Raspberry Pi from scratch – 3

Christophe Blaess

Cet article a été publié dans le numéro 158 (mars 2013) 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. Nous avons déjà obtenu (voir Linux Magazine 155) un système minimal, avec un kernel recompilé et les utilitaires contenus dans le package Busybox. Nous allons étendre les possibilités du système en ajoutant de nouvelles fonctionnalités : réseau, authentification, utilisation des GPIO, etc.

Sécuriser l'accès au système

Jusqu'à présent notre système ne présente aucune sécurité. Dès la fin du boot, il nous propose quatre shells directement accessibles sur les consoles virtuelles **/dev/tty1** à **/dev/tty4** (en utilisant un clavier USB et un écran HDMI) et un shell disponible sur le port série **/dev/ttyAMA0**. Il n'y a pas d'étape de connexion, le shell s'exécute avec l'UID (*User Identifier*) **0**, celui de *root*. Ce comportement est assez fréquent sur les systèmes embarqués profondément enfouis, où aucun point d'accès n'est directement utilisable depuis l'extérieur et où une intervention sur le système nécessite le branchement d'un connecteur spécifique.

Pour le Raspberry Pi, les choses sont différentes puisque nous avons des connecteurs vidéo, HDMI, USB et réseau bien visibles et accessibles. On ne peut pas laisser un accès totalement ouvert sur une carte susceptible d'être connectée à un réseau – ce que nous ferons dans la suite de l'article. Nous allons donc nous connecter sur le système pour créer des comptes utilisateurs et limiter les accès par une phase d'authentification.

Commençons par créer deux fichiers **/etc/passwd** et **/etc/group** avec le compte root. Bien entendu ces manipulations doivent être réalisées sur le Raspberry, pas sur votre PC de compilation !

```
# echo   "root::0:0:root:/root:/bin/sh"   >   /etc/passwd
# echo   "root:x:0:"   >   /etc/group
```

Puis fixons un mot de passe pour le compte *root* :

```
# passwd
Changing password for root
New password:
Bad password: too short
Retype password:
Password for root changed by root
# cat   /etc/passwd
root:3Dtkfq2spiHAI:0:0:root:/root:/bin/sh
```

Même si la commande **passwd** se plaint de la faiblesse de notre mot de passe (j'ai saisi **root**), elle accepte de l'employer, et nous voyons apparaître son hachage MD5 dans le second champ de la première ligne du fichier **/etc/passwd**. Ajoutons un nouvel utilisateur avec des droits restreints par rapport à ceux de *root* :

```
# adduser   rpi
Changing password for rpi
New password:
Bad password: too short
Retype password:
Password for rpi changed by root
# ls   -l   /home/
total 4
```

```
drwxr-sr-x    2 rpi      rpi           4096 Jan  1 00:05 rpi
# cat   /etc/passwd
root:3Dtkfq2spiHAI:0:0:root:/root:/bin/sh
rpi:4o38X5SqxcPQM:10:10:Linux User,,,:/home/rpi/:/bin/sh
#
```

Le compte a bien été créé, avec son mot de passe (**rpi**) et un répertoire personnel.

Modifions à présent le fichier **/etc/inittab** qui jusqu'à présent exécutait directement un shell sur chaque console virtuelle et sur le port série. Voici sa nouvelle version :

L'ensemble des fichiers que nous allons développer dans cet article est disponible dans l'archive **http://www.blaess.fr/christophe/files/glmf/rpi-scratch-03.tar.bz2**.

```
::sysinit:/etc/init.d/rcS

tty1::respawn:/sbin/getty 0 /dev/tty1
tty2::respawn:/sbin/getty 0 /dev/tty2
tty3::respawn:/sbin/getty 0 /dev/tty3
tty4::respawn:/sbin/getty 0 /dev/tty4

ttyAMA0::respawn:/sbin/getty 115200 /dev/ttyAMA0
```

La commande **getty** initialise le périphérique caractère indiqué en second argument en utilisant la vitesse de connexion (octets/seconde) précisée en premier argument. Ensuite elle exécute la commande **login** avant de démarrer un shell.

Désormais chaque accès à notre système demandera une authentification. Nous pouvons également donner un nom à notre système, afin que la demande de login soit plus esthétique. Ceci s'obtient en ajoutant la ligne suivante dans le script de démarrage **/etc/init.d/rcS** :

```
hostname -F /etc/hostname
```

Il faudra inscrire dans le fichier **/etc/hostname** le nom choisi pour votre carte. Par exemple :

```
# echo   "R-Pi"   >   /etc/hostname
```

Après le redémarrage nous obtenons :

```
R-Pi login: rpi
Password:
~ $ whoami
rpi
~ $ exit
R-Pi login: root
Password:
~ #
```

Si vous souhaitez ajuster le *prompt* du shell de manière globale pour tout le système, il suffit de configurer la variable **PS1** dans un fichier **/etc/profile** qui sera lu et interprété par le shell au démarrage. On peut d'ailleurs y compléter le contenu de la variable **PATH**. En voici un exemple :

```
# Fichier /etc/profile

PATH=/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/bin:/usr/local/sbin

# Prompt : user@hostname [chemin]
PS1="\u@\h [\W]"

# Symbole : $ pour utilisateur normal, # pour root
if [ $(id -u) -eq 0 ]
then
    PS1=${PS1}"# "
else
    PS1=${PS1}"$ "
fi
```

La sécurité est assez limitée, car les mots de passe chiffrés sont visibles par tous, sans employer de fichier **/etc/shadow**. Nous pouvons nous contenter de cela si le système n'est pas destiné à recevoir des

connexions d'utilisateurs externes.

> L'arborescence des fichiers reposant sur une carte SD qui dépasse du Raspberry Pi, son niveau de sécurité est nul si un utilisateur peut avoir l'accès physique au système. La meilleure solution pour l'utiliser dans un environnement non surveillé (bureau *open-space*, salle technique, lieu public...) serait de l'incorporer dans un boîtier fixe fermé par une clé ou un cadenas. Une autre possibilité consisterait à chiffrer le système de fichier (en utilisant par exemple LVM2) mais cela peut compliquer le démarrage automatique (s'il faut saisir une *pass-phrase*) et consommer une part non négligeable des ressources CPU.

Ajouter un signe de vie

Une fois le boot achevé, le Raspberry Pi propose donc une connexion sur ses ports consoles et série. Par la suite nous ajouterons un accès par le réseau. Toutefois, pour de nombreux projets embarqués, on n'utilise ni écran, ni liaison série, et la connexion réseau est réservée aux mises à jour. La carte doit fonctionner de manière autonome sans dispositif d'affichage, ce que l'on nomme généralement système « *headless* ». Il est néanmoins intéressant de pouvoir vérifier d'un seul coup d'œil si le démarrage s'est bien déroulé et si le système est dans un état normal. On parle alors de « signe de vie ».

Un signe de vie peut prendre de nombreux aspects différents : signal périodique sur une borne de test, compteur incrémenté régulièrement dans une zone de mémoire partagée, trame vide émise sur une liaison réseau, etc. Le signe de vie le plus courant est le clignotement d'une led. Par chance, le Raspberry Pi est doté de plusieurs leds que le noyau Linux est capable de piloter. En l'occurrence, la led étiquetée « *Ok* » (la plus proche du connecteur audio) va nous servir de signe de vie.

Le support kernel pour les leds est intégré lors de la compilation du noyau en activant l'option « *LED Support for GPIO connected LEDs* » dans le sous-menu « *LED support* » du menu « *Device Drivers* ». Avec le Raspberry Pi, cette option se traduit par l'apparition d'un sous-répertoire **`led0`** dans **`/sys/class/leds`**.

Dans le même sous-menu de configuration du kernel, on peut choisir des déclencheurs (*triggers*) – c'est-à-dire des heuristiques commandant l'allumage ou l'extinction des leds. Le trigger sera sélectionné en écrivant son nom dans **`/sys/class/leds/led0/trigger`**.

Plusieurs heuristiques sont disponibles, en voici quelques-unes :

- **`none`** : la led est pilotée manuellement en écrivant **`0`** ou **`1`** dans **`/sys/class/leds/led0/brightness`** ;

- **`timer`** : (module **`ledtrig-timer.ko`** compilé grâce à l'option « *LED Timer Trigger* » lors de la configuration du noyau) la led clignote suivant un cycle dont on peut contrôler les durées d'allumage et d'extinction en indiquant les valeurs en millisecondes dans les fichiers **`/sys/class/leds/led0/delay_on`** et **`/sys/class/leds/led0/delay_off`** ;

- **`heartbeat`** : (module **`ledtrig-heartbeat.ko`** dépendant de l'option « *LED Heartbeat Trigger* ») pour ce trigger, la led simule un battement de cœur avec une fréquence qui dépend de la charge système (c'est celui qui nous choisirons plus bas) ;

- **`backlight`** : (option « *LED Backlight Trigger* ») : utilisé lorsque la led sert de rétro-éclairage pour un écran, ne nous servira pas ici ;

- **`mmc0`** : intégré directement dans le driver **`mmc-core`**, ce trigger allume la led lors des accès à la carte SD.

Nous activons donc le trigger **`heartbeat`** dans le script de démarrage **`/etc/init.d/rcS`**. Comme nous l'avons compilé précédemment sous forme de module, il est nécessaire de charger ce dernier au préalable. Ajoutons donc les lignes suivantes :

```
modprobe ledtrig-heartbeat
if [ -f /sys/class/leds/led0/trigger ]; then
    echo heartbeat > /sys/class/leds/led0/trigger
fi
```

Après redémarrage, notre Raspberry Pi nous indique par un léger battement de cœur qu'il est opérationnel.

Simplifier le remplissage de /dev

Dans l'article précédent, nous avons lancé, dans **`/etc/init.d/rcS`**, le démon **`mdev`** pour qu'il s'assure du contenu de **`/dev`** et qu'il soit notifié par le kernel à chaque insertion ou extraction de périphérique amovible (clé USB par exemple). Il est possible, depuis le noyau 2.6.32 de simplifier cette étape en utilisant le système de fichiers **`devtmpfs`**. Ce système de fichier permet de disposer d'un répertoire **`/dev`** en mémoire, automatiquement rempli par le kernel au fur et à mesure de la découverte ou de la disparition des périphériques. Le système de fichiers **`devtmpfs`** étant en mémoire, il est possible d'y créer des fichiers si besoin (ou des répertoires comme nous le ferons plus bas). La présence de **`mdev`** n'est donc plus indispensable, et l'on peut remplacer les lignes

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

par

```
mount none /dev -t devtmpfs
```

Le fait de conserver **mdev** peut se justifier si on désire créer des alias ou des liens symboliques sur les fichiers spéciaux ou si l'on veut modifier les droits et appartenances des fichiers créés (ceci en remplissant un fichier **/etc/mdev.conf**).

Pour ce système, j'ai choisi de laisser **devtmpfs** gérer seul les périphériques spéciaux. Il est également possible (en agissant dans le sous-menu « *Generic Driver Options* » du sous-menu « *Device Drivers* » lors de la configuration du kernel) de demander au noyau de monter automatiquement **/dev** dès qu'il a terminé de préparer le système de fichiers initial. Personnellement, je considère qu'il s'agit plutôt d'une tâche dédiée à l'espace utilisateur et non au kernel, aussi ai-je désactivé cette option.

Configuration du réseau

La configuration du réseau est un point important pour un système embarqué. Il s'agit souvent de son unique interface avec un utilisateur humain. Contrairement aux environnements Linux classiques (postes de travail, serveur, etc.), la configuration d'un système embarqué ne cherche pas à être la plus généraliste et la plus portable possible. Nous pourrons prendre directement en considération ce que nous savons du matériel présent (interface, drivers, connectivité) et du réseau sur lequel il sera connecté (adressage statique ou dynamique, routage, résolution de noms), etc.

Le Raspberry Pi contient un contrôleur SMSC Lan9512 fournissant une interface Ethernet 10/100M et deux ports USB 2.0. Le driver Linux correspondant est **smsc95xx** qui doit donc être compilé statiquement dans le kernel (comme nous l'avons fait dans les articles précédents) ou sous forme de module à charger au moment du boot. Toutefois, la détection et l'initialisation de ce contrôleur prennent un temps non négligeable par rapport au script de démarrage du système. Pour effectuer la configuration du réseau, nous allons donc devoir attendre que l'interface Ethernet **eth0** soit disponible.

Il y a plusieurs possibilités pour obtenir ce résultat, la plus simple est de dédier ce travail à un script s'exécutant en arrière-plan et laissant le boot continuer sa progression. Ce script sera nommé **/etc/init.d/rc.network** et assurera l'initialisation des interfaces **lo** et **eth0**.

> On retrouve dans **rc.network** le préfixe **rc** que l'on rencontre dans de nombreux scripts d'initialisations, il s'agit d'un héritage du système CTSS des années soixante, ancêtre d'Unix sur lequel un programme nommé RUNCOM (pour « *run commands* ») exécutait les commandes listées dans ces fichiers. L'habitude a perduré sur la plupart des systèmes Linux de nommer ainsi les scripts de démarrage.

Nous insérons donc dans **/etc/init.d/rcS** la ligne suivante (avant la programmation du signe de vie par exemple) :

```
/etc/init.d/rc.network &
```

Dans ce script, nous allons commencer par initialiser l'interface *loopback* **lo**. C'est facile, son adresse est toujours figée, il s'agit de 127.0.0.1. Ensuite nous attendrons que **eth0** soit disponible en bouclant (avec de petits sommeils d'une seconde) autour de **ifconfig -a** jusqu'à ce que sa sortie standard (examinée avec **grep**) contienne la chaîne de caractères « **eth0** ». Cette méthode n'est peut-être pas la plus élégante qui soit, mais elle convient parfaitement dans le cas d'un système embarqué, dont nous connaissons a priori le comportement.

Une fois que l'interface Ethernet est prête, nous pourrions lui affecter immédiatement une adresse IP statique et lancer les serveurs qui nous intéressent. Toute efficace qu'elle soit cette méthode présente un inconvénient, elle ne permet pas de faire de modification de la configuration réseau sans redémarrer complètement le système. Je préfère différer un peu l'initialisation, en attendant qu'un câble réseau – relié à un switch – soit effectivement branché sur le port Ethernet. Ceci est possible grâce au démon **ifplugd** qui peut surveiller une interface et invoquer un script en cas de modification de son état.

Voici donc mon fichier **/etc/init.d/rc.network** :

```
#! /bin/sh
# rc.network – Interfaces reseau

# ----------------------------------------
# lo - L'interface loopback
  /sbin/ifconfig lo   127.0.0.1

# ----------------------------------------
# eth0 - L'interface ethernet

# Attendre qu'elle soit prete
  while ! /sbin/ifconfig -a | grep eth0 >/dev/null
  do
```

```
    sleep 1
done

/bin/ifplugd -i eth0
```

Lorsque **ifplugd** détecte un changement de connexion sur l'interface **eth0**, il invoque le script **/etc/ifplugd/ifplugd.action** en lui passant en argument le nom de l'interface (**$1**) et le nouvel état (**$2**). Nous allons donc écrire ce script afin qu'il termine la configuration d'adresse.

Si votre Raspberry Pi est destiné à être connecté sans configuration particulière sur un réseau local géré par un serveur DHCP (par exemple une *set-top box* ADSL), il est possible de programmer le script **ifplugd.action** afin qu'il récupère automatiquement les paramètres réseau et se voit affecter une adresse IP personnelle. L'inconvénient, c'est que vous ne connaissez pas a priori cette adresse (sauf configuration particulière dans le serveur DHCP) et que vous ne pourrez pas accéder facilement au Raspberry Pi depuis une autre machine du réseau.

C'est donc un choix dépendant de l'usage auquel le système est destiné. S'il doit servir essentiellement de client réseau (*media player*, poste de travail mobile, etc.) la configuration dynamique par DHCP est parfaitement adaptée. Si son rôle est plutôt celui d'un serveur (HTTP, FTP, point d'entrée SSH, système d'affichage distant, etc.) il est nécessaire de connaître son adresse IP ou son nom d'hôte et l'on préférera un adressage statique en lui attribuant une adresse fixe, et en initialisant manuellement les tables de routage du noyau.

Premier cas – Adressage statique

Voici donc un exemple de script dans lequel l'interface Ethernet est configurée manuellement :

```
#! /bin/sh
# ifplugd.action - Changement d'etat port Ethernet

if [ "$1" = "eth0" ] ; then
    IP_HOST=192.168.3.152
    NETMASK=255.255.255.0
    GATEWAY=192.168.3.254
    IP_DNS=192.168.3.254

    if [ "$2" = "up" ] ; then
        /sbin/ifconfig eth0 ${IP_HOST} netmask ${NETMASK}
        if [ "${GATEWAY}" != "" ] ; then
            /sbin/route  add  default  gw "${GATEWAY}
        fi
        if [ "${IP_DNS}" != "" ] ; then
            echo "nameserver ${IP_DNS}" > /etc/resolv.conf
        fi
    fi
fi
```

Modifiez, bien entendu, les quatre lignes d'adresse avec les paramètres de votre réseau local.

Second cas – Adressage dynamique

À présent nous allons modifier le script pour qu'il démarre le démon **udhcpc** (micro-DHCP client) contenu dans Busybox. Ce dernier récupérera les paramètres au moment de l'activation de l'interface Ethernet.

Le script précédent devient :

```
#! /bin/sh
# ifplugd.action - changement d'etat port Ethernet

if [ "$1" = "eth0" ] ; then
    if [ "$2" = "up" ] ; then
        udhcpc -i eth0 -s /etc/udhcpc/udhcpc.action
    fi
    if [ "$2" = "down" ] ; then
        killall udhcpc
    fi
fi
```

En réalité, je préfère conserver les deux scripts en les renommant **ifplugd.action-dhcp** et

ifplugd.action-statique et en créant un lien symbolique **ifplugd.action** vers celui qui convient.

Le fichier **/etc/udhcpc/udhcpc.action** est appelé par **udhcpc** lorsqu'il a reçu du serveur DHCP les paramètres réseau locaux. Ces paramètres se trouvent alors dans les variables d'environnement suivantes :

- **interface** : l'adaptateur réseau concerné,

- **ip** : l'adresse IPv4 attribuée,

- **netmask** : le masque du sous-réseau local,

- **broadcast** : l'adresse de diffusion pour tout le sous-réseau local,

- **router** : liste des adresses des routeurs sur le sous-réseau,

- **domain** : sous-domaine local,

- **nameserver** : liste d'adresses des DNS.

Il existe d'autres variables qui ne nous concernent pas dans le cadre de ce projet. Le script reçoit également en premier argument sur sa ligne de commande l'une des actions suivantes :

- **bound** : une nouvelle adresse a été attribuée à l'interface,

- **renew** : l'affectation d'adresse a été renouvelée, mais certains paramètres (**nameserver** ou **router** par exemple) peuvent avoir changé.

- **deconfig** : l'attribution d'adresse est supprimée, et l'interface doit être dé-configurée.

Voici donc un script **udhcpc.action** qui répond aux changements d'état signalés par **udhcpc**.

```
#! /bin/sh
# /etc/udhcpc/udhcpc.action - Attribution d'adresse par DHCP

case "$1" in
        bound|renew)
                command="$interface $ip"
                if [ "$broadcast" != "" ]; then
                        command="$command broadcast $broadcast"
                fi
                if [ "$netmask" != "" ]; then
                        command="$command netmask $netmask"
                fi
                /sbin/ifconfig $command

                if [ "$router" != "" ]; then
                        route del default gw 0.0.0.0 dev $interface
                        for r in $router; do
                                route add default gw $r dev $interface
                        done
                fi

                rm /etc/resolv.conf
                if [ "$domain" != "" ]; then
                        echo "search $domain" > /etc/resolv.conf
                fi
                for d in $dns; do
                        echo "nameserver $d" >> /etc/resolv.conf
                done
                ;;

        defconfig)
                /sbin/ifconfig $interface 0.0.0.0
                ;;
esac
```

Services réseau

Lors de la compilation de Busybox, nous avons inclus quelques services réseau. Nous pouvons facilement tester **telnetd** et **httpd** par exemple. La démarrage de ces services peut se faire depuis **/etc/init.d/rcS**, **/etc/init.d/rc.network**, ou tout autre script lancé par **rcS**. C'est cette dernière solution que j'ai adoptée, afin de répartir et organiser plus simplement les actions entreprises au démarrage du Raspberry Pi. Ajoutons donc la ligne suivante dans **rcS** à la suite de celle de lancement de **rc.network**.

```
/etc/init.d/rc.services &
```

et créons le script suivant

```
#! /bin/sh
# rc.services – Lancement des services du systeme

/sbin/telnetd

/sbin/httpd -h /home/httpd
```

Ceci démarrera automatiquement un serveur nous permettant de nous connecter à distance avec un client Telnet. Ainsi il devient possible d'accéder depuis un autre poste :

```
(srv-2)[~]$ telnet   192.168.3.152
Trying 192.168.3.152...
Connected to 192.168.3.152.
Escape character is '^]'.

R-Pi login: rpi
Password:
rpi@R-Pi [rpi]$ uname   -a
Linux R-Pi 3.1.9-glmf #22 PREEMPT Fri Aug 17 16:59:34 CEST 2012 armv6l GNU/Linux
rpi@R-Pi [rpi]$ exit
Connection closed by foreign host.
(srv-2)[~]$
```

La connexion par Telnet n'est pas sécurisée du tout, un outil d'analyse peut révéler facilement le contenu de la session et même le mot de passe employé. Nous allons remédier à cela plus bas en utilisant une connexion sécurisée avec SSH.

Pour tester le serveur HTTP, il faut créer un petit fichier **index.html** que l'on place dans **/home/httpd**. Par exemple ce type de fichier fait parfaitement l'affaire :

```
<html>
    <head>
        <title>Raspberry Pi from scratch</title>
    </head>
    <body>
        <h1>Hello Raspberry World!</h1>
        <p>
            Cette page s'affiche depuis un petit serveur
            http embarqu&eacute; sur mon
            <strong>Raspberry Pi<strong>.
        </p>
    </body>
</html>
```

En dirigeant alors un navigateur sur un PC distant vers l'adresse IP de notre Raspberry Pi, on obtient l'affichage suivant :

Connexion sécurisée SSH

L'accès par Telnet étant nous l'avons dit peu sécurisé, on peut avoir envie de renforcer l'accès à notre système embarqué. En outre, en intégrant un support SSH, nous pourrons bénéficier d'un transfert de fichiers très simple grâce à la commande **scp**.

Plutôt que de déployer un serveur SSH complexe et relativement lourd, il existe une alternative simple et légère : **dropbear**. Ce projet de Matt Johnson permet d'embarquer facilement sur une cible simple un mécanisme de connexion sécurisée. Compilons-le pour notre Raspberry Pi. Tout d'abord téléchargeons les dernières sources.

```
[RaspberryPi]$ wget   http://matt.ucc.asn.au/dropbear/releases/dropbear-2012.55.tar.bz2
[...]
[RaspberryPi]$ tar  xjf  dropbear-2012.55.tar.bz2
```

```
[RaspberryPi]$ cd  dropbear-2012.55/
```

Nous insérons dans notre variable d'environnement **PATH** le chemin d'accès au *cross-compiler* obtenu dans le premier article.

```
[dropbear-2012.55]$ PATH=$PATH:/usr/local/cross-rpi/usr/bin/
[dropbear-2012.55]$ ./configure   --host=arm-linux   --disable-zlib
```

Les options par défaut sont tout à fait correctes, aussi lançons-nous la compilation.

```
[dropbear-2012.55]$ make
[...]
```

La compilation de l'outil **scp** doit être invoquée spécifiquement.

```
[dropbear-2012.55]$ make   scp
[...]
```

Après avoir arrêté le Raspberry Pi et inséré sa carte SD dans l'hôte pour la dernière fois (à partir de maintenant nous pourrons transférer directement les fichiers par le réseau), nous copions les exécutables suivants :

```
[dropbear-2012.55]$ sudo   cp   dropbear   dropbearkey   scp   /media/Root/usr/bin/
```

Puis nous modifions le fichier **/media/Root/etc/init.d/rc.services** en ajoutant les lignes :

```
if [ ! -f /etc/dropbearkey.rsa ]; then
    /usr/bin/dropbearkey -t rsa -f /etc/dropbearkey.rsa
fi

/usr/bin/dropbear -r /etc/dropbearkey.rsa
```

Ce script commence par générer si nécessaire la clé RSA utilisée dans le cryptage employé pour la communication SSH. Puis il lance le serveur, après quoi il devient possible de se connecter à distance ainsi :

```
(srv-2)[~]$ ssh   root@192.168.3.152
The authenticity of host '192.168.3.152 (192.168.3.152)' can't be established.
RSA key fingerprint is cb:27:40:b9:19:f5:07:37:c6:b7:c0:d7:e1:95:0d:76.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.3.152' (RSA) to the list of known hosts.
root@192.168.3.152's password:
root@R-Pi [/root]# uname   -a
Linux R-Pi 3.1.9-glmf #22 PREEMPT Fri Aug 17 16:59:34 CEST 2012 armv6l GNU/Linux
root@R-Pi [/root]# exit
Connection to 192.168.5.28 closed
[~]$
```

Il est également possible d'envoyer un fichier depuis notre hôte vers le Raspberry Pi ainsi :

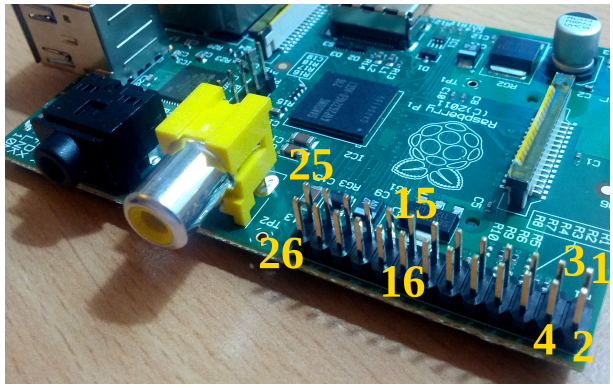
```
(srv-2)[~]$ echo   "ESSAI"   >   fichier
(srv-2)[~]$ scp  fichier  root@192.168.3.152:~/
root@192.168.3.152's password:
fichier                                                 100%    6     0.0KB/s
00:00
(srv-2)[~]$ ssh   root@192.168.3.152
root@192.168.3.152's password:
root@R-Pi [/root]# cat   fichier
ESSAI
root@R-Pi [/root]#
```

Ceci nous évite dorénavant d'avoir besoin d'extraire la carte SD du Raspberry Pi et de l'insérer dans le PC. Tous les transferts de fichiers se feront par le réseau.

Entrées-sorties par GPIO

Lorsque l'on décide d'utiliser un système comme le Raspberry Pi dans un contexte embarqué, c'est habituellement pour communiquer avec d'autres équipements (capteurs, interface IHM spécifique, actionneurs, etc.) suivant des protocoles plus ou moins personnalisés. Pour cela on a généralement recours à la programmation de GPIO (*General Purpose Input Output*) . Il s'agit de broches du micro-processeur que l'on peut affecter individuellement en entrées ou en sorties, et sur lesquelles il est possible d'écrire ou de lire un état logique représenté par une tension électrique. Souvent, il est également possible d'affecter un gestionnaire d'interruption qui sera invoqué automatiquement par le micro-processeur lorsqu'un changement d'état sera détecté.

Le Raspberry Pi ne fait pas exception. Son processeur BCM2835 propose 54 lignes GPIO. Toutes ne sont pas accessibles sur la carte, et une partie d'entre-elles sont déjà employées pour des fonctionnalités prédéfinies. Dix-sept lignes GPIO sont accessibles sur le port d'extension P1. Les broches sont numérotées de bas en haut et de gauche à droite lorsque vous regardez le Raspberry Pi avec le dessin de framboise orienté correctement.



Attention, certaines broches du connecteur ont changé d'affectation avec les révisions du Raspberry Pi.

La numération des déclinaisons du Raspberry Pi est un peu trompeuse. La version dont vous disposez probablement est le modèle B. Le modèle A, quant-à-lui n'est pas encore commercialisé à la date de rédaction de cet article. Il sera proposé à un prix inférieur au modèle B, mais ne comportera qu'un seul port USB et pas de connecteur RJ45.

Le modèle B se décline en deux versions : 1.0 et 2.0. La seconde, disponible depuis le mois de septembre 2012, comporte quelques corrections et améliorations. La plus visible est la présence de deux trous de montage sur la carte, l'un sous la framboise et l'autre sous les leds du réseau.

En outre la version 2.0, initialement distribuée avec 256Mo de mémoire contient 512Mo depuis le mois d'octobre 2012.

Pour déterminer dans un programme le numéro de version (et choisir ainsi le numéro de GPIO par exemple), on peut consulter la ligne **`Revision`** du pseudo-fichier **`/proc/cpuinfo`**.

Hardware Revision	Version Raspberry Pi
`0002`	Modèle B – 1.0
`0003`	Modèle B – 1.0 (sans fusibles)
`0004, 0005, 0006`	Modèle B – 2.0 – 256Mo
`000d, 000e, 000f`	Modèle B – 2.0 – 512 Mo

De plus, le numéro indiqué sur la ligne **`Revision`** sera précédé de **`100`** si le Raspberry Pi est soumis à un sur-cadencement d'horloge (*overclocking*).

Broche du port P1	GPIO rev. 1.0	GPIO rev. 2.0	Fonction
1 et 17			Alimentation +3.3 V
2 et 4			Alimentation + 5 V
6, 9, 14, 20 et 25			Masse électrique
3	0	2	
5	1	3	
7	4	4	
8	14	14	TX UART 0
10	15	15	RX UART 0
11	17	17	
12	18	18	CLK PCM
13	21	27	
15	22	22	
16	23	23	
18	24	24	
19	10	10	SPI 0 MOSI
21	9	9	SPI 0 MISO
22	25	25	
23	11	11	SPI 0 CLK
24	8	8	SPI 0 CE 0
26	7	7	SPI 0 CE 1

Si l'on souhaite réaliser des entrées-sorties pour un montage personnel il est préférable d'utiliser l'un des GPIO sans fonctionnalité spécifique, et n'ayant pas changé entre les versions du Raspberry Pi. On prendra donc de préférence les GPIO 4, 17, 22, 23, 24 ou 25 qui se trouvent respectivement sur les broches 7, 11, 15, 16, 18 et 22 du connecteur.

Attention aux tensions employées ! Les entrées sur les broches GPIO doivent se faire avec des signaux [0, +3.3V] ; évitez de sortir de cet intervalle ou vous risquez de détruire le processeur...

Entrées-sorties depuis l'espace utilisateur

Le noyau Linux nous propose une interface très simple pour accéder aux broches GPIO : il suffit de manipuler des pseudo-fichiers de l'arborescence **`/sys`**. Il est d'abord nécessaire de demander l'accès depuis l'espace utilisateur à une broche GPIO (pour éviter les collisions avec un driver kernel qui serait en train de l'utiliser) puis il devient possible de configurer la broche en entrée ou en sortie. On pourra alors lire ou écrire des données comme dans un simple fichier. Ceci est réalisable avec n'importe quel langage de programmation. L'accès le plus simple : directement depuis le *prompt* du shell.

Demandons au noyau de nous laisser accéder au GPIO 22 (broche 15) par exemple.

```
# cd   /sys/class/gpio/
# ls
export     gpiochip0  unexport
#
```

Il n'y a pas de répertoire **`gpio22`**, aucune autre application de l'espace utilisateur ne l'utilise pour le moment.

```
# echo  22  >  export
# ls
export     gpio22     gpiochip0  unexport
# cd  gpio22/
# ls  -l
```

```
total 0
-rw-r--r--    1 0        0             4096 Jan  1 00:24 active_low
-rw-r--r--    1 0        0             4096 Jan  1 00:24 direction
-rw-r--r--    1 0        0             4096 Jan  1 00:24 edge
lrwxrwxrwx    1 0        0                0 Jan  1 00:24 subsystem ->
../../../../class/gpio
-rw-r--r--    1 0        0             4096 Jan  1 00:24 uevent
-rw-r--r--    1 0        0             4096 Jan  1 00:24 value
#
```

En écrivant **22** dans le pseudo-fichier **`export`** le noyau nous présente un nouveau sous-répertoire pour accéder à notre GPIO.

Nous dirigeons le GPIO 22 en sortie :

```
# echo  out  >  direction
```

En mesurant la tension entre la broche 15 (GPIO 22) et la broche 14 (masse), nous obtenons une tension nulle.

```
# echo  1  >  value
```

Le voltmètre indique à présent 3.3 V

```
# echo  0  >  value
```

La tension retombe à zéro.

Nous pouvons également configurer la broche en entrée :

```
# echo  in  >  direction
```

Relions la broche 15 (GPIO 22) et la broche 14 (masse), puis lisons la valeur de l'entrée :

```
# cat  value
0
```

A présent, relions la broche 15 (GPIO 22) et la broche 17 (+3.3V)

```
# cat  value
1
```

Lorsque nous ne voulons plus accéder à notre GPIO, nous pouvons demander au noyau de cesser de l'exporter dans le pseudo-système de fichiers /sys :

```
# cd  /sys/class/gpio/
# ls
export     gpio22     gpiochip0  unexport
# echo  22  >  unexport
# ls
export     gpiochip0  unexport
#
```

Toutes ces manipulations, réalisées ici depuis le shell peuvent être intégrées dans une application écrite avec n'importe quel langage, il lui suffit d'ouvrir les pseudo-fichiers de **`/sys`** et d'y lire ou écrire les valeurs désirées. On notera que le processus doit disposer des droits *root* au moment de l'ouverture de ces fichiers. Il peut perdre ensuite ses privilèges, par exemple en invoquant l'appel-système **`setuid()`**, et continuer d'écrire dans les fichiers déjà ouverts.

Il y a toutefois une chose qu'une telle application ne peut pas réaliser facilement : traiter des interruptions déclenchées par un changement d'état sur une broche GPIO. Pour cela nous devons nous tourner vers l'espace kernel.

GPIO depuis l'espace noyau

L'accès aux broches GPIO depuis un module kernel est simple, nous disposons d'une API déclarée dans **`<linux/gpio.h>`** comprenant entre autres les fonctions suivantes :

```
// Réservation d'accès.
int  gpio_request (unsigned int gpio, const char * ident);
void gpio_free    (unsigned int gpio);

// Visibilité dans /sys.
int  gpio_export   (unsigned int gpio);
int  gpio_unexport (unsigned int gpio);

// Sens d'accès.
int  gpio_direction_input  (unsigned int gpio);
int  gpio_direction_output (unsigned int gpio);

// Lecture et écriture.
int  gpio_get_value (unsigned int gpio);
int  gpio_set_value (unsigned int gpio, int value);

// Numéro d'interruption associée.
int  gpio_to_irq (unsigned int gpio);
int  irq_to_gpio (unsigned int irq);
```

Nous pouvons facilement écrire un petit module pour gérer les interruptions déclenchées par une broche d'entrée. Continuons par exemple avec la broche 15 que nous avons utilisée jusqu'à présent. Nous installons un gestionnaire d'interruption qui sera invoquée chaque fois qu'un front montant (de 0V à +3.3V) sera détecté sur cette broche.

Pour que le déclenchement du handler soit visible depuis l'extérieur, son travail consistera à changer l'état d'une autre broche, par exemple la broche 16 (GPIO 23). Nous pourrons alors observer les activations du handler avec un oscilloscope, un voltmètre ou même une simple led protégée par une petite résistance.

Voici donc un module qui installera ce gestionnaire d'interruption.

```
/*************************************************************************\
  rpi-irq-gpio.c : gestion d'interruptions GPIO sur Raspberry Pi
\*************************************************************************/

#include <linux/interrupt.h>
#include <linux/module.h>
#include <linux/gpio.h>


// L'entrée se fait depuis la broche 15 (GPIO 22).
#define RPI_IRQ_GPIO_IN  22


// La sortie va sur la broche 16 (GPIO 23).
#define RPI_IRQ_GPIO_OUT 23


// Gestionnaire d'interruption qui inverse l'état de la
// broche de sortie à chaque déclenchement d'interruption
// depuis la broche d'entrée.
static irqreturn_t rpi_irq_gpio_handler (int irq, void * ident)
{
    static int out_value = 0;

    gpio_set_value(RPI_IRQ_GPIO_OUT, out_value);
    out_value = 1 - out_value;

    return IRQ_HANDLED;
}


// Chargement du module .
static int __init rpi_irq_gpio_init (void)
{
    int err;

    // Demander l'accès au GPIO de sortie.
    if ((err = gpio_request(RPI_IRQ_GPIO_OUT, THIS_MODULE->name)) != 0)
        return err;
```

```
    // Demander l'accès au GPIO d'entrée.
    if ((err = gpio_request(RPI_IRQ_GPIO_IN, THIS_MODULE->name)) != 0) {
        gpio_free(RPI_IRQ_GPIO_OUT);
        return err;
    }

    // Configurer la direction du GPIO de sortie.
    if ((err = gpio_direction_output(RPI_IRQ_GPIO_OUT,1)) != 0) {
        gpio_free(RPI_IRQ_GPIO_OUT);
        gpio_free(RPI_IRQ_GPIO_IN);
        return err;
    }
    // Configurer la direction du GPIO d'entrée.
    if ((err = gpio_direction_input(RPI_IRQ_GPIO_IN)) != 0) {
        gpio_free(RPI_IRQ_GPIO_OUT);
        gpio_free(RPI_IRQ_GPIO_IN);
        return err;
    }

    // Installer le handler d'interruption.
    if ((err = request_irq(gpio_to_irq(RPI_IRQ_GPIO_IN),
                           rpi_irq_gpio_handler,
                           IRQF_SHARED | IRQF_TRIGGER_RISING,
                           THIS_MODULE->name,
                           THIS_MODULE->name)) != 0) {
        gpio_free(RPI_IRQ_GPIO_OUT);
        gpio_free(RPI_IRQ_GPIO_IN);
        return err;
    }

    return 0;
}


// Déchargement du module.
static void __exit rpi_irq_gpio_exit (void)
{
    // Retrait du handler d'interruption.
    free_irq(gpio_to_irq(RPI_IRQ_GPIO_IN), THIS_MODULE->name);
    // Liberation des GPIO.
    gpio_free(RPI_IRQ_GPIO_OUT);
    gpio_free(RPI_IRQ_GPIO_IN);
}


module_init(rpi_irq_gpio_init);
module_exit(rpi_irq_gpio_exit);
MODULE_LICENSE("GPL");
```

Pour compiler ce module, il faut un fichier Makefile dans lequel vous devrez configurer les deux lignes **KERNEL_DIR** et **CROSS_COMPILE** en fonction des emplacements respectifs des sources du noyau et de la toolchain que nous avons obtenus dans le premier article.

```
ifneq (${KERNELRELEASE},)

    obj-m += rpi-irq-gpio.o

else
    ARCH          ?= arm
    KERNEL_DIR    ?= ~/RaspberryPi/linux-raspberrypi/
    CROSS_COMPILE ?= /usr/local/cross/rpi/bin/arm-linux-
    MODULE_DIR    := $(shell pwd)

all: modules

modules:
       ${MAKE}  -C  ${KERNEL_DIR}  ARCH=${ARCH}  CROSS_COMPILE=${CROSS_COMPILE}  SUBDIRS=$
{MODULE_DIR} modules

clean:
```

```
    rm -f  *.o  .*.o  .*.o.* *.ko  .*.ko  *.mod.* .*.mod.* .*.cmd
    rm -f Module.symvers Module.markers modules.order
    rm -rf .tmp_versions
endif
```

On compile le module en exécutant simplement **`make`** (sans autre argument) dans le répertoire contenant le fichier source et le **`Makefile`**.

Après transfert sur le Raspberry Pi, nous chargeons le module :

```
# insmod  rpi-irq-gpio.ko
# cat  /proc/interrupts
           CPU0
  3:       1419   ARMCTRL  BCM2708 Timer Tick
 32:       1172   ARMCTRL  dwc_otg, dwc_otg_pcd, dwc_otg_hcd:usb1
 52:          0   ARMCTRL  BCM2708 GPIO catchall handler
 65:        174   ARMCTRL  ARM Mailbox IRQ
 66:          1   ARMCTRL  VCHIQ doorbell
 75:          1   ARMCTRL
 77:        145   ARMCTRL  bcm2708_sdhci (dma)
 83:       2228   ARMCTRL  uart-pl011
 84:        486   ARMCTRL  mmc0
192:          0      GPIO  rpi_irq_gpio
FIQ:              usb_fiq
Err:          0
```

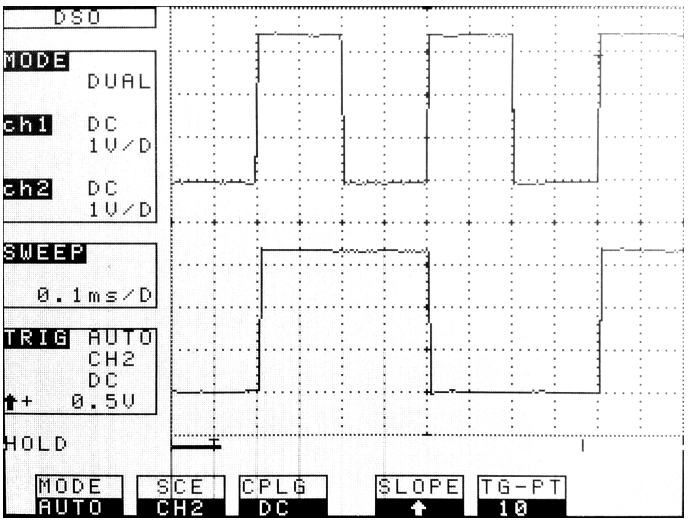
Nous voyons qu'un handler est maintenant installé sur la ligne d'interruption 192 (qui correspond donc vraisemblablement au GPIO 22). Pour le moment aucune interruption ne s'est produite.

Je branche alors un GBF (Générateur de Basses Fréquences) sur la broche 15 du Raspberry Pi, afin qu'il lui envoie un signal carré d'amplitude [0V, +3,3V] avec une fréquence d'environ 2,5 kHz. Au bout de quelques secondes je revérifie **`/proc/interrupts`** :

```
# cat  /proc/interrupts
           CPU0
  3:       1999   ARMCTRL  BCM2708 Timer Tick
 32:       1614   ARMCTRL  dwc_otg, dwc_otg_pcd, dwc_otg_hcd:usb1
 52:      16983   ARMCTRL  BCM2708 GPIO catchall handler
 65:        264   ARMCTRL  ARM Mailbox IRQ
 66:          1   ARMCTRL  VCHIQ doorbell
 75:          1   ARMCTRL
 77:        147   ARMCTRL  bcm2708_sdhci (dma)
 83:       2300   ARMCTRL  uart-pl011
 84:        542   ARMCTRL  mmc0
192:      16984      GPIO  rpi_irq_gpio
FIQ:              usb_fiq
Err:          0
```

Nous observons bien le déclenchement de plus de seize mille interruptions (à raison de 2500 interruptions par seconde, leur nombre progresse très vite).

En branchant la broche 15 (entrée déclenchant l'interruption) sur le premier canal d'un oscilloscope et la broche 16 (sortie de la réponse du handler) sur le second canal, nous obtenons l'affichage suivant.

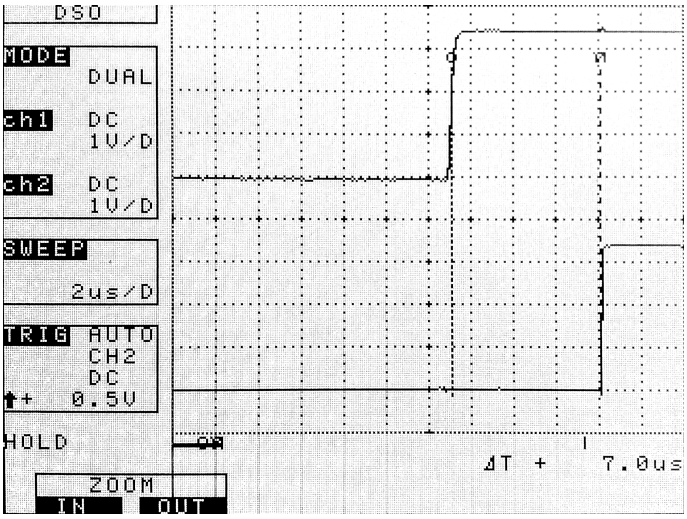


L'écran de l'oscilloscope nous confirme bien ce que nous attendions : à chaque front montant sur la broche 15, une interruption est déclenchée dont le handler fait basculer l'état de la broche 16.

Combien de temps faut-il pour que le Raspberry Pi réponde à une interruption sur une broche GPIO ?

Nous pouvons le savoir en zoomant sur le front montant déclencheur de l'IRQ.

Le résultat est visible sur la capture suivante :



Le temps de latence entre le signal déclencheur et le traitement de l'interruption (mesuré par les deux curseurs) est de 7 microsecondes, ce qui est très honorable pour un processeur de cette gamme. Toutefois dans de nombreux cas d'application, ce n'est pas la durée de latence typique – que nous voyons ici – mais sa durée maximale qui nous intéressera. Il est probable que suivant la charge du système (en tâches et en entrées-sorties), la latence fluctuera sur plusieurs dizaines voire centaines de microsecondes.

Pour améliorer ces performances, il serait nécessaire de se tourner vers des versions du noyau optimisant le comportement temps-réel comme le patch *Preempt-RT* ou l'extension Xenomai.

Conclusion

Nous avons mis au point un petit système minimal pour le Raspberry Pi, que nous avons construit et personnalisé intégralement. Ceci est intéressant lorsqu'on désire mettre en œuvre un système spécialisé pour une application bien précise, sans requérir d'interface utilisateur très avancée. Bien sûr, lorsque le système réclame des composants plus riches (environnement graphique, lecteur vidéo ou audio, compilateurs embarqués sur la cible, etc.) on aura peut-être intérêt à se tourner vers une distribution complète comme Raspbian, ArchLinux, etc. pour éviter des temps de compilation et d'intégration excessifs.

Bien qu'il soit très réduit (une petite cinquantaine de méga-octets), on notera que notre système incorpore tout de même une panoplie assez complète de services réseau, et des capacités de communication (Ethernet, Rs-232, SPI, GPIO, etc.) avancées pour communiquer avec des équipements périphériques.

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 (Raspberry Pi, Pandaboard, Igep v2...).

Christophe Blaess – christophe@blaess.fr