

What else can you do with Android?

Embedded Linux Conference Europe 2010

Date: Tuesday 26th October

Location: DeVere University of Arms Hotel, Cambridge

Room: Churchill Suite

Presenter: Chris Simmonds, chis@2net.co.uk

Copyright © 2010, 2net Limited

1. Making use of Android

NOTE: I tested these instructions using Ubuntu 10.04 Desktop i386. They should work on other Linux configurations too, with maybe a little tweaking.

In this section you will install the Android SDK, create a definition of a virtual device and run the emulator

Install JDK

You will need a version of the Sun Java Development Kit. *Other versions of Java will not work.*

There is a choice here: if you only want to install and use the SDK for writing Android apps then Java 6 is the best choice. But, if you want to build the full Android run-time, as you will if you are porting to custom hardware, then you will need Java 5.

To install Java 6 on Ubuntu just type

```
sudo apt-get install sun-java6-jdk
```

Note: requires that you go to Synaptic->Settings->Repositories and on the Other software tab, enable <http://archive.canonical.com/ubuntu> and then reload the repositories.

To install Java 5...

Download a suitable JDK from java.sun.com, e.g. jdk-5u25-linux-i586.bin, then:

```
cd ~  
sh Downloads/jdk-5u25-linux-i586.bin
```

Inside Android lab notes

Set these shell variables

```
export JAVA_HOME=$HOME/jdk1.5.0_22
PATH=$JAVA_HOME/bin:$PATH
```

Hint: add the lines above to your `.profile` so they are set correctly each time you log in.

Install the Android SDK

Download the SDK "starter pack" from <http://developer.android.com/sdk/index.html>. At the time of writing the current version was r06. Then:

```
cd ~
tar xzf Downloads/android-sdk_r06-linux_86.tgz
cd android-sdk-linux_86

PATH=$HOME/android-sdk-linux_86/tools:$PATH
```

The starter pack contains little besides the tools directory; you need to install at least one SDK platform. There is one platform for each Android release. In this case you are going to install the 2.2 "Froyo" platform.

Run "android" with no parameters so it starts in GUI mode. From the "Available Packages" list, select "SDK Platform Android 2.2, API 8, revision2" & install.

Create an AVD

Since you intend to use the emulator, you need to create a definition of the target device you are going to emulate. This is called an AVD: Android Virtual Device. You can do it using the graphical user interface the *android* tool provides: select "Virtual Devices" and click the "New..." button.

Or you can do it from the command line with

```
android create avd -n AVDtest -t 1
```

where

-n gives the AVD a name and -t selects the target platform. Run "android list" to get a list of target platform ids.

Use "android list avd" for a list of them

If you are curious, the AVD is created in `$HOME/.android/avd` and consists of a configuration file and a place for the emulator to keep user data.

Run the emulator

```
emulator -avd AVDtest
```

Note: the *second* time you boot the device the screen will be locked. You unlock it by pressing the menu key.

Other things you can do

Experiment with AVD settings

When you create an AVD you can specify a skin. The default is HVGA which is 320x480. Other options include

- QVGA (240x320, low density, small screen)
- HVGA (320x480, medium density, normal screen)
- WVGA800 (480x800, high density, normal screen)
- WVGA854 (480x854 high density, normal screen)

You can specify other skins by name, e.g. WVGA (800 x 480) or by resolution.

You can specify an SD card size and image file...

Start the emulator with `-shell` to get a root shell.

Start the emulator with `-show-kernel` to see the kernel messages as the emulator boots.

Install Eclipse and the ADT

I find that the easiest way to deploy Eclipse is to extract the archive to your home directory and launch it with the `-vm` option pointing to the Java binary:

```
cd ~
tar xzf Downloads/eclipse-java-helios-linux-gtk.tar.gz
eclipse/eclipse -vm jdk1.5.0_22/bin/java
```

In Eclipse, go to Help->Install new software. Set the URL to "Work with" to

<https://dl-ssl.google.com/android/eclipse/>

Select "Developer Tools" and click Install.

Install a statically-linked copy of Busybox

Busybox contains a lot of useful Unix commands that are missing from Android, plus it has a decent shell *with proper command line editing*.

```
tar xjf busybox-1.16.1.tar.bz2
cd busybox-1.16.1/
make menuconfig
```

In Busybox Settings->Build Options select **Build BusyBox as a static binary** and **exec prefers applets**, then

Inside Android lab notes

```
make
```

Copy it to Android

```
adb shell  
mkdir /data/bin  
exit
```

Then copy busybox:

```
adb push busybox /data/bin
```

Now you can run busybox from the Android shell and get it to run a nice ash shell for you:

```
/data/bin/busybox ash
```

2. Making Use of Android

Hello World

If you don't have it already, install ant:

```
sudo apt-get install ant
```

Get a list of Android SDK versions installed on your laptop:

```
$ android list targets
Available Android targets:
id: 1 or "android-8"
  Name: Android 2.2
  Type: Platform
  API level: 8
  Revision: 2
  Skins: WQVGA432, QVGA, WVGA854, WQVGA400, WVGA800, HVGA (default)
```

In this case there is only one, and it has an id of 1. Create a project using this target:

```
android create project --target 1 --name Hello \
--path ./helloworld --activity HelloWorld \
--package com.example.HelloWorld
```

Note that the directory helloworld/ has been created and contains a skeleton project. Edit helloworld/src/com/example/HelloWorld/HelloWorld.java and add a TextView as shown in the slides. Build it by typing

```
ant debug
```

Note that helloworld/bin/Hello-debug.apk has been created. Start the emulator and install your package

```
adb install -r bin/Hello-debug.apk
```

In the emulator, click the application launcher icon:



You should see your application there, so click it!

Native code

Download a copy of the ndk from <http://developer.android.com/sdk/ndk/index.html>.

Extract it to your home directory

```
cd ~
unzip Downloads/android-ndk-r4b-linux-x86.zip
```

Next you are going to build one of the sample applications, HelloJni. Create a new Android project

Inside Android lab notes

```
cd ~
mkdir apps
cd apps
android create project --target 1 --name HelloJni --path ./hello-jni --activity
HelloJni --package com.example.hellojni
```

Copy the ndk sample code over the top

```
cp ~/android-ndk-r4b/samples/hello-jni/* hello-jni
```

Build the native code...

```
cd ~/apps/hello-jni
~/android-ndk-r4b/ndk-build
```

Build the project

```
ant debug
```

Run the emulator and then install the project

```
adb install -r bin/Hellojni-debug.apk
```

Test that it works. You should see the string "Hello from JNI !" on the screen and if you run logcat you should see something like this:

```
I/ActivityManager( 58): Start proc com.example.hellojni for activity
com.example.hellojni/.HelloJni: pid=453 uid=10013 gids={1015}
D/dalvikvm( 453): Trying to load lib
/data/data/com.example.hellojni/lib/libhello-jni.so 0x43e372e8
D/dalvikvm( 453): Added shared lib
/data/data/com.example.hellojni/lib/libhello-jni.so 0x43e372e8
D/dalvikvm( 453): No JNI_OnLoad found in
/data/data/com.example.hellojni/lib/libhello-jni.so 0x43e372e8, skipping init
I/ActivityManager( 58): Displayed activity com.example.hellojni/.HelloJni:
1049 ms (total 1049 ms)
```

3. Porting to custom hardware

Merging the Android kernel

This is a question of juggling with three or four kernel trees. They are

1. kernel for your board
2. the mainline kernel it was based on
3. the android kernel
4. the mainline kernel it was based on

It makes life simpler if the mainline kernel is the same in both cases:

```
mainline kernel /--> board kernel
                \--> Android kernel
```

In this case, you “just” need to create a patch of differences between Android and mainline and apply it to the kernel for your board, and then clear up any conflicts. Simple? No, but to make life a bit easier here are some hints.

Install git if you don't have it already

```
sudo apt-get install git
```

Clone the Android kernel tree (this will take a while):

```
git clone git://android.git.kernel.org/kernel/common.git android-kernel
```

List the branches:

```
cd android-kernel
git branch -a
* android-2.6.27
remotes/origin/HEAD -> origin/android-2.6.27
remotes/origin/android-2.6.25
remotes/origin/android-2.6.27
remotes/origin/android-2.6.29
remotes/origin/android-2.6.32
remotes/origin/android-2.6.35
remotes/origin/android-goldfish-2.6.27
remotes/origin/android-goldfish-2.6.29
```

This says that the current branch is android-2.6.27. Suppose you want to use android-2.6.32 as a base, create a new local branch, named android-2.6.32 in this case, which is based on remote branch origin/android-2.6.32:

```
git checkout --track -b android-2.6.32 origin/android-2.6.32
```

Now git branch -a will confirm that you are on android-2.6.32.

Inside Android lab notes

The next stage is to find the differences between the mainline and Android kernels. First you need to synchronise the tags with the mainline kernel (if you list the tags with "git tag -l" you will see that more recent ones are missing):

```
git fetch --tags git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6.32.y.git
```

Now, say you want to generate a list of patches up to mainline 2.6.32.9:

```
git format-patch v2.6.32.9..HEAD
```

Then apply these patches to the kernel for your board:

```
cd your-kernel
for f in android-kernel/*.patch; do
    patch -p1 < $f
done
```

Look out for patches that don't apply cleanly and go and fix them up (have fun!). Finally, modify the kernel configuration to include the Android specific options – see Documentation/android.txt in Linux kernel trees up to and including 2.6.32.

Building the Android Open Source Project code

Install repo

```
curl http://android.git.kernel.org/repo > ~/bin/repo
chmod a+x ~/bin/repo
mkdir working-directory-name
cd working-directory-name
repo init -u git://android.git.kernel.org/platform/manifest.git
```

Then, synchronise with

```
repo sync
```

(Note this will take quite a long time the first time because it will download several GiB of code)

Repo Manifests

Repo is a way of managing multiple git repositories. The list of repositories is held in a manifest, which is a text file in xml format.

When you create a repository using "repo init" you need to give a manifest, default is "default.xml". Add a -m to specify a different manifest, e.g.

```
repo init -u git://android.git.kernel.org/platform/manifest.git -m dalkvik-plus.xml
```

To specify a revision, that is, a particular manifest-branch, use the -b option. For example:

Inside Android lab notes

```
repo init -u git://android.git.kernel.org/platform/manifest.git -b release-1.0
```

You can view the manifest used when the repo was initialised by looking in `.repo/manifest.xml`

Building the default (generic) product

Type

```
cd myandroid
export JAVA_HOME=$HOME/jdk1.5.0_22
PATH=$JAVA_HOME/bin:$PATH
. build/envsetup.sh
m
```

Compiles the run-time tools and file system:

`myandroid/out/host/linux-x86` - tools, e.g. adb

`myandroid/out/target` - target run-time images:
 `target/product/generic/system.img`, `userdata.img`, `ramdisk.img`

`myandroid/target/product/generic/root/` - rootfs

To create the sdk, type

```
make sdk
```

result is in

`out/host/linux-x86/android-sdk_eng.chris_linux-x86/`

and are zipped up into

 Package SDK: `out/host/linux-x86/sdk/android-sdk_eng.chris_linux-x86.zip`