Frédéric Brault • Préface d'Albert Cohen

Hackez Google Android

Introduction à la programmation système

Appropriez-vous le système Android conçu par Google pour équiper les téléphones mobiles et les netbooks et affranchissez-vous des limites habituelles : accédez à toutes les couches basses du système pour exécuter un shell, développer des scripts, installer des modules Linux...

L'image système étudiée dans ce livre est tirée du prototype Android fourni pour les TP d'un cours ambitieux donné en 2009 à l'École Polytechnique, « Composants d'un système informatique », visant à présenter toutes les couches d'un ordinateur, depuis l'interface utilisateur jusqu'au noyau.

EYROLLES

Prix : 12 €
Code éditeur : G85009
ISBN : 978-2-212-85009-3

Préface

Il se vend depuis 2005 plus de téléphones mobiles que de PC. Le succès retentissant des smart-phones et des netbooks entraîne l'émergence quotidienne de nouvelles applications et services : contenus multimédias, outils et formats bureautiques, services d'accès au réseau tels que la voix sur IP, bureau mobile, réalité virtuelle, diffusion pair-à-pair, Web dynamique et collaboratif, applications médicales personnalisées et distribuées, terminaux de micro-paiement, etc. Ainsi voit-on peu à peu le transfert vers ces plates-formes mobiles de tâches traditionnellement dévolues aux PC (portables ou non), y compris en matière de bureautique ou de logiciels d'entreprise.

Plus que de compétition, il convient de parler de *convergence* entre plates-formes généralistes, mobiles et embarquées. Cela n'a pas échappé à des acteurs comme ARM ou Nokia, qui voient une réelle opportunité de renverser la domination Wintel (Microsoft et Intel) sur l'informatique grand public. C'est là qu'intervient la plate-forme Android de Google, qui a le potentiel pour dominer ce marché en pleine effervescence, amené à dépasser prochainement celui des PC.

Je fais d'ailleurs le pari que cette convergence constitue également une opportunité capitale pour que des plates-formes logicielles libres s'imposent comme standards du marché. En effet, les contraintes de compatibilité induisent de tels coûts de développement que

très peu d'acteurs peuvent survivre en vendant des logiciels. Or les logiciels libres permettent des économies d'échelle croissantes : le temps joue largement pour eux, leur maintenance et l'évolution de leurs fonctionnalités étant plus facilement mutualisées. Le modèle est donc viable car l'essentiel du marché se fait et se fera sur les services, les contenus, la publicité et les communautés – y compris les sites de socialisation et les jeux massivement multi-joueurs.

Google a bien compris cela, et depuis fort longtemps, pour le bénéfice de milliards d'utilisateurs indirects de ses contributions importantes à des logiciels libres tels que GNU, Linux ou Mozilla (Firefox). Sans faire exclusivement du logiciel libre ni être une organisation a but non lucratif, Google adopte une politique de contribution lui assurant une popularité qui lui permet d'attirer les meilleurs développeurs, hackers et contributeurs internes ou externes en tout genre.

Au-delà de leurs avantages économiques, les logiciels libres sont un instrument d'émulation et de coopération ; ils sont l'opportunité de replacer une éthique plus saine dans l'économie et la société numérique. Les utilisateurs des plates-formes mobiles sont des acteurs à part entière ; bien plus que de simples consommateurs, ils sont *producteurs* de contenu, *diffuseurs*, *intégrateurs* et *prestataires* de services. Dans un tel monde, n'est-il pas vain de vouloir à toute force maintenir les barrières artificielles que constituent certaines lois régissant la propriété intellectuelle ? Est-il réaliste, techniquement comme socialement, de contrôler les échanges entre individus ? Un monde où les contenus et leur contenant logiciel sont libres est possible, souhaitable... nécessaire surtout.

Découvrir la plate-forme Android est un excellent moyen de comprendre le moteur de cette révolution en marche. À condition de l'attaquer par le milieu et pas uniquement par la programmation d'applications Java. Ce livre vous ouvre les entrailles du système d'exploitation, des multiples machines virtuelles, et de leurs incarnations sur un système mobile et

embarqué complet. Il s'adresse aux étudiants, aux développeurs, aux hackers débutants, et aux citoyens avertis souhaitant maîtriser les ressorts des technologies de l'information.

Albert Cohen
Directeur de recherche INRIA
Professeur chargé de cours à l'École Polytechnique

Table des matières

Avant-propos

Au début de l'année scolaire 2008-2009 a été mis en place à l'École Polytechnique un cours ambitieux intitulé « Composants d'un système informatique ». Ce cours, que nous avons conçu sous la direction d'Albert Cohen avec Fabrice Le Fessant et Louis-Noël Pouchet, avait pour but de présenter toutes les couches d'un ordinateur, depuis l'interface utilisateur jusqu'au noyau.

Une des parties les plus importantes de ce cours, les travaux pratiques, présentait un sérieux défi : les élèves utilisaient chacun leur propre ordinateur portable – avec l'hétérogénéité que cela implique, en terme de systèmes d'exploitations, notamment – mais devaient tous pouvoir faire les exercices portant sur des programmes Java, des scripts shell, des modules Linux.

Après avoir envisagé plusieurs solutions, le choix se porta sur Android, qui n'existait à l'époque que sous la forme d'un émulateur. Cette plate-forme permettait à tous les élèves d'utiliser le même système et gérait directement le langage Java... à ceci près qu'en l'état, il n'était pas possible d'accéder aux couches Linux, ni d'écrire des scripts ou d'utiliser un shell. Il fut donc décidé de modifier l'image système d'Android pour pouvoir s'affranchir de ces limites. Au final, le prototype qui fut fourni aux élèves ressemblait fort à l'image système étudiée dans ce livre !

Lorsque Google annonça en octobre 2008 la mise à disposition du code source, ce fut une motivation supplémentaire pour s'intéresser à ce système. La lecture du code leva le voile sur de nombreux aspects du système qui avaient jusque là dû être devinés par tâtonnement, faute de documentation.

C'est à ce moment que le laboratoire acheta un téléphone G1, pour servir de support à l'étude d'Android. Malheureusement, si la plate-forme logicielle Android est bien libre, il n'en va pas de même avec le G1, qui ne permet pas la modification du système par l'utilisateur : celui-ci n'a pas les privilèges administrateur (*root*).

Quelques semaines plus tard, répondant à la demande des développeurs Android, Google mit sur le marché un téléphone en tout point identique au G1, le Dev Phone 1, qui permettait la modification du système.

Pour les développeurs possédant déjà un téléphone G1, il fallait cependant trouver une autre solution. Celle-ci fut découverte par une équipe de bricoleurs : en exploitant une faille présente dans les premières versions du G1, il s'avéra possible d'obtenir les privilèges `root`...

EN COULISSES La faille miraculeuse

La faille en question est tellement énorme qu'il est difficile d'y croire... Sur les G1 de première génération (numéro RC inférieur à 30), un shell tournant avec les privilèges root est lancé au démarrage du téléphone, probablement à fins de débogage. Toujours est-il qu'il est branché au clavier et peut interpréter tout ce que l'utilisateur tape. Par exemple, si l'utilisateur tape « ls » sur le clavier, cette commande est interprétée par le shell, en tant que root. Le shell n'étant pas connecté à l'écran, l'utilisateur ne s'en aperçoit pas. Il aura par contre un message d'erreur de la part d'une autre application écoutant elle aussi les frappes du clavier, lui indiquant que le contact téléphonique « ls » ne figure pas dans son répertoire.
Exploiter une faille aussi béante est un jeu d'enfant. Il suffit par exemple de taper `telnetd` sur le clavier pour qu'un serveur telnet se lance sur le téléphone, avec les privilèges root !
Bien sûr, une fois la faille mise à jour, elle fut rapidement corrigée dans les versions ultérieures. Et d'autres failles furent découvertes et décrites sur Internet ...

Une fois les droits root acquis, la partie n'était pas gagnée pour autant. S'il était désormais possible de charger ses propres programmes C et Java sur le téléphone, changer complètement l'image système demandait plus de travail.

La procédure suivie, qui s'appuie sur de nombreux points qui seront abordés dans ce livre, est décrite en Annexe à la fin du livre.

1

Google à la conquête des *smartphones* avec Android

D'après Google, Android est une plate-forme logicielle destinée aux appareils mobiles comprenant un système d'exploitation, des applications, et les couches intermédiaires nécessaires pour faire fonctionner le tout. Nous remarquons d'entrée de jeu que Google se place dans le domaine du logiciel. Android n'est donc pas lié à un appareil donné ; il a au contraire vocation à être intégré par différents constructeurs dans des appareils mobiles, moyennant le respect de quelques règles de compatibilité.

En plus des constructeurs de matériel, Google entend fédérer autour d'Android tout un écosystème de développeurs et d'utilisateurs « avertis », créateurs de contenu. Pour leur faciliter la tâche, il a été prévu dès le départ qu'Android serait facilement programmable. C'est le langage Java qui a été retenu et Google fournit aux développeurs de nombreux outils pour écrire et mettre au point leurs programmes : émulateur pour les tests, SDK Java et plugin Eclipse pour le développement, ainsi qu'une documentation fournie, disponible sur internet (www.android.com).

Android peut donc être défini non seulement comme la plate-forme mobile réalisée par Google, mais également comme l'ensemble des apports de cette communauté émergente de développeurs et d'utilisateurs. On retrouve là un modèle bien connu des auteurs et utilisateurs de logiciels libres.

Les différentes couches d'Android

Le schéma suivant illustre la structure de la plate-forme Android, sous forme de couches qui s'empilent :

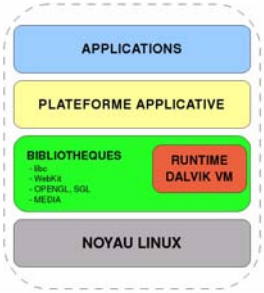


Figure 1–1
L'architecture en couches d'Android

Au plus près du matériel, on trouve le système d'exploitation. Il s'agit d'un noyau Linux 2.6 standard, auquel Google a apporté quelques améliorations, notamment dans le domaine de la gestion de l'énergie (appareil mobile oblige) et de la communication inter-processus. Précisons que pour l'instant, Android tourne sur des processeurs embarqués de type ARM, qui sont nativement gérés par Linux.

Vient ensuite la couche des bibliothèques. Google a écrit pour Android sa propre bibliothèque C, `bionic`, dont la faible taille est adaptée à un environnement embarqué. On trouve aussi d'autres bibliothèques, dont beaucoup seront familières aux utilisateurs de Linux : Webkit, SQLite, OpenGL, SGL, SSL, ainsi qu'une bibliothèque multimédia.

Directement sur cette couche vient se greffer le runtime Android, qui comprend la machine virtuelle Java et ses bibliothèques. Cette machine virtuelle, développée spécifiquement pour Android, porte le nom de **Dalvik virtual machine**. Conçue pour fonctionner dans un environnement embarqué limité en ressources, elle utilise un format d'exécutable compressé (`.dex`), afin de minimiser l'empreinte mémoire.

La couche « plate-forme logicielle » permet quant à elle de mutualiser au maximum les ressources entre applications Java. Elle propose également un moyen pour ces applications d'échanger des données. On retrouve par exemple dans cette couche la boîte à outils graphique qui permet d'afficher des boîtes de dialogue, des boutons, des menus, etc. C'est cette couche qui est rendue disponible au développeur Java au moyen d'un ensemble d'API.

La dernière couche, la seule finalement dont l'utilisateur aura à se préoccuper, est celle des applications. Elle sont sous forme de paquets `.apk`, qui permettent une installation et une désinstallation facile. Certaines sont fournies par l'équipe Android : navigateur web, gestionnaire de contacts, agenda… à vous de compléter la liste avec vos propres programmes !

Le système de fichiers

La hiérarchie des fichiers d'Android peut paraître déroutant pour quiconque a un peu l'habitude des système Unix (et a fortori pour quiconque n'y est pas du tout habitué !). Pas de répertoire `/bin`, ni de `/home`, mais un dossier `/system`. Les fichiers de configuration, que l'on s'attend à retrouver dans `/etc`, sont en fait pour la plupart stockés dans une base de données. Ce système n'a clairement pas été conçu pour être exposé directement à l'utilisateur !

À ces répertoires sont en fait associés (au sens Unix `mount`) les images système suivantes :

- `ramdisk.img` est associée à la partition racine `/`, au format `rootfs`,
- `system.img` est associée au dossier `/system`, au format yaffs2, conçu pour la mémoire flash,
- dans l'émulateur, `/data` est associé au fichier `~/.android/SDK/userdata-qemu.img`, en `yaffs2`,
- enfin, `/sdcard` correspond sur le téléphone à la carte SD, au format FAT – sur l'émulateur, il est possible dans les options de démarrage de charger une image FAT à la place.

> Pour l'émulateur, ces images se trouvent dans le répertoire `tools/lib/images` du SDK.

Android côté utilisateur : le téléphone

Le meilleur moyen d'utiliser Android est de disposer d'un téléphone ou d'un PDA sur lequel la plate-forme est installée. Les premiers modèles (téléphones G1) sont arrivés sur le marché aux USA fin 2008, puis ont été diffusés dans de nombreux autres pays.

En outre, pour répondre à la demande de nombreux développeurs, Google a mis sur le marché un téléphone qui leur est dédié, le Dev Phone 1. Contrairement aux téléphones destinés au grand public, dont les couches basses sont parfois verrouillées (cf. Annexe), le Dev Phone 1 est prévu pour être modifié facilement : il est en effet possible de remplacer l'image système (firmware) par une image personnalisée.

Précisons que tous les téléphones Android acceptent les programmes écrits en Java. Par contre, les manipulations décrites dans ce livre, qui affectent les couches basses (programmation en C/C++, modification des images système) semble être plus aisées avec un Dev Phone.

Figure 1–2
Le téléphone G1

RESSOURCE Liste en ligne des téléphones compatibles

Enfin, la liste des téléphones compatibles Android ayant vocation à évoluer, on pourra consulter une version tenue à jour directement sur le site d'Android. Cette page donne également des conseils pour passer son téléphone en mode debug : Menu->Applications->Development->USB debugging.

- http://developer.android.com/guide/developing/device.html

Côté développeur : installer l'émulateur Android

Bonne nouvelle : il n'est pas nécessaire de posséder un téléphone Android pour pouvoir utiliser la plate-forme. En effet, Google fournit dans sa boîte à outils destinée aux développeurs un émulateur reproduisant l'environnement du téléphone. Cela permet également d'effectuer des tests préliminaires rapidement sur la station de développement, sans prendre aucun risque en cas de mauvaise manipulation. Cerise sur le gâteau, cet émulateur a le bon goût d'être disponible sous de nombreux systèmes : Linux, Windows et Mac OS X.

Pour l'installer, il faut tout d'abord récupérer le SDK sur le site http://developer.android.com. On y choisit dans la section download l'archive zip correspondant à son système d'exploitation. Il est également vivement conseillé de lire la documentation, à l'onglet Installing sur le site web.

Une fois l'archive décompressée, il suffit d'ajouter à la variable d'environnement `PATH` le chemin du répertoire `tools` de l'archive. Sous Linux :

```
export PATH=$PATH:android-sdk-linux_x86-version/tools
```

On rendra cette modification permanente en ajoutant cette ligne au fichier `.bashrc`, par exemple.

Pour vérifier que l'installation s'est bien déroulée, on peut lancer l'émulateur an tapant la commande `emulator` dans une console. On obtient alors l'écran suivant :

Figure 1–3
L'émulateur Android

On peut interagir avec cet émulateur au moyen du clavier, de la souris ou du clavier virtuel affiché à l'écran. Pour accéder au couche basse de l'émulateur, on a recourt à l'outil `adb`. En voici quelques exemples d'utilisation :

```
adb shell
```

Lance un shell sur Android. On peut alors taper des commandes qui seront interprétées par Android.

```
adb push fichier.dat /data/
```

Transfère le fichier `fichier.dat` vers le répertoire `/data/` se trouvant sur Android.

```
adb pull /data/fichier.dat .
```

Récupère le fichier `fichier.dat` se trouvant dans le répertoire `/data` de d'Android, pour le placer dans le dossier courant.

On peut obtenir plus d'informations sur ces commandes grâce à l'option `--help`, ou en consultant la documentation disponible sur http://developer.android.com.

Un des avantages de la commande `adb` est qu'elle fonctionne indifféremment sur l'émulateur et sur un téléphone. Si l'émulateur est lancé sur la machine, les commandes `adb` agiront sur lui. Si un téléphone est branché à une prise USB, alors `adb` agira sur ce téléphone. Le tout de façon transparente pour l'utilisateur.

> Bien que ce ne soit pas l'objet de ce livre, pourquoi ne pas en profiter pour installer aussi les outils développement Java ? La programmation dans ce langage est très facile sous Android et on trouvera une documentation complète et accessible sur le site http://developer.android.com.

Compiler du code C ou C++

2

Vous souhaitez aller plus loin et accéder aux couches basses d'Android à fins de performances et de flexibilité ? Rien d'impossible à condition d'installer les outils adéquats : bien que conçue pour être programmée en Java, la plate-forme Android peut redevenir un véritable système Linux.

- Code natif
- Compilation croisée
- C, C++

Comme nous l'avons vu dans l'introduction, Google a choisi de donner au développeur l'accès aux seules couches « hautes » à travers le langage Java. Il reste cependant possible de contourner cette limite et de programmer directement en C et en C++ pour accéder à toutes les couches, jusqu'au noyau. En effet, une fois maîtrisée la compilation C, rien n'empêche d'écrire des modules pour le noyau Linux.

Sans aller jusque là, nous allons voir que cette approche permet d'accéder aux couches basses d'Android et donc d'avoir accès à plus de performances et de flexibilité. En somme, elle permet d'utiliser Android... comme un véritable système Linux !

> **Attention**
>
> Dans les chapitres qui suivent, on supposera que le lecteur est déjà familiarisé avec le langage C. Il est également nécessaire de disposer d'un système Unix, de préférence Linux, pour procéder à l'installation des outils qui vont être présentés.
>
> 🕮 Blaess C., *Programmation système en C sous Linux*, Eyrolles 2005

Pourquoi ne pas se contenter de Java ?

La question est légitime. Si Android a été prévu pour être programmé en Java, pourquoi se donner tant de mal ? Voici quelques raisons valables (au-delà de l'envie, bien naturelle, de vouloir bricoler un tel engin !) :

- **Les performances**. Ce n'est pas une surprise, le code C compilé sera plus rapide qu'un code Java équivalent. Ceci est d'autant plus vrai que la machine virtuelle Java d'Android (la Dalvik Virtual Machine) n'a pas été conçue pour être particulièrement rapide (voir remarque ci-après). Les programmes pour lesquels la rapidité est un critère important seront donc plutôt écrits en C ou C++, quitte à faire une interface en Java.

- **L'utilisation de programmes existants**. Pouvoir utiliser le langage C ou C++ donne immédiatement accès à une quantité impressionnante de programmes tiers, notamment des programmes Open Source. C'est le cas par exemple de la plupart des programmes destinés aux systèmes Unix. Ceux-ci étant fournis avec le code source, il est souvent possible de les compiler et de les utiliser directement sur Android.
- **L'accès direct au système**. En l'absence de machine virtuelle, un programme C accède directement au système Linux d'Android. Cela permet d'éliminer une grande partie des limitations imposées par la machine virtuelle (souvent pour des raisons de sécurité). On peut par exemple accéder à l'ensemble du système de fichiers, ou lancer des programmes en tâche de fond avec une priorité ajustable.
- **La programmation de pilotes**. Il est certains domaines pour lesquels le langage C est le seul choix possible. C'est le cas, par exemple de la programmation de pilotes de périphériques. Ceux-ci doivent pour des raisons de compatibilité être écrits dans le même langage que le noyau Linux (le langage C, donc...).

DANS LES COULISSES D'ANDROID À propos de performances

La Dalvik Virtual Machine ne possède pas de compilateur JIT (Just in Time). Il s'agit là pourtant d'une optimisation classique pour les machines virtuelles. Les développeurs de Google justifient leur choix ainsi : selon eux, la plupart des programmes Java pour lesquels les performances sont cruciales peuvent soit faire appel à des bibliothèques optimisées fournies par Google, soit utiliser un accélérateur matériel, soit appeler un programme C rapide écrit pour l'occasion.
Précisons que la Dalvik Virtual Machine a été conçue avant tout pour avoir une empreinte la plus faible possible, en terme de mémoire et de consommation d'énergie.

Quelle méthode de compilation choisir : avec ou sans SDK ?

Nous présentons dans ce livre deux manières de compiler du code C ou C++ pour Android. La première est basée sur une chaîne de compilation croisée installée « à la main ». La seconde, qui fournit apparemment des résultats similaires, utilise le SDK Android officiel. La question se pose donc : laquelle de ces deux méthodes faut-il choisir ?

Et comme souvent, la réponse est : « cela dépend ».

Cela dépend tout d'abord du type d'application que l'on souhaite développer.

Pour une application écrite « pour soi »,choisir l'une ou l'autre méthode importe peu. Il en va différemment si l'on souhaite diffuser l'application. En effet, si écrire en C ou C++ n'est officiellement pas permis sous Android, cela l'est encore moins lorsqu'on utilise une chaîne de compilation tierce. Pour l'instant, le seul moyen de placer du code C ou C++ sous Android de façon stable est de développer avec le SDK et de placer son code dans le dépôt officiel sous une licence libre.

> Certaines discussions lues sur les listes de diffusions laissent penser qu'il sera peut-être possible à terme d'écrire et de distribuer des applications écrite en C/C++ comme partie intégrante des paquets `.apk`. Bien évidemment, de telles applications devraient alors être écrites au moyen du SDK.
> Cette idée semble prendre forme avec le NDK :
> ‣ http://groups.google.com/group/android-ndk

Cela dépend également de l'application elle même, de sa taille et de ses dépendances. La bibliothèque C fournie par le SDK, `bionic`, est en effet assez limitée, et certaines fonctionnalités offertes par exemple par la bibliothèque C GNU ne sont pas présentes. À moins de réécrire le code, il n'y a alors pas d'autre solution que de compiler en statique

avec une chaîne telle que `scratchbox`, qui intègre la bibliothèque C GNU. C'est le cas, par exemple, de Busybox, qui ne peut être compilé avec le SDK seul. Notons tout de même que le code ainsi compilé est beaucoup plus volumineux, puisqu'il embarque une partie de la bibliothèque C au lieu d'utiliser du code partagé.

Cela dépend enfin de l'investissement auquel consent le développeur, en terme de temps et de travail. Il semble en effet que le coût initial pour placer son application dans le SDK Android soit plus élevé. Tout d'abord parce qu'il faut compiler tout le SDK avant de pouvoir l'utiliser, ce qui prend du temps, de la place disque et de la patience. En effet, au vu des nombreuses dépendances envers des bibliothèques de développement, qui sont parfois manquantes sur le système, il est rare que la compilation aboutisse au premier coup. Ensuite, le SDK utilise un système de compilation basé sur des fichiers `Android.mk`. Pour un projet qui débute, cela n'est pas forcément un problème. Dans les autres cas, il faut cependant se préparer à devoir réécrire tous les fichiers Makefile !

Tableau 2–1 Avantages et inconvénients à compiler avec le SDK Android

	SDK Android	Toolchain Scratchbox
Compatibilité avec Android	+	-
Facilité de mise en oeuvre	-	+
Richesse de la bibliothèque C	-	+
Compatibilité avec des projets existants	-	+
Taille des programmes	+	-

Outils de compilation à installer

Compilation croisée

Développer en C pour Android nécessite d'avoir au préalable installé un compilateur. En temps normal, la compilation d'un programme C est chose aisée. Le compilateur, généralement fourni par défaut sur les systèmes Unix, sait produire du code machine pour l'ordinateur sur lequel il est installé.

Il en va tout autrement dans notre cas : le compilateur doit produire du code non pas pour l'ordinateur, mais pour Android...

> **Compilation croisée**
>
> Ce cas de figure, souvent rencontré dans le domaine de l'informatique embarquée, porte le nom de compilation croisée (cross compilation en anglais).

Il nous faudra donc installer :

- un compilateur spécial, capable de produire du code non pas pour le processeur de l'ordinateur (par exemple, un processeur Intel type x86), mais pour un processeur ARM (sur lequel tourne Android) ;
- en plus du compilateur, tout un ensemble d'outils appelé binutils : l'éditeur de liens, l'assembleur, etc.
- les versions « ARM » des bibliothèques, et notamment de la bibliothèque C.

Tous ces programmes sont souvent regroupés en des « chaîne de compilation croisée » (toolchain, en anglais), qui facilitent l'installation de l'environnement de développement.

Installation de la chaîne de compilation croisée libre Scratchbox

Scratchbox est une chaîne de compilation croisée libre, disposant d'une version pour processeur ARM. Pour l'installer, il faut tout d'abord télécharger les paquets suivants à cette adresse :

http://scratchbox.org/download/files/sbox-releases/stable/tarball/

- `core-1.0.1`
- `devkit-cputransp-1.0.7`
- `libs-1.0.10`
- `toolchain-arm-linux-2007q1-21-1-07`

Chacune de ces archives devra ensuite être extraite en tant que root grâce à la commande suivante :

```
# tar zxf <archive> -C /
```

Une fois l'installation terminée, il faut configurer Scratchbox.

Configuration et démarrage de Scratchbox

La commande suivante doit être tapée en tant que *root*. On utilisera les valeurs par défaut.

```
# /scratchbox/run_me_first.sh
```

On ajoute ensuite un utilisateur (par exemple l'utilisateur courant), avec la commande suivante (toujours en tant que *root*) :

```
# /scratchbox/sbin/sbox_adduser <utilisateur>
```

L'utilisateur ainsi ajouté fait désormais partie du groupe `sbox`, ce que l'on peut vérifier à l'aide de la commande `groups`. Si ce n'est pas le cas, il peut être nécessaire de redémarrer la session. Une fois membre de ce groupe, on lance Scratchbox en tapant :

```
/scratchbox/login
```

On sort de l'environnent Scratchbox comme d'un shell classique, en tapant `exit`.

Il faut ensuite, en tant que *root*, lancer la commande :

```
# /scratchbox/sbin/sbox_umount_all
```

En effet, pour s'isoler complètement du système hôte et éviter toute interférence entre les composants « natifs » et les composants « croisés », Scratchbox fonctionne dans un `chroot`, sur des partitions virtuelles qui lui sont propres. Ce sont ces partitions qu'il faut « démonter » une fois la session terminée.

Pour redémarrer Scratchbox, on utilisera la commande suivante, en tant que *root* :

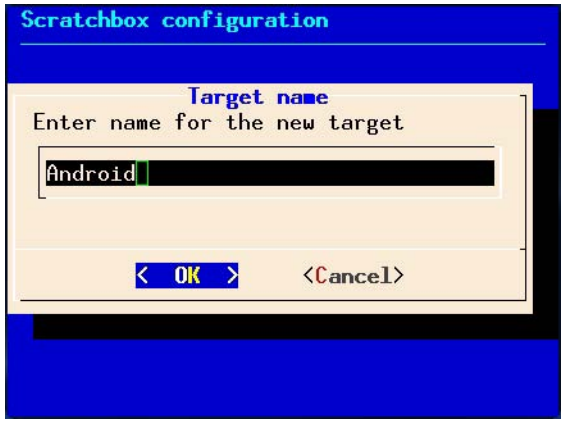
```
/scratchbox/sbin/sbox_ctl start
```

Configuration de la cible Android

Une fois Scratchbox installée, il faut configurer l'environnement pour pouvoir compiler à destination du processeur ARM. On utilisera pour cela la commande `sb-menu`, disponible une fois connecté dans l'environnement Scratchbox. Un menu s'affiche, proposant plusieurs options.

1. On choisira tout d'abord le menu `Setup`, puis l'option `New` pour configurer notre nouvelle cible de compilation. Le nom de la cible importe peu, choisissons par exemple de l'appeler `Android`.

Figure 2–1
Définir une nouvelle cible de compilation

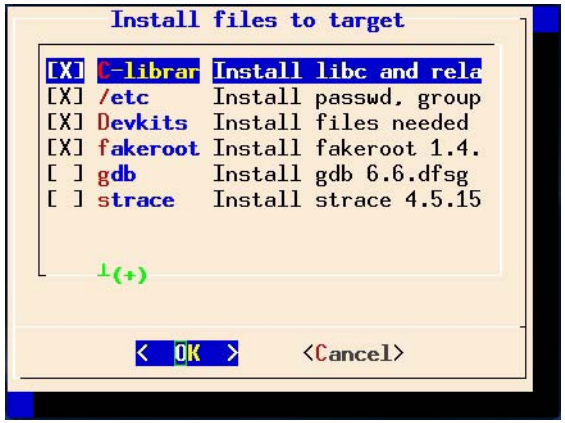


2. Comme compilateur, on choisira **`arm-linux-2007q1-21`** (ce devrait être de toute façon le seul choix disponible).
3. On sélectionnera ensuite **`cputranspdans`** le menu intitulé **`devkits`**.
4. Une longue liste s'affiche, dans laquelle il faut sélectionner **`qemu-arm-0.8.2-sb2`**.
5. Le menu demande alors si l'on désire extraire un rootstrap, à quoi il faut répondre `No`. En effet, un rootstrap est nécessaire si l'on souhaite construire non pas de simples programmes, mais un système entier, avec notamment un système de fichiers.

> Comme nous le verrons dans le prochain chapitre, Android propose d'autres moyens de personnaliser le système de fichiers. Nous n'avons donc pas besoin de cet outil.

6 On répondra par contre Yes à la question suivante, pour installer des fichiers pour notre cible. La liste des options à cocher est indiquée sur la figure ci-après :

Figure 2–2
Fichiers à installer



On répondra enfin Yes à la dernière question, pour utiliser notre nouvelle cible.

Premier programme en C

Pour vérifier que l'installation et la configuration se sont bien passées, nous allons écrire un programme en C très simple, qui se contentera, tradition oblige, d'afficher « Hello Android! » sur la console. À l'aide d'un éditeur de texte, nous allons tout d'abord créer dans le répertoire `/tmp` le fichier `hello.c`, contenant le code suivant :

Contenu du fichier `hello.c` dans le répertoire `/tmp`

```
#include <stdio.h>
int main{
  printf("Hello Android!\n");
  return 0;
}
```

Pour compiler ce fichier, on utilisera la commande suivante, à l'intérieur de l'environnement Scratchbox :

```
gcc -static hello.c -o hello
```

Pour tester notre programme :

1. Démarrer l'émulateur depuis une autre console (en dehors de l'environnement Scratchbox) ou brancher le téléphone sur une prise USB.
2. Transférer le binaire sur Android avec la commande :

```
adb push hello /data/local/
```

3 Puis taper :

```
adb shell /data/local/hello

Hello Android!
```

> **ATTENTION Incompatibilité entre bionic et GCC : nécessité de l'option static**
>
> L'option `static` est ici indispensable. En effet, la bibliothèque C d'Android (**bionic**), est incompatible avec la bibliothèque C GNU. GCC ne sait donc pas, par défaut, créer du code pouvant l'utiliser. Il est donc nécessaire de compiler une application statique embarquant directement le code de la bibliothèque C dont elle a besoin et qui n'utilisera donc pas la bibliothèque C présente dans Android. Cette approche, qui résout les conflits de bibliothèques, a l'inconvénient de créer des fichiers binaires de grande taille.
> Nous verrons dans les prochains chapitres qu'il est possible, en utilisant d'autres outils, de créer des applications dynamiques utilisant **bionic**.

Compiler et configurer la boîte à outils Busybox

Busybox est un programme très populaire dans le domaine de l'informatique embarquée. Véritable « couteau suisse », il regroupe en fait en un seul exécutable plusieurs programmes (les *applets)*. Une technique appelée *multicall binary* permet à Busybox de se comporter comme l'une ou l'autre de ces *applets* suivant le nom sous lequel il est est lancé. En mutualisant le code entre les *applets*, on obtient ainsi, pour une taille minime, un très vaste ensemble de programmes, parmi lesquels un shell, des utilitaires Unix, des outils réseau, etc.

Compiler Busybox présente pour nous deux avantages :

- il permet d'obtenir simplement un environnement Unix quasiment complet ;
- sa configuration et sa compilation sont suffisamment complexes pour permettre de tester notre chaîne de compilation.

1 Récupérez les sources de Busybox avant de le compiler :
http://www.busybox.net/downloads/
La version utilisée ici est la **1.12.1**, mais une version plus récente devrait également convenir.

> **ATTENTION Choisir le bon endroit pour décompresser l'archive**
>
> Scratchbox étant isolé dans un chroot, il n'a pas accès à certains répertoires, notamment les répertoires personnels. Décompressez donc l'archive contenant les sources de Busybox dans un répertoire accessible depuis Scratchbox, par exemple `/tmp`.

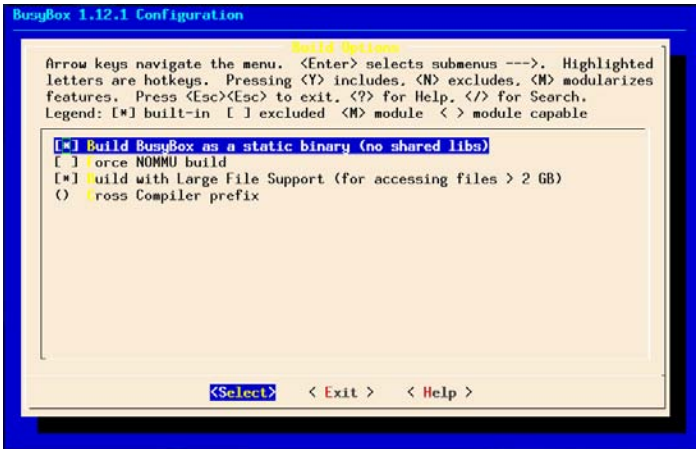
2 Placez-vous dans le répertoire contenant les sources de Busybox et exécutez la commande suivante pour lancer le menu de configuration :

```
make menuconfig
```

> **ATTENTION Sortir de Scratchbox avant d'invoquer la création du menu de configuration**
>
> Pour créer le menu de configuration, des bibliothèques graphiques comme ncurses sont utilisées. Ces bibliothèques n'étant pas incluses dans Scratchbox, la commande make menuconfig doit être tapée hors de cet environnement : assurez-vous d'être bien sorti de l'environnement Scratchbox avec une commande `exit` par exemple.

3. Sélectionnez le sous-menu Busybox Settings, puis l'option Build Options. Comme indiqué sur la figure ci-dessous, il est nécessaire d'indiquer que nous souhaitons compiler Busybox en mode *statique (*voir section précédente).

Figure 2–3
Compiler Busybox en mode statique



4. Le reste du menu permet de choisir les *applets* que l'on veut inclure dans Busybox. N'hésitez pas à en parcourir la liste et à lire leurs descriptions, vous serez sûrement surpris par la diversité des programmes proposés !
5. Une fois votre choix terminé, tapez deux fois sur la touche Echap pour quitter et répondez Yes pour sauvegarder la configuration.

Une fois revenu dans l'environnement Scratchbox, la commande `make` permet de lancer la compilation : un fichier `busybox` est produit au bout de quelques minutes.

Tester Busybox

Pour tester Busybox, transférez le binaire sur l'émulateur ou le téléphone avec la commande :

```
adb push busybox /data/local/
```

Tapez ensuite :

```
adb shell /data/local/busybox
```

Si tout s'est déroulé correctement, cette commande affiche la liste complète des *applets* installés.

Modification du système

3

Programmer en Java ou en C permet de faire des ajouts au système de base Android. Ce chapitre propose d'aller plus loin en modifiant le système de fichiers et le processus de démarrage, pour créer une plate-forme personnalisée.

- Image Système
- Fichier d'initialisation

Nous avons vu comment créer des programmes en C pour les faire fonctionner sur Android. Il est en effet possible de transférer ces programmes sur le téléphone ou l'émulateur et de les utiliser directement sur la plate-forme. Cependant, de telles modifications sont nécessairement limitées : on ne peut installer les applications que dans certains répertoires, on ne peut pas choisir de lancer certains programmes au démarrage, etc.

Dans ce chapitre, nous allons voir comment modifier l'image système d'Android et détailler le processus de démarrage pour nous affranchir de ces limitations.

Nous allons, en fait, créer une version personnalisée d'Android.

L'image système d'Android

L'image système que nous allons modifier est l'image correspondant au fichier `ramdisk.img`, associé à la partition racine. Ce choix est motivé par les raisons suivantes :

- Cette image est facilement modifiable. Il s'agit en fait d'une archive compressée, qui ne nécessite aucun outil particulier pour être extraite et modifiée.
- C'est l'image la plus importante. En effet, elle est associée à la partition racine et contient entre autres le fichier de démarrage.

Afin de créer une image système personnalisée, nous allons nous baser sur une image existante, que nous modifierons. L'image fournie avec l'émulateur semble être un bon point de départ. Nous allons donc la copier depuis le fichier `tools/lib/images/ramdisk.img` du SDK Android vers un répertoire créé pour l'occasion, que nous appellerons `mon_image/`.

Cette image, au format initramfs, est en fait une archive de type cpio compressée. Pour l'extraire, nous allons utiliser les commandes suivantes :

```
gunzip -S.img ramdisk.img
cpio -i -F ramdisk
cpio -t -F ramdisk > contenu
```

La première commande décompresse l'archive, la seconde l'extrait, et la troisième dresse la liste de son contenu, qui nous sera utile ultérieurement pour recréer l'archive.

Une fois ces commandes exécutées, on obtient un ensemble de fichiers. Le plus important, sur lequel nous reviendrons plus tard, est le fichier `init.rc`, le fichier de démarrage. On obtient également quelques répertoires. Ceux-ci ne sont en fait que des points de montage pour d'autres images, ou pour des systèmes de fichiers spéciaux (tel `/proc`). Ils sont donc vides et ne sont « remplis » qu'après le démarrage.

Première modification

Pour ajouter un fichier à l'image système, il faut tout d'abord l'ajouter au répertoire contenant l'archive. Créons par exemple un fichier `test`, dont le contenu importe peu, dans le répertoire `mon_image/`.

```
cd mon_image/
echo ceci est un test > test
```

Il faut ensuite ajouter le nom du fichier à la liste du contenu de l'archive, avec la commande :

```
echo test >> contenu
```

On utilise ensuite `cpio` pour créer une archive selon la liste fournie, puis on la compresse :

```
cat contenu | cpio -o -H newc -O mon_image
gzip -S.img mon_image
```

On obtient une nouvelle image système : `mon_image.img`. Pour la tester, le plus simple est d'avoir recours à l'émulateur, qui dispose d'une option prévue à cet effet :

```
emulator -ramdisk mon_image/mon_image.img
```

Un exemple complet de personnalisation

Dans cette partie, nous allons étudier les étapes qui permettent de créer une image système complète. Cela nous permettra d'aborder tour à tour et de façon concrète les différentes manières de modifier et d'améliorer l'image système.

Installation de Busybox

Comme nous l'avons vu dans le chapitre précédent, Busybox est un programme particulier, qui se comporte de façon différente suivant le nom sous lequel il est lancé. En pratique, cela signifie qu'il faut créer de nombreux liens symboliques pointant vers Busybox,

dont les noms sont ceux des applets. Nous allons donc créer dans notre image système un nouveau répertoire `/bin`, contenant Busybox et les liens symboliques associés aux applets.

1. Tout d'abord, il s'agit de créer le répertoire `bin` dans notre dossier `mon_image`. Il faut ensuite copier le binaire `busybox` dans ce répertoire `bin`.
2. La liste de applets varie selon les options choisies lors de la compilation. Pour la récupérer, il va donc falloir la recréer à partir des sources. En effet, lors du choix des applets, effectué avant la compilation, un fichier d'en-tête contenant entre autres la liste des applets est généré automatiquement. Il s'agit du fichier `include/applet_tables.h`.
3. On peut créer l'ensemble des liens symboliques en se plaçant dans le nouveau répertoire `mon_image/bin` et en lançant la commande suivante après avoir remplacé `source` par le nom du répertoire contenant les sources de Busybox :

```
awk -F\" '/applet_names/,/;/{if($2) system("ln -s busybox " $2)}'
   ➥ source/include/applet_tables.h
```

Une fois les liens créés, on ajoute les fichiers de ce nouveau répertoire `bin` au contenu de l'archive :

```
find bin >> contenu
```

Le but de ce livre n'est certes pas l'étude du langage AWK, mais la commande ci-dessus mérite une explication. Le fichier `applet_tables.h` contient la liste des noms des applets formatée comme un groupe de chaînes de caractères en langage C. Voici un aperçu du fichier :

```
const char applet_names[] ALIGN1 = ""
"[" "\0"
"[[" "\0"
"addgroup" "\0"
"adduser" "\0"
"adjtimex" "\0"
[…]
"zcip" "\0"
;
```

On notera que les différentes chaînes de caractères sont séparées par des `\0`, ou caractères nuls, qui délimitent les chaînes en langage C. Décomposons maintenant la commande `AWK` :

- `/applet_names/,/;/` indique que le traitement n'aura lieu qu'entre la chaîne 'applet_names' et le ';' final. Cela permet d'isoler la partie du fichier qui nous intéresse.
- L'option `-F\"` indique à AWK de séparer les champs à l'aide du caractère `"`. Ce caractère est échappé (précédé d'un `\`) pour éviter que le shell ne l'interprète. Le champ 1 est donc tout ce qui, sur une ligne, précède le premier guillemet, le champ 2, celui qui nous intéresse, se trouve après le premier guillemet et avant le second, etc.
- Le champ 2 contient normalement le nom de l'applet. On s'assure avec `if($2)` que ce champ n'est pas vide. On invoque ensuite le shell avec la commande `ln -s busybox` suivie du nom de l'applet. Par exemple pour la ligne 4 on aura : `ln -s busybox addgroup`.

On a donc bien créé des liens vers Busybox portant le nom des applets. Pour en savoir plus sur le langage AWK :

Blaess C., *Shells Linux et Unix par la pratique*, 2008

Personnaliser la procédure de login

Une fois Busybox correctement installé, la partie n'est pas gagnée pour autant. En effet, le shell fourni par Android par défaut n'est pas celui de Busybox (Ash), et le répertoire `/bin`

nouvellement crée ne fait pas partie du `PATH`, et n'est donc pas consulté... Il faut donc à chaque connexion modifier la variable d'environnement `$PATH` et changer de shell en invoquant manuellement `/bin/ash`.

Sur un « vrai » système Unix, ce travail est habituellement effectué par la procédure de login, une fois que celle-ci a pu authentifier l'utilisateur (le plus souvent grâce à son mot de passe). Nous allons mettre en place une telle procédure.

Le problème qui se pose sous Android est que les fichiers `/etc/passwd` et `/etc/group` sont absents. Ces fichiers stockent en temps normal les informations sur les utilisateurs et les groupes et sont donc requis par le programme `login`. Il va donc nous falloir les créer.

Or, comme nous l'avons constaté au début de ce chapitre, le répertoire `/etc` ne fait pas partie de l'image système que nous sommes en train de modifier. En fait, `/etc` n'est qu'un lien vers `/system/etc`, dont le contenu se trouve sur l'image système `system.img`. Plutôt que de modifier cette image, nous allons nous aussi avoir recours à des liens symboliques – une pratique courante au sein du système de fichiers d'Android.

Choisissons par exemple de placer ces fichiers dans le répertoire `/usr/etc`. Le choix du répertoire importe peu du moment que les liens sont crées correctement. Nous allons donc créer le répertoire `usr/etc` dans le dossier `mon_image/` :

```
mkdir -p usr/etc
```

Plaçons-nous ensuite dans ce répertoire pour créer nos deux fichiers :

```
cd usr/etc
echo root::0:0:root:/home:/bin/ash > passwd
```

On se contente d'un seul utilisateur, l'administrateur (`root`), dont le répertoire personnel sera `/home` et le shell `/bin/ash` (shell fourni par Busybox). Pour des raisons de simplicité, on ne définit pas de mot de passe. On crée ensuite le groupe associé :

```
echo root::0: > group
```

Bien évidemment, on peut ajouter de la sorte plusieurs utilisateurs et définir si on le désire des mots de passe pour chacun d'eux.

Une fois ces opérations terminées, n'oublions pas d'ajouter les fichiers au contenu de l'archive :

```
find usr >> contenu
```

Modifier le fichier de démarrage

Le répertoire `/usr/etc` étant en place, il nous faut maintenant ajouter des liens dans `/etc` pointant vers lui. Le montage des différentes images systèmes et la création éventuelle de liens entre elles se faisant lors du démarrage, il va donc nous falloir modifier le fichier qui contrôle l'étape du boot.

Ce fichier, `init.rc`, fait partie de notre image système. Il est composé d'une série de commandes qui sont exécutées dans l'ordre lors du démarrage. Nous allons l'ouvrir avec un éditeur pour le modifier (les modifications sont mises en exergue) :

```
# Backward compatibility
    symlink /system/etc /etc
    symlink /data/local /home
    symlink /cache /tmp
```

On ajoute sous forme de lien les répertoires `/home` et `/tmp`. Le premier nous servira de répertoire personnel. L'utilisation de `/data/local` est particulièrement judicieuse puisque ce répertoire est monté en écriture et que son contenu est conservé après un redémarrage. Le répertoire `/tmp` pointe en fait vers `/cache` qui a déjà vocation de répertoire « temporaire ». Beaucoup d'applications Unix étant habituées à la présence du dossier `/tmp`, il est recommandé de le créer.

On supprime ensuite les deux lignes suivantes :

```
mount rootfs rootfs / ro remount
mount yaffs2 mtd@system /system ro remount
```

Ces lignes avaient pour rôle de verrouiller en lecture seule :

- le répertoire racine (correspondant à notre image système), ainsi que
- le répertoire `/system` (correspondant à l'image `system.img`).

Garder l'accès en écriture nous permettra notamment de créer des liens symboliques.

> **ATTENTION Limite de l'accès en écriture : des modifications qui ne sont que temporaires**
>
> L'accès en écriture permet des modifications temporaires sur ces deux points de montage. En effet, les images système ne sont pas modifiables depuis Android, et les modifications faites ne sont donc pas enregistrées : elles seront perdues lors d'un redémarrage.
> La seule partition qui échappe à cette règle est la partition `/data` (d'où notre choix pour `/home`).

Enfin, on crée les liens vers `passwd` et `group` en ajoutant ces deux lignes :

```
chown root root /cache/lost+found
chmod 0770 /cache/lost+found
symlink /usr/etc/passwd /system/etc/passwd
symlink /usr/etc/group /system/etc/group
```

Création du fichier de profil

Le fichier `profile` est utile à deux égards. Tout d'abord, il est nécessaire au fonctionnement d'Android, qui repose sur un certain nombre de variables. Ensuite, il permet de personnaliser encore plus l'environnement en y ajoutant ses propres variables, alias, et fonctions.

Lors de son démarrage, le shell Ash fourni par Busybox lit deux fichiers de type `profile` :

- `/etc/profile` est lu en premier, et contient en théorie une configuration commune à tous les utilisateurs du système.
- Ensuite, s'il existe, le fichier `.profile` situé dans le répertoire personnel de l'utilisateur est lu. C'est là que se trouvent des options de configuration plus personnelles, telles que les alias et les variables propres à l'utilisateur.

Nous allons créer un fichier `/etc/profile`. De même que pour le fichier `passwd`, celui-ci se trouvera en fait dans `/usr/etc`, et nous créerons un lien symbolique. Dans `mon_image/usr/etc`, on crée donc le fichier `profile` avec le contenu suivant :

```
export ANDROID_ROOT=/system
export LD_LIBRARY_PATH=/system/lib
export PATH=/bin:/sbin:/system/sbin:/system/bin:/system/xbin
export BOOTCLASSPATH=/system/framework/core.jar:/system/framework/ext.jar:/
system/framework/framework.jar:/system/framework/android.policy.jar:/system/
framework/services.jar
export ANDROID_BOOTLOGO=1
export ANDROID_ASSETS=/system/app
export EXTERNAL_STORAGE=/sdcard
export ANDROID_DATA=/data
export ANDROID_PROPERTY_WORKSPACE=9,32768
```

Seule la ligne surlignée est en fait modifiée. Il s'agit d'ajouter notre nouveau répertoire `/bin` au `PATH` afin de pouvoir lancer directement les commandes offertes par Busybox. Les autres

lignes concernent des variables d'environnement propres à Android. Leurs valeurs sont identiques à celles définies au début du fichier `init.rc`.

Il faut maintenant créer le lien symbolique depuis `/etc` en ajoutant la ligne suivante au fichier `init.rc` :

```
symlink /usr/etc/profile /system/etc/profile
```

Enfin, on ajoute le fichier `profile` au contenu de l'archive :

```
echo usr/etc/profile >> contenu
```

Mise en place d'un serveur Telnet

Afin de bénéficier de tous les avantages offerts par notre procédure de login, nous devons trouver un moyen de la lancer à chaque connexion au shell. Malheureusement, la commande `adb shell` lance par défaut le shell d'Android.

> **Ergonomie Inconfort lié à adb shell**
>
> Il serait bien sûr possible d'utiliser `adb shell /bin/login`, voire de créer un alias. Cependant, outre le manque d'élégance de la solution, on notera que la commande `adb shell` présente quelques problèmes liés aux touches spéciales (telles que les flèches, la tabulation…) qui varient selon le système d'exploitation utilisé, et qui sont fort ennuyeux pour un utilisateur habitué à l'historique des commandes, à la complétion automatique, etc.

Nous allons résoudre plusieurs problèmes du même coup en mettant en place un serveur Telnet. Ainsi, au lieu de nous connecter à Android via `adb shell`, nous utiliserons un client Telnet. Ce dernier étant conçu pour traiter correctement les flèches et autres tabu-

lations, cela nous permettra d'utiliser les fonctionnalités associées dans le shell. De plus, à chaque connexion, Telnet lance automatiquement la procédure de login. Enfin, cela nous permettra de voir comment lancer des applications au démarrage.

Busybox, dont l'étendue des fonctionnalités ne cesse de surprendre, propose une applet faisant office de serveur Telnet : `telnetd`. (Bien évidemment, Busybox fournit aussi un client Telnet, nommé simplement `telnet`.)

Vérifions que cette appletest présente dans Busybox en regardant par exemple s'il existe un lien symbolique nommé `telnetd` dans le répertoire `/bin`. Si ce n'est pas le cas, il faut recompiler Busybox comme expliqué plus tôt, en sélectionnant l'applet `telnetd` dans le menu de configuration. Il faudra également recréer les liens symboliques dans le répertoire `/bin`.

Une fois certain que `/bin/telnetd` est bien présent sur notre système Android, il faut créer un service pour lancer le serveur au démarrage. On ajoutera donc la ligne suivante à la fin du fichier `init.rc` :

```
service telnetd /bin/telnetd
```

On peut enfin générer l'image système à l'aide de ces commandes :

```
cat contenu | cpio -o -H newc -O mon_image
gzip -S.img mon_image
```

Et on peut tester cette image avec l'émulateur :

```
emulator -ramdisk mon_image/mon_image.img
```

Connexion à Android

Une fois mis en place le serveur Telnet et lancé l'émulateur (ou le téléphone), on peut se connecter à Android par le réseau. Par défaut, le serveur Telnet écoute sur le port 23 d'Android, associé au protocole réseau TCP.

Ce dernier ne disposant pas d'adresse propre, il nous faut avoir recours à une redirection de port pour pouvoir le contacter. Nous allons donc associer le port 4444 (par exemple) de l'ordinateur au port 23 d'Android avec la commande suivante :

```
adb forward tcp:4444 tcp:23
```

Il suffit ensuite de lancer un client Telnet vers le port 4444 de l'ordinateur pour se retrouver sur Android :

```
telnet localhost 4444
```

Une fois tapé le nom d'utilisateur (`root`), nous voilà connecté sur Android, directement sur le shell de Busybox (Ash), et dans le répertoire `/home` que nous avons créé !

> Un autre avantage du serveur Telnet est qu'il existe de nombreuses applications Java faisant office de client Telnet, disponibles directement sur Android Market (chercher « Telnet.apk »). Une fois l'application installée, et si le serveur Telnet est bien lancé au démarrage, il suffit de se connecter en local (`localhost`) sur le port 23 pour obtenir un shell sur Android.
> Un émulateur de terminal à peu de frais, en somme...

4

Compiler avec le SDK Android

Depuis que Google a décidé de mettre à disposition le code source d'Android, il est devenu possible d'étudier l'intégralité du système et, surtout, de bénéficier de l'architecture de compilation pour apporter des modifications au plus bas niveau.

- architecture de compilation
- code source
- JNI

Fin 2008, Google a décidé de mettre le code source d'Android à disposition du public sur Internet. Il est donc possible à tout un chacun de le télécharger pour l'étudier dans les moindres détails. Plus intéressant encore, ce code source est livré avec une infrastructure de compilation qu'il est possible de mettre à profit pour écrire ses propres programmes.

Dans ce chapitre, nous allons voir comment ajouter son propre code à l'environnement de programmation Android (le SDK). Nous profiterons de cette occasion pour aborder la technologie JNI, qui permet d'interfacer du code C et du code Java.

Avant de commencer : où trouver plus d'informations

Une petite mise au point s'impose avant de commencer l'exploration du SDK Android. En effet, bien que disponible à tous sur Internet, le code source d'Android s'adresse de toute évidence à un public expérimenté. Contrairement à la partie « programmation Java », pour laquelle Google fournit une documentation détaillée à destination de débutants, il n'y a pas ici d'aide ou de manuel.

Le but de ce chapitre est donc précisément de pallier ce manque d'information, en fournissant non pas une documentation exhaustive, mais plutôt des moyens de comprendre le fonctionnement du SDK et de trouver des réponses par soi-même.

Voici déjà quelques pistes :

- Les listes de diffusion disponibles – la liste android-platform est particulièrement recommandée : http://source.android.com/discuss.
- Le site web http://www.android.com.
- Un document décrivant le SDK, consultable sous forme de brouillon :

http://android.git.kernel.org/?p=platform/development.git;a=blob_plain;f=pdk/docs/build_system.html;hb=HEAD

- Enfin, le code source lui même est probablement la source d'information la plus fiable !

Prérequis en termes d'espace disque et d'équipement logiciel

Avant toute chose, précisons que le code source est volumineux : au moins 2 Gigaoctets, et 6 Go une fois compilé. Il faut donc prévoir de la place sur le disque.

Ensuite, Google conseille d'utiliser la distribution Linux Ubuntu. Ce choix ne semble pas crucial, plusieurs personnes (dont l'auteur !), ayant réussi sans problème à compiler les sources sur une autre distribution. Par contre, il est très fortement conseillé d'utiliser un système configuré en mode 32 bits, pour éviter d'avoir à jongler entre de multiples versions pour les bibliothèques de développement.

> **Alternative Et sous Mac OX ?**
>
> D'après le site, il semble également possible d'utiliser Mac OS.

Enfin, voici une petite liste des logiciels et des bibliothèques qu'il faut avoir installés sur son système :

- Git 1.5.4 ou ultérieur
- le JDK 5 – attention, le JDK 6 n'est pas compatible !

- Python
- libncurses (version développement)
- Flex, Bison, gperf, Zip, Curl

> **Référence** **Liste à jour des logiciels et bibliothèques à installer**
>
> Une liste à jour des logiciels et bibliothèques à installer est disponible en ligne :
> ▸ http://source.android.com/download

Premiers pas avec le SDK Android

Récupérer le code source

Le code est disponible sur un dépôt git, qui permet aux développeurs de travailler en collaboration sur le code, de gérer les différentes versions, etc. Google fournit un outil facilitant l'interfaçage avec ce dépôt de code : `repo`. Afin de récupérer le code, il faut donc télécharger `repo` à l'adresse http://android.git.kernel.org/repo.

On crée ensuite un répertoire où mettre le code source, par exemple `mydroid`. Dans ce répertoire, on initialise `repo` avec la commande :

```
repo init -u git://android.git.kernel.org/platform/manifest.git
```

Ensuite, la commande `repo sync` permet de récupérer l'ensemble des fichiers.

On retrouve alors avec les fichiers suivants, correspondant à différentes parties du code :

```
build           dalvik       frameworks  packages  system
bionic          development  hardware    Makefile  prebuilt
bootloader      external     kernel      out       recovery
```

- Le dossier `build` contient le système de compilation.
- Le dossier `bionic` contient la bibliothèque C du même nom.
- Le code source de la machine virtuelle Java se trouve dans le dossier `dalvik`…

Le tout est documenté sur le site Android : http://source.android.com/projects.

> **ATTENTION Interdépendances nombreuses**
>
> Il est en théorie possible de ne télécharger et de ne compiler que l'un ou l'autre de ces dossiers. Cependant, en raison de nombreuses interdépendances, il est fortement recommandé, au moins la première fois, de tout télécharger et de tout compiler d'un coup.

Compiler le code

Afin de simplifier la vie des développeurs, l'équipe Android met à leur disposition un script regroupant un ensemble de fonctions d'usage courant. Bien que peu documenté, celui-ci se révèle très pratique. Pour l'utiliser, il suffit de le sourcer dans son shell en tapant :

```
. build/envsetup.sh
```

On peut obtenir de l'aide sur ce script en tapant la commande `help`, qui affiche la liste des fonctions disponibles :

```
Invoke ". envsetup.sh" from your shell to add the following functions to your
environment:
- croot:   Changes directory to the top of the tree.
- m:       Makes from the top of the tree.
- mm:      Builds all of the modules in the current directory.
- mmm:     Builds all of the modules in the supplied directories.
- cgrep:   Greps on all local C/C++ files.
- jgrep:   Greps on all local Java files.
- resgrep: Greps on all local res/*.xml files.

Look at the source to view more functions. The complete list is:
cgrep choosecombo chooseproduct choosetype croot findmakefile gdbclient
get_abs_build_var getbugreports get_build_var getprebuilt gettop help
isviewserverstarted jgrep lunch m mm mmm partner_setup pid printconfig
print_lunch_menu resgrep runhat setpaths set_sequence_number
set_stuff_for_environment settitle smoketest startviewserver stopviewserver tapas
tracedmdump
```

Notons la présence des fonctions `m`, `mm` et `mmm`, qui permettent respectivement de lancer la compilation depuis le répertoire racine (`mydroid`), le répertoire courant, ou un répertoire donné en argument.

La commande `choosecombo` fournie par le script va nous permettre de configurer l'environnement de compilation. La commande pose un certain nombre de questions pour déterminer :

- quel est le système cible (simulateur ou téléphone, par exemple), et
- de quel type de compilation il s'agit (mode de débogage activé ou non).

Une fois la configuration choisie, on peut l'afficher avec la commande `printconfig`.

Notre configuration

Pour information, le reste du chapitre a été écrit avec la configuration suivante :

```
TARGET_PRODUCT=generic
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
```

Enfin, on lance la compilation avec la commande `m`. Cela prend longtemps en raison du volume de code à compiler. Il est également probable que la compilation avorte et affiche une erreur. Dans la plupart des cas, cela est dû à une bibliothèque de développement manquante qu'il faut alors installer avant de relancer la compilation.

Si tout se passe comme prévu, on obtient in fine une image système toute neuve que l'on peut tester avec l'émulateur du SDK :

```
out/host/linux-x86/bin/emulator -system out/target/product/generic/ -kernel
prebuilt/android-arm/kernel/kernel-qemu
```

Ajouter du code au SDK : JNI

Un exemple valant mieux qu'un long discours, nous allons écrire un programme que nous allons compiler avec le SDK Android. Ce sera l'occasion de découvrir comment fonctionne le système de compilation. Et ce sera également un excellent prétexte pour découvrir la technologie JNI.

Le but n'est bien sûr pas ici d'expliquer le fonctionnement de JNI. On se contentera de montrer l'utilisation de cette technologie avec Android.

Pour résumer, JNI permet d'interfacer du code écrit en Java avec du code « natif », écrit en C ou en C++. Sous réserve d'utiliser JNI correctement, on peut ainsi appeler depuis une classe Java une fonction écrite en C/C++ dans un autre fichier source et compilé à part.

Cette technologie est particulièrement intéressante dans le cas d'Android. En effet, la machine virtuelle Java d'Android n'a pas pour objectif premier d'être performante en termes de rapidité de calcul. Cependant, avec JNI, ce problème peut très bien être surmonté en écrivant certaines parties du code de l'application en C ou en C++.

Afin d'expliquer l'utilisation de JNI pour Android, l'équipe de développement a mis en ligne un petit projet destiné à servir d'exemple, disponible à l'adresse suivante http://android.git.kernel.org/?p=platform/development.git;a=tree;f=samples/PlatformLibrary;hb=cupcake. Dans la suite du chapitre, nous allons nous baser sur une version simplifiée et commentée de ce code.

> **Méthode Recommandation de Google : écrire une partie en C/C++**
>
> C'est la méthode que recommande l'équipe de Google pour obtenir de bonnes performances pour des applications demandant beaucoup de calculs (traitement multimédia, par exemple).

Le code Java

Commençons par le plus simple, le code Java :

```
package com.Hello;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;
import android.util.Log;
```

```
public class Hello extends Activity {

    static {
       try {
            Log.i("JNI", "Tentative de chargement de libHello.so");
            System.loadLibrary("Hello");
        }
        catch (UnsatisfiedLinkError ule) {
            Log.e("JNI", "WARNING: Echec lors du chargement de libHello.so");
        }
    }

    // La partie importante : définition d'une fonction importée depuis le code C++
    native private int add(int a, int b);

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // appel de la méthode « native »
        int res = add(3,5);

        // Afficher le résultat dans une zone de texte
        TextView tv = new TextView(this);
        tv.setText("C++ vous dit que 3+5 = " + res);
        setContentView(tv);
    }
}
```

Ce code est tout ce qu'il y a de plus classique sous Android. On définit une classe qui hérite d'Activity, puis, grâce à la méthode `onCreate`, on met en place une zone de texte que l'on affiche.

On notera toutefois une différence de taille avec le code présenté habituellement. En effet, par le biais de l'appel de fonction `System.loadLibrary("Hello")`, l'application Java va charger notre bibliothèque de fonctions C++.

Précisons que la fonction `loadLibrary` va chercher la bibliothèque donnée en argument dans un répertoire bien défini (`/system/lib`, sous Android). On peut utiliser la fonction `System.load` à la place, en lui fournissant le chemin complet vers la bibliothèque. Cela permet de placer cette dernière dans n'importe quel dossier.

Comme indiqué dans le commentaire, la ligne `nativeprivate int add(int a, int b);` déclare une fonction « native », c'est-à-dire importée depuis la bibliothèque C++. Cette fonction peut ensuite être appelée comme toute méthode Java, et c'est d'ailleurs ce qui est fait pour additionner 3 et 5.

Enfin, pour faciliter le débogage, on a inséré dans le code des directive `Log.e` et `Log.i` (selon que l'on affiche une erreur ou une simple information). Le journal (log) où sont affichés ces messages est accessible depuis le shell Android avec la commande `logcat`.

Le code C++

Voici le fichier contenant le code C++. Il s'agit d'une version modifiée du code fourni par l'équipe Android.

```
/*
* Copyright (C) 2008 The Android Open Source Project
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
```

```
*      http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*
* Modified (Simplified) by F. Brault (2009), same license applies.
*/

#define LOG_TAG "Hello"
#include "utils/Log.h"


#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>

#include "jni.h"

/*
* La fonction que nous allons appeler depuis Java : additionner deux nombres.
*/
static jint Hello_add(JNIEnv* env, jobject thiz, jint a, jint b) {
    return (jint)(a + b);     (1)
}

/*
* Tableau de méthodes.
*
* Chaque entrée du tableau comporte trois champs : le nom de la méthode,
* sa signature JNI, et un pointer vers l'implémentation.
* Ici, nous ne déclarons que notre fonction d'addition.
*/
```

```
*/
static const JNINativeMethod gMethods[] = {          2
    { "add", "(II)I", (void*)Hello_add }
};


/*
* Enregistrer les fonctions définies dans le tableau précédent auprès de la classe
Java
* Renvoie 0 en cas de succès.
*/
static int registerMethods(JNIEnv* env) {          3
    // nom complet de la classe Java
    static const char* const kClassName = "com/Hello/Hello";
    jclass clazz;

    // Récupérer la classe à partir de son nom
    clazz = env->FindClass(kClassName);
    if (clazz == NULL) {
        LOGE("Impossible de trouver la classe : %s\n", kClassName);
        return -1;
    }

    // Enregistrer toutes les méthodes du tableau
    if (env->RegisterNatives(clazz, gMethods,
            sizeof(gMethods) / sizeof(gMethods[0])) != JNI_OK)
    {
        LOGE("Echec lors de l'enregistrement des methodes pour la classe %s\n"
             , kClassName);
        return -1;
    }

    return 0;
}
```

```
/*
* Cette méthode est appelée automatiquement par la machine virtuelle Java lors du
* chargement de la bibliothèque.
*/
jint JNI_OnLoad(JavaVM* vm, void* reserved) {      ❹
    JNIEnv* env = NULL;
    jint result = -1;

    // Récupération de l'environnement
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: Echec getenv\n");
        goto bail;
    }
    assert(env != NULL);

    if (registerMethods(env) != 0) {
        LOGE("ERROR: Echec enregistrement des méthodes\n");
        goto bail;
    }

    // En cas de succès : renvoyer un numéro de version valide
    result = JNI_VERSION_1_4;

  // En cas d'erreur, result contient -1
bail:
    return result;
}
```

Beaucoup de code pour additionner deux nombres !

Notons que seule la fonction `Hello_add` ❶ implémente la fonction `add` déclarée et appelée dans le fichier Java. On remarque que cette fonction prend plus d'arguments en C++ qu'en Java : cela est dû à JNI, qui ajoute un environnement et un objet à la liste des

arguments. Pour le reste, à part les types utilisés qui sont propres à JNI, le code est du code C++ standard.

Le tableau `gMethods` ❷ sert à stocker la liste des fonctions C++ que nous voulons exporter vers Java. À chacune, on associe un nom (celui qui sera utilisé par la partie Java du code), et une signature. Ici, `(II)I` signifie que la méthode prend en argument deux entiers, et renvoie un entier.

> **Conventions pour les signatures**
>
> Les signatures utilisent une convention particulière, documentée par exemple sur le site suivant :
> ‣ http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/types.html#wp16432 .

La fonction `registerMethods` ❸ se charge d'enregistrer les méthodes se trouvant dans ce tableau auprès de la classe Java identifiée par son nom complet. Cette fonction d'enregistrement est appelée lors du chargement de la bibliothèque, par la fonction `JNI_OnLoad` ❹ (c'est-à-dire lorsque le code Java fait appel à la fonction `System.loadLibrary`).

Pour modifier ce code C++, il faut donc :

1. changer le code de la fonction `Hello_add`, ou ajouter d'autres fonctions ;
2. renseigner le tableau `gMethods` en prenant soin notamment de donner des signatures valides ;
3. enfin, il faut que le nom de la classe dans `registerMethods` corresponde bien à celui de la classe Java qui va charger la bibliothèque.

Compiler avec le SDK Android

Configuration de la compilation

Maintenant que le code est écrit, il faut le compiler avec le SDK.

> **ALTERNATIVE Compiler avec Eclipse**
>
> Notons qu'on pourrait tout à fait choisir de compiler la partie Java directement avec Eclipse. Mais afin d'illustrer le fonctionnement du SDK Android, nous allons tout faire à la main.

Tout d'abord, plaçons-nous dans le répertoire `external` du SDK où se trouvent les applications « externes », développées par des tiers. Nous allons créer un répertoire `test`, et deux sous-répertoires : `java` et `jni`.

Compilation du code Java

On placera le code source Java dans le répertoire `java`, dans le fichier `Hello.java`. Il s'agit ensuite d'écrire, comme pour toute application Java sous Android, un fichier `AndroidManifest.xml` qui la décrit.

Le fichier AndroidManifest.xml pour décrire l'application

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.Hello">
    <application android:label="Hello">
        <activity android:name="Hello">
```

```
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Ensuite, on écrit le fichier `Android.mk`, équivalent des Makefile bien connus des programmeurs habitués à Unix. Ce fichier décrit comment compiler le code :

Le fichier Android.mk, pour configurer la compilation du code Java

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := samples

LOCAL_PACKAGE_NAME := Hello ❶

LOCAL_SRC_FILES := Hello.java    ❷

LOCAL_SDK_VERSION := current     ❸

include $(BUILD_PACKAGE)    ❶
```

❶ Ce fichier définit le nom de l'application que l'on veut créer (`Hello`), ❷ donne la liste des fichiers sources associés (`Hello.java`).

❸ La directive `include $(BUILD_PACKAGE)` demande au système de compilation du SDK de construire un package, c'est-à-dire une application Java `.apk`.

Compilation du code C++

De même, dans le répertoire `jni`, on place le code C++ dans le fichier `Hello.cpp` et on écrit un fichier `Android.mk` :

Le fichier Android.mk, pour configurer la compilation du code C++

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := samples

LOCAL_MODULE:= libHello          (1)

LOCAL_SRC_FILES:= Hello.cpp      (2)

LOCAL_SHARED_LIBRARIES := \      (3)
        libandroid_runtime \
        libnativehelper \
        libcutils \
        libutils

# JNI headers
LOCAL_C_INCLUDES += $(JNI_H_INCLUDE)     (4)

LOCAL_PRELINK_MODULE := false

include $(BUILD_SHARED_LIBRARY)    (5)
```

Cette fois-ci, on cherche à compiler non pas une application .`apk`, mais une bibliothèque partagée (fichier .`so` sous Linux).

(1) On indique le nom de cette bibliothèque (`libHello`), (2) puis les fichiers sources associés (`Hello.cpp`).

❸ Ensuite, on indique les éventuelles dépendances (`LOCAL_SHARED_LIBRARY`).

❹ On demande aussi au système de compilation d'ajouter à la liste des en-têtes C les en-têtes JNI.

❺ Enfin, la directive `include $(BUILD_SHARED_LIBRARY)` permet de compiler une bibliothèque partagée.

Compilation et test

Une fois que tous les fichiers `Android.mk` sont renseignés :

1. Sourcer dans son shell le fichier `build/envsetup.sh` du SDK et vérifier avec `printconfig` que l'on se trouve dans la bonne configuration.
2. Dans le répertoire `java`, taper la commande `mm`, pour obtenir un fichier `Hello.apk`.
3. Dans le répertoire `jni`, taper `mm` pour obtenir le fichier `libHello.so`.

> **ATTENTION Emplacement des fichiers Hello.apk et libHello.so**
>
> Les fichiers `Hello.apk` et `libHello.so` ne sont pas placés directement dans les répertoires où se trouvent les sources, mais dans un sous-répertoire du dossier `out/`, afin d'être directement inclus dans l'image système créée par le SDK. Les nombreux messages affichés dans le terminal lors de la compilation permettent de connaître l'emplacement exact.

Pour tester l'application, il faut utiliser soit l'émulateur, soit le téléphone Android (en le branchant allumé à une prise USB). On installe l'application `Hello.apk` en tapant `adb install Hello.apk`.

Il faut ensuite placer le fichier `libHello.so` dans le répertoire `/system/lib` d'Android (`adb push libHello.so /system/lib/`).

> **Attention**
>
> Pour pouvoir écrire dans ce répertoire, il faut avoir les droits administrateur (root), ce qui n'est pas toujours possible.

Comme indiqué plus haut, on peut contourner cette difficulté, en utilisant la fonction `System.load` dans le code Java : on peut alors placer `libHello.so` dans n'importe quel répertoire, à condition d'en préciser le chemin complet.

Figure 4–1
Notre programme JNI, affichant fièrement que 3 et 5 font 8 !

Figure 4–2

Le même programme, sur un G1 à l'image modifiée !

Modification de l'image système

Cette annexe fait suite à l'avant-propos, nous y expliquons comment à l'époque nous avons remplacé l'image système du G1.

Le téléphone G1 est livré avec une image système fournie par le constructeur. Ce dernier se réserve le droit de pouvoir la mettre à jour, mais il n'est normalement pas possible de la changer. C'est pourtant ce que rêve de pouvoir faire tout bricoleur qui n'a pas la chance de posséder un téléphone Dev Phone – sur ces appareils, il est prévu de pouvoir modifier facilement le système.

Justement, la procédure décrite ici permet de remplacer l'image système du G1 par l'image présentée dans ce livre.

RESSOURCE Les expérimentations en ligne

Notre procédure s'inspire de deux sites Internet :

- http://android-dls.com/wiki/index.php?
  ➥ title=HOWTO:_Unpack%2C_Edit%2C_and_Re-Pack_Boot_Images
- http://android-dls.com/wiki/index.php?title=Keeping_Root

AVERTISSEMENT Attention procédure risquée !

Attention, la procédure est risquée : il y a de fortes chances de détruire complètement l'image système et de rendre le téléphone inopérant.

1 Tout d'abord, une fois connecté en tant que root sur le téléphone, il faut récupérer le contenu des images systèmes.

```
# cat /dev/mtd/mtd1 > /sdcard/mtd1.img
# cat /dev/mtd/mtd2 > /sdcard/mtd2.img
```

2. Afin de ne pas opérer complètement sans filet, on place dans le répertoire `/sdcard` les fichiers `update.zip` et `recovery.img`, obtenus sur les sites cités plus haut.
3. Toujours sur ces sites, on récupère le script `split_bootimg.pl`, qui automatise la dissection des images systèmes. On opère sur l'image `mtd2.img` :

```
./split_bootimg.pl mtd2.img
```

   Ce script sépare l'image en deux parties :
   - le noyau d'un côté (`mtd2.img-kernel`),
   - l'image système de l'autre (`mtd2.img-ramdisk.gz`).
4. L'image système peut être remplacée par une image personnalisée, telle que celle étudiée dans ce livre. On prendra soin néanmoins d'ajouter les fichiers `init.trout.rc` et `logo.rle`, qui ne sont pas présents dans les images de l'émulateur.
5. On peut également en profiter pour changer les paramètres de débogage et de sécurité dans le fichier `default.prop`.
6. Il faut ensuite recréer l'image globale à partir du noyau et du ramdisk. C'est le rôle de la commande `mkbootimg`, fournie dans le SDK Android.

```
mkbootimg --cmdline 'no_console_suspend=1' --kernel mtd2.img-kernel --ramdisk
mydisk.img -o newboot.img
```

7. Une fois cette image chargée sur le téléphone, avec `adb push` (par exemple dans le dossier `/sdcard`), on lance en tant que root les commandes suivantes :

```
cat /dev/zero > /dev/mtd/mtd2
flash_image boot /sdcard/newboot.img
```

Une fois l'opération terminée, il faut redémarrer le téléphone pour connaître le verdict. Soit le téléphone redémarre correctement avec la nouvelle image système, soit il ne redémarre pas. Dans ce cas quelques procédures sont proposées sur le site pour minimiser les dégâts, mais il y a de grandes chances pour que votre téléphone soit devenu un onéreux presse papier...

Nous espérons que cet ouvrage vous aura ouvert des perspectives, et qu'il vous prépare à de bons moments d'amusement... Sur ce, hackez bien et appropriez-vous Android !