Portfolio

Gaël PORTAY

November 12, 2015

Abstract

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'experience dans le monde professionnel. Je suis passionné et curieux : j'aime apprendre et comprendre comment fonctionne les « choses ». Je suis également auto-didacte et j'aime partager mes experiences 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 système d'exploitation au niveau noyau et la partie électronique ; les 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.

Contents

Part I
Board Support Package

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

J'ai développé le logiciel bas niveau (appelé *BSP*) de deux cartes électroniques développées par Overkiz. Ces deux nouvelles platformes 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. J'ai validé le fonctionnement de l'ensemble des composants de la carte électronique : *DELs*, boutons, réseau, ports *USB*, ports séries (*UART*), bus *I2C*, bus *SPI*, mémoires (*NAND* et *RAM*) ainsi que le controlleur de gestion de l'alimentation (*PMIC*).

Le développement de ce *BSP* s'est décomposé en deux parties :

- 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 la première partie de code à s'executer après le *ROM code* du *SoC* d'Atmel. Le *ROM code* est appelé *bootloader* de niveau 1. Le *bootstrap* est quant à lui un *bootloader* de niveau 2. Mon développement s'est déroulé en deux étapes :

- ajout du support de la carte électronique et
- ajout du support d'UBI[4].

1.1 Support des cartes électroniques

Ces deux cartes électroniques sont similaires aux cartes d'évaluations proposées par Atmel du point de vue des composants utilisés (*RAM* et *NAND*). Ce fait a grandement facilité le développement[5] : j'ai pu ré-utiliser les développements déjà effectués par le fabriquant pour initiliser ces deux composants essentielles pour démarrer le noyau *Linux*.

- at91-kizboxmini : `https://github.com/Overkiz/at91bootstrap/commit/d3aec6a5ac990a29947da028f6efb38d01185a15`
- at91-kizbox2 : `https://github.com/Overkiz/at91bootstrap/commit/9dfe562afd954f07aa39c934e63ec480ddb68e02`

[3] System-on-Chip.
[4] Unsorted Block Image.
[5] `https://github.com/Overkiz/at91bootstrap/tree/overkiz`.

1.2 Support d'UBI

Mots clés : *C*, *UBI*.

J'ai utilisé la capacité d'*AT91Bootstrap* de charger et démarrer un noyau *Linux*. Cela, simplie la séquence usuelle de démarrage qui veut que l'on utilise un bootloader de niveau 3 comme u-boot ou barebox.

Le noyau *Linux* étant ammené à évoluer durant le cicle de vie du produit, j'ai été ammené a fiabilisé les mises-à-jour contre des évennements que l'on ne peut pas maitriser comme les coupures de courants. J'ai dupliqué les volumes critiques tels que le noyau et le device-tree et je les ai stockés dans des volumes UBI. Ainsi les mises-à-jours de volumes : controle de l'intégrité (CRC) et maj. J'ai donc du implémenter la gestion de UBI dans le *bootstrap*.

2 Kernel et Device-Tree

Le *device-tree* formalise an niveau noyau les schematiques d'une carte électronique dans un fichier nommé **.dts**. Je me suis muni de ces schematiques afin d'activer les périphériques utilisés par la platforme. Par ailleurs, c'est dans ce fichier que l'on associe les périphériques et leur drivers via l'attribut *compatible*. J'ai pu ensuite etre capable grâce a cette... de générer une configuration spéciale du noyau afin qu'il ne compile que les pilotes nécéssaire à la gestion de la carte électronique.

Ces deux nouvelles cartes électroniques sont intégrées à la dernière version stable noyau *Linux*[6] (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] A partir de la version 4.2.

Part II
Projets Personnels

3 InitRAMFS

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

Je developpe actuellement un système de fichier racine (*rootfs*) minimal axé autour du projet Busybox. Il peut être utilisé par un noyau Linux comme un « Initial RAM-Disk » ou comme un « Initial RAMFS ». InitRAMFS n'a pas pour vocation de concurrencer Buildroot et Yocto. Il fournit simplement une initialisation minimale d'un système sans écran, que l'on qualifie de système *headless*. InitRAMFS ouvre une connection via une liaison série (*login*, *tty*), avec une gestion automatique du réseau et de l'USB.

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* à défaut de présence de serveur *DHCP* sur le *LAN*,
- mdev pour la gestion dynamique des périphérique (périphérique de stockage de masse, interfaces réseaux via USB...),
- syslogd et syslogd pour les messages dit de *log*.

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.

Il configure (et déconfigure) également dynamiquement les interfaces réseaux au branchement (débranchement) des cables via l'applet ifplugd. L'interface obtient une addresse *IP* via *DHCP* ou en *Link-Local* en cas d'erreur.

InitRAMFS utilise un script shell en lieu et place du système d'init traditionnel implémenté par l'applet init. Pas de necessité de run-level à la sysvinit.

J'ai ajouté le support d'OverlayFS afin de palier au problème de ma volatilité d'un système de fichier racine en RAMFS : le redémarrage fait perdre toutes traces de modifications. Un système de fichier, stocké sur un support persistent, est monté par dessus le *rootfs*, puis le répertoire racine est remplacé par ce nouveau point montage combinant les deux ... le RAMFS d'initramfs et le système de fichier peresistant via l'applet chroot. Ainsi, il est possible de modifier les fichiers du rootfs comme de manière persistente. L'initialisation du système se

fait en deux étape : la première consiste init, mount-overlay.

Le sytème 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* et utilisé par le noyau Linux lors de sa compilation pour le 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 complete 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 qui sert à générer l'image (*ARM*, *MIPS*...). J'ai ajouté la possibilité d'utiliser un environnement de compilation croisé via la variable *CROSS_ COMPILE*.

```
$ 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 compiler la compiler.

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

InitRAMFS utiles point de vu d'un développeur. dropbear (serveur et client *SSH*), kexec-tools (execution d'un noyau à chaud) ainsi que toybox (un) qui s'installe à la place ou en plus de Buysbox dans /**media**/**toybox**/. Ces outils externes 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
```

Il est possible de compiler ou cross-compiler un noyau puis d'émuler le système noyau + rootfs dans Qemu (GNU/Makefile, Kconfig, scripts Shell, crosstool-ng, Qemu). Le but étant d'uiliser un maximum d'applet Busybox

InitRAMFS supporte également. Je rencontre actuellement quelques problème avec la gestion du réseau via un pont. Les résolutions. sans necessité les droit administrateurs. QEMU

```
$ make runqemu
```

4 MPKG

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

En parallèle d'*InitRAMFS*, je developpe un système de paquet minimaliste écrit uniquement en *Shell* POSIX. Le but de ce système de paquet est de nécéssiter aucun interpreteur tierce (ni interpretteur *Python*, ni *Lua*, ni *Perl*...). Seulement un interpreteur Shell et quelques outils standard d'un système POSIX (*grep*, *sed*, *tar*, *wget*...). 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) ; il est majoritairement inspiré du formalisme de Debian mais n'est pas compatible avec ce dernier. En effet, le format du paquet est plus simple : une archive *TAR gzippée* (image du *rootfs*). Les méta-données du paquets sont stockées dans un chemin spécifique : **/var/lib/mpkg/<nom-du-paquet>/**. Contrairement au paquets Debian ou le format du paquet est une archive de deux archives *TAR gzippées* : une pour le *rootfs* (data.tar.gz) et une autre pour les *méta-données* (control.tar.gz).

MPKG ne gèrer qu'un minimum de fonctionnalité d'un système de paquet à 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 desinstallation (prerm/postrm)

Ecrire *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.

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`.