7

Optimisation et mise au point du système

Le chapitre précédent nous a permis de configurer le réseau de notre système Linux embarqué et nous disposons d'ores et déjà d'une version très fonctionnelle. Le présent chapitre va s'attacher à quelques points particuliers comme la configuration du clavier local, la mise en place d'un système d'authentification, la gestion de mémoire flash ou la comparaison des types de systèmes de fichier.

Configuration du clavier

Le clavier de la console Linux est configuré par défaut en QWERTY (clavier anglais/américain). Si vous devez faire des manipulations sur la console, il est intéressant de disposer d'une configuration de clavier AZERTY. Le système de configuration comprend d'une part le programme **loadkeys**, permettant le chargement de la disposition du clavier (carte ou *map*), et d'autre part la carte elle-même, contenue dans un fichier *.kmap* ou *.kmap.gz* (compressé avec **gzip**).

Les fichiers *kmap* se trouvent dans le répertoire **/lib/kbd/keymaps**. Dans les distributions classiques, ce répertoire contient un grand nombre de fichiers correspondant aux différents types de clavier selon les architectures matérielles. Dans notre cas, nous ne copierons que le fichier correspondant à un clavier français/latin1, ainsi que les différents fichiers annexes. Par ailleurs, nous copierons également le programme **loadkeys**, ainsi que le décompresseur **gunzip**.

```
mkdir -p /mnt/emb/lib/kbd/keymaps/i386/azerty
cd /lib/kbd/keymaps
cp i386/azerty/fr-latin1.kmap.gz  i386/azerty
```

```
cp -a -r i386/include i386
cp -a -r include .
cp /bin/loadkeys /mnt/emb/bin
cp /bin/gzip /mnt/emb/bin
cp -a /bin/gunzip /mnt/emb/bin
```

On peut maintenant modifier le fichier `/etc/sysconfig/general` de la cible afin de définir le type de clavier.

```
# General configuration
KEYTABLE=fr
KEYMAP=i386/azerty/fr-latin1
```

Puis on peut ajouter la ligne suivante au fichier `/etc/rc.d/rc.S` de la cible, juste avant le lancement de l'interpréteur de commande :

```
# Clavier
/bin/loadkeys /lib/kbd/keymaps/${KEYMAP}.kmap
```

Au démarrage du système, on obtient alors :

```
Loading /lib/kbd/keymaps/i386/azerty/fr-latin1.kmap.gz
```

Mise en place d'un système d'authentification

Un système Linux est par défaut multi-utilisateur et dispose donc d'un système d'authentification permettant aux utilisateurs référencés de se connecter. Cette authentification est le plus souvent basée sur la saisie d'un nom d'utilisateur ou *login*, suivie de l'entrée d'un mot de passe ou *password*. Dans la version actuelle de notre cible, il n'y a pas d'authentification et le démarrage du système se termine par le lancement direct d'un interpréteur de commande en mode super-utilisateur. Cette configuration n'a qu'un but pédagogique et un système réel aura très souvent besoin d'une fonction d'authentification. Celle-ci ne sera pas forcément utilisée sur une console locale car il faut pour cela que le système puisse disposer d'un écran et d'un clavier. L'authentification sera également utile pour une connexion *via* un terminal série ou bien à travers le réseau *via* une connexion Telnet, SSH ou FTP.

Les premières versions d'Unix utilisaient une authentification à travers une liste locale d'utilisateurs définie dans le fichier `/etc/passwd`. Les versions dérivées de System V R4 (SVR4) utilisent en plus le fichier `/etc/shadow` qui stocke les mots de passe cryptés associés aux utilisateurs définis dans `/etc/passwd`. Les distributions Linux utilisent également une telle configuration. Dans une configuration complexe connectée à un réseau local, l'authentification peut se faire à partir d'une base identique à la base locale, mais centralisée sur un serveur utilisant le protocole NIS (*Network Information Service*) développé par SUN Microsystems.

En ce qui concerne le fonctionnement global de l'authentification, Linux utilise également le principe d'Unix en respectant le schéma suivant :

- Le programme `getty` affiche le message d'invite sur une console. Par défaut, ce message correspond à *login*.
- Lorsque `getty` détecte une entrée de caractère sur la console, il la saisit et la passe au programme `login`.
- Le programme `login` se charge de demander le mot de passe à l'utilisateur en affichant par défaut la chaîne Password.
- En cas de saisie correcte du mot de passe, l'interpréteur de commande associé à l'utilisateur, et défini dans `/etc/passwd`, est exécuté.
- En cas de saisie incorrecte, un message Login incorrect est affiché et le processus recommence.

Le lancement du programme `getty` (appelé `agetty` sous Linux) est déclaré dans le fichier `/etc/inittab`.

```
c1:1235:respawn:/sbin/agetty 38400 tty1 linux
c2:1235:respawn:/sbin/agetty 38400 tty2 linux
```

`tty1` et `tty2` correspondent à deux consoles « virtuelles », sur un même écran. On passe d'une console à l'autre en utilisant les combinaisons de touches Ctrl-Atl-F1 et Ctrl-Alt-F2. Pour ajouter une nouvelle console virtuelle, il suffit d'ajouter une ligne au fichier `/etc/inittab` puis de redémarrer le système ou bien de taper `init q` pour forcer le processus `init` à relire son fichier de configuration.

Le système d'authentification décrit ici avait dans sa version initiale une assez forte limitation car la mise en place d'une nouvelle politique d'authentification nécessitait la modification des outils associés comme `login` et `passwd`. Le problème a été résolu de manière élégante en utilisant un système appelé PAM (*Pluggable Authentication Module*) qui permet de modifier le comportement de l'authentification en ajoutant simplement des bibliothèques dynamiques appelées « modules PAM ». Grâce à PAM, il est possible d'ajouter une méthode quelconque d'authentification en ajoutant au système une bibliothèque dynamique et un fichier de configuration. De nombreux modules existent déjà pour s'authentifier auprès d'annuaires de type LDAP ou équivalent.

De ce fait, les programmes comme `login` utilisent, outre celles qui sont habituelles, quelques autres bibliothèques, partagées.

```
# ldd /bin/login
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x4002d000)
        libpam.so.0 => /lib/libpam.so.0 (0x4005a000)
        libdl.so.2 => /lib/libdl.so.2 (0x40062000)
        libpam_misc.so.0 => /lib/libpam_misc.so.0 (0x40066000)
        libc.so.6 => /lib/i686/libc.so.6 (0x40069000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

De même, il est nécessaire d'installer sur le système un certain nombre de fichiers de configuration et bibliothèques partagées situées sur les répertoires `/etc/pam.d` et `/lib/security`.

Il est bien entendu possible de dupliquer ce comportement sur notre cible embarquée mais, dans un éternel souci de réduction de l'empreinte mémoire, nous allons considérer que le système PAM n'est pas utile dans notre cas. Pour ce faire, il sera bien entendu nécessaire de générer de nouvelles versions des programmes concernés ne faisant pas référence au système PAM. Nous devons pour cela installer les archives RPM sources de la distribution Red Hat 7.2, ou bien travailler directement sur les sources des programmes. Une rapide recherche par **rpm** nous indique les noms de paquetages RPM associés au programme **login**.

```
# rpm -qf /bin/login
util-linux-2.11f-9
```

Après récupération du paquetage source associé, nous l'installons sur notre système de développement.

```
rpm -ivh util-linux-2.11f-9.src.rpm
```

Le paquetage en question est maintenant installé dans l'arborescence RPM du système, soit **/usr/src/redhat**. L'archive des sources et les « patches » à appliquer avant compilation sur Red Hat 7.2 sont disponibles sur le sous-répertoire **SOURCES**. L'arborescence des sources, non « patchée » pour l'instant, est disponible sur le sous-répertoire **BUILD**. Le fichier de spécification RPM est disponible sur le sous-répertoire **SPECS**. Pour appliquer les patches, il faut déjà utiliser la commande :

```
rpm -bp /usr/src/redhat/SPECS/util-linux.spec
```

qui va générer une arborescence compatible avec Red Hat 7.2 sur le répertoire **BUILD**. Il faut maintenant modifier les options de compilation du paquetage afin de supprimer le support PAM (HAVE_PAM). On peut également modifier le support des mots de passes masqués (*shadow password*), ce qui permettra de limiter la liste des utilisateurs et mots de passe au fichier **/etc/passwd**.

```
cd /usr/src/redhat/BUILD/util-linux-2.11f
```

En éditant le fichier MCONFIG, on peut modifier les options de compilation du support PAM.

```
# If HAVE_PAM is set to "yes", then login, chfn, chsh, and newgrp
# will use PAM for authentication. Additionally, passwd will not be
# installed as it is not PAM aware.
HAVE_PAM=no

# If HAVE_SHADOW is set to "yes", then login, chfn, chsh, newgrp, passwd,
# and vipw will not be built or installed from the login-utils
# subdirectory.
HAVE_SHADOW=no
```

Le répertoire **login-utils** contient les sources des programmes **login**, **agetty** et **passwd** ; il suffit donc de compiler puis de copier les programmes au bon endroit :

```
cd login-utils
make agetty login passwd
```

```
cp agetty /mnt/smb/sbin
cp login passwd /mnt/emb/bin
```

Au niveau du fichier **/etc/inittab**, et en plus de l'ajout de l'appel à **agetty**, il faut indiquer au système de démarrer en mode multi-utilisateur (*multi-user*) et non plus en mono-utilisateur (*single-user*). Pour cela, on modifie le début du fichier.

```
# Default runlevel.
id:3:initdefault:
# System initialization (runs when system boots).
si:S:sysinit:/etc/rc.d/rc.S

# Script to run when going multi user.
rc:123456:wait:/etc/rc.d/rc.M
```

On ajoute un nouveau script **rc.M** qui pourra contenir des actions à lancer au passage en multi-utilisateur. Pour l'instant, le script contient seulement :

```
#!/bin/sh
# Script to run in multi-user mode
echo "Multi-user mode."
```

Après redémarrage du système, on obtient à la fin :

```
INIT: Entering runlevel: 3
Multi-user mode.
localhost.localdomain login:
```

Remarque

Dans le cas de l'utilisation de BusyBox, le système d'authentification est intégré et peut être mis en place en configurant l'entrée « Login/Password Management Utilities » de la configuration BusyBox accessible par make `menuconfig`.

Configuration des disques flash

Nous avons jusqu'à présent utilisé des disques durs ou disques flash munis d'une interface standard IDE. Ce choix est très avantageux dans la majorité des cas car il ne nécessite pas de configuration particulière du noyau Linux pour le pilotage de ces périphériques. L'interface IDE entraîne cependant un surcoût au niveau du prix du matériel et il est parfois judicieux d'utiliser des mémoires flash ne recourant pas à l'IDE. Dans cette section, nous allons décrire la configuration du noyau pour le pilotage de disque flash M-Systems de type DOC2000 (ou *Disk On Chip*). La principale raison en est simplement la large diffusion de ce produit dans le monde des applications embarquées ; nombreux sont en effet les constructeurs de cartes mère industrielles qui fournissent un emplacement destiné à ce type de composant. La capacité du composant va de 16 à 568 Mo. Nous décrirons aussi brièvement la configuration du pilote MTD pour les mémoires flash de type CFI (*Common Flash Interface*).

La configuration du noyau pour la DOC2000 peut s'effectuer de deux manières :

- en utilisant le pilote fourni par M-Systems sous forme d'un « patch » du noyau,
- en utilisant le driver MTD du noyau 2.4.

> **Attention**
>
> Le pilote MTD fonctionne avec les flash DOC2000 et Millenium mais ne fonctionne *pas* avec les flash Millenium Plus. Pour ces dernières, il est impératif d'utiliser le pilote fourni par M-Systems.

Utilisation du pilote M-Systems

Le pilote est disponible en téléchargement sur le site de M-Systems à l'adresse *http://www.m-sys.com*. Les composants sont répartis sur deux paquetages :

- un paquetage contenant des utilitaires MS-DOS destinés au formatage bas niveau de la flash ainsi que le *BIOS driver* M-Systems (`tffs_dostools_5041.zip` qui contient en particulier `DFORMAT` et `DINFO`) ;
- un paquetage contenant le patch du noyau (pour les versions 2.0, 2.2 et 2.4), la documentation d'installation ainsi que quelques utilitaires comme une version adaptée de LILO (`doc-linux-5.0.0.tgz`).

Les utilitaires DOS nécessitent l'installation d'un système d'exploitation compatible. Si l'on ne dispose pas de licence MS-DOS, on peut utiliser le clone DR-DOS/OPEN-DOS disponible en téléchargement auprès de la société Caldera sur l'URL *http://www.drdos.net*. Cette adresse permet de télécharger les images des trois disquettes d'installation. Les disques flash de type DOC 2000 sont fournis déjà formatés avec une partition de type DOS déjà créée. Pour formater le disque flash, il suffit d'utiliser la commande `DFORMAT` fournie dans l'archive des utilitaires DOS M-Systems.

```
DFORMAT /WIN:D000 /S:DOC504.EXB
```

Le fichier EXB correspond au *firmware* installé sur le disque flash. Une description complète des utilitaires DOS est disponible dans le document fourni avec l'archive `tffs_dostools_5041.zip`.

L'extraction de la deuxième archive crée le répertoire `doc-linux-5.0.0` qui contient le fichier `README.install` décrivant en détail l'installation du pilote sous Linux. Pour appliquer le patch aux sources du noyau, il suffit de faire :

```
cd doc-linux-5.0.0/drivers
./patch_linux linux-2_4-patch driver-patch
./mknod_fl
```

Si le répertoire `/usr/src/linux` n'existe pas (ce qui est souvent le cas sur un système équipé d'un noyau 2.4), il faudra passer le répertoire des sources du noyau à modifier en troisième paramètre du script `patch_linux`. La dernière ligne permet de créer les points d'entrées correspondant au pilote de la flash dans le répertoire `/dev`, soit :

```
# ls -l /dev/msys
total 0
brw-------    1 root     root     100,   0 avr 14 21:47 fla
brw-------    1 root     root     100,   1 avr 14 21:47 fla1
brw-------    1 root     root     100,   2 avr 14 21:47 fla2
brw-------    1 root     root     100,   3 avr 14 21:47 fla3
brw-------    1 root     root     100,   4 avr 14 21:47 fla4
brw-------    1 root     root     100,  64 avr 14 21:47 flb
...
```

Lorsque le patch est appliqué, il suffit de valider le support *M-Systems DOC device support* dans le menu *Block devices* de la configuration du noyau Linux. Si le disque flash est utilisé comme périphérique de démarrage, vous pouvez dans un premier temps valider le support dans la partie statique du noyau (cocher *y* et non *m*). Sachez cependant que la diffusion d'un tel noyau n'est pas conforme à la licence du noyau Linux (la GPL) car le noyau statique généré est constitué de code GPL auquel est ajouté du code non-GPL (venant de M-Systems). Pour être conforme à la GPL, vous devrez utiliser le support par modules, ce qui oblige à créer un disque mémoire de démarrage ou *initrd*. L'utilisation de la fonction **initrd** sera décrite au chapitre suivant.

Lorsque le noyau est recompilé puis installé comme décrit au chapitre 4, on peut d'ores et déjà redémarrer le système et vérifier la détection du disque flash dans les messages de démarrage, ou après coup en utilisant la commande **dmesg**.

```
Flash disk driver for DiskOnChip2000
Copyright (C) 1998,2001 M-Systems Flash Disk Pioneers Ltd.
DOC device(s) found: 2
Fat Filter Enabled
fl_geninit: registered device at major: 100
```

Le disque est dès maintenant accessible comme un disque classique en utilisant le device **/dev/msys/fla** (pour le premier disque flash détecté). Le disque contient par défaut une partition MS-DOS qu'il faudra certainement remplacer par une partition Linux en recourant à l'utilitaire **fdisk** décrit au chapitre 5.

```
# fdisk /dev/msys/fla
Command (m for help): p
Disk /dev/msys/fla: 16 heads, 4 sectors, 1001 cylinders
Units = cylindres of 64 * 512 bytes
        Device Boot    Start       End    Blocks   Id  System
/dev/msys/fla1   *         1      1001     32030   83  Linux

Command (m for help):
```

À partir de là, on peut créer n'importe quel type de partition (ext2, ext3, etc.) sur le device **/dev/msys/fla1**. La partition peut ensuite être montée comme suit :

```
mount -t ext3 /dev/msys/fla1 /mnt/flash
```

On peut également démarrer à partir du disque flash en utilisant la version spéciale de LILO fournie sur l'archive M-Systems, soit sous forme de paquetages RPM, soit sous

forme d'archives au format **tar.gz**. Cette version spéciale est nécessaire car elle prend en compte la géométrie spéciale du disque flash M-Systems, ce qui n'est pas le cas dans la version standard de LILO.

```
# cd doc-linux-5.0.0/lilo/
# ls -l
total 484
-rw-rw-rw-    1 root     root      668 aoû  1  2001 README.lilo
-rw-r--r--    1 root     root    34236 jui 30  2001 doc-lilo-0.21-19.i386.rhat52.rpm
-rw-r--r--    1 root     root    35369 jui 30  2001 doc-lilo-0.21-19.i386.rhat62.rpm
-rw-r--r--    1 root     root   190516 jui 30  2001 doc-lilo-0.21-19.src.rpm
-rw-r--r--    1 root     root    32223 jui 30  2001 lilo-bin.21.tar.gz
-rw-r--r--    1 root     root   183621 jui 30  2001 lilo-src.21.tar.gz
```

Avec l'installation d'un paquetage RPM binaire, on dispose du programme **/sbin/doc-lilo** et du secteur de boot spécial **doc.b** qui remplace le fichier standard **boot.b** utilisé par LILO, comme décrit au chapitre 4.

```
# rpm -ql doc-lilo
/boot/doc.b
/sbin/doc-lilo
```

Un fichier **lilo.conf** adapté au disque flash sera donc du style :

```
boot=/dev/msys/fla
install=/boot/doc.b
map=/boot/map
read-only
vga = normal
# End LILO global section
image = /boot/bzImage-2.4.18_msys
        root = /dev/msys/fla1
        label = linux
```

Dans certains cas, il sera parfois nécessaire de forcer le numéro du disque (paramètre **bios**) par les lignes suivantes dans les paramètres globaux :

```
disk=/dev/msys/fla
bios=0x80
```

Si la partition du disque est montée sur **/mnt/flash** et donc le fichier **lilo.conf** présent sur **/mnt/flash/etc**, on installera LILO sur le secteur de démarrage du disque flash en utilisant la commande :

```
/sbin/doc-lilo -r /mnt/flash -v
```

Après déconnexion des autres périphériques de boot plus prioritaires et redémarrage du système, on devrait démarrer sur le disque flash. Il est bien entendu nécessaire de modifier préalablement les fichiers de configuration qui dépendent du nom du périphérique contenant la partition principale (*root filesystem*), en particulier le fichier **/etc/fstab** présent sur le disque flash.

Attention

Il est également indispensable de créer les entrées spéciales /dev/msys sur le répertoire /mnt/flash/dev *avant* de démarrer le système sur le disque flash.

Certaines versions de LILO sont incompatibles avec le BIOS M Systems installé par DFORMAT. Il est alors conseillé de mettre à jour LILO avec une version plus récente. Les sources de LILO sont disponibles à l'adresse *ftp://sunsite.unc.edu/pub/Linux/system/boot/lilo*. La description du problème et des solutions se trouve dans le fichier README.install contenu dans l'archive doc-lilo-5.0.0.tgz.

Utilisation du pilote MTD

Le projet MTD (*Memory Technology Device*) est mené par David Woodhouse (*dwm2@infradead.org*) et la page d'accueil est disponible sur *http://www.linux-mtd.infradead.org*. Ce pilote a l'avantage d'être totalement Open Source et de supporter un grand nombre de disques flash, dont les produits M-Systems, à l'exception du modèle Millenium Plus. Le but du projet est de mettre en place une interface simple entre le pilote matériel de la flash et les couches supérieures du noyau Linux. Ce projet est également couplé au développement du système de fichier JFFS2 (*Journalling Flash File System* version 2). MTD est intégré à l'arborescence des noyaux 2.4 et 2.6.

Le seul défaut du projet MTD est un manque relatif de documentation concernant la configuration et l'utilisation des composants. Un fichier, **mtd-jffs-HOWTO.txt**, reprend cependant les procédures d'installation et de configuration. Il est disponible depuis la page d'accueil du projet.

Les dernières versions contenant les pilotes et divers utilitaires sont disponibles depuis le serveur *CVS* du projet ou bien sous forme d'une copie journalière au format **tar.gz** (*daily snapshot*) depuis la page d'accueil. Le projet est en perpétuelle évolution et il est donc conseillé de télécharger la dernière version auprès du serveur CVS plutôt que d'utiliser celle fournie dans le noyau Linux 2.4 ou 2.6. Il faut cependant noter que seule la version 2.6 est maintenue à ce jour. Pour ce faire, il faut copier l'arbre CVS en local au moyen des commandes :

```
cd /usr/src
cvs -d :pserver:anoncvs@cvs.infradead.org:/home/cvs login
```

Après saisie du mot de passe *anoncvs*, il faut faire :

```
cvs -d :pserver:anoncvs@cvs.infradead.org:/home/cvs co mtd
```

ce qui a pour effet de créer un répertoire **/usr/src/mtd**. On doit ensuite se positionner dans le répertoire **patches** afin de modifier les sources du noyau courant.

```
cd /usr/src/mtd/patches
sh patchin.sh /usr/src/linux-2.4
```

La configuration du noyau pour l'utilisation de MTD est décrite dans le fichier **mtd-jffs-HOWTO.txt**. Si nous devons utiliser le disque flash comme périphérique de démarrage, il sera plus simple de valider le support MTD dans la partie statique du noyau. La figure ci-après indique la configuration à effectuer dans le menu principal du pilote MTD.

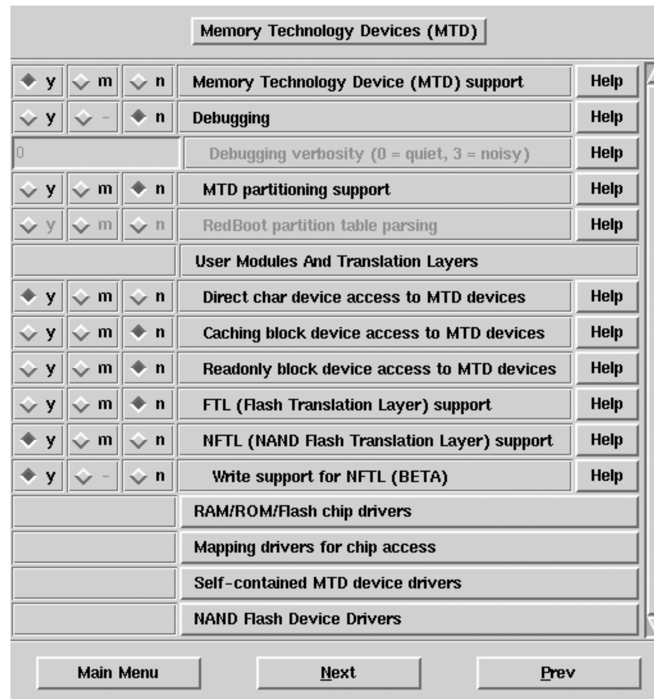

Figure 7-1

Configuration principale de MTD.

En cliquant sur l'option *Self-contained MTD device drivers*, ce qui correspond entre autres aux disques flash M-Systems ; on obtient l'écran de configuration suivant :

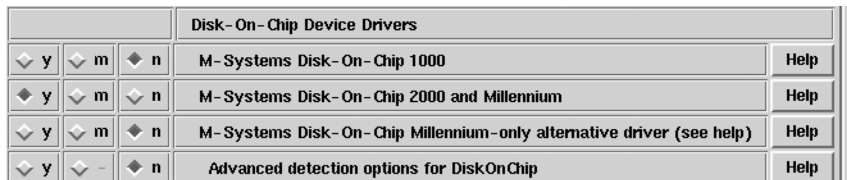

Figure 7-2

Validation du support DOC 2000.

La validation du support du système de fichiers JFFS2 est quant à elle effectuée dans le menu *File systems* du menu principal de configuration du noyau, comme indiqué sur la figure ci-après :

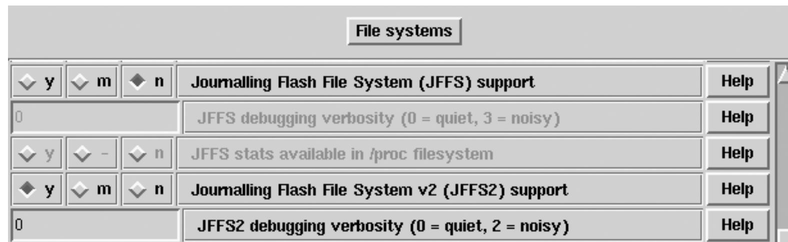

Figure 7-3
Validation du support JFFS2.

Comme pour le pilote M-Systems, il est nécessaire de créer les points d'entrée correspondants dans le répertoire **/dev**. On utilise pour cela le script **MAKEDEV** fourni dans le répertoire **util** des sources du projet MTD.

Après compilation et installation du nouveau noyau, le redémarrage du système doit provoquer l'affichage d'un message, tel que celui-ci :

```
DiskOnChip 2000 found at address 0xD0000
Flash chip found: Manufacturer ID: 98, Chip ID: 73 (Toshiba TH58V128DC)
2 flash chips found. Total DiskOnChip size: 32 MiB
```

Le pilote MTD permet également d'obtenir des informations *via* le système de fichiers virtuel **/proc**.

```
# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 02000000 00004000 "DiskOnChip 2000"
```

Comme pour le pilote M-Systems, on peut manipuler le disque flash avec les utilitaires classiques.

```
# fdisk /dev/nftla
Command (m for help): p
Disk /dev/nftla: 16 heads, 4 sectors, 1001 cylinders
Units = cylindres of 64 * 512 bytes

     Device Boot    Start       End    Blocks   Id  System
/dev/nftla1   *         1      1001     32030   83  Linux

Command (m for help):
```

Une fois la partition créée, on peut alors créer un système de fichier ext2 ou ext3 en utilisant la commande **mke2fs** sur le device **/dev/nftla1**, qui est un device de type bloc. La création d'un système JFFS2 utilise le device de type caractère **/dev/mtd0** et sera décrite dans la prochaine section concernant les systèmes de fichiers. Le montage de la partition ne présente pas de difficulté.

```
# mount -t ext3 /dev/nftla1 /mnt/flash
```

Le boot sur le disque flash nécessite là encore une version modifiée de LILO. Une archive du LILO modifié est disponible sur le répertoire **patches** des sources du projet MTD.

```
# tar tzvf patches/lilo-mtd.tar.gz
-rw-r--r-- dvir/dvir      4540 2000-03-18 16:43:00 boot.b-mtd
-rwxr-xr-x dvir/dvir     51500 2000-03-19 17:44:00 lilo-mtd
-rw-r--r-- dvir/dvir      2361 2000-03-04 19:51:00 lilo21-mtd-patch
```

Un fichier **lilo.conf** adapté est très similaire au fichier précédent.

```
boot=/dev/nftla
install=/boot/boot.b-mtd
map=/boot/map
read-only
vga = normal
# End LILO global section
image = /boot/bzImage-2.4.18_mtd
        root = /dev/nftla1
        label = linux
```

On installe le secteur de démarrage avec la commande :

```
# /sbin/lilo-mtd -r /mnt/flash -v
Patched LILO for M-Systems' DiskOnChip 2000
LILO version 21, Copyright 1992-1998 Werner Almesberger

Reading boot sector from /dev/nftla
Warning: /dev/nftla is not on the first disk
Merging with /boot/boot.b-mtd
Boot image: /boot/bzImage-2.4.18_mtd
Added linux *
Backup copy of boot sector in /boot/boot.5D00
Writing boot sector.
```

Attention

Il est également indispensable de dupliquer sur /mnt/flash/dev les nœuds créés par le script MAKEDEV *avant* de démarrer le système sur le disque flash.

Les mémoires flash CFI (Common Flash Interface)

Le pilote MTD précédemment décrit permet également de piloter les mémoires de type CFI. Le standard CFI a été développé par Intel et permet d'utiliser des mémoires de marques différentes avec la même couche logicielle. Une introduction au standard CFI est disponible en ligne chez Intel à l'adresse *http://developer.intel.com/design/flcomp/cfi_bg.htm.*

L'utilisation nécessite le support générique MTD comme dans le cas des DOC 2000. En plus de cela, il faut sélectionner le type de flash en cliquant sur RAM/ROM/Flash chip driver dans la page principale de configuration MTD. On obtient l'écran suivant :

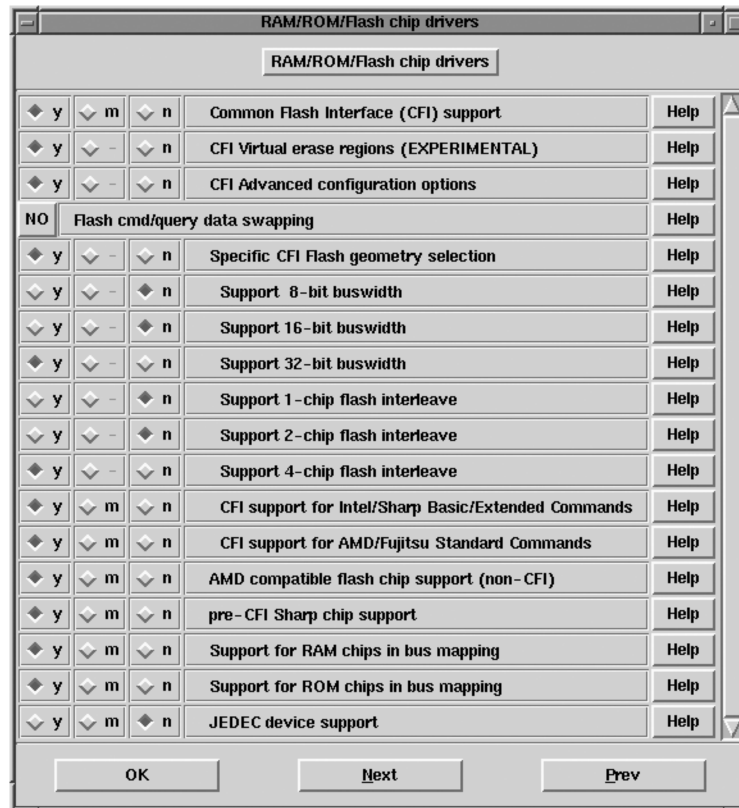

Figure 7-4
Sélection du type de flash.

sur lequel on devra valider le type de flash et éventuellement la géométrie. Si la flash est visible dans l'espace mémoire du processeur, on devra sélectionner les caractéristiques de l'adressage en cliquant sur *Mapping drivers for chip access* dans la page principale de configuration MTD.

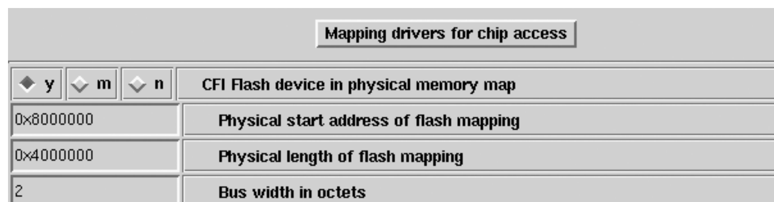

Figure 7-5
Adressage de la flash.

Utilisation d'une clé USB

La clé USB est un périphérique de plus en plus populaire et dont le coût n'a cessé de baisser ces dernières années. On peut aujourd'hui trouver des clés USB de bonne capacité pour quelques dizaines d'euros au point de constituer un support marketing de choix. Certaines sociétés commercialisent des clés USB déjà équipées du système Linux (voir *http://www.flonix.com*) mais il est relativement aisé de mettre en place une distribution réduite sur un tel périphérique. Une telle distribution peut être très intéressante à des fins de démonstration, de tests ou d'outils de maintenance car on pourra alors avoir sa propre distribution dans la poche !

Une clé USB est vue sous Linux comme un disque SCSI. La plupart des distributions actuelles intègrent le support des clés USB en natif (dans le noyau livré) et l'insertion de la clé provoque l'affichage du message suivant dans les traces du système.

```
Sep 21 12:13:26 localhost kernel: hub.c: new USB device 00:07.2-1, assigned address 2
Sep 21 12:13:26 localhost kernel: usb.c: USB device 2 (vend/prod 0xea0/0x6803)
➥is not claimed by any active driver.
Sep 21 12:13:27 localhost kernel: SCSI subsystem driver Revision: 1.00
Sep 21 12:13:27 localhost kernel: Initializing USB Mass Storage driver...
Sep 21 12:13:27 localhost kernel: usb.c: registered new driver usb-storage
Sep 21 12:13:27 localhost kernel: scsi0 : SCSI emulation for USB Mass Storage devices
Sep 21 12:13:27 localhost kernel: Vendor: OTi       Model: Flash Disk        Rev: 1.11
Sep 21 12:13:27 localhost kernel: Type:   Direct-Access         ANSI SCSI revision: 02
Sep 21 12:13:27 localhost kernel: USB Mass Storage support registered.
```

La clé est utilisable à travers le fichier spécial `/dev/sda` et l'on peut bien entendu la partitionner, formater, monter, etc. La clé suivante contient deux partitions de 16 Mo, l'une au format VFAT pour les transferts Windows, l'autre au format EXT3 contenant une mini-distribution Linux. L'existence de cette deuxième partition Linux ne trouble en rien l'utilisation de la clé sous Windows.

Le démarrage sur la clé pose quelques petits problèmes :

- Le BIOS du PC doit permettre le démarrage sur support disque USB. C'est le cas de la majorité des PC spécialisés pour les applications embarquées (EDEN, LEX, etc.) mais ce n'est pas le cas des PC grand public.
- La détection de la clé USB lors du démarrage du noyau Linux est asynchrone par rapport au montage du système de fichiers principal (ou *root filesystem*). De ce fait, un noyau Linux standard ne pourra pas utiliser comme *root filesystem* une partition d'une clé USB. Cependant la modification à apporter au noyau pour permettre cette utilisation est minime et se limite à un patch de quelques lignes appliqué au fichier `init/do_mounts.c`. Il existe plusieurs patches disponibles, celui-ci est accessible depuis le FAQ du site *http://www.linux-usb.org*. Ce problème est également présent pour le noyau 2.6 et le même type de patch sera applicable.

Le principe de la modification est simple : on ajoute une attente de quelques secondes (maximum 5 essais) pour laisser le temps à la clé USB d'être détectée. Ce n'est pas très élégant mais c'est simple et cela fonctionne. Les quelques lignes de code à ajouter à la fin

de la fonction **mount_root()** sont présentées ci-dessous. Il existe d'autres patches disponibles et accessibles depuis la FAQ du site *http://www.Linux-usb.org*.

```
/*
 * Patch for USB boot
 */
          {
            /* begin jordi **************************** */
            static DECLARE_WAIT_QUEUE_HEAD (jordi_queue);
            printk ("\n\n\n--------------------------------\n");
            printk (" WAITING FOR A WHILE (1000) \n");
            printk (" TO DETECT THE USB DISK \n");
            sleep_on_timeout (&jordi_queue, 1000);
            printk ("--------------------------------\n\n\n");
            /* end jordi **************************** */
          }
/* End of patch */
        mount_block_root("/dev/root", root_mountflags);
}
```

Au niveau de la configuration du noyau, il faudra valider les différentes options (SCSI et USB) en statique pour permettre la détection de la clé USB en tant que disque de démarrage. Les options à valider sont simples :

- Valider *SCSI support* et *SCSI disk support* dans le menu *SCSI support*.
- Valider *Support for USB*, le type de contrôleur USB (*OHCI* ou *UHCI*) ainsi que *USB mass storage support* dans le menu *USB support*.

Au niveau du fichier **/etc/fstab**, on fera référence au fichier spécial **/dev/sda1** pour monter le système de fichiers principal.

```
/dev/sda1        /      ext3    defaults        1 1
```

Concernant l'installation de LILO, le principe est le même que pour les disques durs IDE (puisque c'est également un périphérique en mode bloc) sauf qu'il faut utiliser **/dev/sda** au lieu de **/dev/hda** :

```
boot=/dev/sda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
default=linux

image=/boot/bzImage-2.4.27-usb
        label=linux
        read-only
        root=/dev/sda1
```

Les différents types de systèmes de fichiers

Nous avons installé sur notre cible le système de fichier ext3, dérivé de ext2 et très fréquemment utilisé sur Linux. Pour un système embarqué, nous rappelons que la contrainte d'un système de fichier « journalisé », donc résistant aux arrêts imprévus, est très importante.

Ext2/ext3

Le système de fichier ext3 est l'extension journalisée du système de fichier ext2 qui est le type le plus répandu sur les systèmes Linux. Historiquement, ext2 a été développé au MASI, ex-laboratoire de l'université Jussieu de Paris, par Rémy Card, pour une version universitaire d'un système Unix *like* appelé *MASIX*. Les deux systèmes sont compatibles et l'on peut même utiliser une partition ext3 sur un noyau supportant uniquement ext2, mis à part le journal, qui ne sera pas pris en compte. Nous avons utilisé ext3 au chapitre 5 car il est d'un emploi très aisé sur n'importe quel périphérique en mode bloc. Il est intégré à l'arborescence des sources du noyau Linux 2.4 et maintenu par la société Red Hat.

ReiserFS

Contrairement à ext3, ReiserFS fut conçu dès le départ comme un système de fichier journalisé. De conception plus récente, il est très efficace dans le cas de partitions de grandes tailles gérant nombre de petits fichiers. Il a été initialement développé par Hans Reiser à l'université de Stanford à Palo Alto. Hans Reiser a depuis créé une société commerciale autour de ReiserFS (NAMESYS). Bien qu'intégré dans l'arborescence des sources de noyaux Linux 2.4 et 2.6, les dernières versions sont disponibles sur *http://www.namesys.com*. L'utilisation sur la Red Hat 7.2 nécessite l'installation du paquetage `reiserfs-utils`. On peut créer un tel système de fichier sur un périphérique de type bloc en utilisant la commande `mkreiserfs`.

Il est cependant peu adapté à un environnement embarqué car limité à une taille minimale de partition de 32 Mo.

JFFS2

JFFS2 (*Journaling Flash File System, version 2*) est intimement lié au projet MTD décrit dans ce chapitre. Il a pour origine le système JFFS initialement développé par la société suédoise AXIS (*http://developer.axis.com/software/jffs)* sur le noyau 2.0. JFFS et JFFS2 sont maintenant intégrés dans l'arborescence des noyaux 2.4 et 2.6 et les dernières versions sont disponibles sur le serveur CVS du projet MTD décrit précédemment. Outre des améliorations structurelles concernant la gestion des blocs erronés (*bad blocks*) et la récupération d'espace (*garbage collector*), une caractéristique intéressante de JFFS2 par rapport à JFFS est la fonction de compression des données en temps réel. La validation du support de JFFS et/ou JFFS2 se fait au niveau du menu *File systems* de la configuration du noyau.

Pour créer un système de fichier JFFS2, il faut utiliser le programme **mkfs.jffs2** disponible sur la distribution MTD ou sur le serveur CVS dans le répertoire **util**. La procédure est décrite dans le document **mtd-jffs-HOWTO.txt**. En résumé, outre la détection de la mémoire flash décrite aux sections précédentes, la création du système de fichier se résume aux actions suivantes :

```
# mkfs.jffs2 -d /home/jffs2_stuff -o /tmp/jffs2.img
# cp  /tmp/jffs2.img /dev/mtd0
```

La première ligne correspond à la création d'une image du système de fichier dans un fichier unique à partir d'un répertoire contenant l'arborescence à copier. La deuxième ligne copie simplement ce fichier sur le device correspondant à la mémoire flash utilisée. Si la mémoire flash n'a jamais été initialisée, notez qu'il est nécessaire de le faire avec la commande **erase** disponible sur le même répertoire d'utilitaires.

```
# erase /dev/mtd0
```

Pour monter le système de fichier, il suffit de faire :

```
# mount -t jffs2 /dev/mtdblock0 /mnt/flash
```

Notez que l'on utilise le device **mtdblock0** car la commande **mount** nécessite un périphérique en mode bloc.

CRAMFS

CRAMFS (*Compressed ROM File System*) est un système de fichier en lecture seule utilisant l'algorithme de compression de la **zlib** (identique à **gzip**). La taille maximale de chaque fichier est limitée à 16 Mo et la taille totale du système de fichier est limitée à 256 Mo. Pour utiliser CRAMFS, il faut déjà valider le support dans la configuration du noyau dans la rubrique *Filesystems*. CRAMFS utilise une image générée avec l'utilitaire **mkcramfs** qui permet de créer un fichier image à partir d'un répertoire. Les sources de l'utilitaire sont disponibles à l'adresse *http://cvs.bofh.asn.au/cramfs*.

```
# mkcramfs MicroW microw.img
Directory data: 14092 bytes
Everything: 4252 kilobytes
Super block: 76 bytes
CRC: 8647fdf8
```

Le fichier image pourra ensuite être monté en utilisant la fonction de *loopback* du noyau Linux. Il faut pour cela avoir validé en module ou en statique l'option dans le menu *Block devices/Loopback device support* de la configuration du noyau. Cette fonction permet de monter un fichier image (par exemple, un fichier image ISO d'un CD-Rom) exactement comme si l'on montait le support réel. Dans le cas du fichier CRAMFS, on fera :

```
mount -t cramfs -o loop microw.img /mnt/cramfs
```

si **/mnt/cramfs** est un point de montage valide.

Utilisation des disques mémoire

Le noyau Linux offre la possibilité de travailler sur des disques mémoire ou *ramdisks*. L'utilisation d'un disque mémoire a de nombreux avantages, dont la rapidité d'accès aux données mais aussi le faible coût de la RAM par comparaison avec d'autres supports physiques comme la mémoire flash. Pour utiliser des disques mémoire, il faut déjà valider le support dans le noyau Linux avec le menu *Block devices* comme décrit dans la figure ci-après :

Figure 7-6
Support des disques mémoire.

Le principe du noyau Linux est d'allouer automatiquement un certain nombre de disques de taille fixe, cette taille étant par défaut définie dans le noyau. La valeur par défaut est de 4096 octets. On peut modifier cette valeur dans la configuration présentée ci-après, mais également passer la nouvelle valeur lors du démarrage du système à travers le chargeur LILO :

```
LILO: linux ramdisk_size=8192
```

Comme décrit précédemment, cette configuration pourra être ajoutée au fichier **/etc/lilo.conf** grâce à une directive **append** :

```
append="ramdisk_size=8182"
```

Chaque disque mémoire est vu à travers un pilote de type bloc **/dev/ramX**, X variant de 0 à 20. Ce device est utilisable comme n'importe quel périphérique de stockage en mode bloc. On pourra par exemple créer un système de fichier d'une taille de 2 Mo en faisant :

```
# dd if=/dev/zero of=/dev/ram2 bs=1k count=2048
# mke2fs -vm0 /dev/ram2 2048
```

Le système peut ensuite être monté et utilisé comme n'importe quel système de fichier.

```
# mount -t ext2 /dev/ram2 /mnt/tmp
# df /mnt/tmp
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/ram2                 2011        13      1998   1% /mnt/tmp
```

On peut donc imaginer de remplir ce système de fichier avec les composants d'un *root file system* minimal comme nous l'avons fait au chapitre 5. Une fois le système créé, il est possible de compresser le résultat en un fichier unique. Dans l'exemple qui suit, nous

considérons que le système de fichier en question est suffisamment petit pour tenir au maximum sur une disquette après compression avec **gzip**.

```
dd if=/dev/ram2 bs=1k count=2048 | gzip -v9 > /tmp/ram_image.gz
```

Nous pouvons ensuite créer une disquette de démarrage (*bootable*) à partir d'un noyau sur lequel nous aurons validé le support des disques mémoire.

```
dd if=bzImage of=/dev/fd0 bs=1k
```

Il faut ensuite copier l'image compressée du système de fichier sur la même disquette (et sur une autre, si l'espace restant n'est pas suffisant). Si l'on estime que le noyau occupe au maximum 400 Ko et que l'espace doit être suffisant sur la même disquette, on peut copier l'image après le noyau en faisant :

```
dd if=/tmp/ram_image.gz of=/dev/fd0 bs=1k seek=400
```

On indique ensuite au noyau que le root file system sera lu sur une disquette, et ce au moyen de la commande **rdev**.

```
rdev /dev/fd0 /dev/fd0
```

Il reste à indiquer au noyau la localisation du système de fichier à l'aide de la même commande. La manipulation du disque mémoire avec **rdev** est effectuée grâce à l'option **-r** à laquelle on passe le nom du device contenant l'image du disque mémoire (ici, **/dev/fd0** pour la disquette) suivie de la valeur d'un masque décimal de paramétrage. Le tableau suivant donne la signification des valeurs.

Tableau 7-1. Paramètres de rdev -r

Bits	Paramètre	Description
0 à 10	ramdisk_start	Position dans le support
11 à 13	Non utilisé	Fixé à 0
14	load_ramdisk	Chargement du ramdisk
15	prompt_ramdisk	Arrêt avant chargement

Les noms des paramètres indiqués correspondent à ceux que LILO devrait passer au noyau pour obtenir le même résultat que la commande **rdev**.

Les bits 0 à 10 indiquent la position de l'image compressée du disque mémoire par rapport au début du support. Dans notre cas, la valeur est 400. Si l'image est sur une autre disquette, la valeur est 0.

Le bit 14 indique au noyau qu'un disque mémoire doit être chargé et le bit 15 indique au noyau de demander confirmation à l'utilisateur avant de charger le disque mémoire afin de permettre le changement de support si nécessaire. Si nous validons ces 2 bits nous obtenons 2^15 + 2^14 + 400 = 49 552. Si l'arrêt n'est pas nécessaire la valeur sera 2^14 + 400 = 16 784. On en déduit la commande.

```
rdev -r /dev/fd0 16784
```

Dans le cas d'une image copiée sur une deuxième disquette la valeur sera 2^15 + 2^14 + 0 = 49 152, soit la commande :

```
rdev -r /dev/fd0 49152
```

ce qui correspond à une ligne de commande LILO du type :

```
ramdisk_start=0 load_ramdisk=1 prompt_ramdisk=1
```

La technique du disque mémoire est fréquemment utilisée pour la fonction `initrd` qui permet à un noyau de monter un système de fichier principal (*root file system*) initial à partir d'un disque mémoire. Le but est de définir un démarrage en deux phases, une première permettant d'accéder à tous les périphériques (par exemple, un disque SCSI de démarrage) et une seconde phase utilisant un noyau dont la partie statique est réduite au minimum, donc ne contenant pas les pilotes SCSI. Le nom du fichier correspondant à l'image mémoire à charger est défini dans le fichier `lilo.conf`.

```
image=/boot/vmlinuz-2.4.7-10
        label=linux
        initrd=/boot/initrd-2.4.7-10.img
        read-only
        root=/dev/hda1
```

Le fichier `.img` est compressé avec `gzip`, et l'option `-9`. On peut visualiser son contenu en utilisant la fonction *loopback* décrite précédemment :

```
# file initrd-2.4.7-10.img
initrd-2.4.7-10.img: gzip compressed data, deflated, last modified: Fri Apr 26
➥ 10:46:07 2002, max compression, os: Unix
# gunzip -c initrd-2.4.7-10.img > /tmp/i
# ls -l /tmp/i
-rw-r--r--    1 root     root      3072000 avr 29 17:27 /tmp/i
# mount -t ext2 -o loop /tmp/i /mnt/tmp
# ls /mnt/tmp/
bin  dev  etc  lib  linuxrc  loopfs  proc  sbin  sysroot
```

La racine du système de fichier contient un script `linuxrc` qui est exécuté après le chargement du disque mémoire initial et qui permet normalement de basculer sur le système définitif. Le disque mémoire est ensuite monté sur un point de montage `/initrd`. Quand le script `linuxrc` n'existe pas, le système ne sort jamais du disque mémoire et l'on peut donc envisager d'utiliser cette fonction pour un système embarqué minimal.

Un exemple d'utilisation de CRAMFS et disque mémoire

Si l'on observe la structure d'un système Linux, on s'aperçoit qu'une grande majorité du système de fichiers peut être configurée en lecture seule. Mises à part quelques exceptions que nous décrirons plus loin, les parties nécessitant l'accès en lecture-écriture sont les suivantes :

- le répertoire `/var` contenant des données de fonctionnement souvent volatiles ;

- le répertoire `/tmp` encore plus volatile !
- les répertoires de configuration (exemple : définition d'adresse IP, etc.) ;
- les répertoires d'utilisateurs ou applicatifs (exemple : stockage de données enregistrées par le système). Ces derniers pourront être placés sur un véritable disque dur et il est également possible de monter la partition à la demande afin de limiter les risques.

Outre la dernière catégorie que nous ne traiterons pas ici, il est donc relativement simple de mettre en place l'architecture suivante :

- La majorité du système est sur une partition en lecture seule de type CRAMFS.
- Le répertoire `/var` (point de montage) est sur un disque mémoire.
- Le répertoire `/tmp` est un lien symbolique sur /var/tmp.
- Le répertoire de configuration utilisera un format classique (EXT2, EXT3, JFFS2 ou même Minix). La partition est de très faible taille. On peut même imaginer de stocker cette configuration sur une partition non formatée (au format TAR directement écrit sur le fichier spécial de la partition) ce qui limite les risques de problèmes de système de fichiers dus à la coupure d'alimentation. Dans un système réel on pourra même sauver les fichiers de configuration sur une partition créée sur une mémoire sauvegardée de type SRAM.

À titre d'exemple concret nous supposerons que nous utilisons un DiskOnChip de 8 Mo. Cette configuration correspond au cas réel d'un routeur Wi-Fi construit sur la base d'un PC de type PC Light basé sur un processeur VIA C3. Le principe est d'utiliser trois partitions sur la mémoire flash :

- Une première partition `/dev/nftla1` correspondant à `/boot` et contenant le noyau et les composants de LILO. Il n'est pas nécessaire que cette partition soit montée lors du fonctionnement du système mais seulement lors de la mise à jour éventuelle du noyau depuis un environnement de développement. De ce fait, cette partition sera formatée en EXT2.
- Une deuxième partition `/dev/nftla2` contenant le système de fichiers. Elle sera formatée en CRAMFS comme décrit plus loin.
- Une troisième partition `/dev/nftla2` contenant les quelques fichiers de configuration. Elle est de très petite taille et utilise en première approche un formatage EXT2.

Le fichier `/etc/fstab` décrivant les montages sur le système cible aura l'allure suivante :

```
bash-2.05# cat /etc/fstab
/dev/nftla2     /                       cramfs  defaults        1       1
/dev/nftla3     /data                   ext2    defaults        0       0
none            /dev/pts                devpts  mode=622        0       0
none            /proc                   proc    noauto          0       0
/dev/ram0       /var                    ext2    defaults        0       0
```

On note que la partition `/boot` n'est pas montée mais que nous avons une partition `/var` montée sur un disque mémoire `/dev/ram0`. Pour la deuxième partition (système de

fichiers principal), nous utiliserons le système de fichiers CRAMFS. Pour utiliser CRAMFS, il faut disposer du programme **mkcramfs**. La version livrée avec certaines distributions ne fonctionne pas forcément très bien et il vaut mieux récupérer la version officielle disponible sur le site du projet, soit *http://sourceforge.net/projects/cramfs*.

Pour construire une partition CRAMFS sur un support physique on devra tout d'abord créer l'image du répertoire au format CRAMFS.

```
# mkcramfs my_root my_root.img
Directory data: 14092 bytes
Everything: 4252 kilobytes
Super block: 76 bytes
CRC: 8647fdf8
```

On pourra ensuite copier l'image sur la partition par un simple **dd** ou un **cp** :

```
# dd < my_root.img > /dev/nftla2
```

Au niveau du noyau, il faudra bien entendu valider le support de CRAMFS en statique au niveau du menu File systems de la configuration du noyau. Puisque nous utilisons également un disque mémoire, il faudra valider le support Ramdisk le menu Block devices. La taille par défaut de 4 Mo pour le disque mémoire est suffisante pour notre application.

Si nous faisons référence au chapitre 4, nous savons que le démarrage du système est divisé en 5 étapes :

- le démarrage du système par LILO (Linux LOader) ou un programme équivalent type GRUB ;
- le chargement du noyau ;
- le lancement par le noyau du processus **init** (soit **/sbin/init**) ;
- lecture du fichier /etc/inittab par le processus **init**. Ce fichier contenant le nom du fichier de démarrage comme décrit ci-dessous.

```
# System initialization (runs when system boots).
  si:S:sysinit:/etc/rc.d/rc.S
```

- l'exécution du script ci-dessus.

Sachant que le répertoire **/var** est placé dans un disque mémoire, il est nécessaire de construire l'arborescence de ce dernier à chaque démarrage du système. On aura donc dans le fichier **rc.S** les lignes suivantes :

```
# Create /var (EXT2 filesystem)
/sbin/mke2fs -vm0 /dev/ram0 4096

# mount file systems in fstab (and create an entry for /)
# Mount /proc first to avoid warning about /etc/mtab
mount /proc
/bin/mount -at nonfs
```

```
# Populate /var
mkdir -p /var/tmp /var/log /var/lock /var/run /var/spool /var/lib/dhcp /var/run/dhcpc
mkdir -p /var/cron/tabs /var/www/html /var/spool/cron /var/spool/mail /var/ppp
chmod a+rwx /var/tmp
...
```

Outre la création de **/var**, on notera le lien symbolique de **/etc/mtab** vers **/proc/mounts**. Ce lien est nécessaire car **/etc** n'est pas accessible en écriture.

```
lrwxrwxrwx    1 root     root  12 Jun 23  2003 mtab -> /proc/mounts
```

De même, on notera l'utilisation fréquente des liens symboliques afin de permettre à des fichiers créés au démarrage d'apparaître dans le système de fichiers en lecture simple (voir exemple de **/etc/resolv.conf** ci-dessous). Les fichiers variables sont systématiquement placés dans le répertoire **/var/run** :

```
# ls -l /etc/resolv.conf
lrwxrwxrwx   1 root    root     20 Sep 24 07:53 /etc/resolv.conf -> /var/run/resolv.conf
```

Concernant les fichiers de configuration, ils sont placés dans le répertoire **/data** associé à la troisième partition :

```
# ls -l /data/sysconfig/
total 6
-rw-r--r--    1 65534    nobody         41 Jun 23  2003 general
-rw-r--r--    1 65534    nobody        349 Sep 17 21:14 network
-rw-r--r--    1 root     root          174 Jun 23  2003 ppp
-rw-r--r--    1 root     root          150 Sep  6  2003 pppoe
-rw-r--r--    1 root     root           16 Jun 23  2003 syslogd.opts
-rw-r--r--    1 root     root          150 Jun 24  2003 wireless
```

Mise au point des programmes

Utilisation de GDB

La mise au point des programmes ou « débogage » est un mal nécessaire connu de tous les développeurs. Lorsqu'on travaille sur une station Linux, la tâche est facilitée par l'utilisation du débogueur symbolique GDB (*GNU DeBugger*) qui permet d'exécuter le programme en pas à pas, de gérer des points d'arrêt, ou pire encore.

Si le système cible utilise une version très proche de celle du poste de développement, nous essaierons la plupart du temps de régler les problèmes directement sur la station de développement en s'approchant au maximum de l'environnement de la cible. Dans le cas où des problèmes inhérents à l'environnement cible subsistent, GDB offre la possibilité d'effectuer un débogage à distance (*remote debugging*) à travers un lien RS-232 ou une connexion réseau TCP/IP. Même si ce n'est pas directement lié au sujet de l'ouvrage, nous allons brièvement rappeler quelques concepts généraux concernant l'utilisation de GDB.

Si nous considérons le petit programme C qui suit :

```
#include <stdio.h>
#include <stdlib.h>

void affiche (int v)
{
     printf ("valeur= %d\n", v);
}

main (int ac, char **av)
{
     int i;
     for (i = 0 ; i < 5 ; i++)
          affiche (i);
}
```

Ce dernier pourra être compilé sous Linux avec la commande :

```
$ gcc -g -o affiche affiche.c
```

Puis exécuté par :

```
$ ./affiche
valeur= 0
valeur= 1
valeur= 2
valeur= 3
valeur= 4
```

L'option `-g` indique au compilateur d'ajouter les informations de débogage utilisables par GDB. Si nous lançons la même exécution sous la commande **gdb**, nous obtenons :

```
$ gdb affiche
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

Si nous posons un point d'arrêt sur la fonction **affiche()**, et que nous démarrons le programme, nous obtenons le résultat suivant :

```
(gdb) b affiche
Breakpoint 1 at 0x8048466: file affiche.c, line 6.
(gdb) r
Starting program: /home/pierre/tmp/affiche

Breakpoint 1, affiche (v=0) at affiche.c:6
6           printf ("valeur= %d\n", v);
(gdb) c
```

```
Continuing.
valeur= 0

Breakpoint 1, affiche (v=1) at affiche.c:6
6           printf ("valeur= %d\n", v);
(gdb) c
Continuing.
valeur= 1

Breakpoint 1, affiche (v=2) at affiche.c:6
6           printf ("valeur= %d\n", v);
(gdb)
```

Ce type de manipulation très classique ne présente aucune difficulté dans un environnement de station de développement Linux. Supposons à présent que l'on veut mettre au point un programme exécuté sur une machine cible nommée *portable* depuis un poste de développement nommé *duron*. Nous partons du principe que les deux machines sont connectées au même réseau local TCP/IP, mais elles pourraient également être reliées par un simple câble série RS-232.

Pour mettre au point le programme, nous devons bien évidemment disposer du programme **gdb** sur le poste de développement, mais également d'un programme « serveur » sur le poste cible. Ce programme est nommé **gdbserver** et fait partie de la distribution des sources **gdb**. En revanche, il n'est en général pas distribué dans le paquetage RPM binaire car son utilisation est réservée à un usage très particulier. Pour information, ce programme est livré en standard dans les distributions Linux embarquées spécialisées, comme LynuxWorks BlueCat ou Lineo Embedix. Le couple **gdb/gdbserver** peut bien entendu être utilisé sur des architectures différentes si celles-ci sont supportées par **gdb** et **gdbserver**.

Pour compiler **gdbserver**, nous pouvons partir du paquetage RPM source de **gdb** disponible sur *ftp://ftp.redhat.com/redhat/redhat-7.2-en/os/i386/SRPMS/gdb-5.0rh-15.src.rpm*. Nous pouvons également partir des sources de **gdb** disponibles sur *ftp://ftp.gnu.org/pub/gnu/gdb*.

Comme décrit dans ce chapitre, nous pouvons installer le paquetage source au moyen de :

```
rpm -ivh gdb-5.0rh-15.src.rpm
```

puis appliquer les « patches » propres à la Red Hat 7.2 par la commande :

```
rpm -bp /usr/src/redhat/gdb.spec
```

Nous devons ensuite nous positionner sur le répertoire des sources, puis générer le fichier Makefile.

```
cd /usr/src/redhat/BUILD/gdb+dejagnu-20010813/gdb
./configure
```

puis compiler et installer le programme **gdbserver**.

```
cd gdb/gdbserver
make
make install
```

Sur le poste de développement *duron*, nous disposons du source du programme **affiche.c** et de l'exécutable **affiche** compilé avec l'option **-g**.

```
[pierre@duron pierre]$ ls -l
total 32
-rwxr-xr-x    1 pierre   pierre      24622 mai  2 08:19 affiche
-rw-r--r--    1 pierre   pierre        194 mai  2 08:21 affiche.c
```

Sur la machine cible, nous devons disposer d'une copie de l'exécutable **affiche**. Côté cible, nous devons tout d'abord lancer **gdbserver** sur le programme **affiche**.

```
[pierre@portable tmp]$ gdbserver duron:2345 affiche
Process affiche created; pid = 12810
```

Nous choisissons d'utiliser un lien réseau entre les deux machines. Le premier paramètre de **gdbserver** est le nom réseau (ou l'adresse IP) du poste de développement associé au numéro de port TCP à utiliser, ici 2345.

Attention

Nous rappelons que les ports inférieurs à 1024 ne doivent pas être utilisés, car réservés au système. De même, il faut prendre soin de choisir un port non utilisé par une autre application utilisateur.

Côté poste de développement, il suffit de lancer **gdb** avec le nom du programme en paramètre. La sélection de la cible à déboguer s'effectue grâce à la commande **target**.

```
$ gdb affiche
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) target remote portable:2345
Remote debugging using portable:2345
0x40001e60 in ?? ()
(gdb)
```

Côté cible, la ligne suivante est affichée :

```
Remote debugging using duron:2345
```

À partir de là, on peut utiliser **gdb** de manière classique en posant par exemple un point d'arrêt sur la fonction **affiche()**.

```
(gdb) b affiche
Breakpoint 1 at 0x8048466: file affiche.c, line 6.
(gdb) c
Continuing.

Breakpoint 1, affiche (v=0) at affiche.c:6
```

```
warning: Source file is more recent than executable.

6           printf ("valeur= %d\n", v);
(gdb) n
7       }
(gdb)  c
Continuing.

Breakpoint 1, affiche (v=1) at affiche.c:6
6           printf ("valeur= %d\n", v);
(gdb) n
7       }
```

Côté cible, on obtient l'affichage de l'exécution du programme.

```
valeur= 0
valeur= 1
```

Si l'on inhibe le point d'arrêt et que l'on finit l'exécution du programme, l'exécution se termine côté cible et **`gdbserver`** s'arrête. On obtient côté poste de développement :

```
(gdb) dis 1
(gdb) c
Continuing.

Program exited with code 0224.
(gdb)
```

et côté cible :

```
valeur= 2
valeur= 3
valeur= 4

Child exited with retcode = 94

Child exited with status 0
GDBserver exiting
[pierre@portable tmp]$
```

On pourra de la même manière utiliser une connexion par un câble série RS-232 sur le port COM1 (**`/dev/ttyS0`**) en lançant côté cible :

```
gdbserver /dev/ttyS0 nom_programme
```

et côté système de développement :

```
gdb nom_programme
```

puis dans **`gdb`** :

```
(gdb) target remote /dev/ttyS0
```

Si l'on veut utiliser une autre vitesse que 9600 bits/s, on devra préciser l'option **`--baud`** à **`gdb`**.

Utilisation de `strace`

La commande **strace** permet de visualiser les appels systèmes et les signaux utilisés dans un programme. Le résultat de **strace** est assez verbeux comme en témoigne le test effectué sur notre petit programme **affiche**. La sortie de **strace** peut cependant être paramétrée en utilisant l'option **-e**.

```
$ strace ./affiche
execve("./affiche", ["./affiche"], [/* 22 vars */]) = 0
uname({sys="Linux", node="duron.localdomain", ...}) = 0
brk(0)                                  = 0x804964c
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=38627, ...}) = 0
old_mmap(NULL, 38627, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                = 0
open("/lib/i686/libc.so.6", O_RDONLY)   = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0 \306\1"..., 1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=5772268, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
➥ x40021000
old_mmap(NULL, 1290088, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40022000
mprotect(0x40154000, 36712, PROT_NONE)  = 0
old_mmap(0x40154000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x13
➥ 1000) = 0x40154000
old_mmap(0x40159000, 16232, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANON
➥ YMOUS, -1, 0) = 0x40159000
close(3)                                = 0
munmap(0x40017000, 38627)               = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40
➥ 017000
write(1, "valeur= 0\n", 10valeur= 0
)              = 10
write(1, "valeur= 1\n", 10valeur= 1
)              = 10
write(1, "valeur= 2\n", 10valeur= 2
)              = 10
write(1, "valeur= 3\n", 10valeur= 3
)              = 10
write(1, "valeur= 4\n", 10valeur= 4
)              = 10
munmap(0x40017000, 4096)                = 0
_exit(-1073743148)                      = ?
```

La commande **strace** peut être très utile dans le cas de la mise au point d'un pilote de périphérique.

Détection des problèmes de mémoire

Le problème de dépassement de mémoire allouée est la bête noire de tous les développeurs C/C++. En général, cela tient à ce qu'un programme utilise de la mémoire qu'il n'a pas préalablement allouée. Le cas typique est le dépassement de tableau, lorsqu'on alloue un certain nombre d'éléments d'un tableau dynamique avec l'appel système **malloc()** et que l'on utilise ensuite plus d'éléments que le nombre prévu.

Le problème est d'autant plus épineux qu'il n'apparaît pas forcément immédiatement au cours de l'exécution du programme car il a trait à la mémoire disponible, et donc à la charge du système. Dans le cas d'un système embarqué, les conséquences peuvent être dramatiques car le défaut peut apparaître bien après la phase de développement.

Le petit programme C présenté ci-après alloue par exemple 10 cellules d'un tableau de caractères, mais en utilise 20.

```
#include <stdio.h>
#include <stdlib.h>

main (int ac, char **av)
{
  register int i;
  char *tab = calloc (1, 10);

  for (i = 0 ; i < 20 ; i++)
    *(tab+i) = i;

  for (i = 0 ; i < 20 ; i++)
    printf ("tab[%d]= %d\n", i, *(tab+i));
}
```

Après compilation et exécution, le système n'y voit pourtant que du feu.

```
$ gcc -g -o buggy buggy.c
$ ./buggy
tab[0]= 0
tab[1]= 1
tab[2]= 2
tab[3]= 3
...
tab[18]= 18
tab[19]= 19
```

Pour détecter ce type de problème, nous devons utiliser un allocateur de mémoire spécial. Les distributions classiques fournissent le paquetage ElectricFence (la « barrière électrifiée »), qui remplace l'allocateur standard par une version de débogage. Pour cela, il faut installer le paquetage associé.

```
rpm -ivh ElectricFence-2.2.2-8.i386.rpm
```

Il suffit ensuite de compiler le programme en utilisant la bibliothèque **libefence**. Une nouvelle exécution du programme détecte bien un problème de mémoire et le programme est interrompu avec une erreur de segmentation (signal **SIGSEGV**).

```
$ gcc -g -o buggy buggy.c -lefence
$ ./buggy
  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
Segmentation fault
```

Le système doit dès lors générer un fichier **core** qui permet de localiser l'erreur. Si le fichier **core** n'est pas présent, il faut vérifier la configuration de la session grâce à la commande **ulimit -c**. Si la valeur est 0, il faut la modifier en donnant la nouvelle valeur « unlimited ». Une nouvelle exécution du programme générera cette fois-ci un fichier **core** (message *core dumped*).

```
$ ulimit -c
0
$ ulimit -c unlimited
$ ulimit -c
unlimited
$ ./buggy
  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
Segmentation fault (core dumped)
```

Le débogueur **gdb** permet ensuite de localiser l'erreur.

```
$ gdb buggy core
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
Core was generated by `./buggy'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/libefence.so.0...done.
Loaded symbols for /usr/lib/libefence.so.0
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/i686/libpthread.so.0...done.

warning: Unable to set global thread event mask: generic error
[New Thread 1024 (LWP 1671)]
Error while reading shared library symbols:
Cannot enable thread event reporting for Thread 1024 (LWP 1671): generic error
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x080485b5 in main (ac=1, av=0xbffffb64) at buggy.c:11
11          *(tab+i) = i;
(gdb)
```

On peut également exécuter le programme **buggy** directement dans gdb, ce qui conduit au même résultat.

```
$ gdb buggy
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) r
Starting program: /home/pierre/buggy
[New Thread 1024 (LWP 1675)]

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 1024 (LWP 1675)]
0x080485b5 in main (ac=1, av=0xbffffb54) at buggy.c:11
11          *(tab+i) = i;
(gdb)
```

La bibliothèque **efence** permet de détecter les débordements de mémoires mais certains produits commerciaux, beaucoup plus complexes (et plus coûteux), ont d'autres fonctionnalités intéressantes comme la détection de fuites de mémoire. Nous pouvons citer le produit Insure++ édité par Parasoft (*http://www.parasoft.com*).

En résumé

Le présent chapitre a décrit quelques fonctions et méthodes très utiles pour le développement d'un système Linux embarqué. Concernant les disques et mémoires flash, les produits M-Systems sont très répandus et peuvent être utilisés soit avec des pilotes fournis par le constructeur, soit *via* la couche MTD intégrée dans le noyau Linux 2.4. Le pilote M-Systems utilisant une bibliothèque non-GPL, le fait de redistribuer un noyau Linux contenant ce pilote en statique n'est pas conforme à la GPL. On pourra alors utiliser un disque mémoire initial (**initrd**) contenant les modules M-Systems ou bien utiliser le pilote MTD.

La couche MTD permet également de piloter d'autres types de mémoires flash comme les produits compatibles CFI.

Différents formats de système de fichiers incluant des fonctionnalités de journalisation et/ou de compression en temps réel sont utilisables dans un environnement Linux embarqué.