Making a Scratchbox devkit

Valtteri Rahkonen

valtteri.rahkonen@movial.fi

Making a Scratchbox devkit

by Valtteri Rahkonen

Revision history

Version:	Author:	Description:
2005-01-19	Rahkonen	Changed example devkit to use GAR devkit template and Fedora Core 3
2005-01-13	Rahkonen	Added support for Scratchbox 1.0 and devkit template
2004-05-11	Rahkonen	Fixed example devkit section
2004-05-10	Rahkonen	Added example devkit
2004-04-29	Rahkonen	Initial version

Table of Contents

Chapter 1. Introduction

Scratchbox devkits are used to build distribution specific packages inside Scratchbox. Devkits purpose is to provide a set of tools that can be executed on host instead of target device and thus it reduces time that is used to build packages. Usually devkit also provides a set of libraries that is compiled for target device so that configure scripts can find correct libraries and link programs correctly. This way package building is much more faster than doing building process on slow target device or emulated on emulator.

Each distribution usually contains its own specific tools and libraries and thus it needs its own devkit. Devkits should be divided to build and helper tools (actual devkit) and optionally libraries compiled for target platform to provide build dependencies for target platform packages.

Scratchbox development team maintains and distributes Debian specific devkit. Other devkits are developed and maintained by community and currently at least Slackware has its own devkit.

Purpose of this document is to provide information for devkit developers on how devkits are build inside Scratchbox and for maintainers how devkits should be packaged for that they will provide full interoperability with Scratchbox. Focus is on build tools and not in building libraries for target platform (libraries can be built with devkit afterwards).

Chapter 2. Preliminary checklist

First step in creating a devkits requires some reading of distributions documentation.

1. Find a distributions package building tool.
2. Check build tools dependencies and follow dependencies until there are no new dependencies.
3. Check which tools and libraries are already provided by Scratchbox. For example Scratchbox core provides GNU Autotools and doctools package contains many document generation related tools.
4. After list of required packages is finished obtain sources for these new tools and copy these sources to place that can be accessed from Scratchbox (for example systems **/tmp** directory).
5. Find out what kind of environment distribution uses because tools may require environment options that are distribution specific.
6. Find out what kind of directories and configure files distribution packaging tools uses. You may need to patch tools if distributions configuring options do not provide required flexibility to configure tools (for example specific target platform cannot differ from packaging tools or different package database location and so on). These may differ greatly between distributions.

After these preliminary steps we can next start to build devkit.

Chapter 3. Compiling tools inside Scratchbox

3.1. Setting up the build system

Devkit build tools and its dependency libraries are compiled to a host target. For compiling these tools you need to select **HOST** target in Scratchbox (if not already selected) with command:

```
[sbox-TARGET: ~] > sb-conf select HOST
```

Scratchbox team provides and maintains a devkit template package for developers that want to build and use their own tools inside their Scratchbox. Template is basically a GAR build system tree (6) that is modified to work inside Scratchbox. Devkit template package can be found from Scratchbox download area (2). After you have obtained the devkit template it can be extracted with command:

```
[sbox-TARGET: ~] > tar zxvf sb-devkit-template.tar.gz
```

Tools should be installed to devkits directory (**/scratchbox/devkits/<devkitname>**). This way Scratchbox target can be created using devkit (see [1] for target creation information). Also devkit tools should be configured in a way that they are linked only to devkits library directory (**/scratchbox/devkits/<devkitname>/lib**), to host compilers library directory (**/scratchbox/host_shared/lib**) or to Scratchbox tools library directory (**/scratchbox/tools/lib**). If some tools are linked to normal library directories (**/lib** or **/usr/lib**) it will probably break because normal library directories will contain libraries compiled for target platform.

Good thing is that the devkit template will take care of installing and linking your devkit correctly. To achieve this you need to change **DEVKIT** variable from main level's makefile and **SBOX_PREFIX** in devkit directory's category.mk file to correspond your devkits properties. After those changes you should be able to build, install and package your devkit.

However users do not have permissions to write to **/scracthbox/devkit** directory by default. As result of this install directory should be created before installing any tools with following command (needs to be executed as root):

```
# mkdir /scratchbox/devkits/<devkitname>
```

And user should be allowed to write to this directory with following command (needs to executed as root):

```
# chown <username>.<username> /scratchbox/devkits/<devkitname>
```

After above commands are executed you can add tools to devkit template GAR-tree and they can be build and installed.

3.2. Building tools

Adding tools to devkit template GAR-tree is quite straight forward and it can done with following steps:

1. Create a subdirectory for your new tool to devkit subdirectory.
2. Create a makefile for your tool and place it your tools subdirectory.

   Note: You can use the example tool makefile as base of your own tools makefile. Example makefile is found in **devkit/example** directory.

3. Fix at least **GARNAME** and **GARVERSION** variables to correspond your tool in your tools makefile. You should also fine tune **CONFIGURE_ARGS** variable if your tool needs to have some special configure arguments.
4. Place your tool source packages to a place where your GAR build system can access. One option is to place source package to a **files** directory located under your tools directory.

   Note: If you want to use HTTP or FTP to access your source packages you may need to modify **file_locations.mk** or **gar.conf.mk** files to have correct server in **MASTER_SITES** variable.

5. Create source packages checksums with following command:

   ```
   make makesums
   ```

6. Build and install your tool with command:

   ```
   make install
   ```

   Note: You might need to fine tune your makefile so that it will build correctly. Basic rile is that **CONFIGURE_ARGS** contains configure options, **BUILD_ARGS** has variables that affects when tool is build and **INSTALL_ARGS** contains variables that are used when tool is installed.

7. Modify makefile that is used to automatically build all tools from main level (located at **meta/devkit** directory) and add your tool there. Follow instruction in makefile to add them correctly.

These steps should be repeated for each tool that is going to be on your devkit. After you have added all your tools you can proceed to doing post-install steps in Section 3.3.

3.3. After install

Devkit should setup environment in a way that devkit uses normal prefix (i.e. root / directory) to install packages and store package databases. To achieve this following steps should done:

- Store necessary target environment options to devkits subdirectory **etc** in **environment** file.
- Store (and modify if necessary) configure files to their correct places under devkits directory.
- Create **target_setup.sh** script that will copy the above files to target. This script should also create necessary directories under target in order to devkits tools to work.
- Place **target_setup.sh** script to devkits directory. It will be executed when target that uses this devkit is created.
- You can put environment and target_setup.sh to your devkit's build tree. You just need to create a subdirectory and create a makefile so that it will install files to correct places.

3.4. Making a devkit package

Packaging a devkit is fairly simple when using our devkit template. In template main directory you can package the devkit with following command (should be issued as root outside Scratchbox):

```
make tarball deb rpm
```

Packages will be created to devkit templates main directory. After packages are created they should be installed and tested to see that it works properly. Later on we assume that we built a tarball and we will use that in our examples.

3.5. Installing the devkit package

If same Scratchbox installation is used to develop and to install the devkit's package, then old from sources installed version should be removed or moved to a safe place. After the old version of devkit is cleaned up new can be installed with following command (needs to be done as root in root / directory):

```
# tar zxvf scratchbox-<devkitname>-<version>.tar.gz
```

> **Note:** You can also install the devkit package from Debian deb or rpm package if your distribution supports them.

After extracting devkit you should test your new devkit (see Chapter 4).

Chapter 4. Testing devkit

Best way to test devkit is to create a new target, build and install some architecture dependant packages (i.e. packages that contains executables or libraries) with it. Following steps should be done in order to verify that devkit is working as expected:

1. Create a new target that uses target platform compiler and a newly installed devkit (see 1).
2. Select new target with following command:

```
[sbox-HOST: ~] > sb-conf select <targetname>
```

3. Obtain source package for devkit's distribution. Tested package should be selected in a way that it does not depend on anything that is not installed on Scratchbox. Good example is ncurses because it usually (might change between distributions) depends only on C library and build tools that are provided by target compiler and your devkit devkit. Information about package dependencies and package downloads can usually be found from distributions website.
4. Use devkit's package building tool to build package.
5. Install newly created package with package manager that devkit provides.
6. Verify that package was installed to correct place (under target root directory /).
7. Use **file** command to verify that package executables and libraries are compiled to right target.
8. Use **ldd** command to verify that package executables are linked correctly.
9. Execute package's executables that they function properly. Verify from CPU-transparency log (**/tmp/cputransp_<username>.log**) that executed programs were emulated or executed on target device (depending on whether you are using emulator or sbrsh).

If problems are encountered at some point they should be examined to see what went wrong and then fixed. Luckily most problems relate to environment and target directory structure and hopefully fixing environment or setup-target.sh will be enough.

After simple packages builds and installs correctly more complicated packages should be build to verify that devkit works properly. They should be build and tested with similar steps than above. If libraries fulfilling build dependencies for GUI programs (for example GTK2 or Xlib) and some GUI programs using these libraries (for example gtk-demo) builds up correctly it is fair to say that devkit works properly.

> **Note:** You can provide these libraries and programs as a separate rootstrap package. Using rootstrap package can significantly speed up developers work for they do not need to build libraries themselves. For example Debian rootstrap provides target development libraries for developers in addition to Debian devkit.

Chapter 5. Example devkit - Fedora devkit

5.1. RPM dependencies

Fedora uses RPM packaging tool that has some Fedora specific patches. For obtaining required packages and resolving their dependencies rpmfind website [5] was used. Fedora's RPM package depends on following packages:

- beecrypt-devel
- bzip2
- elfutils-libelf and elfutils-devel
- python-devel
- readline
- zlib-devel

Python is already provided by Scratchbox, so it is not necessary.

5.2. Converting packages to tarballs

Here only the beecrypt package is converted. Converting other packages follows same steps.

1. Download beecrypt source package (beecrypt-3.1.0-3.src.rpm) from rpmfind website.
2. Use rpm2cpio and cpio commands to extract source tarball and specs file:

```
# rpm2cpio beecrypt-3.1.0-3.src.rpm | cpio -i
```

3. Copy source tarball to a place that can be accessed from Scratchbox (for example **/tmp** directory).

Other packages need to be extracted the same way.

5.3. Compiling and installing Fedora tools

After sources are obtained they can be installed by following steps:

1. Tools and libraries should be compiled in a way that they are installed to **/scratchbox/devkit/fedora** directory. Before installing any packages that directory needs to be created (as root):

```
# mkdir /scratchbox/devkits/fedora
```

At this point user does not have permissions to write on devkits directory. User needs these permissions because compiling and installation is done inside Scratchbox. Right permissions can be applied with following command:

```
# chown <username>.<username> /scratchbox/devkits/fedora
```

2. Obtain devkit template package (from 2) and unpack it.
3. Start Scratchbox and select 'HOST' target:

```
$ /scratchbox/login
[sbox-TARGET: ~] > sb-conf select HOST
```

4. Change "DEVKIT = template" to "DEVKIT = fedora" in devkit template's main makefile and "SBOX_PREFIX = /scratchbox/devkits/template" to "SBOX_PREFIX = /scratchbox/devkits/fedora" in category.mk file located devkit subdirectory.
5. Necessary directory structure can be created with following commands:

```
[sbox-HOST: ~/sb-devkit-template/devkit] > mkdir beecrypt
[sbox-HOST: ~/sb-devkit-template/devkit] > mkdir bzip2
[sbox-HOST: ~/sb-devkit-template/devkit] > mkdir elfutils
[sbox-HOST: ~/sb-devkit-template/devkit] > mkdir readline
[sbox-HOST: ~/sb-devkit-template/devkit] > mkdir rpm
[sbox-HOST: ~/sb-devkit-template/devkit] > mkdir zlib
[sbox-HOST: ~/sb-devkit-template/devkit] > mkdir environment
```

6. For each tool create files directory and copy makefile to tools directory (here it is done for beecrypt):

```
[sbox-HOST: ~/sb-devkit-template/devkit] > cd beecrypt
[sbox-HOST: ~/sb-devkit-template/devkit/beecrypt] > cp ../example/Makefile .
[sbox-HOST: ~/sb-devkit-template/devkit/beecrypt] > mkdir files
[sbox-HOST: ~/sb-devkit-template/devkit/beecrypt] > cp /tmp/beecrypt-3.1.0.tar.gz files/
```

You need to apply above steps to rest of the tools also.

7. Edit tools makefiles and change the GARNAME and GARVERSION variables to correspond your toolname and tools version. Default options should be fine for beecrypt, bzip2 and readline. For zlib you need to add following to BUILD_ARGS variable in makefile:

```
BUILD_ARGS = AR="host-ar -r"
```

With elfutils you need to remove strip (Scratchbox already provides strip), so you need to add following post-install target in makefile to remove strip:

```
post-install:
        rm $(prefix)/bin/strip
        $(MAKECOOKIE)
```

Rpm package is little bit more tricky. Rpm needs more options to be builded correctly and also its macros needs to be edited after install. With following CONFIGURE_ARGS, BUILD_ARGS and INSTALL_ARGS you can pass correct parameters to rpm:

```
CONFIGURE_ARGS =  CFLAGS="$(CFLAGS) -I$(prefix)/include
-I$(prefix)/include/beecrypt -I$(prefix)/include/readline
-I/scratchbox/compilers/host-gcc/include -I/scratchbox/tools/include
-I/scratchbox/tools/include/python2.3" --prefix=$(prefix)
--without-selinux --enable-posixmutexes --without-javaglue
BUILD_ARGS = LDFLAGS="$(LDFLAGS) -Wl,-rpath -Wl,$(prefix)/lib -Wl,-rpath
-Wl,/scratchbox/tools/lib -Wl,-rpath -Wl,/scratchbox/host_shared/lib
-Wl,-rpath -Wl,/scratchbox/host_shared/lib/python2.3"
INSTALL_ARGS = pylibdir=$(prefix)/lib
```

Also you need to run the configure script manually, so you need to specify CONFIGURE_SCRIPT like this:

```
CONFIGURE_SCRITPS = manual
```

You will also need a target to take care of manual configuring:

```
configure-manual:
        cd $(WORKSRC) && ./configure $(CONFIGURE_ARGS)
        $(MAKECOOKIE)
```

After rpm is installed all its macros will point to devkits directory directory. This will cause problems when installing and building packages, because normal users wont have permissions to write to the devkit directory. These relevant macros will need to be modified so that they will point to target instead of the devkits directory. This can be achieved with following post-install target:

```
post-install:
        cat /scratchbox/devkits/fedora/lib/rpm/macros | sed
"s,%_prefix\t.*,%_prefix\t\t\t/usr,g" > /tmp/macros.tmp &&
cat /tmp/macros.tmp | sed "s,%_usr\t.*,%_usr\t\t\t/usr,g">
/tmp/macros.tmp2 && cat /tmp/macros.tmp2 | sed
"s,%_var\t.*,%_var\t\t\t/var,g" > /scratchbox/devkits/fedora/lib/rpm/macros
        rm /tmp/macros.*
        cat /scratchbox/devkits/fedora/lib/rpm/i386-linux/macros | sed
"s,%_prefix\t.*,%_prefix\t\t\t/usr,g" > /tmp/macros.tmp
        mv /tmp/macros.tmp /scratchbox/devkits/fedora/lib/rpm/i386-linux/macros
        cat /scratchbox/devkits/fedora/lib/rpm/i686-linux/macros | sed
"s,%_prefix\t.*,%_prefix\t\t\t/usr,g" > /tmp/macros.tmp
        mv /tmp/macros.tmp /scratchbox/devkits/fedora/lib/rpm/i686-linux/macros
        $(MAKECOOKIE)
```

8. You will also need to create environment package that will contain target_setup.sh and environment file that will be used to set up target correctly. We already created the environment directory and copied the makefile there, so you will need to create following environment file to files directory:

```
# -*- sh -*-

export LC_ALL=C
export TMPDIR=/var/tmp

target=$(grep ^SBOX_CPU= /targets/links/scratchbox.config | cut -d= -f2)
if [ "$target" = "arm" ]; then
        export SBOX_UNAME_MACHINE=arm
fi
```

   In environment you will export correct platform uname option if you have created an ARM target. You will also need to create following target_setup.sh script to files directory that will setup the new target correctly:

```
#!/scratchbox/tools/bin/bash

mkdir -p /usr/src/redhat/BUILD
mkdir -p /usr/src/redhat/RPMS
mkdir -p /usr/src/redhat/SRPMS
mkdir -p /usr/src/redhat/SOURCES
mkdir -p /usr/src/redhat/SPECS
mkdir -p /var/tmp
mkdir -p /var/lock/rpm
/scratchbox/devkits/fedora/bin/rpmdb --initdb
```

   This script will create necessary directories to your new target and it will also initialize the rpm package database. In environments makefile you do not actually need any configure or build targets you just need to specify a manual install target and define that target as following:

```
install-environment:
        mkdir -p $(prefix)/etc
        cp files/environment $(prefix)/etc
        cp files/target_setup.sh $(prefix)
        $(MAKECOOKIE)
```

   After setting up the environment install process you are ready to proceed to next step.

9. Add tools and environment to "meta/devkit" directory's makefile.
10. Compile and install your packages (in template devkits main directory):

```
[sbox-HOST: ~/sb-devkit-template] > make build
```

11. Create a package of your new devkit (here we will create a tarball):

```
[sbox-HOST: ~/sb-devkit-template] > make tarball
```

5.4. Creating a target for Fedora devkit

For using Fedora devkit a target must be created and Fedora devkit should be selected from list when new target is created (see [1] how to create a target). In our example we will create a ARM target that uses our fedora devkit with following commands:

1. Create a new target:

```
[sbox-HOST: ~] > sb-conf setup FARM -c arm-gcc-3.3.4-glibc-2.3.2 -d
fedora -t qemu-arm
```

2. Install necessary files to target:

```
[sbox-HOST: ~] > sb-conf install FARM -d -e -c -f
```

3. Select your target:

```
[sbox-HOST: ~] > sb-conf select FARM
```

After following steps are done Fedora devkit is ready to be tested.

5.5. Testing Fedora tools

Fedora tools can be tested by building and installing some Fedora packages. Ncurses is nice package in a way that it does not depend on any higher libraries, just C library. However ncurses needs sharutils for building itself. Thus we will use these to test our Fedora devkit. Devkit can be tested with following commands:

1. Obtain Fedora ncurses and sharutils source package from rpmfind website.
2. Build sharutils with following command:

```
[sbox-FARM: ~] > rpmbuild --rebuild --nodeps sharutils-4.2.1-22.src.rpm
```

Option **--nodeps** is required because our Fedora devkit does not provide tools as a rpm packages (this should be done with dummy package that provides all Scratchbox tools and Fedora devkits tools).

3. Install sharutils with following command:

```
[sbox-FARM: ~] > rpm -i --nodeps
/usr/src/redhat/RPMS/arm/sharutils-4.2.1-22.arm.rpm
```

You will see that installing the sharutils package will complain about missing "/sbin/install-info".

4. Build ncurses with following command:

```
[sbox-FARM: ~] > rpmbuild --rebuild --nodeps ncurses-5.4-13.src.rpm
```

5. Install ncurses package with following command:

```
[sbox-FARM: ~] > rpm -i --nodeps
/usr/src/redhat/RPMS/arm/ncurses-5.4-13.arm.rpm
```

Again you will see that installing the sharutils package will complain about missing "/sbin/install-info", but your packages should work correctly.

6. To verify that ncurses package was installed execute command:

```
[sbox-FARM: ~] > rpm -q -a

sharutils-4.2.1-22
ncurses-5.4-13
```

Output should state that sharutils version 4.2.1-22 and ncurses version 5.4-13 are installed.

7. Ncurses package installs **tic** program that can be used for testing. It can be verified to be ARM target binary with following command:

```
[sbox-FARM: ~] > file /usr/bin/tic
/usr/bin/tic: ELF 32-bit LSB executable, ARM, version 1 (ARM), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), not stripped
```

8. After verifying that **tic** is a ARM target executable it can be run with following command:

```
[sbox-FARM: ~] > tic -V
ncurses 5.4.20040724
```

9. Ncurses package can be removed with command:

```
[sbox-FARM: ~] > rpm -e ncurses
```

As shown above Fedora devkit can build and install packages for ARM target platform. However this is not complete devkit and in next section we discuss shortly what is missing.

5.6. Things to do with Fedora devkit

This appendix describes proof of concept devkit creation. This example devkit is far from release version and at least following things should be fixed:

- You could get rid of the complain about install-info. You need to select doctools devkit also and then export the SBOX_REDIRECT_FROM_DIRS so that it will contain the /sbin directory. This should be done in your devkits environment file.
- More Fedora related tools (for example apt-rpm).
- Dummy package that provides build tools (remove need for **--nodeps** option).
- Library package to provide necessary libraries to build packages for Fedora. This is actually an optional feature because this devkit can be used to build necessary libraries. However package that provides most used libraries and tools will speed up developers work.

Bibliography

[1] *Installing Scratchbox (http://www.scratchbox.org/documentation/docbook/installdoc.html)* , Valtteri Rahkonen.

[2] *Scratchbox's devkit template download (http://www.scratchbox.org/donwload/)* .

[3] *Debian GNU/Linux website (http://www.debian.org/)* .

[4] *RPM Package Manager homepage (http://www.rpm.org/)* .

[5] *Rpmfind website (http://www.rpmfind.net/)* .

[6] *GAR Archtecture (http://www.lnx-bbc.org/garchitecture.html)* .