CITY UNIVERSITY OF HONG KONG

DEPARTMENT OF ELECTRONIC ENGINEERING

EE5414 Laboratory - Part II Report

Author:

Wangchen DAI (53623708)

Jingwei HU (53656463)

Instructor:

Dr. L.L.CHENG

November 18, 2014

I. ANDROID SOURCE CODE COMPILATION AND ANDROID BOOTING PROCEDURE

Running Android OS on BeagleBone-XM platform requires following software components

- Boot Image
- Andorid File System
- Other Resources

First of all, Linux Kernel (uImage), Boot Loader (u-boot.in) and Bootstrapping (MLO) are needed to construct a boot image; Then Android Filesystem should be built , in our experiment, we use Android Gingerbread for demonstration; Finally, other media resources, such as video and music are optional to be included for better human-machine interface.

A. Set Up Toolchains

In the experiment, we compile the source code to obtain these essential components using cross-platform C compiler — arm-wabi-gcc 4.4.0. We can permanently include this compiler by adding the following command in "~/.bashrc"

```
export PATH=~/mydroid/prebuilt/linux-x86/toolchain/arm-eabi-4.4.0/bin/:$PATH
```

B. Build Kernel

First, the source code for building kernel is under /mydroid/kernel, thus we change our working directory to that path:

```
cd ~/mydroid/kernel
```

Then we compile these codes by three steps — clean, configure, make

```
make CROSS_COMPILE=arm-eabi- distclean
make CROSS_COMPILE=arm-eabi- omap3_beagle_android_defconfig
make CROSS_COMPILE=arm-eabi- uImage
```

Normally, this process takes about 15 minutes, after which generates Linux Kernel Image 'uImage' in arch/arm/boot/. Fig. 1 shows info displayed in the terminal when the compilation is completed.

Fig. 1.   Kernel successfully compiled

C. Build Boot Loader

We change directory to '∼/mydroid/uboot'. Similar to kernel compilaton, three steps are required:

```
make CROSS_COMPILE=arm-eabi- distclean
make CROSS_COMPILE=arm-eabi- omap3_beagle_config
make CROSS_COMPILE=arm-eabi-
```

It should take less than 5 minutes and 'u-boot.bin' will be generated under the current directory. We list in Fig. 2 compiling information for reference.

D. Build x-loader

From the 'mydroid' sources directory

```
cd ~/mydroid/x-loader
```

Execute the following commands to the kernel sources

```
make CROSS_COMPILE=arm-eabi- distclean
make CROSS_COMPILE=arm-eabi- omap3beagle_config
make CROSS_COMPILE=arm-eabi-
```

This leads to the x-loader Image 'x-load.bin' shown in Fig. 3

To create the MLO file used for booting from a MMC/SD card, sign the x-loader image using the signGP tool found in the Tools directory of the Devkit.

Fig. 2. U-boot successfully compiled

Fig. 3. X-loader successfully compiled

```
../../Tools/signGP/signGP ./x-load.bin
```

The signGP tool will create a x-loader.bin.ift file, that can be renamed to MLO.

E. Build boot.scr

boot.scr is required for booting the system, and it can be generated by mk-bootscr. First download 'TI_Android_GingerBread_2_3_4_DevKit_2_1' which has been provided in the Win7 sharefolder. Then execute the following commands:

```
cd TI_Android_GingerBread_2_3_4_DevKit_2_1/TI_Android_GingerBread_2_3_4_DevKit_2_1/Tools/mk-
    bootscr/
./bootscr
```

F. Build Android Filesystem

Build Android filesystem for Beaglebone-xm by doing following:

```
make TARGET_PRODUCT=beagleboard OMAPES=5.x -j8
make TARGET_PRODUCT=beagleboard droid
```

Fig. 4 depicts the corresponding messages from the terminal.

Fig. 4.   Filesystem successfully compiled

Next we compress the files generated into one single tar package.

```
cd out/target/product/beagleboard
cp -r root/* android_rootfs
cp -r system android_rootfs
sudo ../../../../build/tools/mktarball.sh ../../../host/linux-x86/bin/fs_get_stats android_rootfs
    . rootfs rootfs.tar.bz2
```

Note that rootfs.tar.bz2 is exactly the Android filesystem that we need in this experiment.

G. Build OS image

For simplicity, we rearrange those files compiled from source code into one folder 'beagleboard-xm' as shown in Fig. 5.
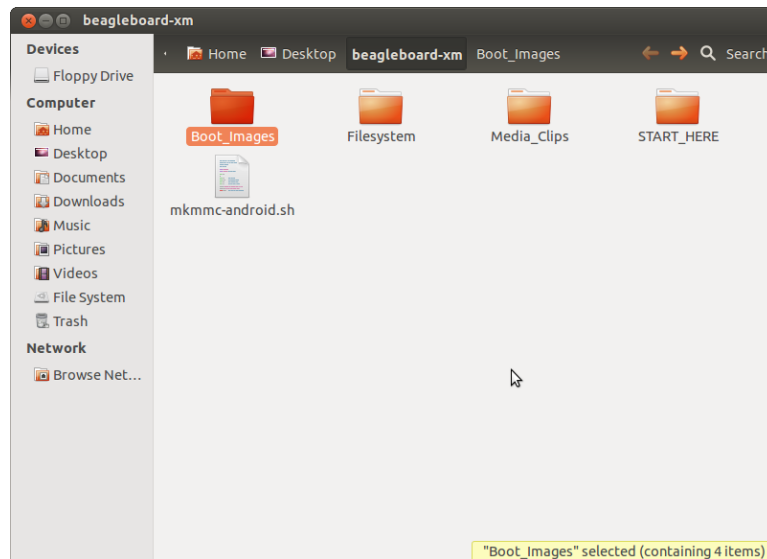
Fig. 5. Pre-built Image

In the subfolder 'Boot_Images', we place boot.scr, MLO, u-boot.bin, uImage; In 'Filesystem', we put rootfs.tar.bz2; Other resources like video and pictures should be laid in 'Media_Clips', which can be found from Lab 1. The shell script 'mkmmc-android.sh' is also the same as used in Lab 1.

Insert your SD card and perform the following commands, a new image would be ready for booting.

```
sudo ./mkmmc-android /dev/sdb MLO u-boot.bin uImage boot.scr rootfs.tar.bz2
```

II. INSTALL AND UNINSTALL APPLICATIONS

Before install applications on the Beagleboard, ADB Daemon and network need to be set up. Enter following commands on host PC to install ADB via apt-get:

```
sudo add-apt-repository ppa:nilarimogard/webupd8
sudo apt-get update
sudo apt-get install android-tools-adb
```

Next, the connection between host PC and Beagleboard needs to be set. To set the Host PC IP to 192.168.194.1:

```
sudo ifconfig eth0 192.168.194.1 netmask 255.255.255.224 up
sudo route add 192.168.194.2 dev eth0
```

then set the IP of the board to 192.168.194.2, we could configure the board via Host PC using the command:

```
sudo minicom -c
```

the network setup command would be:

```
# ifconfig usb0 192.168.194.2 netmask 255.255.255.224 up
```

note that "#" indicates that the command is typed to configure Beagleboard via Minicom. We could PING the IP address of the board from Host PC to see if the connection is set correctly, see Fig. **??**. Input the following commands to configure the on-board ADB service:

Fig. 6. Connection test between Host PC and the board

```
# setprop service.adb.tcp.port 5555
# stop adbd
# start adbd
```
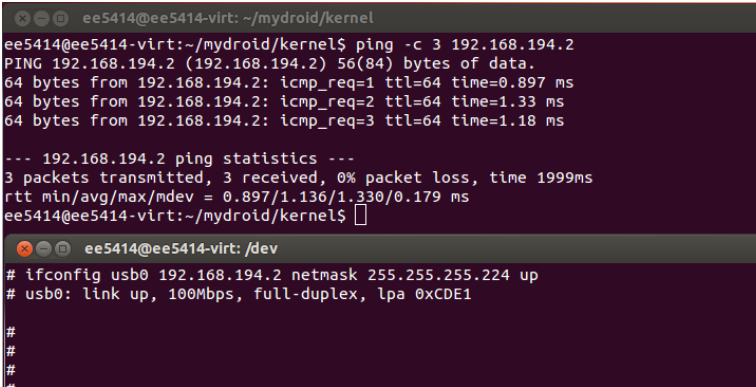
On Host PC, enter the following commands to establish ADB connection:

```
export ADBHOST= 192.168.194.2
adb kill-server
adb start-server
```

To verifying the connection:

```
adb devices
```

Fig. 7 indicates the ADB connection is established successfully. Now, we could also configure the board by either Minicom or ADB Shell. See Fig. 8 After completing ADB connection setup, the on-board Android applications can be managed on Host PC, the following commands are examples of installation/uninstallation applications:

Fig. 7. ADB connection established

Fig. 8. ADB connection established

```
adb install qq.apk
adb uninstall com.android.vending.apk.apk
```

An alternative way to uninstall applications:

```
sudo adb shell
# rm /apk/file/path
```

III.   Q&A

In this section, we answer those questions raised by the sample report.

```
>> Q1 <<
> What is the version of toolchain or gcc you are using?
```

Reply: We can type in the terminal 'which arm-eabi-gcc' to confirm the gcc version we are using. In this experiment, we shall obtain 'arm-wabi-gcc-4.4.0'.

```
>> Q2 <<
> Why is static library and not dynamic library and so on?
```

Reply: Dynamic library is used because we cannot ensure each host machine has the related .dll files for a complete compilation.

```
>> Q3 <<
>  Explain why use objcopy?
```

Reply: Bootloader only accepts .bin(raw binary) files, so if we just have S-record file and then objcopy can help convert it into a different form, e.g. .bin file.

```
>> Q4 <<
> How you make boot.src and why you need to do so.
```

Reply: We have demonstrated how to make boot.src in section I.(E). boot.src is a U-Boot script. It contains instructions for U-Boot. Using these instruction, the kernel is loaded into memory, and (optionally) a ramdisk is loaded. boot.scr can also pass parameters to the kernel.

```
>> Q4 <<
> For the kernel compilation, try to explain the names marked,
> e.g. CLEAN, HOSTCC, HOSTLD, SHIPPED, UPD, CC, AS, GEN, CHK.
```

Reply: In fact, these are nothing but variable alias defined in the makefile under 'kernel' directory. For example, AS indicates the specific assembler of arm-eabi-gcc-4.4.0; HOSTCC indicates the gcc preinstalled in the host Ubuntu OS;