Raspberry Pi from scratch – 2

Christophe Blaess

Cet article a été publié dans le numéro 155 (décembre 2012) de Gnu Linux Magazine France

Pour maîtriser et ajuster parfaitement la configuration de votre Raspberry Pi, je vous propose de construire un système entièrement personnalisé en partant de zéro. Dans le précédent article nous avons préparé une chaîne de compilation, installé les fichiers nécessaires pour le bootloader, compilé et copié le noyau Linux et démarré notre système. Ce qui s'est terminé sur un message d'erreur fatale. À présent nous allons construire un système de fichiers et y inscrire les applications utilisateur nécessaire pour naviguer dans notre environnement.

Kernel panic : no init found, try passing init=...

Comme je l'indiquais en fin d'article précédent, ce message est, de manière assez surprenante, une excellente nouvelle puisqu'il indique que nous avons parfaitement réussi la première phase de mise au point de notre système embarqué. À la mise sous tension, le processeur a chargé les fichiers du bootloader et les a exécutés. Ces derniers ont cherché un fichier nommé **`kernel.img`** (qui est une copie renommée du fichier **`arch/arm/boot/zImage`** produit lors de notre compilation du noyau Linux) et l'ont placé en mémoire avant de lui transmettre le contrôle.

On peut suivre le détail du boot de Linux en examinant la fonction **`start_kernel()`** se trouvant dans le fichier **`init/main.c`** des sources du noyau. Un exercice intéressant consiste à comparer les traces du kernel (obtenues par exemple sur la console série comme décrit dans l'article précédent) et les routines invoquées par **`start_kernel()`** ; on arrive à repérer les étapes principales et à comprendre la succession des opérations réalisées.

Au moment où se produit le « kernel panic » rencontré, le noyau se trouve dans l'état suivant :

- toutes les tables et structures de données internes utiles au fonctionnement du kernel ont été initialisées ;

- le CPU, la mémoire, les interruptions ont été détectés et configurés correctement ;

- les protocoles réseau (par exemple TCP/IP ou UDP/IP) sont prêts à être utilisés ;

- les fonctions d'initialisation des drivers compilés statiquement (pas en modules), dont le matériel a été reconnus, ont été invoquées ;

- le kernel a monté (en lecture seule) le système de fichiers dont le nom a été fourni dans le paramètre de démarrage **`root=/dev/mmcblk0p2`**. Ici il s'agit de la seconde partition de la carte SD du Rasberry Pi. Nous l'avions précédemment formatée en utilisant un système de fichiers **`ext2`**, ce qui est indiqué (bien que cela ne soit pas indispensable) dans le paramètre **`rootfstype=ext2`**.

Pour poursuivre l'initialisation du système, le noyau doit lancer le processus **`init`** (de PID 1). Pour cela, il cherche son fichier exécutable successivement sur **`/sbin`**, **`/etc`** et **`/bin`**. Ceci est parfaitement visible dans la routine **`init_post()`** du fichier **`init/main.c`** mentionné plus haut. Nous allons devoir créer ces répertoires.

> Quel type de formatage utiliser pour l'embarqué ? Il existe plusieurs types de systèmes de fichiers spécialement conçus pour optimiser l'emploi des mémoires flash. Par exemple *Jffs2* ou le plus récent *Ubifs*. Toutefois, ces formats s'utilisent sur des mémoires flash sans contrôleur (prises généralement en charge par le sous-système MTD) et non pas sur des périphériques blocs comme les cartes SD qu'emploie le Rasberry Pi. Pour ces dernières mémoires, on utilise plutôt des systèmes de fichiers classiques comme *Vfat* (qui présente l'inconvénient de ne pas préserver correctement les droits), ou la famille *Ext2/3/4*. C'est cette dernière que j'ai choisie en prenant le système le plus simple (*Ext2*) car nous n'avons aucun besoin de la journalisation offerte par Ext3, ou des extensions proposées par *Ext4*.

Arborescence des fichiers

L'outil Buildroot présenté dans l'article précédent serait tout à fait capable de nous préparer une image complète de l'arborescence des fichiers, contenant tout ce que nous allons construire ici – et même plus – mais je trouve qu'à titre pédagogique (voire ludique) il est plus intéressant de réaliser manuellement l'ensemble de la mise en œuvre de notre système.

Si nous insérons à nouveau la carte SD sur le PC de développement, la partition contenant la racine de l'arborescence est accessible sous **`/media/Root`**. Pour l'instant elle ne contient que le répertoire **`lost+found`** créé automatiquement lors du formatage en **`ext2`** (qui sert lors de la vérification du système de fichiers).

Créons tout d'abord les répertoires principaux :

```
[RaspberryPi]$ cd   /media/Root
[Root]$ ls
lost+found
[Root]$ sudo  mkdir  bin  dev  etc  home  lib  mnt  proc  root  sbin  sys  tmp  usr  var
```

Puis des sous-répertoires qui nous serviront pour la suite :

```
[Root]$ sudo  mkdir  dev/pts  etc/init.d
usr/bin  usr/sbin
[Root]$ cd  -
[RaspberryPi]$
```

La partition étant formatée en **ext2**, le noyau de notre système de développement prend en considération les propriétés et les droits sur les fichiers (contrairement à la partition formatée en **vfat** que nous avons utilisée pour le boot). C'est pour cela qu'il est généralement nécessaire de les manipuler avec les droits *root* par l'intermédiaire de la commande **sudo**.

Outre les répertoires habituels que l'on trouve sur tous les systèmes Linux, nous avons créé :

- **/dev/pts** : un point de montage pour le système de fichiers virtuel **devpts**, qui sert à obtenir des pseudo-terminaux esclave, par exemple, pour **telnet** ou **ssh**.

- **/etc/init.d** : ce répertoire contiendra un script de configuration pour le processus **init** que nous installerons plus loin.

Modules du noyau

Nous avons compilé un kernel contenant tous les drivers nécessaires pour démarrer. Toutefois certains éléments ont été compilés sous forme de modules car nous n'en aurons pas systématiquement besoin. Pour que les outils classiques de chargement (**modprobe** par exemple) les trouvent, ces modules doivent être installé dans un emplacement bien précis de l'arborescence : **/lib/modules/<numero-du-noyau>/**.

Cette installation se fait automatiquement depuis le répertoire des sources du noyau après compilation :

```
[RaspberryPi]$ cd   linux-raspberrypi
[linux-raspberrypi]$ make   ARCH=arm   INSTALL_MOD_PATH=/media/Root   modules_install
  [...]
[linux-raspberrypi]$ ls /media/Root/lib/modules/3.1.9-glmf+/
build                modules.dep.bin      modules.seriomap
kernel               modules.devname      modules.softdep
modules.alias        modules.ieee1394map  modules.symbols
modules.alias.bin    modules.inputmap     modules.symbols.bin
   [...]
[linux-raspberrypi]$ cd  ..
[RaspberryPi]$
```

Processus init

Nous pourrions parfaitement écrire notre propre processus **init**. Dans le cas d'un système embarqué très limité c'est une solution tout à fait légitime que j'ai d'ailleurs rencontrée lors du portage de logiciels initialement écrits pour des micro-contrôleurs. Néanmoins, il est souvent préférable d'utiliser un **init** « standard » car il faut qu'il remplisse deux rôles différents :

- pour terminer le boot, **init** doit réaliser certaines tâches (monter les systèmes de fichiers spéciaux comme **/proc**, remonter la racine du système de fichiers en lecture-écriture, charger les modules supplémentaires du kernel, initialiser les interfaces réseaux, etc.) qu'il est d'usage de sous-traiter à un script shell ;

- pendant le fonctionnement normal du système, **init** doit lire les états de terminaison des processus orphelins (dont le processus père s'est terminé auparavant), ce qu'il réalise à l'intérieur d'un petit *handler* pour le signal **SIGCHLD** (qui lui est envoyé automatiquement par le kernel dans ce cas).

Nous utiliserons donc de préférence un processus **init** classique qui nous permettra d'ajuster les actions d'initialisation du système dans un script shell. Le plus simple sera d'utiliser la version d'**init** incluse dans l'outil **busybox**.

Compilation et installation de Busybox

Le projet Busybox a pour objet de regrouper en un seul fichier exécutable l'essentiel des commandes Unix (environ 400) utiles au quotidien. Ceci simplifie la mise en œuvre d'un système embarqué – en évitant de gérer des dizaines de packages différents – d'autant que l'implémentation des commandes est faite avec un souci d'économie de taille de code, en supprimant les options rarement utiles ou en les rendant désactivables à la compilation. On retrouve Busybox sur la plupart des configurations à base de Linux embarqué, parfois ajusté pour ne contenir qu'une poignée de commandes indispensables.

Le principe de fonctionnement de l'utilitaire **busybox** est simple : lorsqu'il démarre, il examine la ligne de commande qui a servi à l'invoquer – dans le tableau **argv[]** de sa fonction **main()** – afin de savoir sous quel nom il a été appelé et quels sont ses arguments.

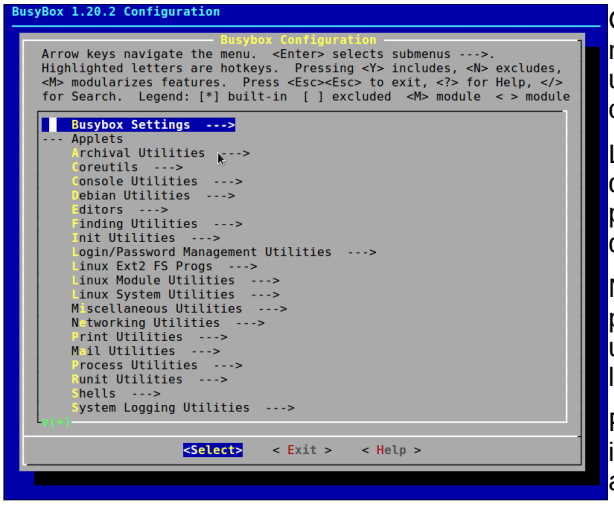
- si on l'a invoqué sous le nom **busybox** sans autres arguments, il affiche sur la sortie standard la liste des commandes (*applets*) qu'il intègre ;

- si il a été appelé sous le nom **busybox** avec des arguments, il attend en première position le nom de l'applet à invoquer, par exemple « **$ busybox ls -l** » auquel cas il se comporte comme la commande **ls** en prenant en considération l'argument **-l** ;

- enfin, si le fichier exécutable **busybox** a été invoqué – via un lien physique ou symbolique – sous un autre nom (par exemple **ls**), il se comporte comme la commande correspondante.

Nous allons compiler Busybox en intégrant un nombre important d'applets, afin de disposer d'un système confortable sur notre Raspberry Pi. Toutefois, il est possible de réduire fortement leur liste si on souhaite construire un environnement plus restreint. Commençons par télécharger la dernière version de Busybox et préparer la compilation :

```
[RaspberryPi]$ wget   http://busybox.net/downloads/busybox-1.20.2.tar.bz2
        [...]
[RaspberryPi]$ tar   xjf   busybox-1.20.2.tar.bz2
[RaspberryPi]$ cd   busybox-1.20.2/
[busybox-1.20.2]$ wget   http://www.blaess.fr/christophe/files/glmf/rpi-scratch-02/config-busybox-1.20.2
[busybox-1.20.2]$ mv   config-busybox-1.20.2   .config
[busybox-1.20.2]$ make   menuconfig
```



Comme nous en avons l'habitude pour la compilation d'un noyau Linux, la commande **make menuconfig** propose une interface semi-graphique pour éditer la configuration contenue dans le fichier **.config**.

L'étendue des commandes implémentées dans les dernières versions de Busybox est impressionnante, on pourra s'en rendre compte en parcourant les sous-menus de configuration.

Notons simplement pour l'instant que nous trouverons un processus **init**, un shell **sh**, et de nombreux utilitaires usuels (**ls**, **cp**, **mv**, etc. et même une version simple de l'éditeur **vi**).

Pour compiler Busybox une fois la configuration terminée il faut préciser l'emplacement du *cross-compiler* que nous avons construit avec Buildroot dans l'article précédent :

```
[busybox-1.20.2]$ make   CROSS_COMPILE=/usr/local/cross-rpi/usr/bin/arm-linux-
  [...]
  DOC      BusyBox.txt
  DOC      busybox.1
  DOC      BusyBox.html
[busybox-1.20.2]$ ls -l busybox
-rwxrwxr-x 1 cpb cpb 887704 juil. 15 18:59 busybox
```

Nous devons maintenant installer ce fichier binaire dans les répertoires de notre cible, et créer les liens

symboliques avec les noms de toutes les applets incluses lors de la configuration. Ceci serait extrêmement fastidieux, aussi laisserons-nous Busybox réaliser ce travail lui-même en invoquant

```
[busybox-1.20.2]$ sudo   make   CROSS_COMPILE=/usr/local/cross-rpi/usr/bin/arm-linux-
install
```

Le point de montage où se trouve la carte SD (**/media/Root**) a été indiqué lors de la configuration de Busybox (menu « *Busybox Settings* », sous-menu « *Installation Options* », Option « *Busybox Installation Prefix* »). Il est très important de bien le configurer avant de lancer la commande d'installation. Vérifions le résultat :

```
[busybox-1.20.2]$ ls /media/Root/bin/
[               dumpkmap    kill           powertop      tar
[[              dumpleases  killall        printenv      tcpsvd
addgroup        echo        killall5       printf        tee
[...]
dnsdomainname   iprule      pipe_progress  sync
dos2unix        iptunnel    pkill          tac
du              kbd_mode    pmap           tail
[busybox-1.20.2]$ ls /media/Root/sbin/
acpid       freeramdisk  logread     pivot_root         switch_root
adjtimex    fsck         losetup     poweroff           sysctl
arp         fsck.minix   lsmod       raidautorun        syslogd
[...]
fbset       insmod       nanddump    sulogin            zcip
fbsplash    klogd        nandwrite   svlogd
fdisk       loadfont     nbd-client  swapoff
findfs      loadkmap     ntpd        swapon
[busybox-1.20.2]$
```

Bibliothèques

Il est possible de demander à Busybox de créer automatiquement les liens sur lui-même lorsqu'il démarre, en ajoutant dans le script de démarrage la commande « **busybox --install** ». Ceci nécessite quand même de créer manuellement les liens **init** et **sh**.

Lors de la configuration de Busybox, on peut choisir (dans le menu « *Busybox Settings* », sous-menu « *Build Options* ») d'activer l'option « *Build Busybox as a static binary* » afin que le fichier exécutable embarque toutes les fonctions dont il a besoin et soit totalement autonome. Ou au contraire, il est possible de désactiver cette option pour que le fichier exécutable emploie des bibliothèques partagées.

La première option est intéressante si le système ne comporte qu'un seul exécutable (ce qui sera le cas pour le moment) et qu'aucun ajout ultérieur d'autres programmes n'est envisagé. Cela évitera d'avoir à gérer l'installation des bibliothèques dynamiques sur la cible.

À l'inverse, dès qu'on envisage d'installer plusieurs fichiers exécutables, il est préférables que le code qu'ils partagent ne soit pas dupliqué sur la cible mais soit partagé dans des bibliothèques dynamiques. Ceci permet en outre de mettre à jour les bibliothèques (corrections, améliorations, etc.) sans avoir besoin de recompiler les exécutables.

Ici, j'ai choisi de compiler Busybox afin qu'il s'appuie sur des bibliothèques dynamiques. Mais il est nécessaire de savoir lesquelles sont nécessaires et où les trouver. Toutes les bibliothèques disponibles pour la cible ont été compilées en même temps que la *cross-toolchain* obtenue avec Buildroot. Il y en a une vingtaine, installées dans **/usr/local/cross-rpi/usr/arm-linux/sysroot/lib**. Pour savoir quelles sont les bibliothèques nécessaires pour notre exécutable, nous pouvons invoquer l'utilitaire **arm-linux-ldd** de la toolchain, néanmoins je préfère toutes les copier sur la cible pour qu'elles soient disponibles lorsque nous ajouterons des applications ultérieurement.

```
[busybox-1.20.2]$ cd   /media/Root/lib/
[lib]$ cp   /usr/local/cross-rpi/usr/arm-linux/sysroot/lib/*   .
```

Fichiers de /dev

Le répertoire **/dev** contient par convention les fichiers spéciaux représentant les périphériques en mode caractère ou bloc. Nous pourrions créer ces fichiers avec la commande **mknod** en nous basant sur les numéros de périphériques d'un système existant ou en consultant le fichier **Documentation/devices.txt** des sources du noyau. Toutefois une solution beaucoup plus élégante existe, qui présente en outre l'avantage d'être compatible avec les drivers modernes qui demandent au noyau de leur attribuer dynamiquement un numéro majeur.

Nous allons invoquer dans notre script de démarrage l'utilitaire **mdev** contenu dans Busybox, qui va se charger de créer dans **/dev** les fichiers spéciaux pour les périphériques déjà identifiés. Ceci s'obtient ainsi :

```
/sbin/mdev  -s
```

En outre, pour les périphériques non encore détectés (parce que leurs drivers sont compilés en modules ou parce qu'ils ne sont pas encore reliés à la carte), nous allons demander au noyau d'invoquer automatiquement **mdev** lorsqu'il les identifie, en inscrivant :

```
echo   /sbin/mdev   >  /proc/sys/kernel/hotplug
```

Il nous faudra ajouter des sous-répertoires dans **/dev** ultérieurement, mais pour le moment nous pouvons laisser ce répertoire vide.

Fichier de configuration

Pour notre première version, la plus simple possible, nous n'allons fournir au système qu'un seul fichier de configuration : **/etc/inittab** qui permet de configurer le comportement du processus **init** (sur les distributions courantes, ce fichier, hérité des Unix Système V tend à disparaître au profit de **systemd**). Notre fichier va contenir les lignes suivantes :

```
::sysinit:/etc/init.d/rcS
tty1::respawn:/bin/sh
tty2::respawn:/bin/sh
tty3::respawn:/bin/sh
tty4::respawn:/bin/sh
ttyAMA0::respawn:/bin/sh
```

La première ligne indique à **init** qu'il doit exécuter au moment du démarrage le script se trouvant dans **/etc/init.d/rcS**. Notez qu'il s'agit du comportement par défaut du **init** de Busybox, même en l'absence de fichier **inittab**.

Les quatre lignes suivantes lancent directement un shell sur les quatre consoles virtuelles du Raspberry Pi, entre lesquelles on peut commuter avec les touches Alt-F1, Alt-F2, Alt-F3 et Alt-F4. La dernière ligne lance également un shell sur le port série RS-232 dont nous avons parlé dans l'article précédent, et sur lequel nous avons jusqu'à présent lu les messages de boot du kernel.

Remarquez bien qu'il n'y a aucune identification des utilisateurs. Dès le démarrage les consoles sont directement accessibles, sans authentification avec les droits *root*. Cette approche est courante du moins durant la phase de mise au point, dans les systèmes embarqués ou – à l'inverse des postes de bureautique – on travaille souvent sous l'identité *root*. Nous nous intéresserons à la sécurisation de ces accès ultérieurement.

Pour copier le script **inittab** sur la cible, nous pouvons le télécharger ainsi :

```
[lib]$ cd   /media/Root/etc/

[etc]$ wget   http://www.blaess.fr/christophe/files/glmf/rpi-scratch-02/inittab
```

Script de démarrage

Enfin, écrire un script de démarrage est la dernière action à réaliser avant de booter notre système personnalisé. Ce script doit au minimum remplir les rôles suivants :

- monter les systèmes de fichiers spéciaux **/proc** et **/sys**,

- remonter l'arborescence principale en lecture-écriture,

- appeler **mdev** pour remplir le répertoire **/dev**,

Nous rajouterons d'autres tâches par la suite.

Voici le contenu du script :

```
#! /bin/sh

mount none /proc -t proc
mount none /sys  -t sysfs
mount / -o remount,rw

/sbin/mdev -s
echo /sbin/mdev >/proc/sys/kernel/hotplug
```

On peut le télécharger directement :

```
[etc]$ cd   /media/Root/etc/init.d

[init.d]$ wget   http://www.blaess.fr/christophe/files/glmf/rpi-scratch-02/rcS
```

Attention à ne pas oublier de le rendre exécutable avec

```
[init.d]$ chmod   +x   rcS
```

Test du système

Après avoir proprement démonté la carte SD, et l'avoir insérée dans son support sur le Raspberry Pi, nous pouvons procéder à un premier boot. Vous avez deux possibilités pour accéder à votre système :

- brancher un écran et un clavier (la souris n'est pas encore utile) sur le Raspberry Pi *via* les interfaces HDMI et USB ;

- brancher un câble sur le port série et utiliser un émulateur de terminal (comme **minicom**) sur votre poste de travail.

Dans un cas comme dans l'autre, vous devriez voir défiler les messages

```
Uncompressing Linux... done, booting the kernel.
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 3.1.9-glmf+ (cpb@Station-CPB) (gcc version 4.5.3 (Buildroot
2012.05) ) #20 PREEMPT Sat Jul 21 09:31:22 CEST 2012
[…]
[    4.785931] usb 1-1.1: New USB device found, idVendor=0424, idProduct=ec00
[    4.805595] usb 1-1.1: New USB device strings: Mfr=0, Product=0, SerialNumber=0
[    4.825187] smsc95xx v1.0.4
[    4.901356] smsc95xx 1-1.1:1.0: eth0: register 'smsc95xx' at usb-bcm2708_usb-1.1,
smsc95xx USB 2.0 Ethernet, b8:27:eb:33:01:2b
/ #
```

Il arrive parfois que le *prompt* soit affiché avant la reconnaissance du contrôleur USB-Ethernet. Dans ce cas, pressez simplement Entrée pour l'afficher à nouveau. Explorons un peu notre environnement :

```
/ # ls
bin         home        lost+found  root        tmp
dev         lib         mnt         sbin        usr
etc         linuxrc     proc        sys         var
```

Nous pouvons vérifier que nous trouvons bien dans **/bin**, **/etc**, **lib/** et **/sbin** les fichiers que nous avons inscrits sur notre carte. Voyons les fichiers spéciaux créés par **mdev** :

```
/ # ls /dev/
1-1.1               ram13               tty23               tty55
1-1.3               ram14               tty24               tty56
[…]
ram11               tty21               tty53
ram12               tty22               tty54
```

Puis observons quelques informations système :

```
/ # cat /proc/cpuinfo
```

```
Processor       : ARMv6-compatible processor rev 7 (v6l)
BogoMIPS        : 697.95
Features        : swp half thumb fastmult vfp edsp java tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xb76
CPU revision    : 7

Hardware        : BCM2708
Revision        : 0002
Serial          : 00000000a033012b
```

Le processeur contient un cœur appartenant à l'architecture ARMv6 (ARM 1176JZF-S) sur un *System-On-Chip* BCM2708.

```
/ # free
             total         used         free       shared      buffers
Mem:        223072         5440       217632            0          188
-/+ buffers:               5252       217820
Swap:            0            0            0
```

Il y a environ 224 Mio libres, ce qui correspond au choix que nous avons fait dans l'article précédent pendant l'installation du *bootloader*.

```
/ # ps
PID   USER     TIME   COMMAND
    1 0          0:01 init
    2 0          0:00 [kthreadd]
    3 0          0:00 [ksoftirqd/0]
[…]
   31 0          0:00 [mmcqd/0]
   36 0          0:00 [flush-179:0]
   39 0          0:00 /bin/sh
   40 0          0:00 /bin/sh
   43 0          0:00 /bin/sh
   46 0          0:00 /bin/sh
   49 0          0:00 /bin/sh
   59 0          0:00 ps
/ #
```

Nous voyons qu'outre les threads du kernel (entre crochets), il y a un processus **`init`** et cinq shells (sur les quatre consoles et sur le port série).

Configuration du clavier

Si vous utilisez votre Raspberry Pi via la liaison série, tout se passe correctement en principe. En connexion directe avec un clavier USB et un écran HDMI vous avez sûrement eu la surprise de voir que votre clavier est reconnu suivant une organisation Qwerty et non Azerty.

Pour configurer le clavier correctement, il faut appeler, au sein du script de démarrage, la commande **`loadkmap`** de Busybox en lui envoyant sur son entrée standard le contenu d'un fichier de configuration. Mais comment obtenir ce fichier ? Simplement en invoquant l'applet **`dumpkmap`** de Busybox après l'avoir recompilé sur un système où le clavier est configuré correctement (par exemple votre PC de développement).

Insérons donc à nouveau la carte SD sur le PC et exécutons :

```
[~]$  sudo    busybox    dumpkmap  >  azerty.kmap
[~]$  sudo    cp   azerty.kmap    /media/Root/etc/
```

Si vous préférez télécharger un fichier **azerty.kmap** tout prêt exécutez plutôt :

```
[~] $  cd   /media/Root/etc/
[etc]$  wget   http://www.blaess.fr/christophe/files/glmf/rpi-scratch-02/azerty.kmap
```

Puis ré-éditons **etc/init.d/rcS** pour ajouter la ligne suivante en fin de fichier :

```
loadkmap < /etc/azerty.kmap
```

Nous pouvons redémarrer le Raspberry Pi et constater que le clavier est bien Azerty.

Arrêt du système

À partir de maintenant, vous disposez d'un système Linux complet, dont la partition principale est montée en lecture-écriture, aussi est-il dangereux de l'éteindre « brutalement » en coupant l'alimentation (certaines écritures sont différées de plusieurs dizaines de secondes).

Pour arrêter proprement votre système Raspberry Pi, il faudra appeler la commande **halt** de Busybox (qui tuera tous les processus actifs et invoquera **sync** pour enregistrer les données contenues dans le cache du disque). J'ai même tendance dans ce cas à remonter le système de fichiers principal en lecture-seule en exécutant :

```
/ # mount    /   -o  ro,remount   ;   halt
Terminated
[ 8558.087640] System halted.
```

Il m'est arrivé de préparer des systèmes embarqués où l'alimentation risquait d'être interrompue inopinément et où il ne fallait pas perdre de données en cas d'arrêt brutal. Pour cela, la racine du système de fichiers était montée en lecture-seule et les répertoires **/tmp**, **/var**, etc. étaient créés sur une partition en mémoire (*ramdisk*) formatée au boot. Le répertoire **/home** dans lequel des données cruciales étaient collectées se trouvait également en mémoire mais était synchronisé toutes les trente secondes avec une partition sur carte SD montée temporairement en lecture-écriture. Pour être exact, la synchronisation se faisait alternativement sur deux partitions différentes, afin de disposer dans le pire des cas (coupure d'alimentation pendant une écriture corrompant la partition) des données vieilles de trente secondes avant le crash sur l'autre partition.

Conclusion

Nous avons mis au point un système Linux embarqué minimal. On peut noter la rapidité du boot, puisqu'il s'écoule à peine six secondes entre la mise sous tension et l'accès au shell. Cette durée peut encore être réduite à cinq secondes si l'on supprime les traces du kernel (en désactivant l'option « *Enable support for printk* » du sous-menu « *Configure standard kernel features* » dans le menu « *General setup* » de configuration du noyau). Toutefois, l'absence de message du kernel est risquée car aucune information ne sera disponible en cas de problème au boot ; cette solution ne doit être envisagée qu'au moment du basculement en phase de production.

Les opérations réalisées dans cet article n'étaient pas très spécifiques au Raspberry Pi, mais s'appliquent à beaucoup d'autres environnements Linux embarqué. Il nous reste encore plusieurs domaines à approfondir sur le Raspberry Pi : la configuration réseau Ethernet, l'authentification des utilisateurs, le son, l'environnement graphique, les entrées-sorties par GPIO... Ceci fera l'objet du prochain article.

Pour en savoir plus

- Le site de documentation principale (officielle et officieuse) sur le Raspberry Pi se trouve à l'adresse : **http://elinux.org/R-Pi_Hub** ;

- Je vous conseille la lecture de la quatrième édition du « *Linux embarqué* » de Pierre Ficheux et Eric Bénard, référence incontournable du domaine ;

- Enfin, on peut trouver sur mon blog à l'adresse **http://christophe.blaess.fr** différents articles sur des systèmes Linux embarqués et temps réel (notamment Pandaboard et Igep v2).

Christophe Blaess – christophe@blaess.fr