tel périphérique. L'organisation de la mémoire, gérée par la MMU, n'est en principe pas du ressort du développeur, *sauf* lorsqu'il veut accéder à un emplacement physique connu, tel que par exemple les registres de configuration du matériel. Dans ce cas, il nous faut effectuer la convertion inverse, de l'espace d'adressage virtuel vers l'espace d'adressage réel. Au niveau de l'espace utilisateur, cette fonctionnalité est fournie par `mmap()`. En espace noyau, il s'agit de `ioremap()`.

L'électronicien ne connaît que la notion d'adresse physique d'un périphérique : il s'agit de l'adresse associée à chaque registre dans la documentation du microcontrôleur. L'informaticien ne connaît que la notion d'adresse virtuelle, telle que manipulée par le système d'exploitation qui tourne au dessus de la MMU. Nous devons faire le lien entre ces deux mondes pour les faire communiquer.

L'instruction [3, chap.9] qui, au niveau du noyau, fait ce lien est `void *ioremap(unsigned long phys_addr, unsigned long size);` Nous allons illustrer son utilisation en complétant les méthodes appelées lors de l'insertion du module pour allumer une LED, et éteidre la LED lorsque le module est retiré de la mémoire. Cet exemple est purement académique puisqu'en pratique, le module `linux-*/drivers/gpio/gpio-xilinx.c` implémente ces fonctionnalités, et bien d'autres, pour cacher les subtilités du matériel au développeur.

Les ressources qui nous aident à comprendre l'organisation des registres de configuration des GPIOs sont décrites à `http://redpitaya.readthedocs.io/en/latest/developerGuide/125-14/extent.html` et la documentation incontournable – à défaut d'être digeste – `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-`
`pdf`. L'adressage des registres liés aux GPIO est décrit dans le chapitre 14 du manuel de l'utilisateur.

Nous complétons le module noyau que nous avons rédigé jusqu'ici par

11. `http://www.makelinux.net/ldd3/chp-6-sect-1`

```
static void __iomem *jmf_gpio;   //int jmf_gpio;
#define IO_BASE2 0xe000a000

int hello_start()  // init_module(void)
{int delay = 1;
 unsigned int stat;
[... gestion horloge GPIO ...]

 if (request_mem_region(IO_BASE2,0x2E4,"GPIO test")==NULL)
     printk(KERN_ALERT "mem request failed");
 jmf_gpio = (void __iomem*)ioremap(IO_BASE2, 0x2E4);
 writel(1<<mio,jmf_gpio+0x204);
 writel(1<<mio,jmf_gpio+0x208);
 writel(1<<mio,jmf_gpio+0x40);

 return 0;
}

void hello_end() // cleanup_module(void)
{printk(KERN_INFO "Goodbye\n");
 release_mem_region(IO_BASE2, 0x2e4);
}
```

en pensant à exploiter les fichiers d'entête `linux/io.h` et `linux/ioport.h`. Ce code ne peut se comprendre qu'en consultant [4, p.1347] reproduit ci-dessous, avec la description des divers registres associés aux GPIO, et [4, p.1580] pour la gestion des horloges.

6.6 Ajouter un *timer* périodique

Le noyau fournit un accès à des *timers* pour cadencer des tâches. Nous nous servons de telles ressources pour faire clignoter la diode périodiquement et afficher un message dans les *logs* du noyau.

Afin d'indiquer que la carte Red Pitaya est en vie, nous désirons représenter le battement de son cœur par une diode qui clignote périodiquement. Le noyau Linux fournit une méthode sous forme de *timer* [3, chap.7].

```
struct timer_list exp_timer;

static void do_something(struct timer_list *t)
{printk(KERN_ALERT "plop");
 mod_timer(&exp_timer, jiffies + HZ);
}

int hello_start()  // init_module(void)
{...
 // init_timer_on_stack(&exp_timer);
 // exp_timer.function = do_something;
 // exp_timer.data = 0;
 timer_setup(&exp_timer,do_something,0);
 exp_timer.expires = jiffies + delay * HZ; // HZ specifies number of clock ticks/second
 add_timer(&exp_timer);
...
}

void hello_end() // cleanup_module(void)
{...
 del_timer(&exp_timer);
}
```

Nous constatons dans `dmesg | tail` que le timer est bien appelé périodiquement, toutes les secondes.

```
[ 5697.089643] registration success
[ 5697.093780] Hello
[ 5698.094083] plop
[ 5699.094146] plop
[ 5700.094227] plop
[ 5701.094286] plop
[ 5702.094348] plop
```

6.7 Sémaphore, mutex et spinlock [5]

Un pilote qui fournit une méthode `read` communique continuellement des informations au processus en espace utilisateur qui les requiert (`cat < /dev/pilote`). Cette condition de fonctionnement n'est pas représentative d'une application pratique, dans laquelle un producteur met un certain temps à générer les données qui seront communiquées à l'espace utilisateur : la méthode `read` se comporte alors comme le consommateur de données. Dans cette architecture de producteur-consommateur, il faut d'une part un mécanisme pour bloquer `read` tant que les données n'ont pas été produites, et d'autre part un mécanisme pour garantir la cohérence des accès à la zone mémoire commune aux deux fonctions de production et de consommation. La première fonctionnalité est fournie par le sémaphore, la seconde par le mutex et son implémentation plus rapide (mais plus gourmande en ressources), le spinlock.

6.7.1 Sémaphore

Le sémaphore se comporte ici comme un compteur qui mémorise le nombre de fois que les données sont produites. Un consommateur qui cherche à obtenir une donnée décrémente le sémaphore : si le sémaphore est déjà à 0, le consommateur (ici la méthode `read`) est bloquée jusqu'à ce qu'un producteur ait incrémenté le sémaphore. Un sémaphore incrémenté plusieurs fois permettra au consommateur d'obtenir plusieurs données.

La définition des constantes et prototypes associés aux sémaphores se résument en

```
1 #include <linux/semaphore.h>
2 struct semaphore mysem;
3 sema_init(&mysem, 0);        // init the semaphore as empty
4 down (&mysem);
5 up (&mysem);
```

Simuler la production périodique de données en incrémentant un sémaphore dans le gestionnaire d'un *timer*. Bloquer la méthode `read` tant que les données n'ont pas été produites.

Que se passe-t-il lorsque nous effectuons une lecture sur le périphérique accessible par `/dev/` alors que le sémaphore s'est déjà incrémenté plusieurs fois ?

Parmi les 5 fonctions au cœur d'un pilote communiquant – init, exit, open, read, release – laquelle permet de ne produire des données que lorsque l'utilisateur en a besoin ? Proposer un mécanisme pour ne produire des informations (incrément du sémaphore) que lorsque l'utilisateur en a besoin.

6.7.2 Mutex et spinlock

Les données produites sont stockées dans une mémoire tampon servant à échanger les informations entre producteur et consommateur. Un mécanisme doit être mis en œuvre pour garantir la cohérence entre les données produites et lues : il ne faut pas écraser une mémoire tampon en cours de lecture, et ne pas chercher à lire des données en cours de production. Le mutex (*MUTually EXclusive*) fournit un vérou binaire – bloqué ou débloqué – qui interdit à deux processus d'accéder simultanément à la même zone mémoire. Avant d'accéder au tampon, chaque processus (producteur ou consommateur) bloque le mutex, interdisant l'autre processus qui chercherait lui même à bloquer le mutex de continuer son exécution. Une fois l'opération – lecture ou écriture – achevée, le mutex est débloqué et l'exécution du processus en attente se poursuit.

La méthode des mutex permet de placer la tâche en mode veille jusqu'à ce que le mutex soit débloqué. Passer la tâche en mode veille libère des ressources du processeur, mais nécessite un changement de contexte qui peut s'avérer lourd et long si les opérations de lecture et écriture nécessitent une faible lattence. Une alternative est le spinlock, qui effectue les mêmes opérations mais en maintenant la tâche active et en testant continuellement l'état du vérou : dans ce cas, le processeur est continuellement sollicité, mais la lattence est moindre et le changement de contexte ne s'opère plus.

La définition des constantes et prototypes associés aux mutex se résument en

```
1 #include <linux/mutex.h>
2 struct mutex mymutex;
3 mutex_init(&mymutex);
4 mutex_lock(&mymutex);
5 mutex_unlock(&mymutex);
```

La définition des constantes et prototypes associés aux spinlocks se résument en

```
1 #include <linux/spinlock.h>
2 static DEFINE_SPINLOCK(myspin);
3 spin_lock_init(&myspin);
4 spin_lock(&myspin);
5 spin_unlock(&myspin);
```

Proposer une implémentation par mutex garantissant la cohérence entre écriture et lecture. Démontrer en modifiant le tampon contenant le texte affiché lors de la requête à `read`, en y insérant le décompte d'un compteur qui s'incrémente à chaque appel du timer simulant la production de données.

Au lieu de modifier le contenu du tableau de caractères contenant le message à afficher dans le gestionnaire de *timer*, créer une tâche qui sera ordonnancée lorsque le *scheduler* en aura le temps : on placera les opérations coûteuses en ressources (mutex, manipulation du tampon) dans cette tâche.

6.8 `sysfs`

L'interaction entre le module et l'utilisateur n'est plus prise en charge par `/dev` mais désormais par un `sysfs` dans l'exemple 6.8 : une entrée dans `/sys` est dynamiquement ajoutée lors du chargement du module. Ainsi, un `platform_device` initialise un point d'entrée dans `/sys`, ici nommé `gpio-simple`. Le chargement du module se traduit par l'initialisation de la plateforme dont le premier argument fournit le nom du pilote (driver) associé : ce même mot clé doit être celui fourni dans la structure `.name` du pilote qui sera initialisé par la méthode `probe` :

```
1  #include <linux/module.h>
2  #include <linux/gpio.h>
3  #include <linux/platform_device.h>
4  #include <linux/err.h>
5
6  static struct platform_device *pdev;
7
8  static int gpio_simple_probe(struct platform_device *pdev)
9  {
10   return 0;
11 }
12
13 static int gpio_simple_remove(struct platform_device *pdev)
14 {
15   return 0;
16 }
17
18 static struct platform_driver gpio_simple_driver = {
19         .probe          = gpio_simple_probe,
20         .remove         = gpio_simple_remove,
21         .driver = {
22                 .name   = "gpio-simple",
23         },
24 };
25
26 static int __init gpio_simple_init(void)
27 { int ret;
28   ret = platform_driver_register(&gpio_simple_driver);
29   if (ret) return ret;
30   pdev = platform_device_register_simple("gpio-simple", 0, NULL, 0);
31   if (IS_ERR(pdev))
32     {platform_driver_unregister(&gpio_simple_driver);
33      return PTR_ERR(pdev);
34     }
35  return 0;
36 }
37
38 static void __exit gpio_simple_exit(void)
39 {
40   platform_device_unregister(pdev);
41   platform_driver_unregister(&gpio_simple_driver);
42 }
43
44 module_init(gpio_simple_init)
45 module_exit(gpio_simple_exit)
46
47 MODULE_DESCRIPTION("GPIO simple driver");
48 MODULE_LICENSE("GPL");
```

Ajouter l'affichage de messages dans les méthodes `probe`, `init`, `exit` et `remove`, et constater leur ordre d'appel, en particulier de `probe` lors de la création du platform_device.

Nous ajoutons ensuite les points de communication avec l'espace utilisateur, qui sera accessible au travers de `/sys/bus/platform/drivers/gpio-simple`

Pour ce faire, une structure de données contenant les points de communication est initialisée au moyen de la macro `DEVICE_ATTR`.

```
static ssize_t
gpio_simple_show(struct device *dev, struct device_attribute *attr, char *buf)
{
        return sprintf(buf, "%d\n", gpio_get_value(gpio));
}

static DEVICE_ATTR(value, 0444, gpio_simple_show, NULL);

static int gpio_simple_probe(struct platform_device *pdev)
{
  err = device_create_file(&pdev->dev, &dev_attr_value);
  if (err < 0)
    goto err_free_irq;

  return 0;
  [...]
}

static int gpio_simple_remove(struct platform_device *pdev)
{
  device_remove_file(&pdev->dev, &dev_attr_value);
  [...]
}
```

Le code devient plus complexe à lire si on ne prend pas pas soin d'analyser `.../buildroot/output/build/linux-headers-XXX/include/linux/device.h` et d'y constater que la macro est définie selon

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
        struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
```

Ainsi, la macro crée dans notre cas une structure `dev_attr_value` (`value` étant le nom fourni en premier argument) qui est fournie en paramètre à `device_create_file()`.

Finalement, nous ajoutons la gestion du timer et le déclenchement de l'évènement associé par :

```
static struct work_struct work;

static void gpio_simple_notify(struct work_struct *ws)
{pr_info("IRQ %d triggered on GPIO %d, value=%d\n",
}

static void do_something(unsigned long data)
{printk(KERN_INFO "plop: %lu",jiffies);
 schedule_work(&work);
}

static int gpio_simple_probe(struct platform_device *pdev)
{[...]
 INIT_WORK(&work, gpio_simple_notify);
}

static int gpio_simple_remove(struct platform_device *pdev)
{device_remove_file(&pdev->dev, &dev_attr_value);
 cancel_work_sync(&work);
  [...]
}
```

puis la communication par le point d'entrée `value` dans le système de fichier associé dans `/sys` à notre pilote. Dans cet exemple, un mécanisme de déclenchement de la tâche est initié lorsqu'une interruption se déclenche. Cet évènement servira ensuite à débloquer la lecture d'un processus en utilisateur lorsque l'interruption se déclenchera.

Démontrer le déblocage d'un fichier en lecture dans le `sysfs` par le déclenchement du timer et le lancement de la tasklet chargée de débloquer le sémaphore.

7 Compiler une image Linux avec support Xenomai (extension-temps réel)

Xenomai a été porté au processeur Zynq [12]. Les modifications amenées à un noyau de la branche officielle ont été adaptées aux noyau `linux-xlnx` et incluses sous forme de patch dans la procédure de compilation de Xenomai par buildroot.

Il **absolument effacer** les patchs inclus par défaut dans buildroot pour Xenomai 3.0.5 : ces patchs entrent en conflit avec nos propres modifications pour le noyaux `linux-xlnx`. Pour ce faire : dans le répertoire de buildroot, `rm packages/xenomai/*.patch`.

Le lecteur désireux de modifier un noyau officiel pourra suivre la procédure proposée par `https://embeddedgreg.com/2017/07/16/getting-started-with-xenomai-3-on-zynq/comment-page-1/`, en sachant que cette utilisation du noyau officiel ne supportera pas l'accès au FPGA depuis GNU/Linux. Cette procédure n'est **pas** à mettre en œuvre avec notre utilisation de buildroot.

La configuration de buildroot qui active le support Xenomai se nomme `redpitaya_xenomai_defconfig`. Ainsi, dans le répertoire `buildroot`, la commande `make redpitaya_xenomai_defconfig` charge cette configuration, qui est mise en œuvre par `make`. Comme auparavant, l'image résultante est placée sur la carte SD par la commande `dd` adéquate.

À l'issue de cette compilation, le noyau doit se charger au démarrage de Linux avec de nouveaux messages indiquant que Xenomai et ses outils associés ont été insérés :

```
sched_clock: 64 bits at 333MHz, resolution 3ns, wraps every 4398046511103ns
I-pipe, 333.333 MHz clocksource, wrap in 12884 ms
```

et

```
hw perfevents: enabled with armv7_cortex_a9 PMU driver, 7 counters available
[Xenomai] scheduling class idle registered.
[Xenomai] scheduling class rt registered.
I-pipe: head domain Xenomai registered.
[Xenomai] Cobalt v3.0.5 (Sisyphus's Boulder)
workingset: timestamp_bits=30 max_order=17 bucket_order=0
```

Afin de qualifier les latences du système temps-réel, nous utiliserons l'outil classique de qualification de Xenomai `latency` par

```
redpitaya> latency
== Sampling period: 1000 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|     -2.798|     -2.435|      2.714|       0|     0|     -2.798|      2.714
RTD|     -2.811|     -2.374|      7.931|       0|     0|     -2.811|      7.931
RTD|     -2.380|     -1.888|      6.692|       0|     0|     -2.811|      7.931
RTD|     -2.806|     -2.393|      6.939|       0|     0|     -2.811|      7.931
RTD|     -2.836|     -2.397|      5.680|       0|     0|     -2.836|      7.931
RTD|     -2.870|     -2.358|      7.046|       0|     0|     -2.870|      7.931
RTD|     -2.883|     -2.388|      8.137|       0|     0|     -2.883|      8.137
```

Les valeurs négatives – aberrantes – sont documentées à `https://embeddedgreg.com/2017/07/16/getting-started-with-comment-page-1/` et nous trouvons bien sur Red Pitaya le pseudo-système de fichiers `/proc/xenomai`, ainsi que l'outil `autotune` dans `/tmp/xeno_zynq/usr/xenomai/sbin` qui ajuste les paramètres du système [13], et permet finalement d'avoir un résultat convenable :

12. `https://session.wikispaces.net/57898/auth/auth?authToken=0eeca7316e2df30bb5b70760aa5847fe8` et en particulier J.H. Brown & B. Martin, *How fast is fast enough ? Choosing between Xenomai and Linux for real-time applications*, Proc. 12th Real-Time Linux Workshop (RTLWS'12) (2012)
13. `autotune --period 10000`

```
redpitaya> latency
== Sampling period: 1000 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      0.993|     10.388|     26.247|       0|     0|      0.993|     26.247
RTD|      0.980|      8.411|     29.984|       0|     0|      0.980|     29.984
RTD|      1.137|      9.887|     25.449|       0|     0|      0.980|     29.984
RTD|      1.180|     11.129|     26.790|       0|     0|      0.980|     29.984
RTD|      1.152|     10.576|     28.543|       0|     0|      0.980|     29.984
RTD|      1.055|     10.865|     27.385|       0|     0|      0.980|     29.984
RTD|      1.063|     10.913|     28.225|       0|     0|      0.980|     29.984
```

L'outil `stress` permet de charger le système et de constater que les latences restent stables malgré l'occupation des ressources par ce programme.

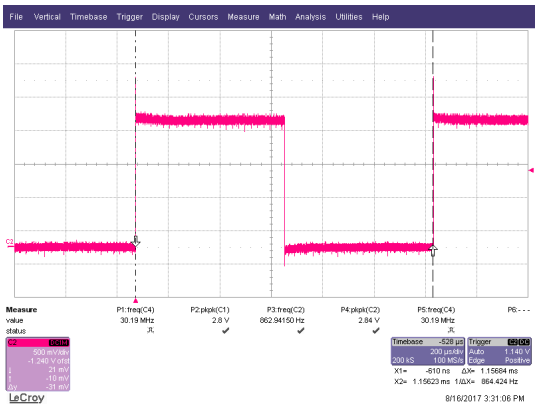
- vérifier que les options temps réel sont actives, et en particulier les applications associées (outils de test – *testsuite* – et `rt-tests`). Pour ce faire, `Target packages` → `Real-Time` → `Xenomai Userspace` → `Install testsuite`. Par ailleurs, sélectionner la version de Xenomai manuellemment en remplissant le champ `Custom Xenomai version` par `3.0.5`,
- il peut être judicieux d'installer quelques fonctionnalités additionnelles accessibles depuis le shell, par exemple `screen` dans `Target packages` → `Shell and utilities` → `screen` ou `Target packages` → `System tools` → `cpuload`.

Les latences de l'option temps-réel [6] se qualifient par `echo "0" > /proc/xenomai/latency` suivi de `latency -p 100`. On observera la variation des latences quand, dans un second terminal, le processeur est chargé par `while ( true ) ; do echo toto;done`. Cette dernière commande se lance dans un second shell accessible par `screen` (CTRL-A c pour créer une nouvelle session du shell, CTRL-A a pour changer de session).

8 Qualification des latences

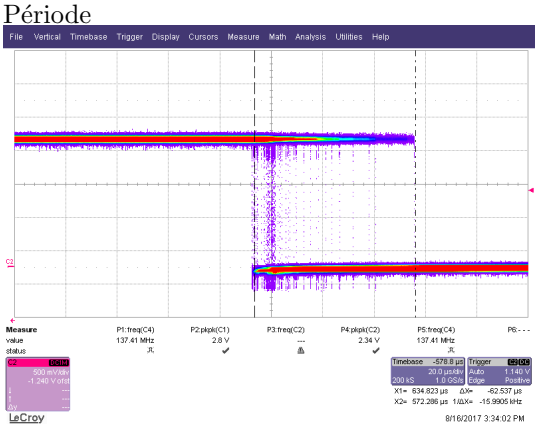
Les quatre cas ci-dessous considèrent les attentes entre deux commutations d'une LED soit par fonction `sleep` (délai prédéfini), soit par un appel à une fonction par *timer*. Hors Xenomai, l'implémentation de cette méthode par signaux donne un résultat catastrophique dès que le processeur est chargé. Dans les autres cas, la qualification des latences se fait en déclenchant un oscilloscope numérique sur le front montant des créneaux et en observant la date de chute du créneau. Si les attentes respectaient parfaitement le chronogramme prévu, les fronts descendants devraient se superposer. L'intervalle de temps dans lequel ces fronts sont observés permet de qualifier la résistance à la charge et le respect de latences maximum dans les diverses conditions analysées. Le cas des modules noyau n'est pas abordé.

8.1 Attente par `sleep`, sans Xenomai

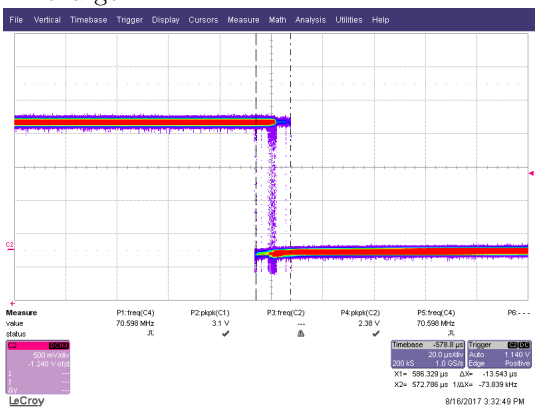


Période

```c
#include <stdlib.h>
#include <stdio.h>
#include "ledlib.h"

int main()
{int status=0;
 unsigned char *h;

 h=red_gpio_init();
 red_gpio_set_cfgpin(h,0); // LED0

 while (1) {
    red_gpio_output(h,0,status); usleep(TIMESLEEP);
//     printf("status=%x\n",status);
    status^=0xff;
 }
 red_gpio_cleanup();
 return 0;
}
```



En charge

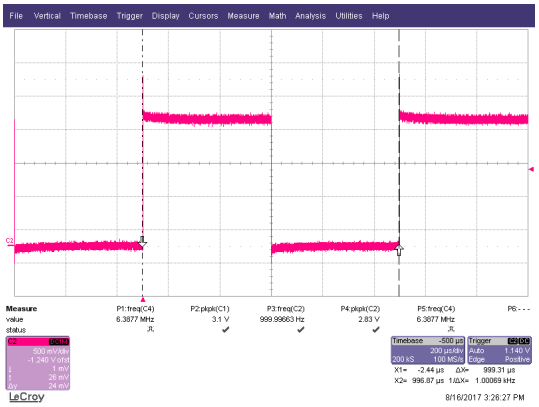


Sans charge

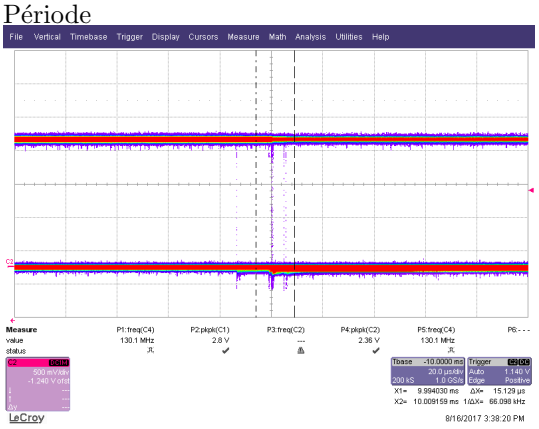
Le programme reprend l'exemple vu plus haut de commutation des diodes, avec une attente de 500 ms. Dans ce cas et tous ceux qui vont suivre, le processeur est chargé par la commande suivante exécutée dans un second terminal connecté à la Red Pitaya : `stress --cpu 8 --io 4 --vm 2 --vm-bytes 128M --timeout 10s` exécuté 5 fois de suite pour une mesure totale qui dure environ une minute.

8.2 Attente par *timer*, sans Xenomai

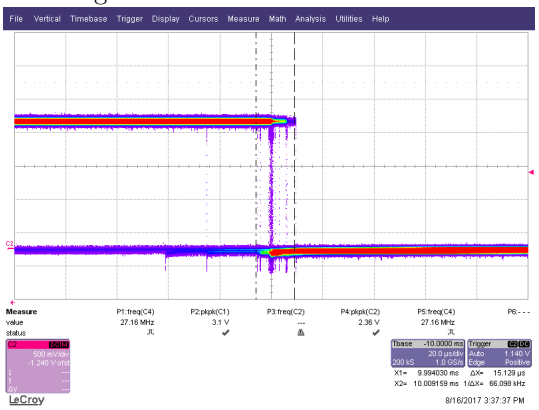
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "ledlib.h"

int status=0; // declenche' a chaque appel du timer => exterieur
unsigned char *h;

void test(int signum) {
    red_gpio_output(h,0,status);
    status^=0xff;
//    printf("%x\n",status);
}

int main()
{struct sigaction sa;
 struct itimerval timer;

 h=red_gpio_init();
 red_gpio_set_cfgpin(h,0); // LED0

 memset(&sa, 0,sizeof(sa));
 sa.sa_handler = &test;
 sigaction(SIGVTALRM,&sa, NULL);
 /* Configure the timer to expire after TIMESLEEP msec ... */
 timer.it_value.tv_sec = 0;
 timer.it_value.tv_usec = TIMESLEEP;
 /* ... and every TIMESLEEP usec after that. */
 timer.it_interval.tv_sec = 0;
 timer.it_interval.tv_usec = TIMESLEEP;
 setitimer(ITIMER_VIRTUAL, &timer, NULL);
 while(1);
 red_gpio_cleanup();
 return 0;
}
```



Période



En charge

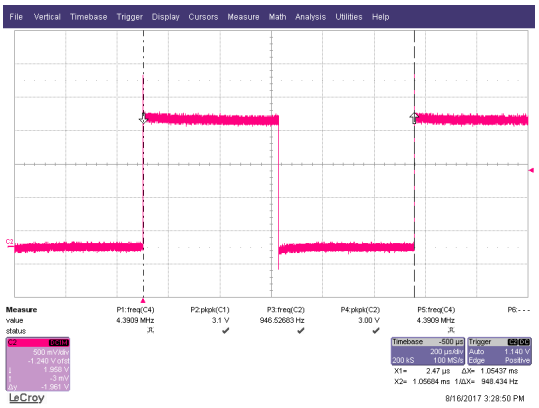


Sans charge

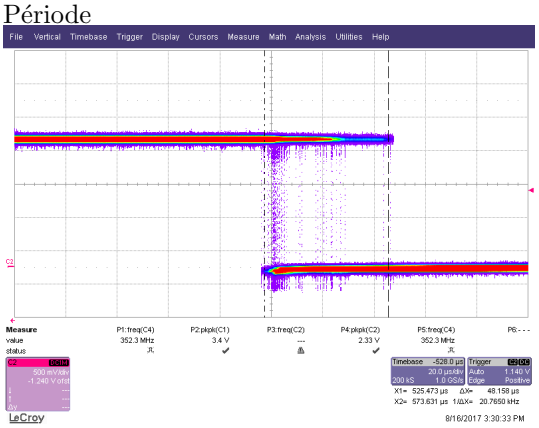
Le résultat est ici catastrophique : non seulement sigalarm n'est pas capable de respecter nos contraintes de période du signal, mais même sans charge des sauts de période sont observés. Cette méthode d'attente ne supporte pas du tout la charge du processeur par ailleurs.

8.3 Attente par `sleep`, avec Xenomai

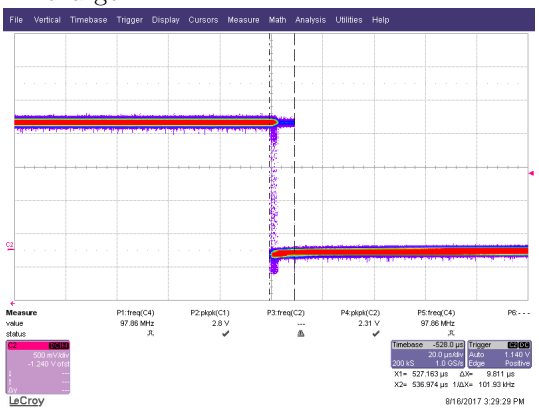
```c
#include <signal.h>
#include <stdio.h>
#include <time.h> // nanosleep
#include <native/task.h>
#include "ledlib.h"

RT_TASK blink_task;
unsigned char *h;

void catch_signal(int sig){}

void blink(void *arg){
 int status=0;
 struct timespec tim = {0,TIMESLEEP*1000};

 rt_printf("blink task\n");
 while(1)
   {red_gpio_output(h,0,status);
    status^=0xff;
    // rt_printf("%d\n",status);
    if (nanosleep(&tim,NULL) != 0)
       {printf("erreur usleep\n");return;}
   }
}

int main(int argc, char *argv[])
{
 h=red_gpio_init();
 red_gpio_set_cfgpin(h,0); // LED0
 // Avoid memory swapping for this program
 mlockall(MCL_CURRENT|MCL_FUTURE);

 signal(SIGTERM,catch_signal);
 signal(SIGINT, catch_signal);

 rt_printf("hello RT world\n");
 rt_task_create(&blink_task, "blinkLed", 0, 99, 0);
 rt_task_start(&blink_task, &blink, NULL);

 pause();

 rt_task_delete(&blink_task);
 red_gpio_cleanup();
 return 0;
}
```



Période



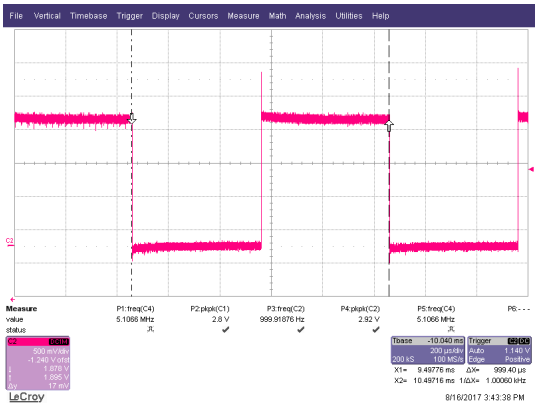
En charge



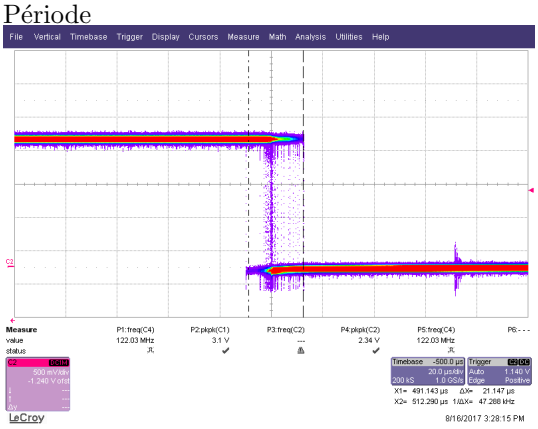
Sans charge

8.4 Attente par *timer*, avec Xenomai

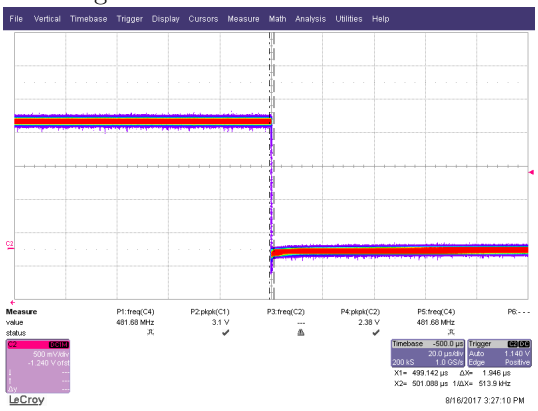
```c
#include <signal.h>
#include <native/task.h>
#include "ledlib.h"

RT_TASK blink_task;
unsigned char *h;

void catch_signal(int sig){}

void blink(void *arg){
 int status=0;

 rt_printf("blink task\n");
 rt_task_set_periodic(NULL, TM_NOW, TIMESLEEP*1000);
 while(1)
   {rt_task_wait_period(NULL);
    red_gpio_output(h,0,status);
    status^=0xff;
   }
}

int main(int argc, char *argv[])
{
 h=red_gpio_init();
 red_gpio_set_cfgpin(h,0); // LED0
 // Avoid memory swapping for this program
 mlockall(MCL_CURRENT|MCL_FUTURE);

 signal(SIGTERM, catch_signal);
 signal(SIGINT, catch_signal);

 rt_printf("hello RT world\n");
 rt_task_create(&blink_task, "blinkLed", 0, 99, 0);
 rt_task_start(&blink_task, &blink, NULL);

 pause();

 rt_task_delete(&blink_task);
 red_gpio_cleanup();
 return 0;
}
```



Période



En charge



Sans charge

9 Configuration du FPGA

Alors qu'un FPGA seul nécessite un environnement de programmation dédié (e.g. sonde JTAG), le Zynq propose d'accéder à la partie programmable du composant (PL) depuis le processeur généralite (PS). Deux méthodes fournissent une telle fonctionnalité : historiquement, Xilinx proposait le pilote `/dev/xdevcfg` dans lequel l'écriture (`cat`) du bitstream se traduisait par la configuration du FPGA. Actuellement, dans un contexte d'homogénéisation des accès aux divers SoC (Systems on Chip) – incluant Altera/Intel et Lattice/Microsemi – une nouvelle infrastructure nommée FPGA Manager est en cours de standardisation. L'intérêt de cette infrastructure est de s'étendre aux *overlays* du *devicetree* qui visent à synchroniser les ressources requises par un bitstream configurant le FPGA, le pilote Linux associé, et le bitstream flashé das le FPGA. De cette façon, le pilote est automatique sollicité lors de la configuration du FPGA avec un bitstream renseigné dans un greffon au devicetree, qui sert d'intermédiaire de communication de configuration avec le noyau Linux.

Avec le noyau que nous proposons, les deux approches sont disponibles. Étant concurrentes et induisant un conflit sur la ressource qu'est le PL, une seule méthode ne peut s'activer à un instant donné. L'ordre de chargement des pilotes

au démarrage fait que la méthode active par défaut est le FPGA Manager. Pour activer `/dev/xdevcfg`, il suffit de retirer tous les modules puis de charger `xilinx_devcfg` qui induira la création de `/dev/xdevcfg` : au démarrage de linux, le système nous informe (`dmesg`) de l'activation de FPGA Manager :

```
FPGA manager framework
fpga_manager fpga0: Xilinx Zynq FPGA Manager registered
fpga-region fpga-full: FPGA Region probed
random: crng init done
```

Nous retirons tous les modules de la mémoire puis chargeons le module `xilinx_devcfg` :

```
rmmod zynq_fpga
rmmod xilinx_devcfg
modprobe xilinx_devcfg
```

qui se traduit par

```
fpga_manager fpga0: fpga_mgr_unregister Xilinx Zynq FPGA Manager
xdevcfg f8007000.devcfg: ioremap 0xf8007000 to dea36000
```

et effectivement nous constatons l'apparition de `/dev/xdevcfg`

```
redpitaya> ls -l /dev/xdevcfg
crw-rw----    1 root     root      240,   0 Jan  1 13:47 /dev/xdevcfg
```

Nous revenons à FPGA Manager par la procédure inverse :

```
redpitaya> rmmod xilinx_devcfg
redpitaya> modprobe zynq_fpga
redpitaya> ls -l /sys/class/fpga_manager/fpga0/
[...]
--w-------    1 root     root          4096 Jan  1 13:52 firmware
```

9.1 Génération du fichier crypté

Vivado génère par défaut un fichier `.bit`. Le pilote s'attend à un autre format contenant un entête particulier. La conversion se fait avec l'utilitaire `bootgen` fourni par le SDK de Vivado.

Cet outil attend un fichier .bif contenant :

```
all:
{
  nom_du_bitstream.bit
}
```

qui sera ensuite fourni à `bootgen` :

```
$VIVADO_SDK/bin/bootgen -image fichier_bif.bif -arch zynq -process_bitstream bin
```

Suite à cette commande un fichier `nom_du_bitstream.bit.bin` est créé dans le répertoire courant.

9.2 Flasher par utilisation directe de `fpga_manager`

Le fichier `.bit.bin` doit être copié/déplacé dans `/lib/firmware`.

Afin d'informer le pilote que le `PL` doit être flashé, et quel bitstream utiliser, la commande suivante est à utiliser :

```
echo "nom_du_bitstream.bit.bin" > /sys/class/fpga_manager/fpga0/firmware
```

La ligne :

```
fpga_manager fpga0: writing nom_du_bitstream.bit.bin to Xilinx Zynq FPGA Manager
```

s'affichera en cas de succès et la LED connectée sur `Prog done` doit s'allumer (LED bleue sur la `RedPitaya`).

9.3 Utilisation d'un overlay pour le `devicetree` pour le flashage

Comme pour l'autre solution, le bitstream doit se trouver dans `/lib/firmware`.

L'intérêt de cette méthode par rapport à la précédente est que lorsque le design contient des IPs qui communiquent avec le processeur, le *devicetree* communique au noyau les informations requises par les pilotes. Généralement dans ce type de cas un pilote est disponible et celui-ci doit être chargé. L'overlay permet, par surcharge et ajout, de fournir, à la fois, le nom du bitstream, et de compléter le devicetree chargé au démarrage avec les pilotes spécifiques à l'application.

Sans rentrer dans tous les détails de ce type de fichier, le code suivant à pour rôle de :

- modifier le nœud `fpga_full`, attribut `target`, correspondant au `fpga_manager` pour lui fournir le nom du binaire à charger au travers de l'attribut `firmware-name` ;
- d'ajouter des sous-nœuds correspondant aux pilotes nécessaires afin de permettre leur chargement. Dans le cas présent le pilote associé est `gpio_ctl` (champ `compatible`), et on fournit l'addresse de base sur la plage mémoire partagée entre le PS et le PL (0x43C00000), la taille de la zone (0x1f) par l'attribut `reg`

```
/dts-v1/;
/plugin/;
/ {
    compatible = "xlnx,zynq-7000";
    fragment@0 {
        target = <&fpga_full>;
        #address-cells = <1>;
        #size-cells = <1>;
        __overlay__ {
            #address-cells = <1>;
            #size-cells = <1>;

            firmware-name = "top_redpitaya_axi_gpio_ctl.bin";

            gpio1: gpio@43C00000 {
                compatible = "gpio_ctl";
                reg = <0x43C00000 0x0001f>;
                gpio-controller;
                #gpio-cells = <1>;
                ngpio= <8>;
            };
        };
    };
};
```

Ce fichier doit ensuite être compilé par la commande :

```
/somewhere/buildroot/output/host/usr/bin/dtc -@ -I dts -O dtb -o ${FILENAME}.dtbo ${FILENAME}.dts
```

avec :

- `-@` pour la génération de symboles qui seront dynamiquement liés lors du chargement dans la mémoire ;
- `-I dts` pour définir le type du fichier d'entrée (DeviceTree Source) ;
- `-O dtb` pour définir le type du fichier de sortie (DeviceTree Binary) ;
- `-o` le nom du fichier généré.

Pour charger le fichier en mémoire deux étapes sont nécessaires :

1. la création d'un répertoire correspondant à notre overlay :
   ```
   mkdir /sys/kernel/config/device-tree/overlays/toto
   ```
   se traduira par la création automatique d'un ensemble de fichiers qui peuplera ce répertoire :
   ```
   redpitaya> ls -l /sys/kernel/config/device-tree/overlays/toto/
   total 0
   -rw-r--r--    1 root     root             0 Jan  1 00:04 dtbo
   -rw-r--r--    1 root     root          4096 Jan  1 00:04 path
   -r--r--r--    1 root     root          4096 Jan  1 00:04 status
   ```
2. le chargement de l'overlay dans le *devicetree* :
   ```
   cat gpio_red.dtbo > /sys/kernel/config/device-tree/overlays/toto/dtbo
   ```
   induira la configuration du FPGA par transfert du bitstream, activation du pilote associé à cet overlay par la méthode "compatible" qui aura été renseignée dans le champ approprié du module.

Il est également possible d'annuler les modifications (revenir à l'état initial) en supprimant, simplement, le répertoire :

```
rmdir /sys/kernel/config/device-tree/overlays/toto
```

10 Interaction PS-PL

10.1 GPIO

Tout l'intérêt du Zynq porte sur la combinaison d'un processeur généraliste (PS) exécutant GNU/Linux, et une matrice de portes logiques reconfigurables (PL). Nous allons démontrer dans l'exemple qui va suivre une synthèse de GPIO qui passe par le PL (et non comme le MIO qui était purement PS auparavant), et comment accéder à ce périphérique par les diverses méthodes que nous avons vues dans ce document :

— accès direct depuis l'espace utilisateur
— accès direct depuis l'espace noyau (module)
— accès depuis un pilote noyau sollicité par le *devicetree* (méthode `compatible`)

La configuration du PL par Vivado dépasse la cadre de cette présentation. Nous utilisons un bloc GPIO fourni par Xilinx comme IP dans Vivado, que nous connectons au PS au travers du bus AXI, standard utilisé sur toute plateforme comprenant un processeur ARM (Fig. 6).

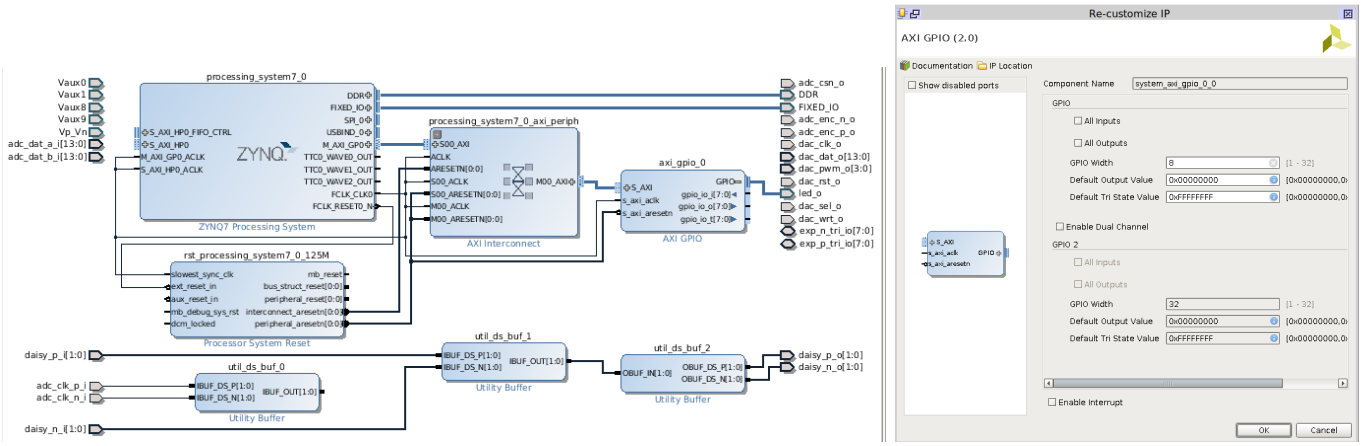


FIGURE 6 – Le PL configuré par un AXI Interconnect pour faire le lien entre le GPIO placé dans le FPGA et le PS.

Comme tout périphérique sur un système numérique, le GPIO est connecté aux trois interfaces du bus AXI que sont le bus de données, le bus de contrôle et le bus d'adresse. La logique du bus de contrôle, relativement complexe, nous est cachée par l'AXI Interconnect, et les seuls points qui nous intéressent sont l'adresse de base des registres de contrôle du GPIO et les divers offsets vers les diverses sous-fonctions. L'adresse de base, susceptible de varier en fonction de la nature des périphériques et leur nombre, est fournie par Vivado (Fig. 7). Les offsets vers les sous-fonctions sont décrites dans la documentation du bloc GPIO fournie par Xilinx [14]. Ainsi, nous savons comment accéder à chaque périphérique du PL depuis le PS, comme nous le ferions classiquement sur un microcontrôleur (Fig. 7).

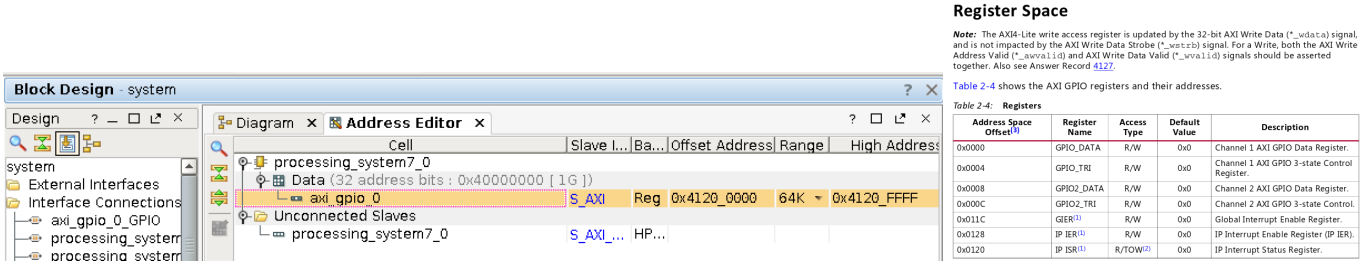


FIGURE 7 – Assignation des adresses des périphériques ajoutés au PL.

10.2 XADC

Le Zynq est équipé d'un convertisseur analogique-numérique lent. Ce périphérique est accessible soit par le PS, soit par le PL. Xilinx fournit un pilote compatible IIO comme il se doit pour ce type de périphérique. En l'absence de configuration du PL, seule la température est lisible de cette façon. Nous nous proposons d'étendre les fonctionnalités du PL en modifiant la configuration du PL en connectant deux broches pour des mesures analogiques.

14. p.10 de `https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf`

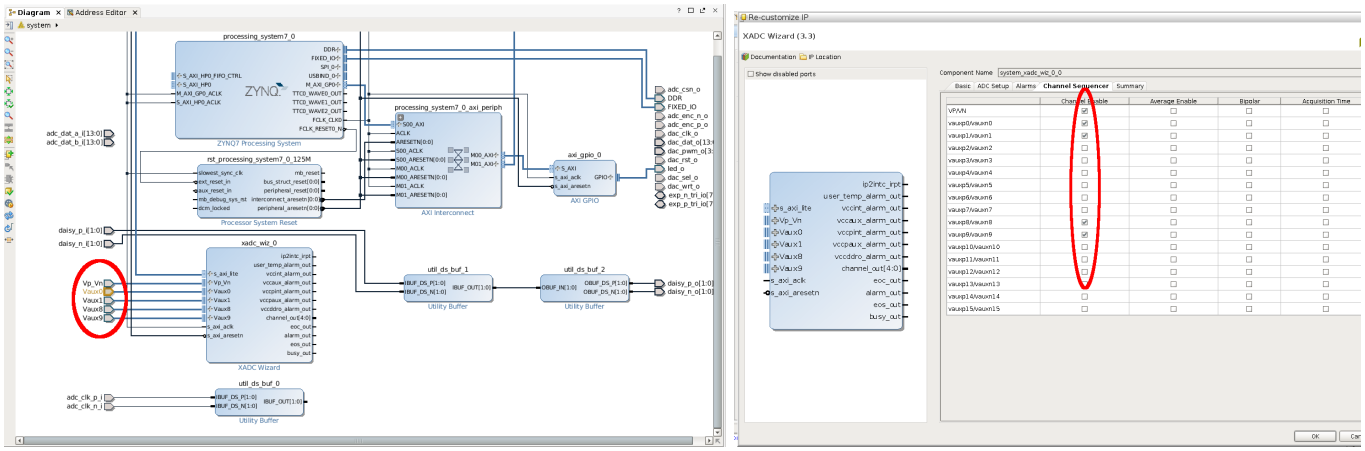


FIGURE 8 – Étapes de configuration du XADC pour connecter les entrées analogiques. Connecter les broches manquantes et lancer l'autoroutage du bus AXI après avoir ajouté le bloc de traitement XADC.

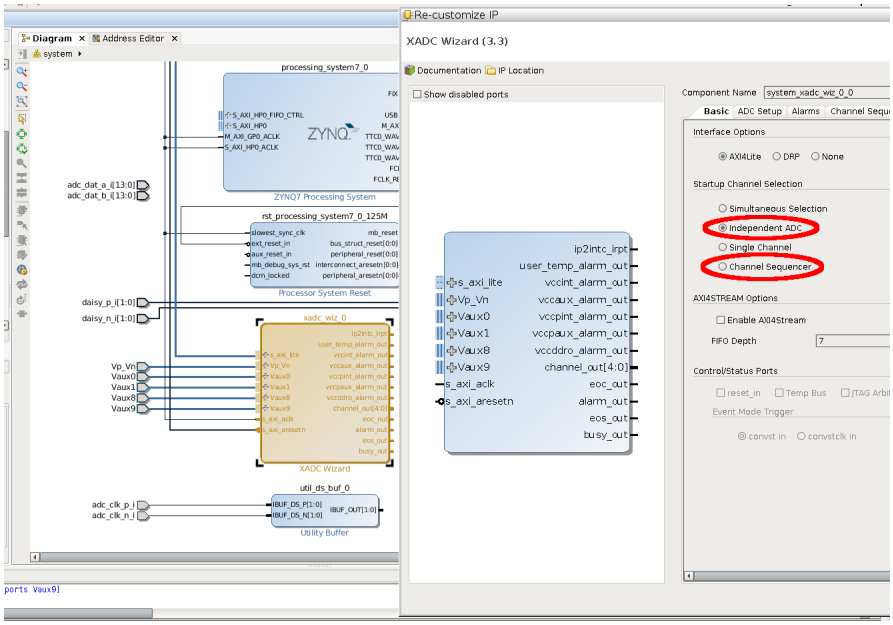


FIGURE 9 – Étapes de configuration du XADC pour connecter les entrées analogiques. Un bug de Vivado impose d'activer le mode Channel Sequencer avant de passer en mode Independent ADC pour pouvoir configurer la liste des canaux actifs.

L'overlay au devicetree qui permettra de communiquer avec cette nouvelle configuration du XADC est

```
/dts-v1/;
/plugin/;
/ {compatible = "xlnx,zynq-7000";

   fragment@0 {
        target = <&fpga_full>;
        #address-cells = <1>;
#size-cells = <1>;
__overlay__ {
#address-cells = <1>;
#size-cells = <1>;
firmware-name = "system_wrapper.bit.bin";
};
};
fragment@1 {
target = <&adc>;
__overlay__ {
xlnx,channels {
```

```
#address-cells = <1>;
#size-cells = <0>;
channel@0 {reg = <0>;};
channel@1 {reg = <1>;};
channel@2 {reg = <2>;};
channel@9 {reg = <9>;};
channel@10 {reg = <10>;};
};
};
};
};
```

qui surcharge l'entrée `&adc` déjà existante dans le devicetree de la Red Pitaya. L'organisation et les options du pilote Xilinx pour leur ADC sont décrites dans le documentation du noyau sous `Documentation/devcietree/bindings/iio/adc/xili`
Nous retrouvons en effet dans `arch/arm/boot/dts/zynq-7000.dtsi` des sources du noyau la configuration d'un Zynq-7000 et son entrée ADC

```
adc: adc@f8007100 {
   compatible = "xlnx,zynq-xadc-1.00.a";
   reg = <0xf8007100 0x20>;
   interrupts = <0 7 4>;
   interrupt-parent = <&intc>;
   clocks = <&clkc 12>;
};
```

qui est surchargée par notre nouvelle configuration. Le pilote Xilinx, compatible `xlnx,zynq-xadc-1.00.a`, est configuré pour respecter IIO tel que décrit dans la documentation citée ci-dessus. Il suffira donc de modifier la configuration du devicetree sans avoir à re-écrire de code dans le pilote pour accéder aux nouvelles ressources implémentées dans le bitstream connectant l'XADC au PS au travers du bus AXI. Nous vérifions l'emplacement du pilote par

```
$ LINUX/drivers/iio/adc$ grep xlnx * | grep adc
xilinx-xadc-core.c:     { .compatible = "xlnx,zynq-xadc-1.00.a", (void *)&xadc_zynq_ops },
```

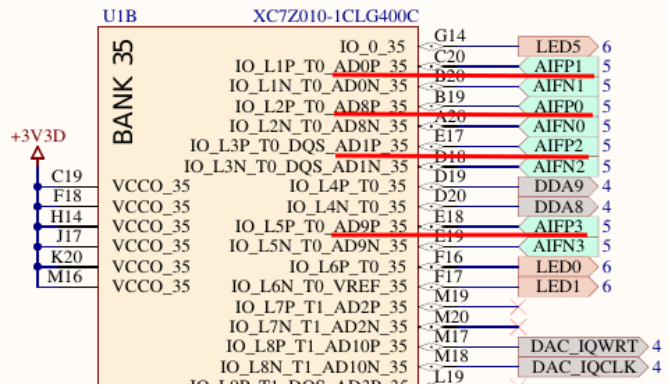
qui est un pilote IIO.

L'adresse de XADC connecté au PS est documentée en page 1160 de `ug585-Zynq-7000.pdf` : les registres de configuration de l'XADC depuis le PS débutent en 0xf8007100. Afin de tenir compte de la configuration de l'XADC dans le bitstream et de reproduire cette configuration décrite dans le devicetree lors du chargement du pilote IIO, on pensera à compiler le pilote `xilinx_xadc.ko` sous forme d'un module. Une fois chargé, le périphérique IIO est accesible depuis `/sys/bus/iio/devices/iio:device0`.

Le schéma `Red_Pitaya_Schematics_STEM_125-10_V1.0.pdf` nous informe de la relation entre la sérigraphie sur la nomenclature des broches et les fonctions dans le Zynq : AI0 est connecté à AD8, AI1 est connecté à AD0, AI2 est connecté à AD1 et AI3 est connecté à AD9, qui explique la nomenclature du pilote IIO :

- broche 16 E2=in_voltage8_vaux9_raw
- broche 13 E2=in_voltage9_vaux8_raw
- broche 14 E2=in_voltage11_vaux0_raw
- broche 15 E2=in_voltage10_vaux1_raw

Pour un accès direct aux registres du XADC par le PL, leurs adresses sont décrites dans la documentation XADC Wizard v3.0 (référence PG091, version April 1, 2015) : par exemple le registre donnant accès à l'ADC8 se trouve à l'offset 0x260 par rapport à l'adresse de base fournie par Vivado.



A Commande par serveur TCP/IP

Un exemple très simple, ne permettant qu'une connexion unique, de serveur TCP/IP est fournie ci-dessous.

```
#include <sys/socket.h>
#include <resolv.h>
#include <unistd.h>
#include <strings.h>
#include <arpa/inet.h>

#define MY_PORT          9999
#define MAXBUF           1024

int main()
{int sockfd;
 struct sockaddr_in self;
 char buffer[MAXBUF];

 sockfd = socket(AF_INET, SOCK_STREAM, 0);  // ICI LE TYPE DE SOCKET

 bzero(&self, sizeof(self));
 self.sin_family = AF_INET;
 self.sin_port = htons(MY_PORT);
 self.sin_addr.s_addr = INADDR_ANY;

 bind(sockfd, (struct sockaddr*)&self, sizeof(self));
 listen(sockfd, 20);

 while (1)
 {struct sockaddr_in client_addr;
  int taille,clientfd;
  unsigned int addrlen=sizeof(client_addr);

  clientfd = accept(sockfd, (struct sockaddr*)&client_addr, &addrlen);
  printf("%s:%d connected\n", inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
  taille=recv(clientfd, buffer, MAXBUF, 0);
  send(clientfd, buffer, taille, 0);
  close(clientfd);
 }
 close(sockfd);return(0);  // Clean up (should never get here)
}
```

L'accès à ce serveur se fait par exemple par `telnet IP 9999` pour effectuer des transactions sur le port 9999 de l'interlocuteur d'adresse IP, ou au moyen de `netcat` par `echo "message | nc IP port`.

B Modifications à la configuration buildroot

Outils non-standard nécessaires aux travaux pratiques

Espace utilisateur	(`make menuconfig`)
screen	
boa	
file	
stress	
vim	(activer *show packages that are also provided by busybox*)
nano	

Modifications au noyau pour permettre l'accès par diverses méthodes aux GPIOs et LEDs, ainsi que configuration du FPGA

Espace noyau	(`make linux-menuconfig`)
passer LEDs en module	`Device Drivers → LED Support →` <M> `LED Class Support` et <M> `LED Support for GPIO connected LED`
passer GPIO en module	`Device Drivers → GPIO Support → Memory mapped GPIO drivers →` <M> `Xilinx GPIO support` et <M> `Xilinx Zynq GPIO support`
xdevcfg en module	`Device Drivers → Character devices → Xilinx Device Configuration`
FPGA Manager en module	`Device Drivers → FPGA Configuration Support →` <M> `FPGA Configuration Framework`

Références

[1] P. Ficheux, *Les distributions "embarquées" pour Rasberry PI*, Opensilicium 7 (2013)

[2] P. Kadionik, P.Ficheux, *Temps réel sous LINUX (reloaded)*, GNU/Linux Magazine France HS 24 (2006), disponible à `http://pficheux.free.fr/articles/lmf/hs24/realtime/linux_realtime_reloaded_final_www.pdf` et `http://www.unixgarden.com/index.php/gnu-linux-magazine-hs/temps-reel-sous-linux-reloaded`

[3] J. Corbet, A. Rubini, & G. Kroah-Hartman, *Linux Device Drivers, 3rd Ed.*, O'Reilly, disponible à `http://lwn.net/Kernel/LDD3/`

[4] *Zynq-7000 All Programmable SoC Technical Reference Manual, rev. 6 Dec. 2017*, Xilinx (2017)

[5] C. Blaess, *Interactions entre espace utilisateur, noyau et matériel*, GNU/Linux Magazine France Hors Série 87 (Nov.–Déc. 2016)

[6] C. Blaess, *Solutions temps réel sous Linux*, Eyrolles (2012)

Activation du serveur X11

La Red Pitaya ne possède pas d'interface pour communiquer avec l'utilisateur au travers d'un écran. Cela ne nous empêche pas de pouvoir exécuter des interfaces graphiques, grâce au serveur X11 qui tournera sur la Red Pitaya et affichage sur le client qu'est le PC (X11 intervertit la convention classique de client et de serveur).

Pour activer le support graphique, le support de X11 est activé dans buildroot par le menu `X.org` dans `Target Packages` → `Graphic libraries and applications (graphic/text)`. On pensera par ailleurs à installer `xauth` afin de générer les certificats autorisant d'exporter l'interface graphique de la Red Pitaya vers l'ordinateur hôte.

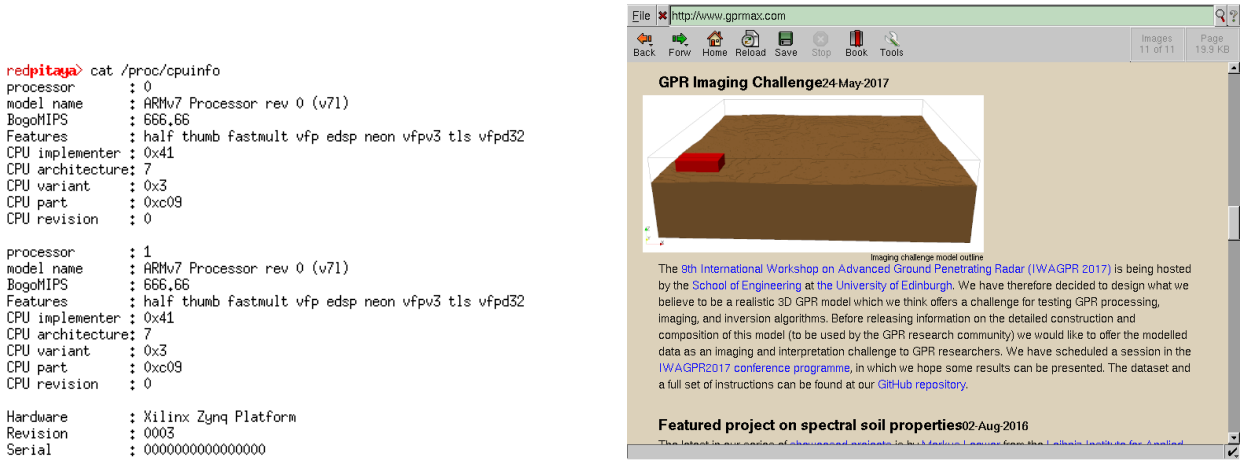


FIGURE 10 – Gauche : Xterm exécuté depuis la Red Pitaya avec un affichage exporté vers l'hôte. Droite : Dillo permet de visualiser des pages HTML depuis la Red Pitaya.

.1 Interface graphique programmée en Qt5

Qt5 est un ensemble de bibliothèques programmées en C++ permettant de générer du code compatible multiplate-formes, que ce soit entre systèmes d'exploitation (MS-Windows, X11 notamment sur GNU/Linux, OS X ou iOS chez Apple) ou plateformes matérielles (x86, MIPS, ARM, SPARC ...). Malgré sa lourdeur, inhérente à toute interface graphique, elle permet de mettre en valeur les performances du système d'exploitation exécuté sur système embarqué.

La compilation de Qt5 par buildroot nécessite le support C++ – qui n'est pas actif par défaut. Recompiler la *toolchain* avec support C++ est garanti en effaçant le répetoire `output` (`rm -rf output`) de buildroot et en re-lancçant `make`.

Par ailleurs, Qt5 n'embarque plus ses polices, et fait appel à un mécanisme (`fontconfig`) trop lourd pour un système embarqué. La façon la plus simple d'embarquer de façon statique les polices de caractère est d'inclure le paquet `liberation` dans `Target packages` → `Fonts, cursors, icons, sounds and themes` et sur la Red Pitaya, après s'être connecté sur la plateforme embarquée, de créer un lien symbolique entre l'emplacement des polices (`/usr/share/fonts/liberation`) et l'emplacement attendu par Qt5 (`/usr/lib/fonts`) :

```
cd /usr/lib
ln -s /usr/share/fonts/liberation/ fonts
```

Une fois le système installé, il nous faut développer une application de test. Le programme trivial proposé (pour Qt4, mais fonctionnel avec Qt5) à `http://doc.qt.io/qt-4.8/gettingstartedqt.html` est sauvé dans un fichier d'extension `.cpp` (Fig. 11).

```cpp
// https://wiki.qt.io/How_to_Use_QTextEdit

#include <QApplication>
#include <QTextEdit>

int main(int argc, char **argv)
{
 QApplication app(argc, argv);

QTextEdit *txt = new QTextEdit();
 txt->setText("Hello, world!");
 txt->append("Appending some text...");

txt->show();
 return app.exec();
}
```


```cpp
// http://doc.qt.io/qt-4.8/gettingstartedqt.html

#include <QApplication>
#include <QTextEdit>
#include <QPushButton>
#include <QVBoxLayout>

int main(int argv, char **args)
{
 QApplication app(argv, args);

 QTextEdit *textEdit = new QTextEdit;
 textEdit->setText("Hello, world!");
 textEdit->append("Appending some text...");

 QPushButton *quitButton = new QPushButton("&Quit");
 QObject::connect(quitButton, SIGNAL(clicked()), qApp, SLOT(quit()));

 QVBoxLayout *layout = new QVBoxLayout;
 layout->addWidget(textEdit);
 layout->addWidget(quitButton);

 QWidget window;
 window.setLayout(layout);
 window.show();

 return app.exec();
}
```

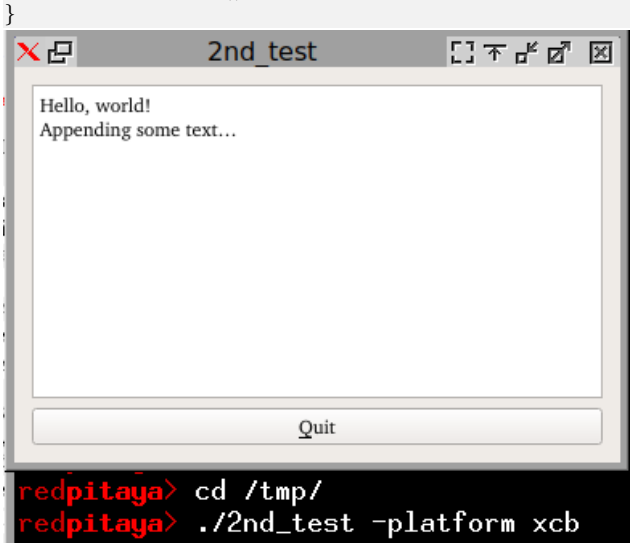


FIGURE 11 – Gauche : exemple d'éditeur de texte, objet fourni par la bibliothèque Qt. Droite : exemple à peine plus évolué avec un bouton.

⚠ L'extension du fichier est importante : une extension `.c` se solde par l'appel à `gcc` qui ne comprend pas la syntaxe du C++ et ne sait pas appeler les bibliothèques appropriées.

Qt propose un mécanisme de compilation qui génère le `Makefile` à partir d'un fichier de configuration d'extension `.pro`. Ce fichier est généré automatiquement par `qmake -project`. Cependant, nous devons prendre soin d'appeler `qmake` depuis le répertoire `output/host/usr/bin` de buildroot *et non le qmake potentiellement installé sur l'hôte (PC).* Ayant généré le fichier de configuration `.pro`, nous y ajoutons les modules de Qt appelés par notre programme, dans notre cas le module widgets, par

```
QT += widgets
```

pour finalement obtenir un fichier de configuration de la forme

```
TEMPLATE = app
TARGET = Qt
INCLUDEPATH += .

QT += widgets

# Input
SOURCES += 2nd_test.cpp
```

Nous appelons à nouveau `$BR/output/host/usr/bin/qmake` – cette fois sans l'option `-project` – pour générer le Makefile, et finalement nous convertissons le code source en binaire exécutable par `make`.

Une fois la compilation achevée, deux résultats sont obtenus :
- l'image (rootfs) GNU/Linux à destination de la cible ARM comporte les bibliothèques Qt,
- l'hôte (le PC) est muni des outils Qt de configuration et de compilation des programmes écrits en C++ selon une syntaxe faisant appel aux bibliothèques Qt.

Le programme s'exécute depuis la plateforme Red Pitaya en précisant que la sortie graphique se trouve sur le périphérique *xcb* : après transfert du fichier vers la plateforme cible qu'est la Red Pitaya, nous exécutons `Qt -platform xcb` et, si tout se passe bien, une fenêtre d'éditeur de texte s'affiche (Fig. 11, bas)

Une fois la méthode de compilation comprise, la langage Qt lui-même est documenté et des exemples fournis dans l'archive à `$BR/output/build/qt5base-5.*/examples`. Nous avons ainsi validé le bon fonctionnement de l'exemple `gui/analogclock` ou `widgets/digitalclock` (Fig. 12).

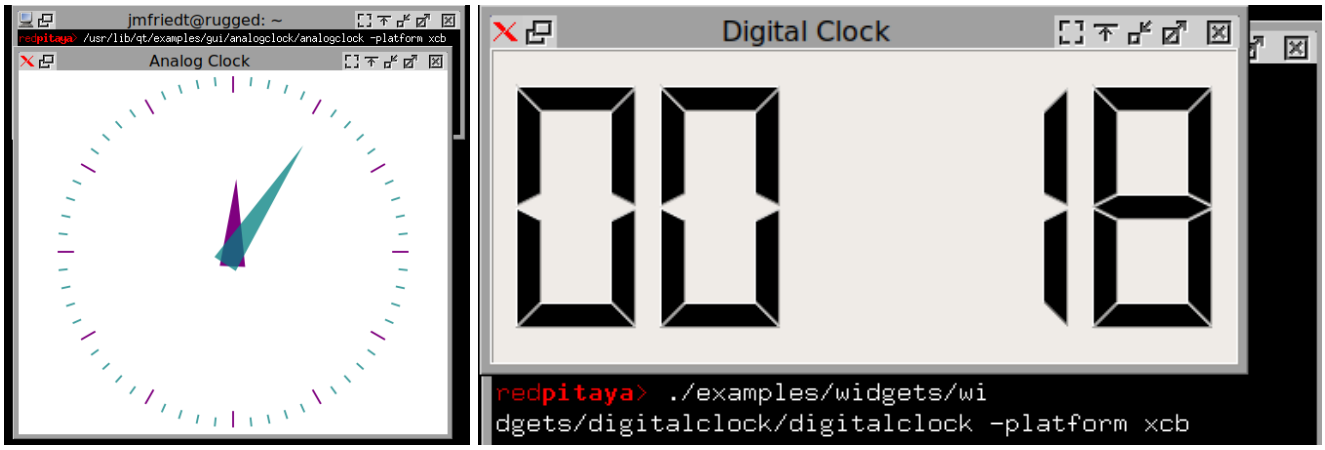


FIGURE 12 – Gauche : l'exemple `analogclock`. Droite : l'exemple `digitalclock`. Tous deux, disponibles dans `/usr/lib/qt/examples/`, sont exécutés sur la plateforme `xcb` fournie en argument de l'option `-plateform` du l'exécutable.