Portfolio

Gaël PORTAY

3 décembre 2015

Résumé

Je suis ingénieur en informatique, spécialisé en *Linux Embarqué*. J'ai découvert l'informatique à l'âge de 10 ans. Aujourd'hui je totalise un peu plus de 6 ans d'expérience dans le monde professionnel. Je suis passionné et curieux : j'aime apprendre et comprendre comment fonctionne les « choses ». Je suis également autodidacte et j'aime partager mes expériences et mes connaissances avec les autres.

Mon travail étant la partie immergée de l'iceberg, il est par conséquent assez difficile de montrer le fruit de mon travail par des images ou des photos. En lieu et place [1], j'expliquerai comment et avec quels outils j'ai réalisé mes travaux. De même, certains de mes développements étant du domaine du propriétaire, je ne présenterai ici que mes développements libres.

Je mets en évidence deux projets pour démontrer mes compétences : le *BSP* [2] pour la partie électronique ainsi que la partie système d'exploitation au niveau noyau ; et deux projets personnels pour la partie système d'exploitation au niveau espace utilisateur. L'annexe montre mon implication vis-à-vis de la communauté du Logiciel Libre.

Voici une liste non exhaustive de mes compétences : langage *C/C++* (libc, STL), mécanisme de *polling* (epoll), *Git*, *Systèmes Linux* (espace utilisateur et noyau), scripts *Shell* (POSIX, redirection, pipe...), *Python*, *Makefile*, *Autotools*, *Kconfig*, *Yocto*, *crosstool-ng*, *QEMU*...

1. Plus tard, j'illustrerai mes propos par quelques schémas.
2. Board Support Package : le logiciel bas niveau d'une carte électronique.

Table des matières

Première partie

Board Support Package

Mots clés : *C*, *noyau*, *device-tree*, *électronique*, *SoC*, *bootloader*.

J'ai développé le logiciel bas niveau (appelé *BSP*) de deux nouvelles cartes électroniques développées par Overkiz. Ces deux plateformes sont basées sur deux système-sur-puce d'*Atmel* (ou *SoC*[3]) : SAM9G25 et SAMA5D31.

Mon travail consistait à faire fonctionner notre distribution *Linux* maison sur ces deux nouvelles plateformes. J'ai validé le fonctionnement de l'ensemble des composants utilisés sur les deux cartes électroniques : les *DELs*, les boutons, le réseau (*Ethernet*), les différents bus de communication (*UART*, *USB*, *I2C* et *SPI*), les différentes mémoires (*NAND* et *RAM*) ainsi que le contrôleur de gestion de l'alimentation (*PMIC*[4]).

Le développement de ces deux *BSP*s se décompose en deux parties distinctes :
- le *bootstrap* : comme chargeur d'amorçage (*bootloader*) pour démarrer un noyau *Linux*, et
- le *device-tree* : pour définir la carte électronique au niveau du noyau *Linux*.

1 AT91Boostrap

Mots clés : *C*, *UBI*, *bootloader*.

Le *bootstrap* est le premier logiciel à s'exécuter après la mise sous tension du *SoC* d'*Atmel*. Il est chargé par le *ROM code* interne du *micro-processeur*. Le *ROM code* est appelé *bootloader* de niveau 1 tandis que le *bootstrap* est appelé *bootloader* de niveau 2. Le rôle du *bootstrap* est d'initialiser les mémoires afin de charger un logiciel depuis une mémoire morte (*ROM*) en mémoire vive (*RAM*) et d'exécuter son contenu.

Mon développement s'est déroulé en deux étapes :
- ajout du support de la carte électronique et
- ajout du support d'UBI[5].

1.1 Support des cartes électroniques

Ces deux nouvelles cartes électroniques sont similaires aux cartes d'évaluations proposées par *Atmel* du point de vue des composants utilisés (*RAM* et *NAND*). J'ai ré-utilisé les développements effectués par le fabriquant pour initialiser correctement ces deux mémoires, notamment en terme de timings.

3. System-on-Chip.
4. Power Management Integrated Circuit.
5. Unsorted Block Image.

1.2 Support d'UBI

Mots clés : *C*, *UBI*.

Le noyau *Linux* étant amené à évoluer durant le cycle de vie du produit, j'ai fiabilisé ses mises-à-jour contre des évènements que l'on ne peut pas maîtriser comme les coupures de courants. Pour ce faire, j'ai dupliqué tous les éléments critiques pouvant être soumis à des mises-à-jour et j'ai utilisé la technologie *UBI* pour détecter leur intégrité via les *méta-données*[6] liées à cette technologie.

Chaque élément critique est par conséquent stocké deux fois dans la partition *UBI*. Une première fois sous son propre nom (exemple : « **kernel** ») et une seconde fois sous son nom suivi du suffixe « *-spare* » (exemple : « **kernel-spare** »).

La procédure de mise-à-jour consiste à mettre à jour d'abord le volume principal, puis le volume de rechange. Ainsi, si la coupure de courant a lieu lors de la mise-à-jour du premier volume, alors on détecte sa non-intégrité grâce à la technologie *UBI* et on charge le second. Si la coupure de courant a lieu lors de la mise-à-jour du second volume, on charge quoi qu'il arrive le premier volume car celui ci est valide. La procédure de mise-à-jour doit prendre en compte une reprise sur erreur et reprendre la mise-à-jour là où elle a été interrompue.

Comme ni le *ROM code*, ni le *bootstrap* ne prennent en charge la gestion d'*UBI*, j'ai implémenté cette fonctionnalité[7].

2 Kernel et Device-Tree

Le *device-tree* est un langage qui formalise les schématiques d'une carte électronique an niveau noyau. Muni des schématiques des deux nouvelles plateformes, j'ai créé leur fichiers descriptifs en activant uniquement les périphériques utilisés. Par ailleurs, c'est dans ce fichier que l'on associe un périphérique à son *pilote* (*driver*). Grâce à ce couple périphérique/pilote, j'ai généré une configuration spéciale du noyau pour chaque plateforme afin que ce dernier n'intègre que les pilotes nécessaires à la gestion de la carte électronique.

Ces deux nouvelles cartes électroniques sont intégrées à la dernière version stable du noyau *Linux*[8] (on parle de *mainline*).

— at91-kizboxmini : `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/arch/arm/boot/dts/at91-kizboxmini.dts?id=refs/tags/v4.2`
— at91-kizbox2 : `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/arch/arm/boot/dts/at91-kizbox2.dts?id=refs/tags/v4.2`

6. Le marqueur de mise-à-jour du volume dans la table des volumes et le Contrôle de Redondance Cyclique (*CRC*).
7. Cette fonctionnalité est en cours de soumission pour être intégrée au projet. Elle est disponible via l'*URL* `https://github.com/Overkiz/at91bootstrap/commit/be87d62fef7d11f7cf77c182ec9e76f89bdc9c78`
8. A partir de la version 4.2.

Deuxième partie

Projets Personnels

3 InitRAMFS

Mots clés : *Busybox*, *OverlayFS*, *GNU/Makefile*, *Kconfig*, *crosstool-NG*, *QEMU*.

Je développe actuellement un système de fichier racine (*rootfs*) minimal axé autour du projet Busybox. Il peut être utilisé par un noyau *Linux* soit comme un « Initial RAM-Disk » soit comme un « Initial RAMFS ».

InitRAMFS n'a pas pour vocation de concurrencer Buildroot ou Yocto. Il fournit simplement système minimal pour des systèmes sans écran (*headless*). Le système ouvre une connexion via une liaison série (*login*, *tty*) et gère automatiquement les périphériques ainsi que la configuration réseau.

Le cœur du projet est architecturé autour de l'exécutable *Busybox* compilé en *statique* et de plusieurs *scripts Shell* de ma propre invention. Il tire partie de plusieurs *applets Busybox* pour remplir les tâches minimales d'un tel système. Il utilise notamment :

- ifplugd, udhcpc et zcip pour la configuration automatique du réseau via *DHCP* ou la configuration en *Link-Local*,
- mdev pour la gestion dynamique des périphériques (périphériques de stockage de masse, interfaces réseaux via *USB*...),
- syslogd et syslogd pour les messages dit de *log*.

Le but étant de solliciter un maximum d'*applet Busybox*.

InitRAMFS gère de manière dynamique le montage des périphériques de stockage de masse (comme une *clé USB*) dans le point de montage /**media**/<**nom-ou-id-du-périphérique**> grâce aux *applets* mdev et blkid et d'un *script Shell* automount.

InitRAMFS configure (et dé-configure) également dynamiquement les interfaces réseaux au branchement (et débranchement) des câbles via l'applet ifplugd. L'interface réseau obtient alors soit une adresse *IP* attribuée par le serveur *DHCP* via l'applet udhcpc, soit une adresse *IP* en *Link-Local* via l'applet zcip si aucun serveur *DHCP* n'est présent sur le *LAN*.

InitRAMFS utilise un script Shell en lieu et place du système d'init traditionnel implémenté par l'applet init.

J'ai ajouté le support d'OverlayFS afin de palier au problème de la volatilité d'un système de fichier racine en RAMFS. En effet, tout redémarrage fait perdre toutes traces de modifications.

Le système de compilation est basé sur un *Makefile* utilisant les spécificités additionnelles de *GNU/Make*. Il est donc indispensable d'utiliser *gmake* pour

construire l'*initramfs*. Par exemple, j'utilise les règles avec des « :: ».

La commande suivante génère le fichier *initramfs.cpio* : une archive *CPIO* contenant le *rootfs*. Elle est ensuite utilisée par le noyau *Linux* lors de sa compilation pour la concaténer à son image et faire du noyau un système complet en une seule et même image.

```
$ make [initramfs.cpio]
```

La commande suivante permet de générer une image du noyau complète avec son *rootfs* concaténé.

```
$ make kernel [KIMAGE=zImage]
```

InitRAMFS se destinant au domaine de l'embarqué, l'architecture cible du processeur est souvent différente de celle de la machine servant à générer l'image (*ARM*, *MIPS*...). J'ai ajouté la possibilité d'utiliser un environnement de compilation croisé via la variable *CROSS_ COMPILE* (tout comme pour le noyau *Linux*).

```
$ make CROSS_COMPILE=arm-unknown-gnueabi-
```

Si la chaine de compilation croisée (*toolchain*) n'existe pas, *InitRAMFS* utilise le projet crosstool-NG pour la générer. La commande suivante permet de la compiler.

```
$ make CROSS_COMPILE=arm-unknown-gnueabi- toolchain
```

InitRAMFS se veut utile du point de vue d'un développeur *BSP*. Le projet intègre d'autre projets comme dropbear (serveur et client *SSH*), kexec-tools (execution d'un noyau à chaud) ou encore toybox (un autre projet comme *Busybox* qui s'installe à la place ou en plus dans /**media**/**toybox**/). Ces autres outils sont également compilés en statique.

Par ailleurs, j'ai ajouté la possibilité de pouvoir configurer l'image générée avec Kconfig : le même système utilisé par le noyau, *BuildRoot* ou encore *crosstool-NG*.

```
$ make menuconfig
```

InitRAMFS supporte également l'émulation via QEMU. Je rencontre actuellement quelques problème avec la gestion du réseau via un pont. Les résolutions *DNS* ne semble pas fonctionner. Une configuration spéciale permet de se passer des droits administrateurs *root*.

```
$ make runqemu
```

4 MPKG

Mots clés : *POSIX*, *script Shell*, *grep*, *sed*, *tar*, *wget*.

En parallèle au développement d'*InitRAMFS*, je développe un système de paquet minimaliste écrit uniquement en *Shell* POSIX. *MPKG* ne nécessite aucun interpréteur tierce (ni *Python*, ni *Lua*, ni *Perl*...) ; il recquière uniquement un interpréteur *Shell* et quelques outils standards que l'on retrouve sur n'importe quel système *POSIX* minimal (*grep*, *sed*, *tar*, *wget*...). Par conséquent, il est possible de l'utiliser avec un simple système comme *Busybox*.

MPKG se veut moins complexe et plus léger que les systèmes de paquets existants (*Debian* ou *RPM*[9]) ; il est majoritairement inspiré du formalisme de *Debian* mais n'est en aucun cas compatible avec ce dernier.

Le format binaire du paquet est simple : une archive *TAR gzippée* représentant une archive des données à partir de la racine (*rootfs*). L'extension est **.tgz**. Les *méta-données* du paquet sont stockées dans un chemin spécifique : **/var/lib/mpkg/<nom-du-paquet>/**.

Ce format est à comparer à celui utilisé par *Debian* où ils ont fait le choix de créer une archive plus complexe en séparant les données utiles du paquet et des *meta-données*. Un paquet *Debian* est une archive *AR* de deux sous archives *TAR gzippées* : une pour le *rootfs* (**data.tar.gz**) et une autre pour les *méta-données* (**control.tar.gz**). Cette archive *AR* contient également un fichier supplémentaire définissant le version binaire du format du paquet (**debian-binary**).

MPKG ne gère que les fonctionnalités essentielles d'un système de paquet. A savoir :

- un nombre restreint de mots-clés pour les *meta-informations* du paquet : nom du paquet, version et la liste des dépendances.
- les scripts de pré et post installation (*preinst*/*postinst*)
- les scripts de pré et post désinstallation (*prerm*/*postrm*)

Écrire *MPKG* en *Shell POSIX* me permet d'exprimer mes connaissances dans ce langage de script, en usant du mécanisme de parallélisation (pipe) et des redirections pour le rendre plus performant.

9. RedHat Package Manager.

A Contributions

Voici une liste de mes contributions dans différents projets libres :

- **Linux** : `https://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git/log/?id=refs%2Ftags%2Fnext-20150610&qt=grep&q=PORTAY`
- **OPKG** : `http://git.yoctoproject.org/cgit/cgit.cgi/opkg/log/?qt=grep&q=PORTAY`
- **cURL** : `https://github.com/bagder/curl/commits?author=gazoo74`
- **AT91Bootstrap** :
  - `https://github.com/linux4sam/at91bootstrap/commits?author=gazoo74`
  - `https://github.com/linux4sam/at91bootstrap/pull/25`
- **Dropbear** : `https://github.com/mkj/dropbear/commits?author=gazoo74`

Ainsi que lien vers mes dépots hebergés par GitHub : `https://www.github.com/gazoo74`.