



Prise en main de  
Scratchbox

Version: 1.01

Auteur: Hugues HIEGEL

# Prise en main de Scratchbox

Hugues HIEGEL  
hugues.hiegel@openwide.fr

Avril 2009



Prise en main de  
Scratchbox

Version: 1.01  
Auteur: Hugues HIEGEL

## MODIFICATIONS

<b>VERSION</b>	<b>DATE</b>	<b>AUTEUR(S)</b>	<b>DESCRIPTION</b>
1.0	20/04/09	H. Hiegel	Création
1.01	20/05/09	P. Ficheux	Relecture et correction mineures

## Table des matières

1. Présentation de Scratchbox.....	4
2. Installation de Scratchbox.....	5
Depuis les paquetages binaires.....	5
Depuis les sources.....	5
3. Installation de l'environnement de développement.....	7
Installation de la chaîne de compilation croisée.....	7
3.1. Chaîne de compilation sous forme binaire.....	7
3.2. Chaîne de compilation sous forme de sources.....	8
Mise en place du système de fichiers cible.....	8
4. Mise en oeuvre de Scratchbox.....	11
Initialisation d'un profil Scratchbox.....	11
Utilisation de notre profil.....	11
4.1. Compilations de référence.....	12
4.2. Compilation vers une cible non gérée.....	13
4.3. Installation dans le système de fichiers cible.....	14
5. Cas de figure : Kerberos 5.....	16
Cross-compilation « témoin ».....	16
Compilation à l'aide de Scratchbox.....	17
6. Mesures.....	19
7. Références.....	20

## 1. Présentation de Scratchbox

Dans le cadre du développement de projets pour l'embarqué, la principale contrainte rencontrée par les développeurs est la mise en place d'un environnement de compilation croisée, et son utilisation.

A priori, l'installation d'une chaîne de compilation croisée s'effectue une seule fois, et on peut l'oublier par la suite. Son utilisation même, en revanche, dépend fortement des projets à compiler. En effet, bien que l'on trouve de plus en plus de projets pour l'embarqué, il n'est pas rare d'avoir besoin d'un quelconque programme, dont les sources n'ont pas été structurées pour une compilation croisée.

Pourtant, il aurait suffi que les Makefiles du projet en question aient été écrits en tenant compte de variables « standardisées » telles que ARCH, CROSS ou CROSS\_COMPILE.

Pour de petits projets, il est facile de réécrire les Makefiles en ce sens. Mais sur des projets de plus grande envergure, ou ayant des scripts de compilation bien plus complexes, la tâche devient tout de suite gargantuesque.

Une solution serait de compiler ces vilains canards directement sur la cible. Il va de soi que l'installation d'un environnement de développement complet sur une cible dédiée à l'embarqué est assez incongrue. D'autre part, la puissance des processeurs visés est généralement bien en deçà des possibilités offertes par les stations de travail modernes.

Scratchbox se propose donc d'utiliser un environnement chrooté<sup>[1]</sup>, en utilisant qemu, afin de permettre la compilation d'un projet comme s'il tournait directement sur la plate-forme cible.

Nous allons présenter l'installation de cet outil, sa configuration et sa mise en œuvre.

En guise d'exemple de code à compiler, nous prendrons Kerberos5, connu pour poser des problèmes à la cross-compilation.

## 2. Installation de Scratchbox

### . Depuis les paquetages binaires

Scratchbox est disponible sous forme de paquetage Debian, Ubuntu, Gentoo et Fedora.

La liste des dépôts Debian pour Scratchbox est disponible ici : [www.scratchbox.org/download](http://www.scratchbox.org/download)

Sur la Debian par exemple, après avoir ajouté les dépôts mentionnés dans mon `/etc/apt/sources.lists` et mis à jour la liste des paquetages, il suffit de faire :

```
$ apt-get install scratchbox2
```

### . Depuis les sources

Cependant, il peut être intéressant d'installer Scratchbox directement à partir des sources, en particulier si vous utilisez une distribution sur laquelle il n'existe aucun dépôt pour scratchbox.

Ces sources sont disponibles, sur le site officiel, sous git, il vous faudra donc au préalable avoir installé cet outil (Paquetages « git » et « git-core » pour une Debian).

Récupérons les sources depuis le site officiel :

```
$ git-clone git://anongit.freedesktop.org/git/sbox2 ~/tools/sbox2
Initialized empty Git repository in /home/hugues/sbox2/.git/
remote: Counting objects: 4781, done.
remote: Compressing objects: 100% (2210/2210), done.
remote: Total 4781 (delta 3162), reused 3647 (delta 2379)
Receiving objects: 100% (4781/4781), 1.19 MiB | 326 KiB/s, done.
Resolving deltas: 100% (3162/3162), done.
```

Lisons le README pour savoir comment compiler et installer les sources :

```
$ cd ~/tools/sbox2
$ grep -i "to build" -A5 README

To build SB2:

$ cd sbox2
$ ./autogen.sh
$ make install prefix=$HOME/sb2
```

Tout simplement.

On remarquera cependant que la procédure de compilation est assez particulière, car le README nous demande de lancer directement « make install » avec la variable « prefix » positionnée comme souhaité (vous l'adapterez évidemment à votre environnement).

Exécutons donc les commandes suggérées :



Prise en main de  
Scratchbox

Version: 1.01

Auteur: Hugues HIEGEL

```
$. /autogen.sh  
[...]  
$ sudo make install prefix=/usr/local  
[...]
```

Cela suppose que vous ayez les droits sudo pour la commande « make install », sinon il vous faudra utiliser un autre répertoire d'installation.

Voilà pour l'installation de Scratchbox, nous n'avons rencontré ici aucune difficulté particulière.

Maintenant, il nous faut mettre en place un environnement de développement.

## 3. Installation de l'environnement de développement

Pour utiliser Scratchbox, nous avons besoin de plusieurs éléments :

1. une chaîne de compilation croisée,
2. un système de fichiers adapté à la cible (binaires, bibliothèques, includes système),
3. un projet à compiler :)

### . Installation de la chaîne de compilation croisée

Pour la chaîne de compilation croisée, sa génération est une tâche assez ardue si l'on ne connaît pas exactement les versions de gcc, libc et binutils à utiliser.

Cependant, comme pour n'importe quel type de programme, il est possible d'en installer une :

- soit à partir de sources ; pour ce faire, le mieux est de recourir à un environnement de développement dédié à l'embarqué, tel que buildroot par exemple.
- soit à partir d'un paquetage binaire ;

En principe, si vous avez besoin de lire ce chapitre pour installer une chaîne de compilation, je vous recommande fortement de commencer par le paquetage binaire !

#### 3.1. Chaîne de compilation sous forme binaire

Généralement, les vendeurs de matériel embarqué fournissent également une chaîne de compilation croisée, basée sur gcc, le plus souvent sous forme de paquetages RPM ou de tarball.

Sinon, vous pouvez en trouver dans votre distribution favorite : si vous avez pu installer Scratchbox via le gestionnaire de paquetages de votre distribution, il y a fort à parier que vous ayez également accès à quelques chaînes précompilées, fournies également avec Scratchbox. C'est le cas sur Debian avec les paquetages « scratchbox-toolchain-\* », généralement ciblées pour ARM.

```
$ apt-cache search toolchain | grep "\<arm\>.*\<uclibc[0-9]*\>"
scratchbox-toolchain-arm-gcc3.4-uclibc0.9.28 - arm-gcc3.4-uclibc0.9.28 compiler
for Scratchbox
scratchbox-toolchain-arm-gcc4.1-uclibc20061004 - arm-gcc4.1-uclibc20061004
compiler for Scratchbox
scratchbox-toolchain-arm-gcc3.4.cs-uclibc - arm-linux-gcc3.4.cs-uclibc0.9.27
compiler for Scratchbox
scratchbox-toolchain-arm-gcc3.4.cs-uclibc-sf - arm-linux-gcc3.4.cs-uclibc0.9.27-
nofpu compiler for Scratchbox
scratchbox-toolchain-arm-uclibc - arm-gcc-3.3.2-uclibc-snapshot-20040229
compiler for Scratchbox
```

Pour l'exemple, nous allons partir sur « scratchbox-toolchain-arm-gcc4.1-uclibc20061004 », car utilisant des versions de gcc et de uClibc les plus récentes, parmi celles proposées.

```
$ apt-get install scratchbox-toolchain-arm-gcc4.1-uclibc20061004
$ dpkg -L scratchbox-toolchain-arm-gcc4.1-uclibc20061004 | head -n4
/.
/scratchbox
/scratchbox/compilers
/scratchbox/compilers/arm-gcc4.1-uclibc20061004
```

Nous avons donc une chaîne de compilation croisée pour ARM/uClibc, installée dans /scratchbox/compilers/arm-gcc4.1-uclibc20061004/.

### 3.2. Chaîne de compilation sous forme de sources

Nous pouvons recourir à buildroot pour télécharger et compiler une chaîne croisée directement à partir des sources. Une version récente nous proposera un ensemble basé sur uClibc 0.9.30 et Busybox 1.14 par exemple.

Reférez-vous à la documentation de buildroot pour plus d'informations, nous ne détaillerons pas les étapes dans ce document.

Notre chaîne de compilation installée, nous pouvons mettre en place le système de fichiers cible.

#### . Mise en place du système de fichiers cible

La génération d'un système de fichiers cible n'est pas très compliquée en soi, mais pour bien faire il faudrait y passer du temps.

Nous allons donc recourir à un outil tel que buildroot, vu précédemment, pour générer une image de système de fichiers racine (RootFS). Étant conçu pour cela, il le fait très bien.

Deux choses importantes sont à configurer, à travers le menu de configuration de buildroot : les options relatives à la chaîne de compilation à utiliser (menu « Toolchain »), et l'image à générer (dans « Target filesystem options »).

Les options spécifiques à BusyBox (menu « Package selection for target ») pourront être laissées aux valeurs par défaut.

Les options spécifiques à la cible (menus « Target \* ») devront être configurées par vos soins, selon vos besoins. Les « Target options » pourront être laissées à leurs valeurs par défaut.

```
$ svn co svn://uclibc.org/trunk/buildroot ~/tools/buildroot
[...]  
$ cd ~/tools/buildroot ; make menuconfig
```

La target a été configurée ici pour une cible ARM (en bleu) :

```
.config - buildroot v2009.05-svn Configuration  
-----  
Buildroot Configuration  
-----  
Target Architecture (arm) --->  
Target Architecture Variant (generic_arm) --->  
Target ABI (OABI) --->  
Target options --->  
Build options --->  
Toolchain --->  
Package Selection for the target --->  
[...]
```

Pour le menu « Target options », à vous de configurer les différentes variables selon vos besoins. Ceci dit, les valeurs par défaut nous conviennent.

Dans le menu « Toolchain », nous allons partir sur la chaîne de compilation installée précédemment à partir de paquetages binaires. Choisissons donc le type « External binary toolchain » :

```
.config - buildroot v2009.05-svn Configuration  
-----  
Toolchain  
-----  
Toolchain type (External binary toolchain) --->  
(libc.so.0) The core C library from the external toolchain  
[...]  
(/scratchbox/toolchains/arm-gcc4.1-uclibc20061004) External toolc  
($(ARCH)-linux-uclibc) External toolchain prefix  
-----  
<Select> < Exit > < Help >
```

On remarquera que j'ai laissé la valeur par défaut de « libc.so.0 » pour la deuxième option du menu. En effet, si l'on regarde dans le dossier lib/ de notre toolchain, on trouve un lien symbolique nommé libc.so.0 et pointant vers la libuClibc :

```
$ cd /scratchbox/compilers/arm-gcc4.1-uclibc20061004/lib
$ ls -l libc.so.0
lrwxrwxrwx 1 root root 19 2009-04-14 12:09:15 libc.so.0 -> libuClibc-0.9.29.so
```

Enfin, il ne nous reste plus qu'à configurer les options relatives à l'image du système de fichiers racine à générer.

Pour cette dernière, j'ai opté personnellement pour le choix d'une image au format `.cpio` : elle sera facile à extraire, et elle pourra même être directement intégrée dans un kernel en tant qu'`initrd` ou `initramfs` ; les autres options ayant été désactivées :

```
.config - buildroot v2009.05-svn Configuration
-----
Target filesystem options
-----
↑(-)-----
[ ] tar the root filesystem
[*] cpio the root filesystem
    Compression method (none) --->
    () also copy the image to...
[ ] initramfs for initial ramdisk of linux kernel
[ ] romfs root filesystem
-----
<Select>  < Exit >  < Help >
```

Bien entendu, il ne tient qu'à vous de choisir les options qui répondent le plus à vos attentes.

Lançons la génération de l'image, et décompactons-la dans un dossier de travail :

```
$ make
[...]
```

```
$ mkdir -p ~/target ; cd ~/target
$ cpio -i < ~/tools/buildroot/binaries/uclibc/rootfs.arm.cpio
cpio: dev/ttyXX: Cannot mknod: Opération non permise
cpio: dev/null: Cannot mknod: Opération non permise
[...]
```

```
2772 blocks
```

Les erreurs liées à la génération des périphériques spéciaux dans `/dev` sont tout à fait normales, il est nécessaire d'être `root` pour appeler la commande « `mknod` ». Nous n'en aurons pas besoin. À ce stade, nous avons notre environnement quasi-complet dans le dossier `~/target`.

Nous pouvons désormais faire entrer Scratchbox en scène.

## 4. Mise en oeuvre de Scratchbox

Scratchbox fonctionne sur la base de profils, chaque profil ayant un nom défini par l'utilisateur, et permet l'utilisation d'une chaîne de compilation donnée dans un système de fichiers donné.

### . Initialisation d'un profil Scratchbox

Dans la page man de « sb2 », nous avons cet extrait :

```
[...]
CONFIGURATION
  To configure sb2, do something like this:

  mkdir $HOME/buildroot
  cd $HOME/buildroot
  [fetch a rootfs from somewhere and extract it here]
  sb2-init -c qemu-arm TARGET /path/to/cross-compiler/bin/arm-linux-gcc

  To change default scratchbox2 target:

  sb2-config -d another_target

[...]
```

Déplaçons-nous donc à la racine de notre répertoire target, et appelons « sb2-init » avec les paramètres indiqués, pour générer notre profil « arm-uclibc » :

```
$ cd ~/target
$ sb2-init -c qemu-arm arm-uclibc /scratchbox/compilers/arm-gcc4.1-
uclibc20061004/bin/arm-linux-uclibc-gcc
[...]
```

Si tout s'est bien déroulé, on peut constater que nous avons une cible « arm-uclibc » correctement configurée :

```
$ sb2-init | tail -n5
Target arm-uclibc:
configured at 2009-04-22_17:54:40 by user 'hugues', with command
( cd /home/hugues/target;
sb2-init -c qemu-arm arm-uclibc /scratchbox/compilers/arm-gcc4.1-uclibc20061004/
bin/arm-linux-uclibc-gcc )
```

Nous avons désormais un environnement prêt à l'emploi.

### . Utilisation de notre profil

Nous allons prendre, comme base, un classique « Hello World ! », se résumant à quelques lignes :

```
1
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hello World !\n");
7     return 0;
8 }
```

Avec le Makefile suivant :

```
1
2 .PHONY: all clean
3
4 EXEC=helloworld
5
6 SRC=helloworld .c
7 OBJ=$(SRC:.c=.o)
8
9 all: $(OBJ)
10 $(CC) $(OBJ) -o $(EXEC)
11
12 clean:
13 $(RM) $(OBJ)
14 $(RM) $(EXEC)
```

## 4.1. Compilations de référence

Lançons une première compilation sur la machine hôte, sans aucun paramètre :

```
$ make
cc -c -o helloworld.o helloworld.c
cc helloworld.o -o helloworld
$ ./helloworld
Hello World!
```

Nous nous retrouvons avec un exécutable nommé « helloworld ».

Relançons une deuxième compilation, cette fois-ci en spécifiant le cross-compileur que nous avons installé précédemment (après avoir lancé un « make clean ») :

```
$ CC=/scratchbox/compilers/arm-gcc4.1-uclibc20061004/bin/arm-linux-uclibc-gcc
make
/scratchbox/compilers/arm-gcc4.1-uclibc20061004/bin/arm-linux-uclibc-gcc -c -
o helloworld.o helloworld.c
/scratchbox/compilers/arm-gcc4.1-uclibc20061004/bin/arm-linux-uclibc-gcc
helloworld.o -o helloworld
```

```
$ ./helloworld
/scratchbox/tools/bin/misc_runner: /targets/links/scratchbox.config: No such
file or directory
$ file helloworld
helloworld: ELF 32-bit LSB executable, ARM, version 1, dynamically linked (uses
shared libs), not stripped
```

Nous avons bien un exécutable pour ARM, impossible à lancer sur notre machine hôte.  
En revanche, dans un tel cas de figure, Scratchbox ne nous est d'aucune utilité.

## 4.2. Compilation vers une cible non gérée

Étudions un autre cas de figure : on reprend les mêmes sources que précédemment, mais dans le Makefile nous allons forcer l'appel au compilateur gcc hôte :

```
[...]
9 all: $(OBJ)
10 gcc $(OBJ) -o $(EXEC)
[...]
```

Il est certes évident qu'un tel Makefile n'est pas correctement écrit, mais il va nous permettre d'illustrer tout l'intérêt de Scratchbox.

Lançons la compilation en spécifiant notre cross-compilateur, après un « make clean » :

```
$ CC=/scratchbox/compilers/arm-gcc4.1-uclibc20061004/bin/arm-linux-uclibc-gcc
make
/scratchbox/compilers/arm-gcc4.1-uclibc20061004/bin/arm-linux-uclibc-gcc -c -
o helloworld.o helloworld.c
gcc helloworld.o -o helloworld
/usr/bin/ld: helloworld.o: Relocations in generic ELF (EM: 40)
helloworld.o: could not read symbols: File in wrong format
collect2: ld returned 1 exit status
make: *** [all] Erreur 1
```

À la première ligne, on voit que notre cross-compilateur a bien été appelé, grâce aux règles automatiques de make (ici, la règle « .c.o »), qui fait appel à la variable \$(CC) pour appeler le compilateur.

En revanche, pour la compilation du binaire final, nous appelons directement gcc. Et là, ça coince.

Regardons maintenant ce qui se passe si on relance la compilation à l'aide de Scratchbox (après avoir effectué un « make clean ») :

```
$ sb2 make
cc -c -o helloworld.o helloworld.c
gcc helloworld.o -o helloworld
```

Tout de suite, ça passe beaucoup mieux...

Vérifions le binaire généré :

```
$ file helloworld
helloworld: ELF 32-bit LSB executable, ARM, version 1, dynamically linked (uses
shared libs), not stripped
$ ./helloworld
/scratchbox/tools/bin/misc_runner: /targets/links/scratchbox.config: No such
file or directory
$ sb2 ./helloworld
Hello World!
```

Voilà qui est très intéressant !

À partir d'un Makefile ne contenant aucune indication de cross-compilation, nous avons réussi à générer un binaire pour ARM.

De plus, Scratchbox utilisant qemu, nous pouvons même exécuter les binaires compilés pour vérifier qu'ils fonctionnent bien ! Et ce, sans avoir eu besoin d'apprendre à utiliser QEMU.

Nous avons donc réussi, à partir d'un Makefile bancal ne gérant que le cas classique de compilation pour l'hôte courant, à générer un binaire pour une autre architecture processeur. Nous avons également réussi à l'exécuter directement. Il nous manque cependant une dernière chose à voir : la gestion de la cible « install ».

### 4.3. Installation dans le système de fichiers cible

Rajoutons une règle « install » à notre Makefile :

```
1
2 .PHONY: all clean install
[...]
```

15

```
16 install:
17   mkdir -p /usr/bin
18   cp $(EXEC) /usr/bin
```

Certes, c'est tout aussi rigide que notre règle « all », mais c'est clairement voulu : rien n'est prévu ici pour effectuer l'installation dans une autre arborescence que celle de notre système de fichiers racine. Un Makefile correctement écrit comporterait l'utilisation d'une variable « \$ (DESTROOT) », en plus de « \$(PREFIX) ».

Étudions la réaction de Scratchbox :

```
$ sb2 make install
mkdir -p /usr/bin
cp helloworld /usr/bin
cp: ne peut créer le fichier régulier `/usr/bin/helloworld': Permission non
accordée
make: *** [install] Erreur 1
```

Sans surprise, il a essayé de copier notre binaire directement dans le dossier /usr/bin de notre système courant. À moins d'être root, nous n'en avons pas les droits, fort heureusement.



C'est là qu'intervient une notion très intéressante de Scratchbox : les modes. En l'occurrence, ici, ce qui nous intéresse, c'est le mode « install » :

```
$ sb2 -m install make install
mkdir -p /usr/bin
cp helloworld /usr/bin
```

Aucune plainte... Vérifions donc le contenu de notre dossier « ~/target/usr/bin » :

```
$ ls ~/target/usr/bin/helloworld
/home/hugues/target/usr/bin/helloworld
```

Notre exécutable a été correctement installé dans le système de fichiers cible, sans aucune mention particulière dans notre Makefile !

Nous avons appris deux choses :

1. De base, scratchbox permet d'intercepter tout appel aux binaires locaux pour les remplacer, si besoin, par un binaire de cross-compilation
2. Dans le mode « install », Scratchbox intercepte tout paramètre référençant un dossier système et le remplace par son équivalent dans notre système de fichiers cible.
3. Dans aucun cas de figure nous n'avons eu besoin des droits root.

Nous voilà parés pour attaquer un logiciel plus complexe : Kerberos 5.

## 5. Cas de figure : Kerberos 5

La compilation et l'installation de Kerberos 5 n'a pas été prévue pour une autre cible que le système hôte. Ce paquetage logiciel étant assez imposant, il constitue un excellent exercice d'application pour Scratchbox.

Récupérons les sources de l'implémentation Libre MIT de Kerberos 5 sur le site <<http://web.mit.edu/kerberos/>> :

```
$ mkdir -p ~/kerberos ; cd ~/kerberos
$ wget http://web.mit.edu/kerberos/dist/krb5/1.6/krb5-1.6.3-signed.tar
[...]

$ tar xvf krb5-1.6.3.tar
krb5-1.6.3.tar.gz
krb5-1.6.3.tar.gz.asc
$ tar xvzf krb5-1.6.3.tar.gz
[...]
```

Nous nous retrouvons avec un dossier « krb5-1.6.3 », contenant deux dossiers : « doc/ » et « src/ », et un fichier README.

### . Cross-compilation « témoin »

Dans le dossier « krb5-1.6.3/src », nous trouvons un script « configure ». Testons donc la configuration avec l'option standard « --host » :

```
$ PATH=$PATH:/scratchbox/compilers/arm-gcc4.1-uclibc20061004/bin ./configure
--prefix=/usr/local --host=arm-linux-uclibc
[...]
checking for constructor/destructor attribute support... configure: error:
Cannot test for constructor/destructor support when cross compiling
```

Ça coince méchamment, Kerberos n'a visiblement pas été prévu pour ça. C'est dommage, car c'était bien parti...

La plupart des scripts configure permettent l'utilisation d'une variable « CROSS » ou « CROSS\_COMPILE », contenant une chaîne de caractères à positionner avant l'appel à tout outil de développement : gcc, ld, objdump, strip, etc... Cette variable est ensuite concaténée au nom de l'outil à appeler : \$(CROSS)gcc, \$(CROSS)ld, \$(CROSS)ar, etc..., quand cela est nécessaire.

Vérifions si le script « configure » gère cette variable :

```
$ make distclean
$ grep -r CROSS
./lib/krb5/krb/rd_req_dec.c:#elif defined(_NO_CROSS_REALM)
./lib/krb4/g_pw_in_tkt.c:#ifdef CROSSMSDOS
./lib/krb4/g_pw_in_tkt.c:#ifdef CROSSMSDOS
```

On ne retrouve strictement aucune référence à « CROSS » (les résultats ci-dessus ne concordent pas)...

Une première « solution » rapide consisterait à repasser derrière tous les Makefiles :

```
$ find -iname Makefile | wc -l
106
```

Soit un peu plus d'une centaine de Makefiles à corriger.  
À y réfléchir, non, ce n'est peut être pas une très bonne solution.

Après information, il est possible de cross-compiler Kerberos5, avec la ligne de commande qui suit :

```
$ cross_compiling=${cross_compiling=yes,yes} \  
  {krb5_cv_attr_constructor_destructor=yes,yes} \  
  ac_cv_func_regcomp=${ac_cv_func_regcomp=yes,yes} \  
  ac_cv_printf_positional=${ac_cv_printf_positional=yes,yes} \  
  ac_cv_file_etc_environment=${ac_cv_file_etc_environment=no,no} \  
  ac_cv_file_etc_TIMEZONE=${ac_cv_file_etc_TIMEZONE=no,no} \  
  enable_thread_support=${enable_thread_support=no,no} \  
  enable_thread=${enable_thread_support=no,no} \  
  KRB5_AC_ENABLE_THREADS=${KRB5_AC_ENABLE_THREADS=no,no} ./configure \  
  --enable-shared --without-krb4 --without-tcl \  
  --host=arm-linux --prefix=/opt/media/LxNETES3.2/LxNETES \  
  --with-libcurses=~/target/usr/lib/libcurses.a
```

Rien que ça... Il est facile d'imaginer le temps perdu si Kerberos5 n'est qu'un paquetage parmi plein d'autres posant les mêmes soucis, attendant leur intégration dans un projet bien plus vaste.

Regardons donc Scratchbox.

## . Compilation à l'aide de Scratchbox

Lançons la configuration depuis un shell Scratchbox :

```
$ sb2  
[SB2 simple arm-uclibc] hugues@groumpf src $ ./configure --prefix=/usr
```

Ça échoue lors de la détection de la libcurses.

Téléchargeons-la, et appliquons le même principe :

```
$ sb2 ./configure --prefix=/usr  
[...]  
$ sb2 make  
[...]  
$ sb2 -m install make install  
[...]
```

Un coup d'oeil à notre ~/target nous indique que la libcurses a été correctement installée<sup>[2]</sup> :



```
$ cd ~/target
$ ls -la usr/lib/*ncurses*
-rw-r--r-- 1 hugues users 406678 2009-mai-06 11:32:08 usr/lib/libncurses.a
-rw-r--r-- 1 hugues users 130670 2009-mai-06 11:32:26 usr/lib/libncurses++.a
-rw-r--r-- 1 hugues users 2107630 2009-mai-06 11:32:08 usr/lib/libncurses_g.a
```

Reprenons la configuration de Kerberos5, puis la compilation :

```
$ sb2
[SB2 simple arm-uclibc] hugues@groupmf src $ ./configure --prefix=/usr
[...]

[SB2 simple arm-uclibc] hugues@groupmf src $ make
```

Quelques erreurs de compilation au niveau des appels système « rresvports », « ruserok », et « ruserpass » sont contournées, pour l'exemple, et étant simplement commentées dans le code<sup>[3]</sup>.

Relançons la compilation.

La génération s'est parfaitement bien déroulée. Lançons l'installation :

```
$ sb2 -m install
[SB2 install arm-uclibc] hugues@groupmf src $ make install
```

Et c'est terminé. Au final, Scratchbox nous aura au moins permis la compilation de la librairie Kerberos5 pour une architecture cible différente de notre machine hôte.

## 6. Mesures

Voici un récapitulatif des différentes mesures effectuées :

Projet	Commande	Mesure	Ratio
HelloWorld	make	0,07s	-
	make CROSS=...	0,05s	0,7x
	sb2 make	0,50s	7x/10x

Pour HelloWorld, la simplicité du code par rapport au travail effectué par scratchbox explique l'écart de 7x le temps de compilation de base, et 10x par rapport au temps de cross-compilation.

Projet	Commande	Mesure	Ratio
libncurses	./configure make	16s 2min 40s	- -
	./configure --host=arm-linux-uclibc make	12s 2min 30s	0,75x 0,9x
	sb2 ./configure sb2 make	34s 3min	2x 1,10x

Dans le cas de la libncurses, nous avons une base de comparaison intéressante car la quantité de code à compiler est plus proche de ce qu'on pourra attendre d'un module industriel.

Les temps de configuration et de compilation en natif et en cross-compilation sont très proches. En revanche, la configuration avec sb2 est 2x plus lente, tandis que la compilation est un peu plus longue qu'en natif, ce qui reste très honorable.

Il conviendra toutefois, autant que possible, préférer la cross-compilation directe.

Projet	Commande	Mesure	Ratio
Kerberos5	./configure make	1min 15s 3min 30s	- -
	sb2 ./configure sb2 make	3 min 5 min 30s	2,5x 1,5x

Dans le cas de Kerberos5, nous avons un ratio qui passe de 2,5x pour la configuration, à 1,5x le temps de compilation native. Au total, 8 minutes 30 secondes de compilation avec Scratchbox contre 4 minutes 45 secondes pour la compilation native, soit un ratio de près du double. C'est beaucoup.

Mais comme nous l'avons vu, la difficulté de configuration d'un tel paquetage pour la cross-compilation risque de dépasser largement ces quelques minutes de compilation supplémentaires.

En conclusion, Scratchbox reste une solution viable pour mettre en œuvre très rapidement, au détriment d'un temps de génération approchant du double, une compilation croisée pour un projet qui n'a pas été conçu pour. Dans la majorité des cas, cependant, il serait préférable d'utiliser directement les facilités de cross-compilation proposées.



Prise en main de  
Scratchbox

Version: 1.01

Auteur: Hugues HIEGEL

## 7. Références

1. Scratchbox2, sur freedesktop : <<http://freedesktop.org/wiki/Software/sbox2>>
2. Scratchbox, site officiel (obsolète) : <<http://www.scratchbox.org/>>
3. Buildroot, site officiel : <<http://buildroot.uclibc.org/>>
4. Busybox, site officiel : <<http://www.busybox.net/>>
5. Kerberos5, site officiel : <<http://web.mit.edu/kerberos/>>

- 1 **environnement chrooté** : utilisation de la commande «chroot» pour modifier temporairement la cible du répertoire racine «/». Toutes les références au système de fichiers, comme /etc, /bin, /lib, etc... prendront leur racine au niveau du dossier chrooté. S'utilise généralement à des fins de sécurité : « service chrooté », « compte utilisateur chrooté », mais peut aussi s'utiliser dans le cadre de tests ou, comme ici, de développement.
- 2 En réalité, l'installation s'est mal déroulée avec le compilateur de scratchbox. Les essais ont été effectués à nouveau avec un compilateur généré avec buildroot, plus récent, et nous sommes arrivés à des résultats bien plus concluants. Il faut noter que le choix du compilateur, de manière générale, n'est pas le fruit du hasard, et qu'ici nous nous concentrons plutôt sur la mise en oeuvre de scratchbox que sur la cross-compilation elle même.
- 3 Pour bien faire, il faudrait trouver les bonnes versions de la libc/uClibc, mais d'une part, ce n'est pas notre objectif, et d'autre part, les appels à rsh/rlogin sont quelque peu dépréciés..