Hands-On Exercises for

Embedded Android

v. 2012.02

WARNING:
The order of the exercises does not always follow the same order of the explanations in the slides. When carrying out the exercises, carefully follow the exercise requirements. Do **NOT** blindly type the commands or code as found in the slides. Read every exercise **in its entirety** before carrying out the instructions.

These exercises are made available to you under a Creative Commons Share-Alike 3.0 license. The full terms of this license are here:

https://creativecommons.org/licenses/by-sa/3.0/

Attribution requirements and misc.:

- This page must remain as-is in this specific location (page #2), everything else you are free to change; including the logo :-)

- Use of figures in other documents must feature the below "Originals at" URL immediately under that figure and the below copyright notice where appropriate.

- You are free to fill in the space in the below "Delivered and/or customized by" section as you see fit.

(C) Copyright 2010-2012, Opersys inc.

These exercises created by: Karim Yaghmour

Originals at: www.opersys.com/community/docs

Delivered and/or customized by:

Day 1

Introduction to Embedded Android

1. Get the repo tool (see slide 83)
2. Download gingerbread branch of the AOSP (see slide 83)
3. Have a look at the following websites to get a good grasp of available resources:
    • source.android.com
    • developer.android.com
    • tools.android.com
    • cyanogenmod.com
    • wiki.cyanogenmod.com
    • elinux.org/Android_Portal
    • www.omappedia.org/wiki/OMAP_Android_Main

Concepts and Internals

1. Get and install the JDK
2. Get and install Eclipse (make sure OpenJDK is installed along w/ Eclipse)
3. Get and install the SDK. If you are running a 64-bit Ubuntu, make sure you have "ia32-libs" installed using: "sudo apt-get install ia32-libs".
4. Get and install the ADT plugin for Eclipse
5. Use the Android SDK and AVD manager to download the 2.3.3 platform. Don't download any other platform as the more platforms you download, the more time it will take. One platform is sufficient for what we will be doing.
6. Create a virtual machine using the "android" tool
7. Create a HelloWorld program and run it

Android Open Source Project

1. Install all ubuntu packages required for the AOSP build
2. Apply any required symbolic link fixes
3. Apply any required patches and/or fixes to the AOSP
4. Build the AOSP
5. Check out the size of the images generated in [aosp]/out/target/product/generic
6. Start the emulator and go to the settings to check which version of the AOSP thinks it's running.
7. Modify the AOSP to add the HelloWorld app created in the previous section. Use an Android.mk similar to this in your [aosp]/packages/apps/HelloWorld/ directory (you can copy one from another app in [aosp]/packages and edit it):

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

LOCAL_MODULE_TAGS := optional

LOCAL_SRC_FILES := $(call all-java-files-under, src)

LOCAL_PACKAGE_NAME := HelloWorld

include $(BUILD_PACKAGE)

# Use the following include to make our test apk.
include $(call all-makefiles-under,$(LOCAL_PATH))
```
7. Recompile, launch emulator and HelloWorld app

Kernel Basics
1. Check kernel version in emulator ("cat /proc/version")
2. Fetch kernel for Android from android.git.kernel.org. Use "git" to fetch the "qemu" kernel.
3. Revert the checked-out kernel to 2.6.29. Use "git checkout android-goldfish-2.6.29".
4. Get the kernel configuration from emulator. Use "adb pull" to retrieve the "/proc/config.gz" file from the running emulator generated by the previous AOSP build
5. Apply the retrieved configuration to the kernel downloaded earlier
6. Use menuconfig to add support for modules and module unloading to that kernel. These are two separate options. The former is on the top-most menu and the latter is visible when you enter the modules sub-menu from the top menu.
7. Cross-compile kernel with the ARM toolchain in [aosp]/prebuilt/linux-x86/toolchain/arm-eabi-4.4.0/bin/
8. Boot the emulator with newly-built kernel and check the kernel version run by the emulator
9. Check the new kernel's version ("cat /proc/version")

Day 2

Linux root filesystem

1. Get and install the codesourcery ARM toolchain. Download the GNU/Linux toolchain: http://www.codesourcery.com/sgpp/lite/arm/portal/release1803 . FYI: There's another toolchain provided from CodeSourcery marked as plain "EABI", but that is for bare-metal compiling and we won't be using it.

2. Create yourself an embedded Linux workspace script such as the one on slide 111 for housing a build environment for "classic" embedded Linux components.

3. Create the workspace layout (i.e. directories) listed in slide 110 within the workspace pointed to by the previous step.

4. Create yourself a root filesystem that includes the glibc shared libraries as in slides 115 and 121. Note that you'll need to replace CODESOURCERY_DIR with the location where you installed your CodeSourcery toolchain in exercise #1 of this section.

5. Download, configure, build and install BusyBox using the codesourcery ARM toolchain into the root filesystem you just created.

Native Android user-space

Preamble:

- To the exception of exercise 5, it is implied that you should build the AOSP and run the emulator after each exercise to check that your changes have indeed been correctly reflected in your newly modified AOSP.
- Once built, the AOSP will have an [aosp]/out/target/product/generic/ directory containing a directory and a corresponding image for:
  - The root filesystem (root/ and ramdisk.img)
  - /system (system/ and system.img)
  - /data (data/ and userdata-qemu.img)
- If you modify the AOSP or the linux root filesystem created earlier, you may need to erase those directories and images from [aosp]/out/target/product/generic/ to force the AOSP's build system to regenerate them. Sometimes, those directories and images don't get updated properly, hence the need to erase them and using "make" within [aosp]/ to force their re-creation.
- Do **NOT** get rid of [aosp]/out/target/product/generic/ itself or any of the parent directories. If you do so, you will have to wait for the AOSP to rebuild itself. Only get rid of the previously-mentioned directories on a case-by-case basis as needed.

1. Modify AOSP build system to copy content of your rootfs to its default ramdisk. You will need to:

- Amend [aosp]/system/core/rootdir/Android.mk to make it look like this (a. additions are

in bold, b. don't forget to use TABS, not spaces for the "my_dir" make target's command, c. PATH_TO_MY_GLIBC_ROOTFS is a placeholder you need to replace with the location of the filesystem created in the exercises of the previous section):

```
...
    $(TARGET_OUT_DATA)

my_dir:
    cp -a $(PATH_TO_MY_GLIBC_ROOTFS)/* $(TARGET_ROOT_OUT)

$(DIRS): my_dir
...
```

- Amend [aosp]/system/core/include/private/android_filesystem_config.h's "static struct fs_path_config android_files[]" to add an entry for "lib/*" so that the execute bit is preserved for all files in that directory.

2. Modify init so that it starts BusyBox httpd by default.
- You will need to add something similar to the following to your init.rc:
```
service httpd /usr/sbin/httpd
        oneshot
```
If you add this at the bottom of the file, make sure there's a line-wrap after the last line.
- Make sure the system path includes "/usr/sbin". That path is set in the init.rc file which is placed into your rootfs. The original copy of it is in [aosp]/system/core/rootdir/. The path setting line in your init.rc should then look something like:
```
export PATH /sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin :/usr/sbin
```

3. Modify the AOSP so that ADB uses BusyBox' shell instead of the default Android shell.
- ADB is in [aosp]/system/core/adb
- You need to modify the SHELL_COMMAND macro in services.c
```
#if ADB_HOST
#define SHELL_COMMAND "/bin/sh"
#else
//#define SHELL_COMMAND "/system/bin/sh"
#define SHELL_COMMAND "/bin/sh"
#endif
```
- Also make sure the system path ($PATH) points to the BusyBox "binaries". The path setting line in your init.rc should then look something like:
```
export PATH /bin:/sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin :/usr/sbin
```

4. Write a C program that is packaged into the "embedded Linux" root filesystem and that opens a socket to listen for a connection and prints out a message to the console accordingly. Write a C program that is built as part of the native Android utilities in frameworks/base/cmds and that connects to a sockets and prints out a message stating that the connection was successful. Examples can be found here (we recommend you use the first one):

http://www.linuxhowtos.org/C_C++/socket.htm

http://www.pcs.cnu.edu/~dgame/sockets/sockets.html

To have a C component build against Bionic:
- Add it as [aosp]/frameworks/base/cmds/foo
- Make sure "foo" is added to [aosp]/build/core/user-tags.mk
- Make sure you have an appropriate Android.mk within [aosp]/frameworks/base/cmds/foo to get your app to build. Here's a sample:
```
LOCAL_PATH:= $(call my-dir)
```

```
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
        foo.cpp

LOCAL_SHARED_LIBRARIES := \
        libcutils

LOCAL_MODULE:= foo

include $(BUILD_EXECUTABLE)
```

5. Create a setup where a second emulator instance is able to view pages served by the httpd running on a first emulator instance. See the "Using the Android Emulator" documentation for more information and how it makes use of "telnet" to connect to a special control port of the emulator. Note that there are "magic" IPs and that 10.0.2.2 in the emulator points to your host's loopback interface (127.0.0.1).

A quick introduction to Java

1. Create and run a HelloWorld in Java
2. Create a java program that creates two threads that call a same synchronized function that increments a number shared by the two threads and prints out the number's value along with a string identifying the caller. Have a look at the separate and combined use of "static" and "synchronized".

System server

1. Use the following tools to observe the System Server's behavior: logcat, dumpsys, dumpstate
2. Use strace to monitor the operation of dumpsys. You should see dumpsys using ioctl() to talk to the binder driver (/dev/binder).
3. Add your own system service that prints out a message to the logs when it is called. If you copy and paste from the slides, make sure the pasting doesn't introduce "magic characters" that will cause the compiler to complain about your code.
4. Build the AOSP and use "logcat" to check that your service is started upon system startup.
5. Modify the HelloWorld program merged earlier into the AOSP to invoke your new system service's callback. Make sure your Android.mk **doesn't** include a "LOCAL_SDK_VERSION" tag.
6. Implement the dump() function in your system service to allow dumpsys to poke it for status. The prototype is:

```
@Override
public void dump(FileDescriptor fd, PrintWriter pw, String[] args) {
...
}
```

Day 3

Linux device driver

1. Create a char device driver that implements a circular buffer over the read/write file-ops. Refer to LDD3 slide-set for both code examples and a Makefile. Do not implement blocking/waking semantics. If there's no content available, just return 0 bytes. If you need inspiration, have a look at the driver at the following address as the basis for your own driver: http://tldp.org/LDP/lkmpg/2.6/html/x569.html

LDD3 is here: http://lwn.net/Kernel/LDD3/

- It's best to use misc_register() instead of register_chrdev() as it will register your char dev <u>and</u> fire off a hotplug event.
- Don't forget you'll need something like this to build your `module`:
  `$ make ARCH=arm CROSS_COMPILE=~/aosp-2.3.4/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin/arm-eabi-`

2. Load your driver into your custom-built kernel on your emulator. Use adb to push the ".ko" to "/data/local" temporarily. "/data" is mounted from an image which is persisted to disk at runtime. That image, however, will get destroyed when the AOSP is used to generate an SDK. A more permanent location for the driver would be a "/lib/modules" directory in the embedded Linux filesystem we created earlier.

3. Create a device node in /dev for your device (see /proc/devices for its major number.)

4. Use "cat" and "echo" to "read" and "write" to your "device". For example, "cat /dev/foo" will read from the device and "echo *string* > /dev/foo" will write to it.

HAL

1. Create a libopersyshw.so that exposes your driver's read/write functionality through basic read/write functions.

- Place libopersyshw as "opersyshw/" in [aosp]/sdk/emulator/; the latter contains some HAL implementations for the virtual devices in the emulator. Refer to [aosp]/sdk/emulator/gps for an example.
- You will need to ensure that the driver is loaded prior to libfoo.so being loaded by the System Server. To do that, modify the init.goldfish.sh script in [aosp]/system/core/rootdir/etc to run insmod on the .ko file.
- If you had used the "misc_register()" call in your driver to register it as a miscellaneous character driver, you will need to modify the ueventd.rc file in [aosp]/system/core/rootdir/ in order to ensure the the /dev entry created for your device has the proper access rights set to it. By the default, an entry will be created but it will only be accessible to "root" and they System Server doesn't run as root. Sample ueventd entry:
  `/dev/mychar          0660    system    system`

2. Write the JNI code that allows the functions of your libopersyshw.so to be exposed to Java.

3. Extend the previously-implemented system server to allow it to invoke the read/write functions provided by libopersyshw.

- To test the behavior of the entire opersys stack, hard-code calls to write() and read() in the Opersys Service's constructor in OpersysService.java.
- Remember that you will need to modify the AIDL file matching the Opersys Service in order to expose the new read() and write() calls through Binder.

4. Implement a dump() within your system server in order to allow the dumpsys utility to retrieve the number of calls read() and write() having been made since system startup.

Day 4

HAL use

1. Change the HelloWorld app added to the AOSP to use the system server's calls to do read/write operations to libopersyshw

Android Framework

1. Amend the framework to make the Opersys system service available to apps through binder

2. Modify the status bar in order to remove the phone signal icon

Custom SDK

1. Create your own SDK allowing developers to call Opersys as any other system service

2. Configure Eclipse to use your newly-built SDK

3. Create a new app in eclipse that uses the newly-built SDK to call on the Opersys system service

Day 5

Supplemental Exercises

1. Modify the driver to use "misc_register()" instead of "register_chrdev()" to register the device. "misc_register()" will cause a hotplug event to be sent to user-space which will be caught by ueventd. Change ueventd's ueventd.rc to create the /dev entry with the proper rights. You should then not need to manually create a node in /dev entry for your device any more.

2. Implement the ioctl() call of the driver to:
>   a. Zero out the content of the circular buffer
>   b. Poll the driver to see if there's new data in the buffer
>   c. Get last write time-stamp
>   d. Get the number of calls to read()
>   e. Get the number of calls to write()
>   f. Set the entire content of the buffer to a given character value

3. Modify the OpersysManager's API and the stack underneath it all the way to the driver to provide access to the following methods:
>   void clearBuffer(), which empties the circular buffer
>   boolean isThereContent(), which tells the caller whether there's content to read
>   long getLastWriteTime(), which returns the time at which the last write happens
>   int getReadStat(), which returns the number of read() calls made to the driver
>   int getWriteStat(), which returns the number of write() calls made to the driver
>   void setBufferToChar(char), which sets the buffer's content to a char value

4. Update the SDK and make sure you can use these calls from within an app created within Eclipse.

5. Add a "void enableIntentOnContent()" and "void disableIntentOnContent()" to the OpersysService that makes it so that the service periodically polls the driver to see if it's got new content and broadcasts a new Intent when new content is found. Use the driver's ability to give you the time of the last write to avoid sending an Intent twice for the same addition to the buffer.

6. Modify your app to call on "enableIntentOnContent()" and catch the Intent with a BroadcastReceiver.