GNU/Linux sur Playstation Portable

S. Guinot & J.-M Friedt

1er février 2009

La Playstation Portable (PSP) est un ordinateur, généralement utilisé comme console de jeu, à base de processeur MIPS [1], fournissant quelques interfaces avec l'utilisateur (boutons, port série, USB et wifi), un clavier de bonne qualité et plus de 32 MB de RAM sans MMU [2]. Il s'agit donc d'un environnement idéal pour faire tourner uClinux. Basées sur divers projets actifs sur le web, nous présentons les étapes pour installer un système uClinux fonctionnel sur cette plateforme, et l'ajout d'une interface avec clavier PS2 pour faciliter les développements. Il s'agit là de bases qui doivent encourager le lecteur à contribuer au portage d'uClinux sur PSP puisque le support de nombreux périphériques est encore absent du noyau.

La Playstation Portable (http://en.wikipedia.org/wiki/PlayStation_Portable, PSP) fournit une plateforme contenant un processeur généraliste basé sur une architecture MIPS R4000 (architecture `-mips3` de `gcc`), 32 MB de RAM et la capacité à exécuter un jeu depuis son support de stockage de masse non volatile (Memory Stick). Il s'agit donc d'un environnement presque idéal pour installer une version embarquée de GNU/Linux sur système sans gestionnaire de mémoire : uClinux. Des progrès ont récemment été fait en ce sens, que nous proposons de présenter ici. Nous allons développer la mise en œuvre d'une toolchain de crosscompilation pour générer le code à destination de la PSP, et le fonctionnement du bootloader qui se présente comme un jeu exécuté depuis la carte de stockage Memory Stick : ces outils sont disponibles sous forme d'archive afin de faciliter leur obtention. Afin de rendre la programmation sur PSP fonctionnelle, nous nous proposerons d'ajouter une interface PS2-RS232 afin de connecter un clavier au port série de la PSP et ainsi disposer d'une interface de communication pratique.

1 Les toolchains

Nous allons présenter deux ensembles d'outils de compilation : dans un premier temps les outils pour compiler le *bootloader* qui apparaît comme un jeu pour la PSP, et d'autre part les outils pour compiler le noyau uClinux et les outils associés. La première partie n'est réellement intéressante que pour le développeur désireux de modifier le *bootloader* et y ajouter ses propres fonctions.

Le second ensemble d'outils est utile pour compiler le noyau uClinux et les outils associés pour l'utilisateur (principalement `busybox`).

Contrairement au format d'exécutables ELF habituellement utilisé par GNU/Linux sur plateforme disposant d'un gestionnaire matériel de mémoire, uClinux utilise le format d'exécutables `binary flat` (BFLT). Il est basé sur le format `a.out`, et fournit un sous ensemble des fonctionnalités proposées par le format ELF : plus petit et plus rapide à charger, les exécutables dans ce format sont mieux adaptés aux systèmes embarqués aux ressources réduites.

La chaîne de cross-compilation MIPS – fonctionnelle puisqu'un certain nombre de routeurs exploitent des processeurs basés sur cette architecture [1] – doit être modifiée afin de générer des binaires FLAT au lieu des habituels ELF. Pour cela, un éditeur de liens spécial doit être utilisé : `elf2flt`, indisponible à l'origine pour architecture MIPS [2].

Le "paquet" `elf2flt` est composée des éléments suivants :

- le script shell `ld-elf2flt`
- le programme `elf2flt` capable de convertir des binaire ELF au format BFLT.
- le programme `flthdr` qui permet d'éditer les entêtes flat.

[1] une des architectures de processeur les plus courantes, issues des recherches à l'université de Stanford, en opposition notamment avec l'architecture SPARC développée à Berkeley

[2] l'absence de gestionnaire de mémoire a des implications quant aux fonctionnalités supportées par le matériel, notamment concernant l'efficacité de changements de contexte lorsque le scheduler passe d'un processus à l'autre, allocation de mémoire ou création de processus. Ces différences expliquent la création d'une branche dédiée du noyau Linux pour ces architectures : uClinux à www.uclinux.org

Le script `ld-elf2flt` va venir se substituer au binaire `ld` original. `ld` est lui renommé en `ld.real`. Si le script `ld` est invoqué avec l'option `-elf2flt`, alors `ld.real` puis `ld-elf2flt` sont successivement utilisés pour générer un binaire au format ELF puis au format BFLT. Dans le cas contraire, `ld.real` est utilisé seul. En conséquence, le comportement de la toolchain n'est modifié que lorsque l'option `-elf2flt` est passée à l'éditeur de liens. Exemple de compilation :

```
# mipsel-psp-gcc -Wl,-elf2flt -o test test.c
# ls
test.gdb test
```

À noter que le programme `elf2flt` produit deux fichiers. Le binaire avec le suffixe `.gdb` contient des informations utiles au débogage et peut être utilisé par exemple avec `gdbserver`. Le format FLAT n'embarque que quelques sections indispensables à l'exécution d'un binaire : `.text`, `.data` et `.bss`. Il ne contient donc pas les sections de debogage que l'on trouve habituellement dans un binaire ELF.

En résumé, la principale difficulté pour construire une toolchain MIPS-PSP est d'inclure les programmes `elf2flt` à la suite `binutils`.

Pour mieux comprendre le format BFLT, une bonne source est le loader du noyau GNU/Linux. Son code se trouve dans le fichier `fs/binfmt_flat.c`. Une autre source incomplète mais utile est le document http://www.beyondlogic.org/uClinux/bflt.htm.

1.1 La compilation de jeux pour PSP

Nous avons déjà décrit dans ces pages [3] la modification logicielle à effectuer sur sa PSP pour pouvoir exécuter des jeux depuis la carte mémoire et compiler ses propres jeux au moyen de la PSP toolchain [3] et du PSP SDK [4]. Avec l'avènement de nouveaux firmwares sur les PSP plus récentes que janvier 2007 (date de rédaction du précédent article) et de la nouvelle PSP Slim, le lecteur se reportera au web pour les dernières nouvelles à ce sujet. La compilation du bootloader et surtout la lecture du code associé nous semble cependant instructive pour comprendre comment exécuter un noyau Linux sur toute plateforme sur laquelle un système tourne déjà. Le bootloader décrit ici (Fig. 1) a été écrit par J. Mo [4], et ne diffère de celui de C. Mulhearn [5] que par sa capacité à charger une image compressée par `gzip` : il faudra donc penser à compiler la `zlib` pour PSP telle que décrite à http://www.psp-programming.com/tutorials/c/lesson04.htm.

À noter que au cours de l'exécution du bootloader, les routines du système d'exploitation Sony natif (que nous visons à remplacer par GNU/Linux) sont encore disponibles, et avec elles l'usage de fonctions telles que l'initialisation des ports série ou du cache du processeur qui ne sont pas nécessairement documentées (ou désassemblées) pour être accessibles sous Linux. C'est donc le dernier moment, avant de lancer Linux, d'initialiser les périphériques dont la gestion n'est possible *que* par le système d'exploitation Sony (par exemple l'affichage en mode texte à l'écran est encore accessible par `pspDebugScreenPrintf()`).

Le code qui suit s'occupe de charger le fichier contenant l'image, de le décompresser, de le copier en RAM dont l'adresse commence en 0x88000000 [6], d'activer les divers périphériques (port série associé au casque audio et cache du processeur), pour finalement exécuter les instructions à partir du début de la RAM.

```
#define KERNEL_ENTRY        0x88000000
#define KERNEL_PARAM_OFFSET 0x00000008
#define KERNEL_MAX_SIZE     (size_t)( 4 * 1024 * 1024 )   /* 4M */

#define printf              pspDebugScreenPrintf

BOOL loadKernel(void ** buf_, int * size_)
{
  gzFile zf;
  void * buf;
```

[3] http://ps2dev.org/psp/Tools/Toolchain/psptoolchain-20070626.tar.bz2
[4] http://ps2dev.org/psp/Projects/PSPSDK

```
  int size;

  zf = gzopen( s_paramKernel, "r" );
  buf = (void *)malloc( KERNEL_MAX_SIZE );
  size = gzread( zf, buf, KERNEL_MAX_SIZE );
  gzclose( zf );
  *buf_ = buf;
  *size_ = size;
}
/*-------------------------------------------------------------------------*/
void transferControl(void * buf_, int size_)
{
    KernelEntryFunc kernelEntry = (KernelEntryFunc)( KERNEL_ENTRY );

    /* prepare kernel image */
    memCopy( (void *)( KERNEL_ENTRY ), buf_, size_ );

    uart3_setbaud( s_paramBaud );
    uart3_puts( "Booting Linux kernel...\n" );

    kernelEntry( 0, 0, kernelParam );
}
```

Le bootloader (Fig. 1) – disponible notamment dans l'archive associée à cet article sur la page web des auteurs à http://jmfriedt.free.fr/ – se compile donc comme un jeu pour PSP par `make kxploit` pour fournir les deux répertoires classiques `pspboot` et `pspboot%`.

```
$ make kxploit
psp-gcc -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall -D_PSP_FW_VERSION=150   -c -o main.o main.c
psp-gcc -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall   -c
psp-gcc -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall -D_PSP_FW_VERSION=150  -L. -L/usr/local/pspdev/psp/sdk/lib
...
```

FIG. 1 – Le bootloader permettant d'exécuter un système GNU/Linux sur PSP se présente comme un jeu, lancé depuis la carte mémoire Memory Stick. Un espace total de l'ordre de 1,5 MB est nécessaire sur la carte.

Recompiler son propre bootloader est l'occasion d'en étudier le fonctionnement : la copie en mémoire volatile (RAM) du système d'exploitation Sony est écrasée pour être remplacée par l'image en *raw binary* de uClinux. Ce code est directement exécutable sur l'architecture cible, sous réserve d'avoir été compilé pour un emplacement approprié en RAM : nous trouverons dans `target/device/mips/psp/linux26.config` la définition de cette adresse sous forme de `CONFIG_PSP_ADDRESS_BASE=0x80000000`

```
jmfriedt@(none):/mnt/sde1/psp/game$ ls -l pspboot
total 1056
-rwxr-xr-x 1 root root 163636 2008-01-04 16:09 eboot.pbp
```

```
-rwxr-xr-x 1 root root    522 2008-01-26 16:26 pspboot.conf
-rwxr-xr-x 1 root root 860798 2008-05-08 15:12 vmlinux.bin
jmfriedt@(none):/mnt/sde1/psp/game$ ls -l pspboot%
total 32
-rwxr-xr-x 1 root root 8292 2008-01-04 16:09 eboot.pbp
```

Nous placerons dans le même répertoire de la carte MemoryStick l'image uClinux complète contenant le noyau et le système de fichier, ainsi que le bootloader associé.

1.2 Les outils pour compiler uClinux pour MIPS

Nous nous sommes efforcés de fournir un environnement cohérent (Fig. 2) pour compiler le noyau uClinux et les applications associées, notamment `busybox` qui fournira tous les outils GNU habituels. Nous avons dans ce but utilisé et modifié le framework de cross-compilatation `buildroot` (version 20071216). Ces modifications permettent de construire une toolchain fonctionnelle à destination du MIPS de la PSP. Un noyau Linux supportant un maximum de périphériques et une image comportant quelques utilitaires de busybox ont également été intégré au `buildroot`. L'archive complète est accessible au moyen de Mercurial par
`hg clone http://hg.sequanux.org/buildroot-psp`
qui crée le répertoire `buildroot-psp`. Une fois dans le dépôt, la commande `hg log` montrera les modifications apportées à l'arborescence `buildroot-20071216`. Le fichier de configuration proposé, `buildroot-20071216-linuxonpsp.config`, est renommé en `.config` avant de `make oldconfig && make`. Buildroot ira alors chercher toutes les archives nécessaires à la compilation. Comme pour toute compilation de noyau, les bibliothèques de développement `ncurses-dev` sont nécessaires.

Notre contribution dans ce projet est la réalisation d'un environnement de développement cohérent (Fig. 2) issu de `buildroot`, utilisant un même ensemble d'outils (*toolchain*) pour compiler le noyau et les applications en espace utilisateur (notamment busybox).

FIG. 2 – Gauche : effet d'une incohérence entre le passage de paramètres par les programmes GNU et le noyau Linux, telle qu'elle peut apparaître lorsque deux toolchains distinctes sont utilisées pour compiler ces deux ensembles de programmes. Ce type de problème est éliminé par l'utilisation d'une unique toolchain générée par `buildroot` pour compiler à la fois le noyau et les applications, tel que nous le proposons ici. Droite : compilation réussie, la séquence classique de boot de Linux se concluant par une console à l'écran.

1.3 Description et mise en œuvre de `buildroot`

`buildroot` est un ensemble de Makefile qui permet de générer une *toolchain* (outils pour compiler des programmes pour une cible donnée), un noyau Linux et un *rootfs* (image de système d'exploitation contenant fichiers de configuration et exécutables). `buildroot` permet d'automatiser le téléchargement des sources et des correctifs (*patch*), la configuration, la compilation et l'installation

des programmes dans le *rootfs*. Différents formats d'image *rootfs* sont possibles : squashfs, cramfs, ext3, archive cpio, initramfs, etc... Dans le cas de la PSP, le fichier chargé en mémoire est obtenu par compression avec `gzip` de l'image binaire *raw* `buildroot-psp/project_build_mipsel/linuxonpsp/linux-2.6.22/a`
Cette image contient un noyau et le *rootfs* embarqué sous la forme d'un initramfs. L'option `BR2_TARGET_ROOTFS_INITRAMFS=y` devra donc être sélectionnée lors de la configuration du `buildroot`.

Étant donné que l'arborescence de `buildroot` peut sembler au premier abord assez déroutante, nous allons en présenter les principaux répertoires et expliquer sommairement leur rôle :

- le répertoire `toolchain` contient les Makefiles nécessaires pour compiler la toolchain. Par exemple, `toolchain/gcc` contient de quoi construire `gcc` pour la cible visée, dans toutes ses configurations supportées :
  ```
  $ tree -L 1 toolchain/gcc/
    toolchain/gcc/
    |-- 3.3.5
    |-- 3.3.6
  ...
    |-- 4.2.1
    |-- Config.in
  ...
    |-- gcc-uclibc-3.x.mk
    |-- gcc-uclibc-4.x.mk
  ```
  Les fichiers `gcc-uclibc-3.x.mk` et `gcc-uclibc-4.x.mk` sont les fragments de makefile qui seront utilisés pour compiler gcc en fonction de la configuration sélectionnée.
- lors de la construction de la toolchain, les différents composants seront compilés sous le répertoire `toolchain_build_xxxx` : pour la PSP, `xxxx` vaut `mipsel`.
- `package` contient les makefiles des applications. Par exemple, `wget` (non busybox) se trouve dans :
  ```
  $~/buildroot$ tree package/wget/
    package/wget/
    |-- Config.in
    `-- wget.mk
  ```
  `wget.mk` est le fragment de Makefile permettant de compiler `wget`. Un rapide coup d'oeil à ce fichier nous présente les cibles intéressantes :
  ```
  $~buildroot/package/wget$ cat wget.mk
  ...
  wget: uclibc $(TARGET_DIR)/$(WGET_TARGET_BINARY)

  wget-clean:
  rm -f $(TARGET_DIR)/$(WGET_TARGET_BINARY)
  -$(MAKE) -C $(WGET_DIR) clean

  wget-dirclean:
  rm -rf $(WGET_DIR)
  ...
  ```
  À la racine du buildroot, la commande `make wget` permet d'ajouter l'application wget au *rootfs*. `make wget-clean` permet de nettoyer le répertoire de compilation de wget (ie `build_mipsel/wget`). `make wget-dirclean` est la cible utilisée pour supprimer ce même répertoire. En règle générale, les cibles `$(app)`, `$(app)-clean` et `$(app)-dirclean` sont présentes pour toutes les applications.
  La même étude sur le répertoire de `busybox` présente une arborescence incluant toutes les versions successives et les correctifs associés.
- à quelques exceptions près les applications (ou *packages*) sont compilées sous `build_mipsel` (variable `BUILD_DIR` dans les différents makefile de paquets). Il s'agit d'un répertoire créé dynamiquement lors de la compilation. Chaque paquet y est décompressé, patché, configuré et compilé.

```
$~/buildroot$ ls build_mipsel/
fakeroot-1.8.10  makedevs  psposk2  staging_dir
```

Pour la PSP par exemple, le répertoire `build_mipsel/psposk2` est utilisé pour cross-compiler l'application On-Screen Keyboard. On notera également la présence du répertoire `build_mipsel/staging_dir`. Il est désigné dans les différents makefiles par la variable `STAGING_DIR` et est en quelque sorte la racine de l'environnement de cross-compilation. Les variables `CFLAGS` et `LDFLAGS` telles que définies par `buildroot` vont respectivement aller pointer (entre autre) vers `$(STAGING_DIR)/usr/include` et `$(STAGING_DIR)/usr/lib`. Lorsqu'une bibliothèque est compilée, elle n'est pas immédiatement intégrée au *rootfs*. Elle est tout d'abord installée dans le répertoire `STAGING_DIR`. C'est une étape importante, surtout si une application à compiler ultérieurement dépend de cette bibliothèque. Le script `configure` de cette application aura sans doute besoin de détecter la bibliothèque ainsi que les fichiers d'entêtes afin d'activer correctement les fonctionnalités et les options avant la compilation. Dans le cas d'une dépendance critique non résolue, la configuration ne sera tout simplement pas possible. Une bonne pratique lors du packaging d'une application est d'utiliser systématiquement le répertoire `STAGING_DIR` pour installer une application (`make install`). Les fichiers voulus sont ensuite recopiés du `STAGING_DIR` vers le *rootfs*. Cette étape fait office de filtre et permet par exemple d'écarter les fichiers de documentation ou alors les programmes qui ne seront pas utilisés sur le système cible. Elle permet aussi de modifier les permissions de certains fichiers.

– les applications principales comme `busybox` ou le noyau Linux ne sont pas compilées sous le répertoire `BUILD_DIR` mais sous `project_build_mipsel/linuxonpsp` (variable `PROJECT_BUILD_DIR`) :

```
$~/buildroot$ ls project_build_mipsel/linuxonpsp/
buildroot-config  busybox-1.9.0  linux-2.6.22  linux-2.6.22-bk  root
```

On remarquera le répertoire `root`. Il est désigné dans les makefiles par la variable `TARGET_DIR`. Comme son nom le laisse deviner, il s'agit du répertoire cible *rootfs* qui sera utilisé pour générer une image finale au format souhaité (ext3, cpio, squashfs, cramfs, etc...). Explorer ce répertoire permet d'observer la composition de l'espace utilisateur du système cible :

```
$~/buildroot$ tree project_build_mipsel/linuxonpsp/root/
|-- bin
|   |-- busybox
|   |-- cat -> busybox
|   |-- cp -> busybox
[...]
|-- etc
|   |-- TZ
|   |-- fstab
|   |-- group
|   |-- hostname
|   |-- hosts
|   |-- init.d
|   |   |-- S20urandom
[...]
|   |-- services
|   `-- shadow
|-- home
|   `-- default
|-- init -> sbin/init
|-- lib
|   |-- libgcc_s.so -> libgcc_s.so.1
|   `-- libgcc_s.so.1
|-- linuxrc -> bin/busybox
|-- proc
|-- root
|-- sbin
```

```
|   |-- getty -> ../bin/busybox
|   |-- init -> ../bin/busybox
|   `-- mdev -> ../bin/busybox
[...]
```

Ce répertoire peut également être utilisé à des fins de développement ou de débogage. En effet il constitue un excellent *NFS root*. Un des avantages d'accéder à un *rootfs* au travers d'un partage NFS est de rendre une application (re)compilée directement disponible sur le système cible sans avoir à reflasher ce dernier. Dans le cas de la PSP, l'absence d'interface réseau facilement exploitable empêche l'utilisation de cette technique. La PSP ne dispose que d'un contrôleur wifi et aucun driver pour l'instant ne permet de l'exploiter. De plus, les interfaces wifi sont de très mauvaises candidates pour accéder à un root filesystem via NFS car elles nécessitent souvent la collaboration d'un programme utilisateur (comme `wpa_supplicant`) afin d'initialiser la connexion réseau.

- le répertoire `target` contient tous les makefiles nécessaires à la compilation du *rootfs* final. Par exemple, lors de la génération d'une image au format `squashfs`, le fragment de Makefile `target/squashfs/squashfsroot.mk` sera utilisé. Nous y trouvons aussi le squelette du répertoire `TARGET_DIR` (*target skeleton*) :

  ```
  $~/buildroot$ ls target/generic/target_skeleton/
  bin  dev  etc  home  lib  mnt  opt  proc  root  sbin  tmp  usr  var
  ```

  L'utilisateur peut utiliser ce répertoire pour inclure dans son système de fichier un script qui n'est pas associé à une application particulière.
  Le fichier `target/generic/mini_device_table.txt` contient lui la définition des fichiers spéciaux statiques du root filesystem final. Les permissions de certains fichiers sensibles (`/etc/passwd` ou `/etc/shadow` par exemple) y sont également définies. Lors de la création de l'image finale le programme `makedev` se basera sur ce fichier `device_table.txt`. La compilation ne disposant bien sûr pas des permissions root, la créations des fichiers spéciaux est rendue possible en utilisant `makedev` en combinaison avec le programme `fakeroot`.

  ```
  $~/buildroot$ cat target/generic/mini_device_table.txt
  ...
  /etc/shadow             f       600     0       0       -       -       -       -       -
  /etc/passwd             f       644     0       0       -       -       -       -       -
  ...
  # Sony Memory stick
  /dev/ms0                b       640     0       0       31      200     -       --
  ```

  `/dev/ms0` est le fichier spécial bloc qui permet d'accéder au memory stick Sony. L'utilisateur peut modifier ce fichier à sa convenance pour ajouter d'autres nœuds (*nodes* sous `/dev`).
- comme pour le noyau Linux, le fichier de configuration se trouve sous la racine du buildroot et porte le nom de .config. Il se modifie par `make config`, `make menuconfig` etc ...

En résumé, le buildroot se divise en 2 grandes familles de répertoires : ceux contenant les Makefiles, package et toolchain, et les répertoires de travail créés à la compilation.

Pour finir, on soulignera un des défauts de `buildroot`. Il s'agit d'une dépendance assez forte avec l'environnement host de cross-compilation. Lors de la configuration des applications, des outils comme `pkg-config` par exemple, ont la facheuse tendance en cas de recherche infructueuse d'aller examiner le répertoire host `/usr/lib/pkgconfig/`. Dès lors, on imagine que la génération d'un *rootfs* sur un host où par exemple `dbus` est installé peut poser un certain nombre de problèmes cocasses : au mieux, une erreur de compilation et au pire, un comportement suspect à l'exécution... Afin de parer à ce genre de désagréments, une bonne pratique est de cross-compiler au sein d'un environnement *chroot* minimaliste et maîtrisé. Une installation `debootstrap` d'une debian stable (ou même instable) fournit un environnement host de cross-compilation très décent. Ce genre de procédure est décrit à http://wiki.easyneuf.org/index.php/Buildroot_HOWTO.

2 Le noyau Linux pour PSP

Nous allons maintenant présenter quelques uns des composants du noyau Linux pour la console PSP.

2.1 Le driver memory stick Sony

Le driver `ms_psp` permet d'accéder aux memory sticks Sony ProDuo. Les memory sticks standards ne sont eux pas supportés.

L'accès bas niveau au memory stick Sony est fourni par l'IPL SDK. Les fichiers sources concernés sont `arch/mips/psp/ipl_sdk/memstk.c` et `include/asm-mips/ipl_sdk/memstk.h`. Les fonctions exportées sont :

- `int pspMsInit(void)`
- `int pspMsReadSector(int sector, void *addr)`
- `int pspMsWriteSector(int sector, void *addr)`

La partie du driver s'interfaçant avec la couche bloc du noyau Linux se trouve dans le fichier `drivers/block/ms_psp.c`. La structure de ce pilote bloc [9] est plutôt classique.

On notera cependant quelques particularités : Le driver `ms_psp` ne définit pas la méthode `request_fn`. De plus il implémente sa propre méthode `make_request_fn`. Habituellement, les drivers blocs implémentent la méthode `request_fn` pour réaliser les opérations d'entrées/sorties demandées et ne définissent pas la méthode `make_request_fn`. La fonction standard `__make_request()` mise à disposition par le module `block_core` est alors utilisée par défaut. Le rôle de cette fonction est d'organiser la file d'attente des requêtes avant de la soumettre au driver via la méthode `request_fn`. L'intérêt de classer les requêtes est évident. Prenons l'exemple d'un périphérique de type disque dur. Les déplacements de la tête sur le disque doivent impérativement être optimisés et ce travail est celui de la fonction `__make_request()`. Cette dernière insère les nouvelles requêtes dans la file de requêtes du driver de façon à optimiser les accès au médium et la rapidité des transferts. Pour cela, les requête concernant des secteurs mémoires contigus pourront être concaténées. Le tri des requêtes proprement dit est réalisé par un ordonnanceur (ou encore élévateur) bloc. Les élévateurs proposés par le noyau Linux sont : *noop*, *CFQ (Complete Fairness Queueing)*, *deadline* ou encore *anticipatory*. Le lecteur curieux pourra trouver des informations en se référant à la documentation du noyau (`Documentation/block/`) ou encore en consultant [10, chap. 14 "Block Device Drivers"]. Dans le cas du memory stick Sony, dont la mémoire est de type flash, la problématique du temps d'accès à un secteur est moins cruciale que pour un disque dur. C'est pour cela que le driver `ms_psp` court-circuite l'ordonnaceur bloc en implémentant sa propre méthode `make_request_fn` : `psp_ms_make_request()`. Cette fonction ne classe pas et n'insère pas les BIO (blocs d'entrées/sortie) dans la file d'attente de requêtes mais transfère directement les données. Cette implémentation évite la gestion coûteuse d'une queue de requêtes. Cette stratégie a cependant un inconvénient. Ne pas organiser les accès au médium en fonction de l'adresse des secteurs demandés ne permet pas de grouper les accès sur plusieurs secteurs mémoire contigus du memory stick. Par exemple, la lecture d'un fichier peut nécessiter l'accès à plusieurs secteurs contigus sur le périphérique. Dans ce cas, l'implémentation du `ms_psp` est telle que les secteurs seront lus un à un. Autant de commandes que de secteurs demandés seront transmises au memory stick.

```
static int psp_ms_make_request(request_queue_t * queue_, struct bio * bio_)
{
bio_endio( bio_,
     bio_->bi_size,
     psp_ms_transfer_bio( (psp_ms_partition_t *)queue_->queuedata,
  bio_ )
);
return 0;
}

static int psp_ms_transfer_bio
(
  psp_ms_partition_t * partition_,
  struct bio * bio_
)
{
  struct bio_vec * bvec;
  int i, rt;
```

```
  void * buf;
  sector_t sector, numSectors;

  /* Calculate the start sector */
  sector = bio_->bi_sector + partition_->startSector;

  bio_for_each_segment( bvec, bio_, i )
  {
    buf = __bio_kmap_atomic( bio_, i, KM_USER0 );
    //numSectors = bio_cur_sectors( bio_ );
    numSectors = bvec->bv_len >> 9;

    if ( bio_data_dir( bio_ ) == READ )
    {
      rt = psp_ms_read( buf, sector, numSectors );
    }
    else
    {
      rt = psp_ms_write( buf, sector, numSectors );
    }

    /* always call unmap regardless of the operation succeeded or not */
    __bio_kunmap_atomic( bio_, KM_USER0 );

    if ( rt < 0 )
    {
      DBG(( "Failed to %s MS (%d,%d) with error of %d\n",
            ( bio_data_dir( bio_ ) == READ ) ? "read" : "write",
            sector, numSectors, rt ));
      return -EIO;
    }

    sector += numSectors;
  }

  return 0;
}
```

Les `structure bio` transmises à la fonction `psp_ms_make_request()` puis à la fonction `psp_ms_transfer_bio()` ne seront composées que d'un seul segment (`struct bio_vec`). En effet, la fusion de requêtes bio adjacentes normalement effectuée par l'élévateur bloc n'est pas implémentée par le driver `ms_psp`. La macro `bio_for_each_segment()` ne produira donc qu'une seule itération. Étant donné qu'un segment est d'au mieux de la taille d'une page (4096 octets), la fonction psp_ms_read() sera donc appelée pour transmettre au maximum 8 secteurs (de 512 octets).

```
static int psp_ms_read(void * buf_, int sector_, int numSectors_)
{
  int t, i, rt;
  char * readBuf;

  if ( down_interruptible( &s_psp_ms_rw_sem ) != 0 )
  {
    return -EBUSY;
  }

  for ( readBuf = (char *)buf_, i = 0; i < numSectors_; i++ )
  {
    rt = psp_ms_read_sector( readBuf, sector_ + i );
    if ( rt < 0 )
    {
      DBG(( "%s: Failed to read sector %d, err=%d\n",
            PSP_MS_NAME, sector_ + i, rt ));
      break;
    }

    readBuf += PSP_MS_SECTOR_SIZE;
```

```
  }

  up( &s_psp_ms_rw_sem );
  return rt;
}
```

De plus l'IPL ne permet de traiter qu'un secteur à la fois. La fonction `psp_ms_read_sector()` sera donc appelée pour chacun des secteurs à lire.

Assurément le driver `ms_psp` et l'IPL permettent de dialoguer avec un memory stick mais des optimisations sont possibles :

- Ajouter à l'IPL le support pour transférer les secteurs contigus du memory stick via une seule commande.
- Utiliser un élévateur bloc afin d'agglomérer les requêtes concernant des secteurs adjacents.

Un pilote `memstick` a récemment été intégré au noyau Linux. Son implémentation est bien plus optimisée et complète que celle du driver `ms_psp`. Il supporte par exemple les memory sticks standards. Une amélioration intéressante serait donc d'intégrer ce driver au noyau Linux PSP et d'y ajouter un driver host PSP en regroupant et complétant les fonctions bas niveau exportées par l'IPL. Pour l'instant, les contrôleurs hosts supportés par le pilote `memstick` sont les lecteurs de cartes TI Flash Media et JMicron JMB38X.

2.2 Le driver du joypad

Le noyau Linux pour PSP fournit un driver (`psp_joypad`) permettant d'utiliser le joypad et les boutons de la console. Le code source de ce driver est localisé dans le fichier `drivers/input/joypad_psp.c`. Contrairement à ce que le chemin (*path*) peut laisser penser, ce pilote n'utilise malheureusement pas la couche `input` du noyau. Il s'agit en fait d'un pilote de type caractère classique.

La connaissance et le support de l'architecture de la PSP par le noyau Linux ne permettent pas encore d'associer une ligne d'interruption aux mouvements du joypad ou à l'appui d'un bouton. Le driver a donc recours au *polling* : un *thread* kernel dédié collecte à intervalle régulier la valeur du registre de statut des boutons et du joypad. L'accès bas niveau à ce registre est fourni une fois de plus par l'IPL SDK via la fonction `_pspSysconGetCtrl1()`.

```
static int __init psp_joypad_init()
{
  [...]

  INIT_LIST_HEAD( &s_psp_joypad_queue_list );

  /* Launch the daemon thread here */
  s_psp_joypad_thread_id = kernel_thread( psp_joypad_thread,
                                          NULL,
                                          CLONE_FS | CLONE_SIGHAND );
  [...]
}

static int psp_joypad_thread(void * unused_)
{
  unsigned long val;
  struct list_head * pos;

  while ( !s_psp_joypad_thread_terminated )
  {
    if ( psp_joypad_read_input( &val ) )
    {
      DBG(( "%s: %08x\n", PSP_JOYPAD_NAME, val ));
```

```
      /* Turn the LCD back on whenever a button is pressed */
      psp_lcd_on();

      /* Feed the data to all registered queues */
      if ( down_interruptible( &s_psp_joypad_queue_list_sem ) == 0 )
      {
        list_for_each( pos, &s_psp_joypad_queue_list )
        {
          (void)psp_joypad_queue_push( LIST_TO_QUEUE( pos ), val );
        }
        up( &s_psp_joypad_queue_list_sem );

        wake_up_interruptible( &s_psp_joypad_wait_queue );
      }
    } // end if

    msleep( 1000 / PSP_JOYPAD_SAMPLE_RATE );
  } // end while

  return 0;
}
```

La fonction `psp_joypad_thread()` est donc le cœur du driver Joypad PSP. Elle accomplit les tâches suivantes :

1. collecter périodiquement (toutes les 50 millisecondes) la valeur du registre de status du joypad en appelant la fonction `psp_joypad_read_input()`.
2. ajouter les valeurs recueillies dans une file d'attente type FIFO.
3. et enfin réveiller les processus en attente de lecture.

Un processus souhaitant connaître l'état des boutons et du joypad peut par exemple utiliser la méthode `read()` exportée par le driver `psp_joypad`. Pour cela, il lui suffit d'ouvrir le fichier spécial `/dev/joypad` (majeur 11, mineur 200) et de d'utiliser l'appel système `read()`. Ce dernier conduit le fil d'exécution du processus jusqu'à la méthode `psp_joypad_fop_read()` (exportée par le pilote). Le processus est ensuite inscrit dans la file d'attente de lecture du driver. Il sera réveillé lorsque des données seront disponibles. Un exemple d'utilisation de cette interface pourra être étudié en consultant le code source du PSP OSK (PSP On Screen Keyboard).

Le driver PSP Joypad fonctionne relativement bien, mais des améliorations sont possibles.

Tout d'abord, le driver pourrait et devrait utiliser le module *input* proposée par le noyau Linux. Ce module offre un certain nombre de facilités et permet notamment d'exporter une interface standard vers l'espace utilisateur. Par exemple, le joypad et les boutons de la PSP deviendraient directement utilisables pour toutes les applications supportant l'interface *evdev*. Ce qui est le cas du serveur X. La documentation du noyau (`Documentation/input/input-programming.txt`) présente un exemple simple d'implémentation pour un *input device driver*. Cet exemple pourrait constituer un excellent squelette pour une nouvelle version du pilote PSP Joypad.

Une autre amélioration possible serait évidemment d'identifier et d'utiliser la ligne d'interruption associée au joypad et aux boutons de la PSP.

2.3 Le driver série

La PSP est munie de plusieurs ports série : soit la liaison filaire par le connecteur à côté du jack pour le casque, soit la liaison infrarouge. Bien qu'une interruption soit associée aux évènements survenus sur les ports de communication asynchrones (UART défini comme l'interruption 0 [7]),

l'implémentation proposée ici exploite – probablement pour des raisons historiques – la communication en sondant périodiquement l'état de ces ports, en même temps que la gestion d'autres évènements sous le contrôle de l'interruption timer (interruption CPUTIMER numéro 66 dans [7]).

La communication par port série UART3 a été implémentée dans `linux/drivers/serial/serial_psp.c`

```
void psp_uart3_txrx_tick()
{
  if ( !s_psp_uart3_port_data.shutdown &&
       ( s_psp_uart3_port_data.txStarted ||
         ( !s_psp_uart3_port_data.rxStopped &&
           !( PSP_UART3_STATUS & PSP_UART3_MASK_RXEMPTY ) ) ) )
  {
    up( &s_psp_uart3_port_data.sem );  // reveille la reception du char
  }
}
```

Ce gestionnaire d'interruption réveille un thread noyau chargé de tester la présence d'un caractère dans la file de réception de l'UART (Fig. 3) : `psp_port_txrx_thread()`. Le réveil se fait donc par déclenchement d'une interruption timer, initialisée dans `psp.c` sous `arch/mips/psp` (fonction `psp_cputimer_handler()`). Lors de la lecture des données, l'accès au port est bloqué par un mutex `init_MUTEX_LOCKED( &portData->sem );` qui est initialisé par le thread noyau `psp_port_txrx_thread()`.

On retrouve ici une initialisation telle que celle présentée plus haut pour le joypad :

```
static int __init psp_serial_modinit()
{
[...]
portData->txThreadId = kernel_thread( psp_port_txrx_thread,
                       port,
                       CLONE_FS | CLONE_SIGHAND );
[...]}
```

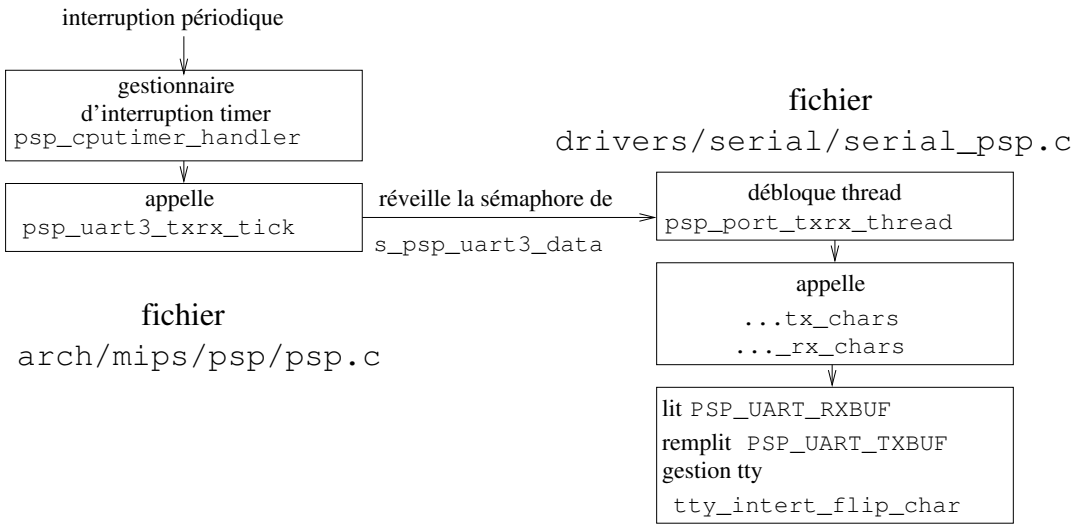

FIG. 3 – Interaction entre le gestionnaire d'interruption timer et de communication avec les ports série pour périodiquement sonder l'état de ces ports et interagir avec la console.

La gestion des ports séries de la PSP, nommés `/dev/ttySRCi`, est décrite dans le fichier `serial_psp.c` du sous répertoire `drivers/serial` de l'arborescence du noyau. Nous y trouverons d'une part une gestion du protocole de communication fortement inspirée de la description de la configuration de l'UART proposée dans l'IPL SDK et dans les logiciels dédiés à la PSP, et d'autre part l'interfaçage du port série avec un terminal – et notamment une console – probablement l'originalité la plus intéressante de ce driver.

3 Exécution et console série

Étant donné que le bootloader (le jeu exécuté en natif sur l'OS Sony de la PSP) se charge d'initialiser les périphériques en exploitant les fontions disponibles initialement sur la console, Linux n'a pas besoin de se charger de cette tâche : on ne trouvera donc pas de fonction `boot` comme c'est habituellement le cas dans `linux-2.6.22/arch/`, mais seulement un appel aux fonctions spécifiques à la PSP lors du démarrage du kernel par `arch/mips/kernel`. Néanmoins, les fonctions dédiées à l'architecture de la console se retrouvent bien dans `linux-2.6.22/arch/mips/psp`, et notamment la gestion des interruptions et la réinitialisation périodique du chien de garde *watchdog*.

3.1 Ajout d'un clavier

L'injection de caractères dans la couche tty [8, chap. 18 "TTY Drivers"] par le gestionnaire de port série (fonction `tty_flip_char()` de `psp_uart3_rx_chars()` s'exploite de deux façons : soit en connectant un PC sur lequel tourne un logiciel de communication sur le port série (`screen` ou `minicom` par exemple), soit en connectant un clavier de PC dont le protocole de communication a été transformé en RS232. La première solution est la plus simple mais la moins pratique puisqu'elle nécessite un PC, réduisant quelque peu l'intérêt d'une PSP sous GNU/Linux. La seconde solution est abordée en détail ici.

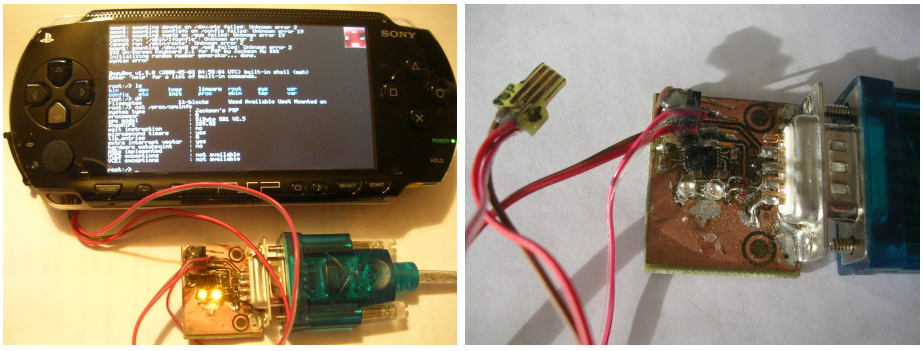

FIG. 4 – Circuit de conversion RS232-connecteur série de la PSP, à côté du connecteur jack audio. Il s'agit simplement d'un MAX3232 ou équivalent transformant les niveaux ±12 V en 0-3,3 V. Ce montage permet de communiquer au moyen de `minicom` par exemple avec un système GNU/Linux sur PSP lorsqu'une console est connectée au port série `/dev/ttyS2`.

Du point de vue utilisateur, la création d'une console sur le port série (Fig. 4) s'obtient par l'ajout dans `/etc/fstab` de la ligne

```
# Put a getty on the serial port
ttyS2::respawn:/sbin/getty -L ttyS2 115200 vt100
```

Cette fonctionnalité entre évidemment en conflit avec la disponibilité d'un clavier sur le port série : il est donc dans ce cas nécessaire de désactiver la réception des caractères pour injection dans la couche clavier. Réciproquement, l'activation du clavier sur le port série nécessite de désactiver la console qui y est associée pour éviter tout conflit.

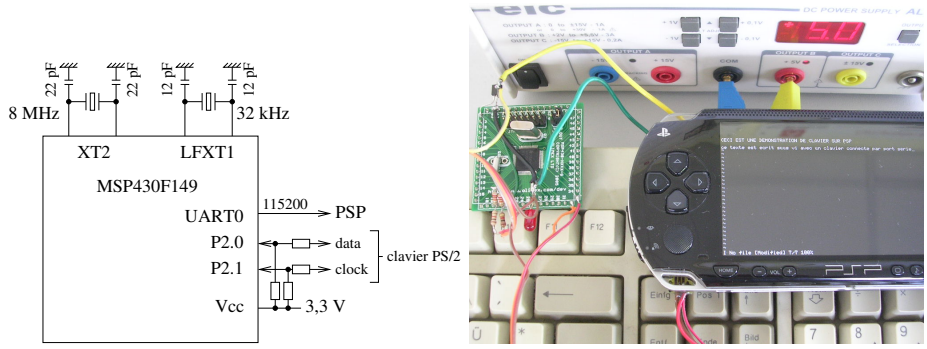

FIG. 5 – Circuit de conversion entre un clavier PS2 et un format RS232 compatible avec la PSP. Un microcontrôleur est dédié à cette tâche : le protocole synchrone bidirectionnel de communication avec les périphériques PS2 y est implémenté de façon logicielle. Seul un caractère sur deux est transmis à la PSP puisqu'un clavier transmet le code de touche à chaque appui *et relâchement* de la touche concernée. Sur la figure de droite, `vi` est fonctionnel sur PSP : la console devient ainsi un outil de travail autonome ne nécessitant pas un ordinateur additionnel pour communiquer.

Nous avons ajouté la communication avec un clavier PS2 par la voie la plus simple : conversion de PS2 (protocole synchrone bidirectionnel) en RS232 (protocole asynchrone avec des voies séparées pour l'émission et la réception de données) au moyen d'un MSP430. Le travail est minimal puisqu'un exemple quasiment fonctionnel est fourni avec `msp-gcc` (version de `gcc` pour MSP430) dans `examples/mspgcc/pc_keyboard`. Sans entrer dans les détails du protocole PS2 qui est parfaitement géré par le microcontrôleur, nous nous contentons de modifier ce programme pour communiquer par le port série au lieu d'afficher les caractères sur un écran, et de retirer un code clavier sur deux puisque appuyer *et relâcher* une touche induisent tous deux la transmission du code correspondant. Le `main()` du programme flashé dans le MSP430 devient donc pour notre application :

```
while (1) {                          //main loop, never ends...
    LPM0;                            //sync, wakeup by irq
    processRawq();
    while( (c = getkey()) ) {
        if ((c >> 8) == 0) {

            if (c=='\n') {putc(10);} // putc(13);}
            else
            if (makebreak==0) {putc(c);makebreak=1;}
               else makebreak=0;
        } else {
            switch (c) {
                case CAPS:
                    sendkeyb(0xed);
                    delay(1000);
                    sendkeyb((s & 1) ? (BIT0|BIT1) : 0);
                    break;
                case F1:
                   putc('h'); putc('e'); putc('l'); putc('l'); putc('o');
            }
        }
    }
}
```

Ces tâches très simples peuvent être exécutées par n'importe quel microcontrôleur : nous avons pour notre part utilisé un MSP430F149 (Fig. 4) à ces fins. Le programme se compile au moyen de `msp430-gcc` et des binutils associés que nous avions décrit par ailleurs [11].

3.2 Utilisation des boutons

En complément du clavier relié au port série – ou pour les moins électroniciens qui ne désirent pas implémenter le montage présenté précédemment – une solution purement logicielle est fournie sous la forme du PSP OSK (On Screen Keyboard). Il s'agit d'une interface dédiée à la PSP qui affiche quelques touches sur l'écran, le choix de la touche étant effectué par combinaison des boutons de contrôle de la PSP. Sans permettre de taper un long texte (les amateurs de SMS me contrediront peut-être), cette interface permet au moins de visualiser le contenu de quelques fichiers ou tester quelques fonctionnalités nouvellement compilées dans busybox (Fig. 6).

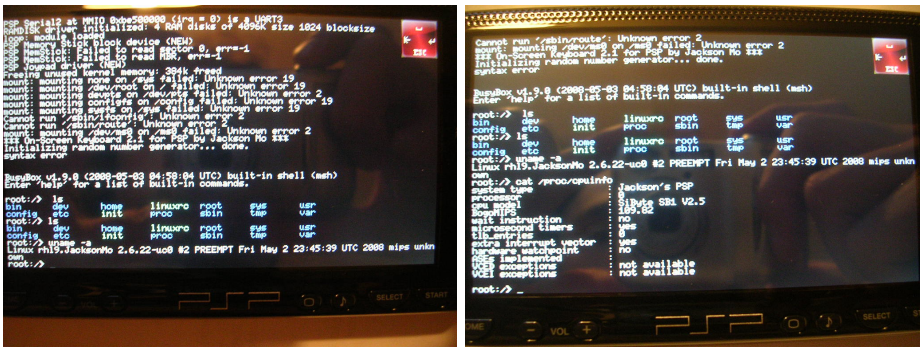

FIG. 6 – L'OSK est visible en haut à gauche de l'écran de ces photos de la PSP. Il s'agit du carré rouge contenant les symboles de lettres dont le choix se fait par des combinaisons des boutons de la PSP. Bien que d'un accès peu confortable, il permet de valider la compilation du système avec des commandes simples telles que `dmesg`, `uname -a` (gauche), ou `cat /proc/cpuinfo` (droite).

4 Développements additionnels

Nous nous sommes contentés dans cette présentation d'installer un noyau Linux et quelques outils additionnels afin de rendre la console exploitable. La gestion de l'écran graphique est implémentée sous forme de framebuffer (`drivers/video/pspfb.c` dans l'arborescence du noyau) et donne donc accès à tous les programmes qui supportent ce périphérique (incluant notamment ceux compilés avec la bibliothèque SDL ou au moyen de qtopia, la version embarquée de Qt).

Le matériel inclus dans la PSP est en grande partie propriétaire et non-documenté. Les applications fonctionnant sous PSP font appel aux fonctions fournies par le système d'exploitation Sony : il s'agit des nombreux appels aux fonctions dont les noms commencent par `sce`, référencées dans le PSP SDK disponible à http://ps2dev.org/psp/Projects. Nous ne savons pas ce que font ces fonctions : nous nous contentons de les appeler aveuglément et d'en attendre une réponse. Cette solution n'est pas exploitable par GNU/Linux exécuté en remplacement de l'OS Sony, puisque dans ce cas les fonctions ne sont plus utilisables (l'espace RAM contenant l'OS Sony, et donc ces fameuses fonctions `sce*`, a été effacé pour laisser place à Linux).

Une solution consiste alors à profiter du travail des contributeurs à `IPL` qui désassemblent certaines fonctions fondamentales de l'OS Sony et fournissent sous forme de code source des fonctions effectuant les mêmes opérations. L'archive de ces codes sources est disponible à http://www.mediafire.com/?etdnoonpnfj. Dans l'arborescence de développement du noyau Linux pour PSP, ces fonctions se retrouvent dans `linux-2.6.22/include/asm-mips/ipl_sdk` pour les prototypes, et `linux-2.6.22/arch/mips/psp/ipl_sdk`. Ainsi, seule l'étude des codes désassemblés du système d'exploitation natif de Sony permettra de découvrir le fonctionnement de périphériques aussi complexes que les interfaces wifi ou USB. Un effort en ce sens a déjà permis l'utilisation des Memory Stick Pro depuis uClinux grâce aux outils fournis dans le IPL SDK http://exophase.com/psp/booster-releases-psp-ipl-sdk-2190.htm.

5 Conclusion

Nous avons présenté l'exploitation d'une console de jeu, la Playstation Portable, comme environnement de développement pour uClinux sur une architecture intéressante à base de processeur MIPS. Cette activité a introduit un certain nombre de tâches qui se retrouvent sur tout port d'un système d'exploitation sur une nouvelle architecture embarquée :

- écrire un bootloader qui initialise les périphériques du processeur et charge le système d'exploitation en mémoire
- modifier le système d'exploitation afin de l'adapter à sa propre application. Ici, l'affichage se fait sur écran graphique, tandis que la communication se fait soit au moyen des boutons de la PSP, soit au moyen d'un clavier qui s'adapte sur le port série dont est équipée la console
- identifier le matériel afin de développer les drivers appropriés pour utiliser au mieux les fonctionnalités du système d'exploitation.

À notre connaissance, la majorité des périphériques ne sont pas exploitables à ce jour : seules les MemoryStick Pro sont supportées, le port USB et le wifi ne sont pas documentés. Il s'agit donc d'une plateforme pour laquelle les développeurs sont encore susceptibles de fournir des contributions pertinentes.

Références

[1] É. Heitor, *Jugamos a a Fonera*, GNU/Linux Magazine France **94** (Mai 2007)

[2] Xiptech est une société qui met à disposition un certain nombre d'outils et une toolchain à destination de l'architecture MIPS : *Porting uClinux to MIPS* est disponible à http://www.xiptech.com/download/porting.zip

[3] J.-M Friedt, *Introduction à la programmation sur PlayStation Portable*, GNU/Linux Magazine France **92**, Mars 2007

[4] le site web de Jackson Mo "Linux On PSP" regroupe les outils dont nous nous sommes servis pour générer un buildroot fonctionnel : jacksonm88.googlepages.com/linuxonpsp.htm

[5] le site original df38.dot5hosting.com/~remember/chris/ n'est plus actif : un complément qui s'en approche est http://www.bitvis.se/articles/psplinux.php

[6] TyRaNiD, *The Naked PSP*, présentation disponible à http://ps2dev.org/psp/Tutorials/PSP_Seminar_from_Assembly.download

[7] groepaz/hitmen, *Yet another PlayStationPortable Documentation*, disponible à http://hitmen.c02.at/files/yapspd/psp_doc/, et en particulier la liste des interruptions de la PSP à http://hitmen.c02.at/files/yapspd/psp_doc/chap9.html

[8] J. Corbet, A. Rubini & G. Kroah-Hartman, *Linux device drivers*, O'Reilly (2005) disponble à http://lwn.net/Kernel/LDD3

[9] T. Petazzoni & D. Decotigny, *Conception d'OS : pilotes de périphériques blocs*, GNU/Linux Magazine France **80** (janvier 2006), pp. 74-90

[10] Daniel P.Bovet & Marco Cesati, *Understanding the Linux kernel*, O'Reilly (November 2005 Third Edition)

[11] J.-M. Friedt, A. Masse, F. Bassignot, *Les microcontroleurs MSP430 pour les applications faibles consommations – asservissement d'un oscillateur sur le GPS.*, GNU/Linux Magazine France 98, Octobre 2007

Simon Guinot est développeur de systèmes embarqués et noyau Linux, membre de l'association Sequanux pour la diffusion du logiciel libre en Franche Comté. Il est joignable sur IRC sur #sequanux, serveur irc.freenode.net.

Jean-Michel Friedt est membre de l'association Projet Aurore pour la diffusion de la culture scientifique et technique à Besançon. Afin de maîtriser diverses architectures de processeurs en vue du développement de systèmes embarqués, il s'intéresse à l'exploitation d'outils libres en vue de porter des logiciels complexes (par exemple le noyau Linux) à des plateformes aux ressources réduites. Les consoles de jeu offrent une plateforme souvent disponible, ou tout au moins accessible à moindres coût.