

Raspberry Pi from scratch - 1

Christophe Blaess

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

Le succès du petit système Raspberry Pi n'est plus à démontrer. Alliant un prix de revient modique et un potentiel informatique prometteur, il s'impose comme une base expérimentale incontournable pour Linux embarqué. Mais il est dommage de se contenter d'utiliser des images ou des packages précompilés sur cette plate-forme dédiée aux hackers... Je vous propose de construire votre système entièrement personnalisé en partant de zéro.

Découverte du Raspberry Pi

Dans le courrier du matin une enveloppe blanche dépasse, un peu plus épaisse que les autres et portant la mention « *Royal Mail International* ». Expéditeur : Farnell. À l'intérieur une petite boîte en carton contenant la fameuse carte Raspberry Pi au format d'une carte de crédit. Pas de notice ni de documentation, toutes les informations seront à chercher sur Internet, essentiellement dans des forums ou des wikis.

Mon premier réflexe est évidemment de vérifier si la carte fonctionne. Je branche donc dans le connecteur micro-USB le câble du chargeur de mon téléphone. Une LED rouge s'allume instantanément et aucune autre activité ne se manifeste. Le Raspberry Pi ne contient pas de système d'exploitation intégré (contrairement par exemple des *Beagleboard*, *Pandaboard*, *Igep*, etc. qui embarquent d'origine un petit système Linux). Il n'y a d'ailleurs pas de mémoire flash accessible et nous devons ajouter une petite carte SD contenant tout le système d'exploitation.



Pour être honnête, je dois avouer que j'ai téléchargé à ce stade une image *Arch Linux Arm* sur <http://www.raspberrypi.org/downloads> que j'ai copiée sur une carte SD pour vérifier le bon fonctionnement de ma carte, mais j'ai rapidement décidé de l'effacer et de reconstruire le système en repartant de zéro.

Chaîne de compilation croisée

Le premier choix qui se pose lorsque l'on doit se lancer dans un projet Linux embarqué est celui de la chaîne de compilation. Nous devons disposer d'une **chaîne de compilation croisée** (*cross toolchain*) fonctionnant dans notre environnement de travail habituel – disons un PC sous Linux – et capable de produire du code fonctionnant sur notre plate-forme cible, ici le Raspberry Pi. Il y a plusieurs possibilités pour obtenir une chaîne de compilation croisée, mais toutes emploient les utilitaires de la *Gnu Compiler Collection*.

Tout d'abord de nombreuses distributions Linux proposent des packages précompilés de toolchain produisant du code pour des processeurs différents. La famille Arm ne fait pas exception et on trouve facilement une cross-toolchain pour processeur Arm générique sur la plupart des distributions. Néanmoins je trouve qu'il est encore plus intéressant de compiler soi-même sa propre chaîne de compilation, d'autant que nous pourrions préciser le processeur cible et obtenir ainsi une meilleure optimisation des bibliothèques.

Si nous devons prendre tous les packages nécessaires pour générer la cross-toolchain et les compiler manuellement, cela serait non seulement très long, mais également très fastidieux, car pour résoudre de nombreuses incompatibilités entre les différentes versions des packages, on a recours à un nombre incalculable de *patches* à appliquer avant compilation. Heureusement certains outils sont capables de sélectionner les bonnes versions de chaque package et de préparer la chaîne de compilation

Que recouvre le terme de « chaîne de compilation croisée » ? Cela inclut un ou plusieurs compilateurs (par exemple **gcc**, **g++**, **gnat**, etc.) des outils de manipulations des fichiers binaires (éditeur de liens **ld**, assembleur **as**, archiveur **ar**, etc.) ainsi que des bibliothèques pour la plate-forme cible (dont la **libc**). On ajoute en général un débogueur à distance comme **gdb** et **gdbserver**.

pour nous. Citons par exemple *Crosstool*, *Crosstool-NG*, *Scratchbox*, *OpenEmbedded* ou un autre environnement qui a ma préférence : **Buildroot**.

À l'inverse de *Crosstool-NG* par exemple qui ne fait que préparer la chaîne de compilation, *Buildroot* peut aller beaucoup plus loin en compilant un noyau Linux, un *bootloader*, des utilitaires système ; il peut même préparer une arborescence de fichiers sous forme d'image prête à flasher sur la cible ! Toutefois cet aspect ne m'intéresse pas ici, je veux simplement obtenir la cross-toolchain.

Notons que le choix de *Buildroot* pour préparer la chaîne de compilation n'est pas anodin : cela implique que nous utiliserons une bibliothèque C spécialement conçue pour l'embarqué : la **µClibc**. Cette bibliothèque (créée initialement pour le projet *µClinux*) est une implémentation condensée de bibliothèque C. À la différence de la *Gnu Glibc*, elle n'embarque pas systématiquement de fonctionnalités de paramétrage dynamique de son comportement par des variables d'environnement, ni de code de débogage ou de diagnostic, ni de messages internationalisés, de caractères larges multi-octets, etc. Si l'on préfère incorporer dans la chaîne de compilation une *Glibc* complète (ou sa version allégée *eGlibc*) on se tournera plutôt vers l'outil *Crosstool-NG*.

Pourquoi insister ainsi sur la bibliothèque C ? Celle-ci est le point de passage obligé pour les applications de l'espace utilisateur qui désirent avoir recours à un service du noyau en invoquant un **appel système** (pour écrire dans un fichier, connecter une socket, allouer de la mémoire, s'endormir quelques millisecondes, etc.). L'implémentation de l'appel système se trouve dans la bibliothèque C : il s'agit souvent d'une trappe (interruption logicielle) qui transmet le contrôle au kernel pour qu'il exécute le service demandé. La bibliothèque C est donc un élément indispensable pour tout système embarqué, que l'on installera immédiatement après le noyau Linux.

Utilisation de Buildroot

Notre utilisation de *Buildroot* sera minimale au regard de ses possibilités. Commençons par télécharger sa dernière version et décompresser l'archive :

```
[~]$ mkdir RaspberryPi
[~]$ cd RaspberryPi
[RaspberryPi]$ wget http://buildroot.uclibc.org/downloads/buildroot-2012.05.tar.bz2
[...]
[RaspberryPi]$ tar xjf buildroot-2012.05.tar.bz2
[RaspberryPi]$ cd buildroot-2012.05/
```

Une fois dans le répertoire de *Buildroot*, je vous encourage à télécharger le fichier de configuration que j'ai préparé, et à le renommer sous le nom **.config** ainsi :

```
[RaspberryPi]$ wget http://www.blaess.fr/christophe/files/glmf/rpi-scratch-01/config-buildroot
[...]
[RaspberryPi]$ mv config-buildroot .config
```

Il est possible d'examiner et de modifier cette configuration avec :

```
[RaspberryPi]$ make menuconfig
```

Le fichier de configuration demande la création d'une toolchain optimisée pour processeur **Arm 1176-jzf-s**, avec compilateurs C et C++. Elle sera installée dans le répertoire **/usr/local/cross-rpi** ce qui signifie qu'il vous faudra les droits *root* pour lancer la compilation. Si cela pose problème, vous pouvez choisir d'installer la *toolchain* dans un dossier personnel en modifiant l'option « *Host Dir* » du menu « *Build Options* » (attention, il faut indiquer un chemin absolu depuis la racine de l'arborescence des fichiers).

Buildroot est configuré pour ne rien compiler d'autre que cette toolchain (mais rien ne vous empêchera par la suite de lui en demander plus...). On lance la compilation avec :

```
[RaspberryPi]$ sudo make
```

Après quelques minutes, nous pouvons vérifier l'installation de la chaîne de compilation en appelant **arm-linux-gcc**, qui fonctionne sur notre PC et produit du code au format Arm.

```
[buildroot-2012.05]$ /usr/local/cross-rpi/usr/bin/arm-linux-gcc -v
Utilisation des specs internes.
COLLECT_GCC=/usr/local/cross-rpi/usr/bin/arm-linux-gcc
COLLECT_LTO_WRAPPER=/usr/local/cross-rpi/usr/libexec/gcc/arm-unknown-linux-uclibcgnueabi/4.5.3/lto-wrapper
Target: arm-unknown-linux-uclibcgnueabi
Configuré avec: /home/cpb/RaspberryPi/buildroot-2012.05/output/toolchain/gcc-4.5.3/configure --prefix=/usr/local/cross-rpi/usr --build=i686-pc-linux-gnu --host=i686-pc-linux-gnu --target=arm-unknown-linux-uclibcgnueabi --enable-languages=c,c++ --with-sysroot=/usr/local/cross-rpi/usr/arm-unknown-linux-uclibcgnueabi/sysroot --with-build-time-tools=/usr/local/cross-rpi/usr/arm-unknown-linux-uclibcgnueabi/bin --disable-cxa_atexit --enable-target-optspace --disable-libgomp --with-gnu-ld --disable-libssp --disable-multilib --enable-tls --enable-shared --with-gmp=/usr/local/cross-rpi/usr --with-mpfr=/usr/local/cross-rpi/usr --with-mpc=/usr/local/cross-rpi/usr --enable-threads --disable-decimal-float --with-float=soft --with-abi=aapcs-linux --with-
```

```
arch=armv6zk --with-tune=arm1176jzf-s --with-pkgversion='Buildroot 2012.05' --with-
bugurl=http://bugs.buildroot.net/
Modèle de thread: posix
gcc version 4.5.3 (Buildroot 2012.05)
[buildroot-2012.05]$ cd ..
[RaspberryPi]$
```

Compilation du kernel

L'élément probablement le plus spécifique d'une plate-forme embarquée est le noyau Linux. Contrairement aux kernels fournis avec les distributions pour postes de travail ou serveurs, nous ne voulons pas d'un noyau générique capable de fonctionner sur une multitude de machines différentes mais d'une configuration bien ajustée, contenant tous les drivers, protocoles, systèmes de fichiers indispensables sans en ajouter plus que nécessaire.

La petite complication de cette étape vient du fait que le noyau Linux standard ne dispose pas encore de support pour le Raspberry Pi. Deux possibilités s'offrent à nous : télécharger un noyau standard et lui appliquer une dizaine de patches pour ajouter le support nécessaire ou utiliser les sources d'un noyau contenant déjà les drivers adaptés. La seconde solution est la plus simple, mais il existe plusieurs versions disponibles : celle officielle de la Raspberry Pi Foundation (un noyau 3.1.9 au moment de la rédaction de ces lignes) que je vais employer ici, et d'autres portages vers des versions plus récentes du kernel comme la branche maintenue par Chris Boot (sur github.com/bootc/).

La technique de compilation est classique pour Linux embarqué. Téléchargeons le noyau :

```
[RaspberryPi]$ git clone https://github.com/raspberrypi/linux linux-raspberrypi
```

Puis plaçons dans le répertoire des sources du kernel un fichier de configuration que nous renommons en **.config** :

```
[RaspberryPi]$ cd linux-raspberrypi
[linux-raspberrypi]$ wget http://www.blaess.fr/christophe/files/glmf/rpi-scratch-
01/config-linux
[linux-raspberrypi]$ mv config-linux .config
```

et nous pouvons vérifier et modifier la configuration avec

```
[linux-raspberrypi]$ make ARCH=arm menuconfig
```

En particulier, vous pouvez modifier l'option « *Local version* » du menu « *General Setup* » afin d'inscrire un identifiant – par exemple vos initiales – qui apparaîtra en suffixe du numéro de noyau après le boot. Puis lançons la compilation ainsi :

L'option ARCH permet de sélectionner l'architecture cible lors de la compilation du kernel, il faut l'indiquer à chaque étape de la préparation du noyau. L'option **CROSS_COMPILE** est un préfixe ajouté par le **Makefile** du noyau devant les commandes **gcc**, **ld**, **as**, etc. ainsi le compilateur invoqué ici sera **/usr/local/cross-rpi/usr/bin/arm-linux-gcc**.

```
[linux-raspberrypi]$ make ARCH=arm
CROSS_COMPILE=/usr/local/cross-rpi/usr/bin/arm-linux-
```

```
[linux-raspberrypi]$ ls -l arch/arm/boot/zImage
-rwxrwxr-x 1 cpb cpb 2680880 juil. 12 07:37 arch/arm/boot/zImage
[linux-raspberrypi]$
```

Préparation de la carte SD

Nous avons obtenu une image de noyau prête à l'emploi. Encore faut-il l'installer sur le Raspberry Pi. Pour cela nous allons préparer une carte SD (de préférence de classe 6 comme c'est recommandé par RS Electronics) avec deux partitions :

- la première partition sera formatée au format **vfat** et contiendra, outre le *bootloader* dont nous parlerons ci-dessous, l'image du noyau ;
- la seconde partition, au format **ext2** contiendra la racine du système de fichiers principal. Nous traiterons de sa construction dans le prochain article (ainsi que du choix de **ext2**).

J'utilise une carte SD de 2 Go, je vais attribuer 128 Mio à la première partition et le reste de la carte à la seconde. Lorsque j'insère la carte SD dans le lecteur sur mon PC de développement elle est vue comme **/dev/sdb** (ce que je vois dans les traces du kernel avec la commande **dmesg**).

Attention à bien employer le nom qui lui est attribué sur *votre* système, pour ne pas détruire le système de fichiers d'un autre disque !

Initialement la carte ne contient qu'une seule partition – que je vais supprimer – au format Fat 32.

```
[RaspberryPi]$ sudo fdisk /dev/sdb
Commande (m pour l'aide): p
Disque /dev/sdb : 1973 Mo, 1973420032 octets
[...]
Périphérique Amorçe Début Fin Blocs Id Système
/dev/sdb1 2048 3854335 1926144 c W95 FAT32 (LBA)
Commande (m pour l'aide): d
Partition sélectionnée 1
Commande (m pour l'aide):
```

J'ajoute une première partition de 128 Mio :

```
Commande (m pour l'aide): n
Partition type:
  p primary (0 primary, 0 extended, 4 free)
  e extended
Select (default p): p
Numéro de partition (1-4, par défaut 1): 1
Premier secteur (2048-3854335, par défaut 2048): (Entrée)
Utilisation de la valeur par défaut 2048
Dernier secteur, +secteurs or +taille{K,M,G} (2048-3854335, par défaut 3854335): +128M
```

On lui associe le type Fat 32 et l'attribut « bootable »

```
Commande (m pour l'aide): t
Partition sélectionnée 1
Code Hexa ( taper L pour lister les codes): c
Type système de partition modifié de 1 à c (W95 FAT32 (LBA))
Commande (m pour l'aide): a
Numéro de partition (1-4): 1
```

Puis une seconde partition de type Linux :

```
Commande (m pour l'aide): n
Partition type:
  p primary (1 primary, 0 extended, 3 free)
  e extended
Select (default p): p
Numéro de partition (1-4, par défaut 2): (Entrée)
Utilisation de la valeur par défaut 2
Premier secteur (264192-3854335, par défaut 264192): (Entrée)
Utilisation de la valeur par défaut 264192
Dernier secteur, +secteurs or +taille{K,M,G} (264192-3854335, par défaut 3854335): (Entrée)
Utilisation de la valeur par défaut 3854335
Commande (m pour l'aide): w
[RaspberryPi]$
```

Je vais formater ces deux partitions en leur attribuant deux noms faciles à identifier. Attention, encore une fois, à employer les noms de périphériques blocs qui correspondent aux partitions sur *votre* système (pas nécessairement `/dev/sdb1` et `/dev/sdb2`)

```
[RaspberryPi]$ sudo /sbin/mkfs.vfat -n Boot /dev/sdb1
mkfs.vfat 3.0.12 (29 Oct 2011)
[RaspberryPi]$
[RaspberryPi]$ sudo /sbin/mkfs.ext2 -L Root /dev/sdb2
mke2fs 1.42 (29-Nov-2011)
[...]
[RaspberryPi]$
```

Ainsi lorsque j'insère à nouveau la carte SD dans mon PC, les deux partitions sont montées automatiquement sous `/media/Boot/` et `/media/Root/`.

Installation du bootloader

Les systèmes Linux embarqués sont généralement chargés par le *bootloader* U-boot, mais ce n'est pas le cas du Raspberry Pi, qui dispose de son propre outil de démarrage. Nous pouvons télécharger ainsi ces firmwares précompilés (les sources ne sont malheureusement pas disponibles) :

```
[RaspberryPi]$ git clone https://github.com/raspberrypi/firmware bootloader
[...]
[RaspberryPi]$ ls bootloader/boot/
arm128_start.elf  bootcode.bin          kernel_emergency.img  loader.bin
arm192_start.elf  COPYING.linux         kernel.img            start.elf
arm224_start.elf  kernel_cutdown.img    LICENCE.broadcom
[RaspberryPi]$
```

Les fichiers qui nous intéressent sont les suivants :

- Les deux fichiers de firmware **bootcode.bin** et **loader.bin** initialisent le GPU et le cœur Arm.

- Le fichier **start.elf**, est une copie (au choix) de **arm128_start.elf**, **arm192_start.elf** ou **arm224_start.elf** qui attribuent respectivement 128Mio, 192Mio ou 224Mio de mémoire au processeur Arm et le reste des 256Mio de mémoire au contrôleur graphique. Pour ce premier essai je m'intéresse essentiellement au mode console aussi utiliserai-je le fichier **arm224_start.elf** qui donne un maximum de mémoire au CPU.

- Le fichier **kernel.img** est l'image du noyau Linux à démarrer, nous le remplacerons par le **zImage** obtenu dans les paragraphes précédents.

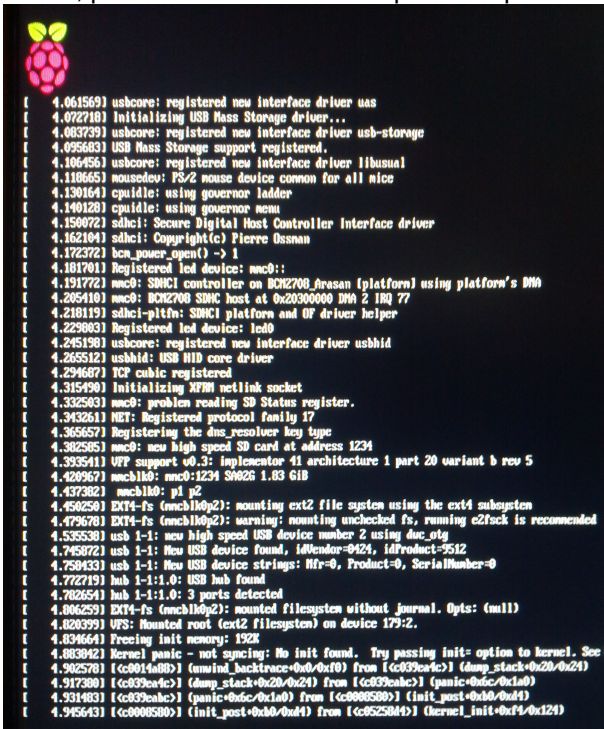
- Les arguments essentiels pour le kernel ont déjà été embarqués dans l'image lors de la compilation (dans le menu « *Boot Options* » de la configuration) mais des paramètres supplémentaires peuvent être ajoutés dans le fichier **cmdline.txt**. La présence de ce dernier est indispensable et il ne doit pas être vide, nous allons donc y inscrire un simple espace.

Voici donc l'installation des cinq fichiers sur la partition de démarrage.

```
[RaspberryPi]$ cp bootloader/boot/bootcode.bin /media/Boot/
[RaspberryPi]$ cp bootloader/boot/loader.bin /media/Boot/
[RaspberryPi]$ cp bootloader/boot/arm224_start.elf /media/Boot/start.elf
[RaspberryPi]$ cp linux-raspberrypi/arch/arm/boot/zImage /media/Boot/kernel.img
[RaspberryPi]$ echo ' ' > /media/Boot/cmdline.txt
```

Nous pouvons alors démonter notre carte SD et l'insérer dans le Raspberry Pi pour un premier boot.

Branchons un écran sur le connecteur HDMI et alimentons la carte. La LED rouge s'allume dès la mise sous tension, puis la LED verte scintille pour indiquer les accès à la carte SD.



À quoi correspondent les options du noyau qui ont été inscrites lors de sa configuration ?

- **rootwait** : attendre (éventuellement indéfiniment) sans échouer que la partition contenant l'arborescence des fichiers soit prête, ceci est nécessaire lorsque l'initialisation du périphérique bloc correspondant peut être longue (notamment pour les disques USB) ;

- **root=/dev/mmcblk0p2** : la racine de l'arborescence des fichiers se trouve sur la seconde partition de la première (et seule) carte SD ;

- **rootfstype=ext2** : cette partition est formatée en utilisant le système de fichiers **ext2** ;

- **console=tty1** : envoyer les messages du noyau vers le premier terminal (sur le port HDMI) ;

- **console=ttyAMA0,115200n8** : envoyer également les messages du noyau vers le port série (voir l'encadré ci-après) ;

- **loglevel=10** : envoyer sur les consoles mentionnées ci-dessus tous les messages avec un niveau d'urgence inférieur à 10 (les niveaux les plus urgents étant les plus faibles) ;

- **smc95xx.turbo_mode=N** : désactiver le mode turbo sur le contrôleur Ethernet, cette option est conseillée car des « *kernel panic* » ont été observés sur des charges réseau importantes, notamment avec BitTorrent ;

- **dwc_otg.lpm_enable=0** : désactiver le « *Link Power Managment* » et la mise en veille du contrôleur USB.

Sur l'écran une framboise (*raspberrypi*) s'affiche et les messages du noyau défilent, puis se terminent sur un beau « *kernel panic* » !

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

Tout ça pour ça ? Et oui ! Ce message de panique du noyau est en fait une excellente nouvelle : il signifie que le bootloader est bien installé, que le kernel a été correctement compilé (par notre *toolchain*) et que sa configuration lui permet de mener à bien son démarrage. Arrivé à ce point, le noyau Linux a terminé son travail d'initialisation, il a reconnu les périphériques disponibles et le système est totalement prêt à fonctionner, mais pour cela, le kernel doit passer le relais aux applications de l'espace utilisateur.

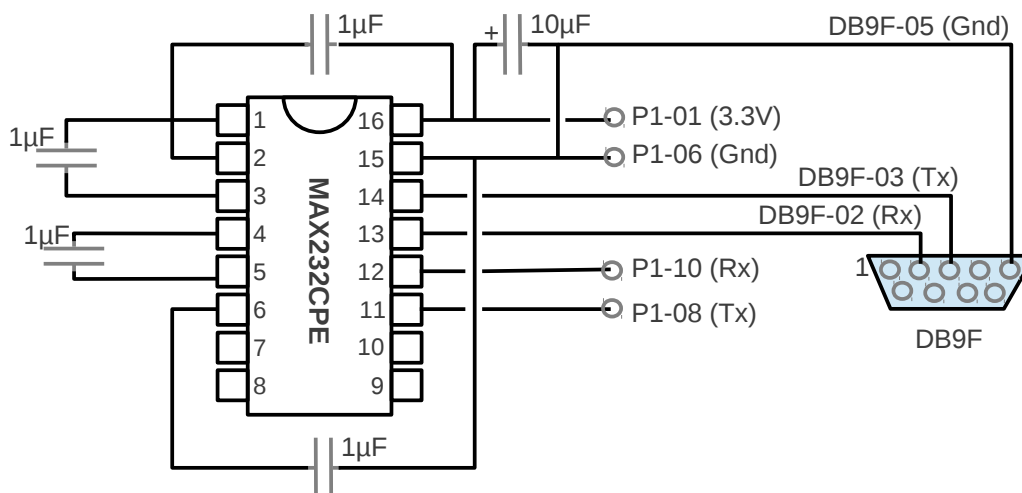
Pour ce faire, il lance un processus nommé **init**. Néanmoins pour démarrer ce processus, il faut que le kernel trouve le fichier exécutable dans l'arborescence. Par convention le fichier **init** est recherché successivement dans **/sbin**, **/bin** et **/etc**. En cas

d'échec, le noyau tente, en dernier recours, de lancer un shell **/bin/sh** puis échoue sur le message de panique. Comme nous n'avons pour le moment rien écrit sur la seconde partition (la racine de l'arborescence des fichiers), il ne peut évidemment trouver ni **init** ni **sh**.

Nous allons devoir remplir le système de fichiers principal, ce qui fera l'objet du second article.

Face à un système embarqué, l'un de mes premiers réflexes est de chercher à obtenir une console sur un port série. Ceci offre, entre autres, l'avantage de pouvoir dérouler tranquillement les messages du noyau pour vérifier sa configuration, contrairement à l'écran connecté sur le port HDMI qui ne nous affiche que les dernières lignes.

Le Raspberry Pi offre bien un port série, mais il est dissimulé dans le connecteur d'extension P1. En outre les signaux sont à un niveau [0, +3.3V] non compatible avec la norme RS-232. Une adaptation de niveaux électriques est nécessaire. Elle se fait facilement à l'aide d'un montage très classique à base du composant MAX232CPE.



Les broches du port P1 du Raspberry Pi peuvent être identifiées sur la photo du connecteur. Celles qui nous concernent sont :

- broche 1 : l'alimentation +3.3V
- broche 6 : la masse électrique
- broche 8 : la ligne de transmission série Tx
- broche 10 : la ligne de réception série Rx

Les sorties du composant MAX232CPE, au niveau RS-232 devront être branchées sur une prise DB9 femelle sur les broches suivantes :

- broche 2 : Entrée TX
- broche 3 : Sortie TX
- broche 5 : Masse électrique

Avec ce branchement, il vous faudra insérer un câble de liaison « null-modem » entre la prise DB9 et celle du PC (éventuellement via un adaptateur USB/Série). Si vous le souhaitez, vous pouvez éviter cette étape en utilisant une prise DB9 mâle sur laquelle on intervertira le branchement des broches 2 et 3. Ainsi vous pourrez relier directement votre Raspberry Pi sur l'adaptateur USB/Série de votre PC.



Pour que le kernel Linux envoie ses traces vers le port série, il est nécessaire de lui fournir en argument **console=ttyAMA0,115200n8**. Le port **ttyAMA0** est le premier port série AMBA, et on ajoute la configuration série (115200 bits/seconde, pas de parité, 8 bits de données) après la virgule. Ceci a été inscrit dans le menu « *Boot Options* » de la configuration du kernel en supplément de **console=tty1** qui lui demande d'envoyer les traces vers la sortie HDMI.

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 ;
- Chris Boot effectue des portages des patches Raspberry sur des noyaux plus récents, son site se trouve à l'adresse <http://www.bootc.net/> ;
- Je vous conseille également la lecture de la quatrième édition du « *Linux embarqué* » de Pierre Fichoux 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).